Efficient Model Checking of PSL Safety Properties

Tuomas Launiainen Keijo Heljanko

Tommi Junttila

Aalto University

School of Science

Department of Information and Computer Science

PO Box 15400, FI-00076 Aalto, Finland

{Tuomas.Launiainen,Keijo.Heljanko,Tommi.Junttila}@tkk.fi

April 8, 2011

Abstract

Safety properties are an important class of properties as in the industrial use of model checking a large majority of the properties to be checked are safety properties. This work presents an efficient approach to model check safety properties expressed in PSL (IEEE Std 1850 Property Specification Language), an industrial property specification language. The approach can also be used as a sound but incomplete bug hunting tool for general (non-safety) PSL properties, and it will detect exactly the finite counterexamples that are the informative bad prefixes for the PSL formulae in question. The presented technique is inspired by the temporal testers approach of Pnueli and co-authors, but unlike theirs, our approach is aimed at finding finite counter-examples to properties. The new approach presented in this paper handles a larger syntactic subset of PSL safety properties than earlier translations for PSL safety subsets and has been implemented on top of the open source

NuSMV 2 model checker. The experimental results show the approach to be a quite competitive model checking approach when compared to a state-of-the-art implementation of PSL model checking.

1 Introduction

Safety properties are an important class of properties as in the industrial use of model checking a large majority of the properties to be checked are safety properties. Safety properties are also interesting from the point of view that they can be reduced to invariant checking without a blow-up in the number of state variables in the system to be checked. This enables a larger variety of model checking algorithms to be applied to them, such as the use of interpolants [22], that are restricted to invariant properties.

In this work we present an approach for model checking of safety properties expressed in PSL (IEEE Std 1850 Property Specification Language), an industrial property specification language. PSL combines temporal operators from linear temporal logic (LTL) with regular expressions, and is therefore strictly more expressive than LTL [6]. Our approach can also be used as a sound but incomplete bug hunting tool for general (non-safety) PSL properties, and it will detect exactly the finite counterexamples that are the informative bad prefixes (see Section 2.3) for the PSL formulae in question. The semantics in our approach is based on [11] which coincides with the latest revision of the semantics of PSL [10]. Our approach extends to PSL the approach of [19] for finding informative bad prefixes for linear temporal logic (LTL) formulae. Thus our approach is sound for all PSL formulae in the following sense: If our approach finds a counterexample then a counterexample exists by the semantics of PSL. Otherwise, if our approach does not find a counterexample, then there is no informative bad prefix ([19], see also Section 2.3) for the PSL formula in question. On the technical level our approach is inspired by [18], and similar

to [23], but instead of general (non-safety) properties with temporal testers on infinite words or words ending in a deadlock, our approach is tailored for discovering finite counter-examples to safety properties with transducers. Additionally, our paper uses an updated version of the PSL semantics. The transducer we generate is translated into a NuSMV observer module that has an invariant specification. This technique resembles the classical LTL model checking approach where LTL formulas are translated into Büchi-automata that are implemented as SMV modules with fairness constraints [9]. Our technique differs from that by searching for finite word counterexamples instead of infinite ones. This paper is an extended version of [21], with fixes, complete proofs, and more examples.

There are a number of papers that encode smaller syntactic subsets of PSL safety properties than our encoding. The most widely known is the so called safety simple subset [1, 16, 15, 4]. Our encoding handles a strictly larger syntactic subset of PSL safety properties but is not directly suitable for runtime monitoring, as we use nondeterminism in the generated transducers for better succinctness. Our approach is designed to be used in combination with model checking algorithms and is thus significantly more succinct than the approach of [12] tailored to be used in runtime monitoring of PSL in a simulation setting. The approach of [19] for encoding informative bad prefixes for LTL formulae has been implemented in the `scheck` tool [20] in the context of explicit state model checking. Our approach is different in the way that it is a symbolic model checking approach detecting all informative bad prefixes of PSL formulae.

One could argue that using the liveness-to-safety reduction [25, 26, 2] eliminates the need for any specialized model checking algorithms for safety properties as it can reduce model checking of general (non-safety) properties to invariant checking, and as such only optimizing algorithms for invariant checking would suffice. However, this reduction doubles the number of system state variables and is thus often

impractical from an efficiency perspective. While our approach also reduces the model checking problem to invariant checking, ours adds much less state variables[1]. This is an important distinction especially with model checking techniques such as symbolic model checking with BDDs [5] that are quite sensitive to the number of state bits in the model. A more traditional approach to model checking general (non-safety) properties with BDDs is to find accepting cycles using nested fixpoint computations, see e.g. [13]. There is also a symbolic algorithm with a better theoretical worst-case complexity [3]. The problem with these fixpoint algorithms is that their use often leads to slow running times in BDD-based model checkers when compared to simple invariant checking used by algorithms for safety properties.

In our experiments we compare to the state-of-the-art symbolic encoding of all PSL properties [8]. The experiments show that the approach presented in this work is a very efficient model checking approach. Especially in combination with BDD-based symbolic model checking it avoids the use of costly algorithms used to find accepting cycles with BDDs and instead relies on simple and more efficient invariant checking.

The structure of the rest of the paper is as follows. Section 2 describes the syntax and semantics of PSL, as well as introduces the central notion of informative bad prefixes. In Section 3 transducers are introduced and it is shown how one can construct a transducer for a PSL formula. In Section 4 the implementation of model checking based on transducers encoded as NuSMV modules is described. Section 5 reports on the experiments and Section 6 presents the conclusions.

[1] I.e. only a linear amount w.r.t. the size of the PSL formula, where the size of a Sequential Regular Expression is the number of states in the automaton that represents them.

2 PSL syntax and semantics

This section formally defines the syntax and semantics of PSL and is based on the revised standard [10, 11]. The semantics are divided into three variants: strong, neutral, and weak. The three variants are identical for infinite paths, but differ for finite paths. The full set of PSL operators is supported by this work, including several operators that can be easily rewritten using those presented here. The PSL extensions with local variables suggested in [10] as well as the Optional Branching Extension are not considered.

2.1 Syntax

This section presents the syntax of PSL, which is similar to the one in [10]. Assume a non-empty set of atomic propositions AP. The syntax of Sequential Extended Regular Expressions (SERE) is defined by the grammar

$$\begin{aligned} r \quad ::= \quad & [*0] \mid p \mid \neg p \mid r_1[+] \mid r_1 \cdot r_2 \mid r_1 \circ r_2 \mid \\ & r_1 \cup r_2 \mid r_1 \cap r_2, \end{aligned}$$

where p varies of over atomic propositions in AP and r_1 and r_2 are SEREs. A SERE defines a language of words over the alphabet 2^{AP}. Intuitively, $[*0]$ denotes the empty word, $r_1[+]$ is the Kleene plus operator, $r_1 \cdot r_2$ is the concatenation of two SEREs, $r_1 \circ r_2$ is the concatenation of two SEREs with an overlap of a single state, and $\cup$ and $\cap$ denote the standard union and intersection. Additionally, we write $r[*]$ to denote $r[+] \cup [*0]$, **true** to denote $(p \cup \neg p)$, and **false** to denote $(p \cap \neg p)$ for some $p \in AP$. PSL also uses a conjunction operator $(r_1 \;\&\; r_2)$ to denote words that match both operand SEREs but one need not be matched tightly. We omit this operator because it can be rewritten with $(r_1 \cap (r_2 \cdot \mathbf{true}[*])) \cup ((r_1 \cdot \mathbf{true}[*]) \cap r_2)$.

The syntax of PSL formulae is defined by the grammar

$$\phi \quad ::= \quad p \mid \neg\phi_1 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \,\mathbf{U}\, \phi_2 \mid \phi_1 \,\mathbf{R}\, \phi_2 \mid$$

$$\mathbf{X!}\, \phi_1 \mid \mathbf{X}\, \phi_1 \mid r \mapsto \phi_1 \mid r \diamond\!\!\rightarrow \phi_1 \mid r \mid r!,$$

where ϕ_1 and ϕ_2 are PSL formulae, r is a SERE, and p varies over atomic propositions in AP. As usual, we also use the abbreviations $\mathbf{true} \equiv (p \vee \neg p)$ and $\mathbf{false} \equiv (p \wedge \neg p)$ for some $p \in AP$. The $\mathbf{X!}$, $\mathbf{X}$, $\mathbf{U}$ and $\mathbf{R}$ operators are the "strong next", "weak next", "until" and "releases" temporal operators also used in LTL. The key difference between PSL and LTL are the SERE operators. Using SEREs in temporal properties allows a different style of specifying some properties that are equivalent to LTL properties, but also specifying properties that would not be possible to express with LTL. An example of such a property would be $((a \cdot a)[+] \cdot b)!$, since LTL has no means of ensuring that a holds in an even number of states. The tail conjunction operator $\diamond\!\!\rightarrow$ is the dual of the standard PSL tail implication operator $\mapsto$. Given a word, tail implication $r \mapsto \phi_1$ states that whenever a prefix of the word matches the SERE r, then the corresponding postfix with one state of overlap must satisfy ϕ_1; tail conjunction $r \diamond\!\!\rightarrow \phi_1$ holds if there exists some prefix of the word that matches r and the corresponding overlapping postfix satisfies ϕ_1. The tail conjunction is not part of the official PSL syntax but is also introduced in [8] to allow transforming formulae into negation normal form.

2.2 Semantics

As previously mentioned, the semantics of PSL are divided into three different variants. All of the three variants are identical when considering only infinite paths. The variants treat finite paths differently, however. Intuitively, all operators are considered to be strong operators in the strong semantics, i.e. they require that the path is of sufficient length to satisfy the requirements of the operator. Correspond-

ingly, in the weak semantics all operators are considered to be weak operators, i.e. if the path is too short to know if a continuation would satisfy the requirements of the operator, it is assumed to be satisfied. The neutral semantics considers strong operators to be strong and weak operators to be weak.

We define a state s to be the set of atomic propositions that hold in it, i.e. $s \subseteq AP$. The set of all possible states is denoted by S, i.e. $S = 2^{AP}$. A path is a finite or an infinite sequence of states. In the following definitions, $s \in S$ while $\pi, \pi_1, \pi_2, \ldots \in S^\star$ are finite paths. For each SERE r, we define three different languages: $\mathcal{L}(r)$, $\mathcal{L}_{\text{pref}}(r)$, and $\mathcal{L}_{\text{loop}}(r)$. The first is the base language of the SERE, similar to the one used in other flavours of regular expressions. The second is the prefix language, defined as the set of proper prefixes of paths in the base language. The third is the loop language, which consists of the infinite paths where some repetition operator is repeated infinitely many times. These three languages for regular expressions are needed to define the three variants of PSL semantics. The base language $\mathcal{L}(r) \subseteq S^\star$ of a SERE r is defined inductively as follows:

- $\mathcal{L}([*0]) = \{\varepsilon\}$, where ε is the empty path.
- $\mathcal{L}(p) = \{s \in S \mid p \in s\}$ and $\mathcal{L}(\neg p) = \{s \in S \mid p \notin s\}$ for each $p \in AP$. Note that this does not restrict other atomic propositions from being true in s.
- $\mathcal{L}(r[+]) = \{\pi \mid \exists n \geq 1 : \pi = \pi_1\pi_2 \ldots \pi_n \text{ and } \forall i, 1 \leq i \leq n : \pi_i \in \mathcal{L}(r)\}$.
- $\mathcal{L}(r_1 \cdot r_2) = \{\pi_1\pi_2 \mid \pi_1 \in \mathcal{L}(r_1) \text{ and } \pi_2 \in \mathcal{L}(r_2)\}$.
- $\mathcal{L}(r_1 \circ r_2) = \{\pi_1 s \pi_2 \mid \pi_1 s \in \mathcal{L}(r_1) \text{ and } s\pi_2 \in \mathcal{L}(r_2)\}$.
- $\mathcal{L}(r_1 \cup r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$.
- $\mathcal{L}(r_1 \cap r_2) = \mathcal{L}(r_1) \cap \mathcal{L}(r_2)$.

As an example, the path $\{a, b\}\{a, d\}\{b\}$ belongs to $\mathcal{L}(a[+] \cdot (b \cup c))$ but the path $\{a, b\}\{a, d\}\{a\}$ does not.

The prefix language $\mathcal{L}_{\text{pref}}(r) \subseteq S^\star$ for a SERE r is defined to consist of all finite, proper prefixes of paths in $\mathcal{L}(r)$:

$$\mathcal{L}_{\text{pref}}(r) = \{\pi \in S^\star \mid \exists \pi' \in S^+ : \pi\pi' \in \mathcal{L}(r)\}.$$

As an example, the path $\{a,b\}\{a,b\}$ is in $\mathcal{L}_{\text{pref}}(a[+] \cdot (b \cup c))$ but $\{a,b\}\{b\}$ is not.

The loop language $\mathcal{L}_{\text{loop}}(r) \subseteq S^\omega$ for a SERE r is defined inductively as follows:

- $\mathcal{L}_{\text{loop}}([*0]) = \emptyset$.
- $\mathcal{L}_{\text{loop}}(p) = \emptyset$.
- $\mathcal{L}_{\text{loop}}(r[+]) = \{\pi_1\pi_2 \mid \pi_1 \in \mathcal{L}(r[+]) \cup \{\varepsilon\}$ and $\pi_2 \in \mathcal{L}_{\text{loop}}(r)\} \cup \{\pi \mid \pi \in S^\omega, \pi = \pi_1\pi_2\pi_3 \ldots, \forall i \in \mathbb{Z}_+ : \pi_i \in \mathcal{L}(r)\}$. This case characterises the loop language. The first case captures those words where a repetition is done infinitely many times inside r, and the second case is the concatenation of infinitely many words from r, i.e. making the outermost repetition infinitely many times.
- $\mathcal{L}_{\text{loop}}(r_1 \cdot r_2) = \mathcal{L}_{\text{loop}}(r_1) \cup \{\pi_1\pi_2 \mid \pi_1 \in \mathcal{L}(r_1)$ and $\pi_2 \in \mathcal{L}_{\text{loop}}(r_2)\}$.
- $\mathcal{L}_{\text{loop}}(r_1 \circ r_2) = \mathcal{L}_{\text{loop}}(r_1) \cup \{\pi_1 s \pi_2 \mid \pi_1 s \in \mathcal{L}(r_1)$ and $s\pi_2 \in \mathcal{L}_{\text{loop}}(r_2)\}$.
- $\mathcal{L}_{\text{loop}}(r_1 \cup r_2) = \mathcal{L}_{\text{loop}}(r_1) \cup \mathcal{L}_{\text{loop}}(r_2)$.
- $\mathcal{L}_{\text{loop}}(r_1 \cap r_2) = \mathcal{L}_{\text{loop}}(r_1) \cap \mathcal{L}_{\text{loop}}(r_2)$.

For example, the language $\mathcal{L}_{\text{loop}}(a[+] \cdot (b \cup c))$ contains the infinite path $\{a\}^\omega = \{a\}\{a\}\ldots$, but also for example the infinite path $\{a,b\}^\omega = \{a,b\}\{a,b\}\ldots$, since $\{a,b\} \in \mathcal{L}(a)$. Another example is the language $\mathcal{L}_{\text{loop}}((a[+] \cdot (b \cup c))[+])$ which contains the infinite path $\{a\}^\omega$, but also the infinite paths $(\{a\}\{b\})^\omega$, $(\{a\}\{c\})\{a\}^\omega$, and $(\{a\}\{b\}\{a\}\{a\}\{c\})^\omega$, among others, since the infinite repetition can be done with either of the repeat operators.

Assume a non-empty path $\pi = s_1 s_2 \ldots \in S^+ \cup S^\omega$. For each $i \in \mathbb{Z}_+$ and each $v \in \{\text{strong}, \text{neutral}, \text{weak}\}$, we define the relation $\models_i^v$ by the following inductive rules:

- $\pi \models_i^v p$ iff $p \in s_i$, where $p \in AP$.
- $\pi \models_i^v \neg\phi$ iff $\begin{cases} v = \text{strong and } \pi \not\models_i^{\text{weak}} \phi, \text{ or} \\ v = \text{neutral and } \pi \not\models_i^{\text{neutral}} \phi, \text{ or} \\ v = \text{weak and } \pi \not\models_i^{\text{strong}} \phi. \end{cases}$
- $\pi \models_i^v \phi \wedge \psi$ iff $\pi \models_i^v \phi$ and $\pi \models_i^v \psi$.
- $\pi \models_i^v \phi \vee \psi$ iff $\pi \models_i^v \phi$ or $\pi \models_i^v \psi$.
- $\pi \models_i^v \mathbf{X}! \, \phi$ iff $\begin{cases} i < |\pi| \text{ and } \pi \models_{i+1}^v \phi, \text{ or} \\ v = \text{weak and } i = |\pi|. \end{cases}$
- $\pi \models_i^v \mathbf{X} \, \phi$ iff $\begin{cases} i = |\pi| \text{ or } \pi \models_{i+1}^v \phi, \text{ and} \\ v \neq \text{strong or } i < |\pi|. \end{cases}$
- $\pi \models_i^v \phi_1 \, \mathbf{U} \, \phi_2$ iff $\begin{cases} \exists j, i \leq j \leq |\pi| : (\pi \models_j^v \phi_2) \wedge (\forall k, i \leq k < j : \pi \models_k^v \phi_1), \text{ or} \\ v = \text{weak and } \pi \in S^+ \text{ and } \forall k, i \leq k \leq |\pi| : \pi \models_k^v \phi_1. \end{cases}$
- $\pi \models_i^v \phi_1 \, \mathbf{R} \, \phi_2$ iff $\begin{cases} \forall j, i \leq j \leq |\pi| : (\pi \models_j^v \phi_2) \vee (\exists k, i \leq k < j : \pi \models_k^v \phi_1), \text{ and} \\ v \neq \text{strong or } \pi \in S^\omega \text{ or } \exists k, i \leq k \leq |\pi| : \pi \models_k^v \phi_1. \end{cases}$
- $\pi \models_i^v r \mapsto \phi$ iff $\begin{cases} \forall j, i \leq j \leq |\pi| : s_i \dots s_j \in \mathcal{L}(r) \Rightarrow \pi \models_j^v \phi, \text{ and} \\ v = \text{strong} \Rightarrow s_i \dots s_{|\pi|} \notin \mathcal{L}_{\text{pref}}(r). \end{cases}$
- $\pi \models_i^v r \diamond\!\!\rightarrow \phi$ iff $\begin{cases} \exists j, i \leq j \leq |\pi| : s_i \dots s_j \in \mathcal{L}(r) \wedge \pi \models_j^v \phi, \text{ or} \\ v = \text{weak and } s_i \dots s_{|\pi|} \in \mathcal{L}_{\text{pref}}(r). \end{cases}$
- $\pi \models_i^v r! \text{ iff } \begin{cases} \exists j, i \leq j \leq |\pi| : s_i \dots s_j \in \mathcal{L}(r), \text{ or} \\ v = \text{weak and } \pi \in S^+ \text{ and } s_i \dots s_{|\pi|} \in \mathcal{L}_{\text{pref}}(r). \end{cases}$

- $\pi \models_i^v r$ iff $\begin{cases} \pi \models_i^v r!, \text{ or} \\ \pi \in \mathcal{L}_{\text{loop}}(r), \text{ or} \\ v \neq \text{strong and } \pi \in S^+ \text{ and } s_i \dots s_{|\pi|} \in \mathcal{L}_{\text{pref}}(r). \end{cases}$

We use $\pi \models^v \phi$ to denote $\pi \models_1^v \phi$ and say that "ϕ holds on π", or that "π satisfies ϕ", if $\pi \models^v \phi$.

Example 2.1. An example of a useful PSL formula is $(\texttt{Nil}[*] \cdot \texttt{Acquire} \cdot \texttt{Nil}[*] \cdot \texttt{Release})[+] \diamond\!\!\rightarrow \mathbf{X}!\ (\texttt{Nil}\ \mathbf{U}\ (\texttt{Destroy} \wedge\ \mathbf{XG}\ \texttt{Nil}))$. Here we use `Nil` as an abbreviation for $\neg\texttt{Acquire} \cap \neg\texttt{Release}$ and $r[*]$ as an abbreviation for $r[+] \cup [*0]$. It states that on the path `Acquire` and `Release` must alternate, beginning with `Acquire`, ending with `Release`, containing at least one of both, and after the last `Release`, `Destroy` must eventually follow. It is also assumed that in the system only one of `Acquire`, `Release`, and `Destroy` may be true simultaneously. (This could be verified with a different formula). The time between actions is specified to be irrelevant. For instance, the path

$$\{\}\{\texttt{Acquire}\}\{\}\{\texttt{Release}\}\{\}\{\}\{\texttt{Destroy}\}$$

satisfies the formula.

In the rest of the paper, all formulae are assumed to be written in negation normal form where negations only appear in front of atomic propositions. The following equalities can be used to rewrite formulae: $\neg(\phi \vee \psi) \equiv (\neg\phi \wedge \neg\psi)$, $\neg(\phi \wedge \psi) \equiv (\neg\phi \vee \neg\psi)$, $\neg(\mathbf{X}!\ \phi) \equiv (\mathbf{X}\ \neg\phi)$, $\neg(\mathbf{X}\ \phi) \equiv (\mathbf{X}!\ \neg\phi)$, $\neg(\phi\ \mathbf{U}\ \psi) \equiv (\neg\phi\ \mathbf{R}\ \neg\psi)$, $\neg(\phi\ \mathbf{R}\ \psi) \equiv (\neg\phi\ \mathbf{U}\ \neg\psi)$, $\neg(r \mapsto \psi) \equiv (r \diamond\!\!\rightarrow \neg\psi)$, and $\neg(r \diamond\!\!\rightarrow \psi) \equiv (r \mapsto \neg\psi)$. With these, every formula can be rewritten to an equivalent one in the negation normal form.

In the rest of the paper we will only apply the strong semantics for finite paths. Since $\mathbf{X}!\ \phi$ and $\mathbf{X}\ \phi$ are equivalent in the strong semantics, we only use $\mathbf{X}!\ \phi$ from

now on. For the same reason we use $r \diamond\!\!\rightarrow \mathbf{true}$ instead of r and $r!$. Strong semantics is used because it has the desired property that if a finite prefix of an infinite or a finite path satisfies a property with this semantics, then the whole path satisfies the property with any of the three semantics (weak, neutral, or strong) presented in [10, 11]. This is formally captured by the following:

Proposition 2.2. *[11] If $\pi \models^{\text{strong}} \phi$ holds for a finite path $\pi = s_1 ... s_k$ and a PSL formula ϕ, then $\pi' \models^{v} \phi$ holds for all finite or infinite paths π' having π as a prefix and for all semantics $v \in \{\text{strong}, \text{neutral}, \text{weak}\}$.*

This is convenient when searching for finite counter-examples to properties. If a path satisfies the negation of some property with the strong semantics, then every extension of it satisfies the negated property as well, and therefore the path is a counter-example for the non-negated property. Thus, finite counter-examples for a PSL formula ϕ can be searched for with the following steps:

- Negate ϕ to get $\neg\phi$.
- Transform $\neg\phi$ to the equivalent negation normal form formula ψ.
- Search for a finite path that satisfies ψ with the strong semantics.

The negation normal form is used because negating a formula changes the semantic variant it needs to be evaluated in. Pushing negations next to atomic propositions allows us to evaluate all sub-formulas with the strong semantics. This would not be possible if arbitrary negations were allowed. E.g. the formula $\neg(\mathtt{a} \cdot \mathtt{b} \mapsto \mathtt{c}\ \mathbf{U}\ \mathtt{d})$ holds in the strong semantics if the path begins with $\{a\}\{b\}$ and $\mathtt{c}\ \mathbf{U}\ \mathtt{d}$ does not hold in the second state with the weak semantics. After the transformation to the formula $\mathtt{a} \cdot \mathtt{b} \diamond\!\!\rightarrow \neg c\ \mathbf{R}\ \neg d$, only strong semantics needs to be considered.

2.3 Informative bad prefixes

As mentioned above, if a finite prefix π of an infinite or a finite path satisfies a property ϕ under the strong semantics (i.e. $\pi \models^{\text{strong}} \phi$), the whole path satisfies the property under any of the three semantics (weak, neutral, or strong). In the model checking context this means that if $\pi \models^{\text{strong}} \neg\phi$ holds, then the property ϕ cannot hold on the path π (or any extension of it) and thus π serves as a finite counterexample for ϕ. Following the terminology of [19], we formalize this by defining that a finite path $\pi \in S^\star$ is an informative bad prefix for a formula ϕ if $\pi \models^{\text{strong}} \neg\phi$. For the LTL subset (i.e. PSL formulae without tail implications and tail conjunctions and thus without SEREs), the semantics here is equivalent to the definition of informativity in [19] and thus the definition of informative bad prefixes is also equivalent to the one in [19].

As our model checking approach constructs an observer (defined in the next two sections) for the negation $\neg\phi$ of the formula ϕ under consideration and the observer uses the strong semantics to accept paths of the observed system, we can find all the finite paths in the system that violate ϕ and are informative bad prefixes for ϕ. However, observe that some safety formulae do not have informative bad prefixes. As an example of such "pathologically safe" formulae (taken from [19]), the LTL safety formula $\psi = (\mathbf{G}\ q) \vee (\mathbf{G}\ r) \vee (\mathbf{G}\ (q \vee \mathbf{FG}\ p) \wedge \mathbf{G}\ (r \vee \mathbf{FG}\ \neg p))$, where $\mathbf{G}\ \phi \equiv \mathbf{false}\ \mathbf{R}\ \phi$ and $\mathbf{F}\ \phi \equiv \mathbf{true}\ \mathbf{U}\ \phi$, does not have informative bad prefixes although no finite path with $\neg q$ and $\neg r$ holding in some states can be extended to a path satisfying ψ.

As a consequence, our model checking approach cannot detect (i) counter-examples to such "pathologically safe" formulae or (ii) infinite counter-examples to general (non-safety) formulae. However, it should be reminded that this is exactly what the strong semantics for PSL described above dictates, and our approach exactly matches the strong semantics for finite paths.

On the other hand, observe that also general (non-safety) properties can have informative bad prefixes and these are detected by our model checking approach. As an example, $\tau = (\mathbf{F}\ \neg p) \wedge (\mathbf{G}\ \neg r)$ is a non-safety property as the infinite path $\{p\}\{p\}...$ satisfying $\neg\tau$ does not have a finite bad prefix (i.e. a prefix that cannot be extended to a (finite or infinite) path satisfying τ). But τ also has informative bad prefixes, such as $\{p\}\{p, r\}$, and these are detected by our approach. In fact, any linear time property can be decomposed into a conjunction of two parts, where the other part is pure a non-safety property, and the other is a pure safety property [24]. A more detailed discussion on the classification temporal properties can be found in [7].

3 Observers

This work uses a custom formalism, similar to temporal testers in e.g. [18, 17], for defining observers to PSL formulae. The custom formalism makes it easy to combine observers for sub-formulae into an observer for the whole formula. They are also relatively simple to convert directly to NuSMV modules.

3.1 Transducers

Transducers in this paper are a symbolic variant of finite state automata. Unlike traditional automata, their state and input are represented by a set of boolean variables, and they can signal acceptance at multiple points in the execution. In this work the latter property is used to build transducers that accept at those points in the execution where a PSL formula holds. Formally, a transducer T is a tuple $(Q, Q^{\text{in}}, q^{\text{out}}, I, F, \delta)$, where:

- Q is a finite set of state variables.
- Q^{in} is a finite set of input variables, disjoint from Q.

- $q^{\text{out}} \in Q$ is the output variable.
- Every subset of $Q \cup Q^{\text{in}}$ is a state of the transducer. A variable v is said to be true in a state $s \subseteq Q \cup Q^{\text{in}}$ if and only if $v \in s$.
- $I \subseteq 2^{Q \cup Q^{\text{in}}}$ is the set of initial states of the transducer.
- $F \subseteq 2^{Q \cup Q^{\text{in}}}$ is the set of final states of the transducer, not to be confused with accepting states in traditional finite state automata.
- $\delta \subseteq 2^{Q \cup Q^{\text{in}}} \times 2^{Q \cup Q^{\text{in}}}$ is the transition relation.

Let $T = (Q, Q^{\text{in}}, q^{\text{out}}, I, F, \delta)$ be a transducer. An execution of T is a finite non-empty sequence of transducer states, $s_1, s_2, \ldots, s_n$, such that $s_1 \in I$, $s_n \in F$, and $\forall i, 1 \leq i < n : (s_i, s_{i+1}) \in \delta$. The set of initial states I restricts what may be the first state of an execution, the set of final states F restricts what may be the last state of an execution, and the transition relation restricts what states may be adjacent in the sequence. An input of T is a sequence $\pi = p_1, \ldots, p_n \in (2^D)^+$, where each p_i is a set of input variables from some input domain D such that $Q^{\text{in}} \subseteq D$. An execution $s_1, s_2, \ldots, s_n$ of T is defined to be an execution for π if the execution and the input path agree on input variables, i.e. for every input variable $v \in Q^{\text{in}}$ and for every $i, 1 \leq i \leq n$ the following holds: $v \in s_i \Leftrightarrow v \in p_i$. The transducer accepts at a state s_i of an execution if $q^{\text{out}} \in s_i$.

Example 3.1. The following example of a transducer, presented in Figure 1, accepts at states which precede a state where its sole input variable i is true. This is equivalent to accepting at states where the formula $\mathbf{X}!\ i$ is true. The output variable is q, which is also the only state variable. The state-space is every subset of $\{q, i\}$, i.e.:

- $\emptyset$, corresponding to the situation where the input i is not true and won't be in the next state either,

- $\{q\}$, corresponding to the situation where the input i is not true but it will be in the next state,
- $\{i\}$, corresponding to the situation where the input i is true but it won't be in the next state, and
- $\{q, i\}$, corresponding to the situation where the input i is true and will be in the next state as well.

All of these states are initial, since any of the situations can occur in the initial state. The final states are the ones where the output is not true, namely $\emptyset$ and $\{i\}$. This is because i can not be true in the next state when the execution is in its last state. An example of an execution of the transducer is: $\{q\}, \{i\}, \{q\}, \{q, i\}, \{i\}$, which corresponds to the input sequence $\emptyset, \{i\}, \emptyset, \{i\}, \{i\}$.

Formally, the transducer is defined as the tuple $T_{\mathbf{X}!} = (\{q\}, \{i\}, q, I, F, \delta)$ such that (i) all states are initial: $I = 2^{\{q,i\}}$, (ii) a state is final iff q is false in it: $F = \left\{s \in 2^{\{q,i\}} \mid q \notin s\right\}$, and (iii) the transition relation δ is defined so that the variable q is true in a state iff the variable i is true in the next state: $(s, s') \in \delta$ iff $(q \in s) \Leftrightarrow (i \in s')$.

3.2 Transducer composition

Transducer composition is a way to combine two transducers so that one transducer can use information from the other. This is done by plugging the output variable of one transducer to one input variable of the other in a circuit-like manner.

Here we use $S[a/b]$ to denote that an element a from the set S is renamed to b. Similarly, $(S, S')[a/b]$ is used to denote $(S[a/b], S'[a/b])$ for a pair of sets.

Now let $T_1 = (Q_1, Q_1^{\text{in}}, q_1^{\text{out}}, I_1, F_1, \delta_1)$ and $T_2 = (Q_2, Q_2^{\text{in}}, q_2^{\text{out}}, I_2, F_2, \delta_2)$ be two transducers such that $Q_1 \cap Q_2 = \emptyset$. The composition of T_1 and T_2, with respect to some input variable $q^{\text{in}} \in Q_2^{\text{in}}$, is denoted as $T_1 \rhd_{q^{\text{in}}} T_2$, and defined as ($Q_\rhd$, $Q_\rhd^{\text{in}}$,

$q^{\text{out}}_{\triangleright}$, $I_{\triangleright}$, $F_{\triangleright}$, $\delta_{\triangleright}$), where:

- $Q_{\triangleright} = Q_1 \cup Q_2$,
- $Q^{\text{in}}_{\triangleright} = Q^{\text{in}}_1 \cup (Q^{\text{in}}_2 \setminus \{q^{\text{in}}\})$ and the plugged input variable cannot exist in Q^{in}_1, i.e. $q^{\text{in}} \notin Q^{\text{in}}_1$,
- $q^{\text{out}}_{\triangleright} = q^{\text{out}}_2$,
- $I_{\triangleright} = \{s_1 \cup s_2[q^{\text{in}}/q^{\text{out}}_1] \mid$

  $s_1 \in I_1, s_2 \in I_2,$ and $q^{\text{out}}_1 \in s_1 \Leftrightarrow q^{\text{in}} \in s_2,$ and

  $\bigwedge_{v \in Q^{\text{in}}_1 \cap Q^{\text{in}}_2} v \in s_1 \Leftrightarrow v \in s_2\}$,
- $F_{\triangleright} = \{s_1 \cup s_2[q^{\text{in}}/q^{\text{out}}_1] \mid$

  $s_1 \in F_1, s_2 \in F_2,$ and $q^{\text{out}}_1 \in s_1 \Leftrightarrow q^{\text{in}} \in s_2,$

  and $\bigwedge_{v \in Q^{\text{in}}_1 \cap Q^{\text{in}}_2} v \in s_1 \Leftrightarrow v \in s_2\}$, and
- $\delta_{\triangleright} = \{(s_1 \cup s_2, s'_1 \cup s'_2)[q^{\text{in}}/q^{\text{out}}_1] \mid$

  $(s_1, s'_1) \in \delta_1, (s_2, s'_2) \in \delta_2,$ and

  $(q^{\text{out}}_1 \in s_1 \Leftrightarrow q^{\text{in}} \in s_2) \wedge (q^{\text{out}}_1 \in s'_1 \Leftrightarrow q^{\text{in}} \in s'_2),$ and

  $\bigwedge_{v \in Q^{\text{in}}_1 \cap Q^{\text{in}}_2} [(v \in s_1 \Leftrightarrow v \in s_2) \wedge (v \in s'_1 \Leftrightarrow v \in s'_2)]\}$.

Note that the condition $q^{\text{in}} \notin Q^{\text{in}}_1$ can always be satisfied by renaming if necessary.

The intuitive description of the composition is that the initial states, final states and the transitions from each transducer are combined, but only when they agree on the value of the variable to be plugged and the input variables that they share. States and transitions where the transducers disagree on these variables are dropped.

Example 3.2. The following example of transducer composition combines the previous example with itself to create a transducer that accepts when the formula **X!X!** i is true. First, a copy of the transducer is made with the state variable renamed to p and the input variable renamed to j to avoid conflicts. This means

that the semantics of the variable q is to be true when $\mathbf{X!}\ i$ holds and the semantics of the variable p is to be true when $\mathbf{X!}\ j$ holds. Let $T_{\mathbf{X!}}$ be the transducer in the previous example, and $T'_{\mathbf{X!}}$ be the copy. The transducer $T_{\mathbf{X!X!}}$ is then the composition $T_{\mathbf{X!}} \rhd_j T'_{\mathbf{X!}}$, which is presented in Figure 2. Each transition pair that agrees on the plugged variable is combined, e.g. $(\{i\}, \{q\})$ and $(\{p\}, \{p, j\})$ together yield the transition $(\{p, i\}, \{q, p\})$. Now that q is plugged to j, p holds when $\mathbf{X!}\ q$ holds, meaning that it holds when $\mathbf{X!X!}\ i$ holds. The output variable is p, every state is initial, and the final states are $\emptyset$ and $\{i\}$.

3.3 Transducers for formulae

In this section we define how transducers for formulae are built inductively, starting from atomic propositions. We first formally define what "a transducer for a formula" means:

Definition 3.3. A transducer T is a transducer for a formula ϕ if for every input $\pi = p_1, \ldots, p_n \in (2^D)^+$, where $AP \subseteq D$, the following hold:

- there exists an execution $\gamma = s_1, \ldots, s_n$ of T for π such that T accepts at a state s_j iff $\pi \models_j^{\text{strong}} \phi$, and
- there are no executions $\gamma = s_1, \ldots, s_n$ of T for π such that T accepts at a state s_j and $\pi \not\models_j^{\text{strong}} \phi$.

The first condition in the definition states that there exists at least one execution of the transducer that correctly evaluates the formula at every state of the given input. The second condition states that the other executions the transducer may have must be under-approximations of the correct evaluation. That is, the transducer cannot signal that the formula evaluates to true at a state even though it does not; however, it may signal that a formula evaluates to false even though it evaluates to true. This is similar to the behaviour of a non-deterministic finite

automaton. If a single execution from many possible ones is an accepting one, then acceptance is assumed.

Atomic propositions are represented by input variables, meaning that $AP \subseteq D$, and transducers for larger formulae are built with composition from transducers for their sub-formulae as explained below.

3.3.1 Logical operators

The transducer for the $\vee$-operator has two input variables for the operands and a single state variable that is also the output variable. The initial and final state constraints, as well as the transition are defined so that the state variable is true exactly when at least one of the input variables is true. Formally, the transducer is $T_\vee = (Q, Q^{\text{in}}, q^{\text{out}}, I, F, \delta)$, where:

- $Q = \{q\}$,
- $Q^{\text{in}} = \{q^{\text{left}}, q^{\text{right}}\}$,
- $q^{\text{out}} = q$,
- $I = F = \{\emptyset, \{q^{\text{left}}, q\}, \{q^{\text{right}}, q\}, \{q^{\text{left}}, q^{\text{right}}, q\}\}$, and
- $\delta = \{(s, s') \mid q \in s \Leftrightarrow (q^{\text{left}} \in s \vee q^{\text{right}} \in s)$ and
  $q \in s' \Leftrightarrow (q^{\text{left}} \in s' \vee q^{\text{right}} \in s')\}$.

The transducer for the entire formula $\phi_1 \vee \phi_2$ is obtained as the composition $T_{\phi_1 \vee \phi_2} = T_1 \rhd_{q^{\text{left}}} (T_2 \rhd_{q^{\text{right}}} T_\vee)$. The transducers for the $\wedge$ and $\neg$-operators are defined in a similar way.

Lemma 3.4. *If T_1 and T_2 are transducers for ϕ_1 and ϕ_2, respectively, then $T_{\phi_1 \vee \phi_2} = T_1 \rhd_{q^{\text{left}}} (T_2 \rhd_{q^{\text{right}}} T_\vee)$ is a transducer for the formula $\phi_1 \vee \phi_3$.*

Proof. Omitted. □

3.3.2 The next operator

The transducer $T_{\mathbf{X}!}$ for the next operator is presented in Example 3.1. The transducer for the entire formula $\mathbf{X}!\ \phi$ is obtained with the composition $T_\phi \rhd_i T_{\mathbf{X}!}$, where T_ϕ is a transducer for ϕ.

Lemma 3.5. *If T_ϕ is a transducer for ϕ, then $T_\phi \rhd_i T_{\mathbf{X}!}$ is a transducer for the formula $\mathbf{X}!\ \phi$.*

Proof. Omitted. □

3.3.3 The until operator

First, recall the strong semantics of the until operator for a finite path $\pi = s_1...s_n$:

$\pi \models_i^{\text{strong}} (\phi_1\ \mathbf{U}\ \phi_2)$ iff $\exists j, i \leq j \leq |\pi| : (\pi \models_j^{\text{strong}} \phi_2) \wedge (\forall k, i \leq k < j : \pi \models_k^{\text{strong}} \phi_1)$.

The intuition behind the transducer for the until-operator is that the until-formula $\phi_1\ \mathbf{U}\ \phi_2$ holds if and only if ϕ_2 holds, or ϕ_1 holds and the whole formula holds in the next state i.e. $\phi_1\ \mathbf{U}\ \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \mathbf{X}!\ (\phi_1\ \mathbf{U}\ \phi_2))$. We make use of this by having a variable $q^{\mathbf{U}}$ that represents the truth value of the formula. The transition relation restricts the variable so that it is true when ϕ_2 is true or when ϕ_1 is true and the variable itself is true in the next state. The transducer for the until operator is $T_{\mathbf{U}} = (Q, Q^{\text{in}}, q^{\text{out}}, I, F, \delta)$, where:

- $Q = \{q^{\mathbf{U}}\}$,
- $Q^{\text{in}} = \{q^{\text{left}}, q^{\text{right}}\}$,
- $q^{\text{out}} = q^{\mathbf{U}}$,
- $I = 2^{Q \cup Q^{\text{in}}}$,
- $F = \left\{s \mid q^{\mathbf{U}} \in s \Leftrightarrow q^{\text{right}} \in s\right\}$, and
- $\delta = \{(s, s') \mid q^{\mathbf{U}} \in s \Leftrightarrow (q^{\text{right}} \in s \vee (q^{\text{left}} \in s \wedge q^{\mathbf{U}} \in s'))\}$.

The transducer for the entire formula ϕ_1 **U** ϕ_2 is obtained as the composition $T_{\phi_1 \mathbf{U} \phi_2} = T_1 \rhd_{q^{\text{left}}} (T_2 \rhd_{q^{\text{right}}} T_{\mathbf{U}})$, where T_1 is a transducer for ϕ_1 and T_2 is a transducer for ϕ_2.

Lemma 3.6. *If T_1 and T_2 are transducers for ϕ_1 and ϕ_2, respectively, then $T_{\phi_1 \mathbf{U} \phi_2} = T_1 \rhd_{q^{\text{left}}} (T_2 \rhd_{q^{\text{right}}} T_{\mathbf{U}})$ is a transducer for the formula ϕ_1 **U** ϕ_2.*

Proof. See Appendix A. □

3.3.4 The releases operator

The strong semantics for the releases-operator in the case of finite paths is

$$\pi \models_i^{\text{strong}} (\phi_1 \mathbf{R} \phi_2) \text{ iff } \exists j, i \leq j \leq |\pi| : (\pi \models_j^{\text{strong}} \phi_1) \wedge (\forall k, i \leq k \leq j : \pi \models_k^{\text{strong}} \phi_2).$$

The intuition behind the transducer for the releases-operator is that the formula ϕ_1 **R** ϕ_2 holds if and only if both ϕ_1 and ϕ_2 hold, or ϕ_2 holds and the entire formula holds in the next state. As with the until-operator, we use a variable $q^{\mathbf{R}}$ to represent the truth value of the entire formula, and the transition relation restricts it so that it holds when both ϕ_1 and ϕ_2 hold or when ϕ_2 holds and the variable holds in the next state. The transducer for the releases operator is $T_{\mathbf{R}} = (Q, Q^{\text{in}}, q^{\text{out}}, I, F, \delta)$, where:

- $Q = \{q^{\mathbf{R}}\}$,
- $Q^{\text{in}} = \{q^{\text{left}}, q^{\text{right}}\}$,
- $q^{\text{out}} = q^{\mathbf{R}}$,
- $I = 2^{Q \cup Q^{\text{in}}}$,
- $F = \left\{s \mid q^{\mathbf{R}} \in s \Leftrightarrow \left(q^{\text{left}} \in s \wedge q^{\text{right}} \in s\right)\right\}$, and
- $\delta = \{(s, s') \mid q^{\mathbf{R}} \in s \Leftrightarrow q^{\text{right}} \in s \wedge (q^{\text{left}} \in s \vee q^{\mathbf{R}} \in s'))\}$.

For the entire formula $\phi_1 \mathbf{R} \phi_2$, we define the transducer $T_{\phi_1 \mathbf{R} \phi_2} = T_1 \rhd_{q^{\text{left}}} (T_2 \rhd_{q^{\text{right}}} T_{\mathbf{R}})$, where T_1 is a transducer for ϕ_1 and T_2 is a transducer for ϕ_2.

Lemma 3.7. *If T_1 and T_2 are transducers for ϕ_1 and ϕ_2, respectively, then $T_{\phi_1 \mathbf{R} \phi_2} = T_1 \rhd_{q^{\text{left}}} (T_2 \rhd_{q^{\text{right}}} T_{\mathbf{R}})$ is a transducer for the formula $\phi_1 \mathbf{R} \phi_2$.*

Proof. See Appendix A. □

It can also be noted that in the strong semantics $\phi_1 \mathbf{R} \phi_2$ is equivalent to $\phi_1 \mathbf{U} (\phi_1 \wedge \phi_2)$, so this transducer could be formed with the transducers for the until and conjunction operators. Our implementation considers the releases operator separately, however, so we present it here.

3.3.5 Tail implication

In describing the tail implication $r \mapsto \phi$ and tail conjunction $r \diamond\!\!\rightarrow \phi$, we use $AP(r)$ to denote the set of atomic propositions appearing in the SERE r. For each SERE r we also define a function $\ell_r : 2^{Q^{\text{in}} \cup Q} \to 2^{AP(r)}$ that maps the states of a transducer to the atomic propositions in r that hold in the state: $\ell_r(s) = s \cap AP(r)$. Additionally, in both cases we assume that the base language $\mathcal{L}(r)$ is not empty; if $\mathcal{L}(r) = \emptyset$, $r \mapsto \phi$ can be rewritten to **true** and $r \diamond\!\!\rightarrow \phi$ can be rewritten to **false**.

The intuition behind the transducer for the tail implication operator is that an automaton is created for the SERE r, and multiple copies of the automaton are simulated, which yields the matches for the SERE. When a simulated copy accepts, ϕ should hold.

Let $A_r = (Q_r, \Sigma, \delta_r, q_0, F_r)$ be a finite, non-deterministic automaton that satisfies the following requirements: (i) at least one state in F_r is reachable from every state, i.e. there are no rejecting states, (ii) there are no ε-transitions, (iii) $\mathcal{L}(A_r) = \mathcal{L}(r)$, and (iv) $\Sigma = 2^{AP(r)}$. From basic automata theory we know that such an automaton can always be constructed. Using A_r, we can construct a trans-

ducer

$$T_{r\mapsto} = (Q, Q^{\text{in}}, q^{\text{out}}, I, F, \delta)$$

for the tail implication operator, where:

- $Q = Q_r$,
- $Q^{\text{in}} = AP(r) \cup \{q^\phi\}$, where q^ϕ is the input variable that signals when ϕ holds,
- $q^{\text{out}} = q_0$, the initial state of A_r,
- $I = 2^{Q \cup Q^{\text{in}}}$,
- $F = \{s \mid s \cap Q_r = \emptyset\}$, and
- $\delta = \{(s, s') \mid \bigwedge_{(v,\sigma,v') \in \delta_r} ((v \in s \wedge \ell_r(s) = \sigma) \Rightarrow v' \in s')$ and $F_r \cap s' \neq \emptyset \Rightarrow q^\phi \in s\}$.

The transducer for the entire formula $r \mapsto \phi$ is obtained with the composition $T_{r\mapsto\phi} = T_\phi \rhd_{q^\phi} T_{r\mapsto}$, where T_ϕ is a transducer for ϕ. Intuitively, the first part of the transition relation handles simulating copies of the automaton for the SERE, and the second part states that when the simulated automaton accepts, ϕ must hold. The final state constraint states that no copies of the simulated automata can be left running, which takes care of the requirement that the suffix of the path cannot belong to the prefix language of r.

Since input to finite state automata is given on a transition from one state to another, combining them with transducers in the described way introduces a slight inconvenience. The atomic propositions that are used for the input of the automaton are a part of the states in the transducer, as opposed to the transitions. Therefore, in the transducer, the state of the automaton changes one step after the input. Combined with the fact that the final state of the transducer cannot contain state variables from the automaton, in some cases the transducer needs to be run for two additional steps even though a counter-example has already been detected.

Lemma 3.8. *If T_ϕ is a transducer for ϕ, then $T_{r\mapsto\phi} = T_\phi \rhd_{q^\phi} T_{r\mapsto}$ is a transducer for the formula $r \mapsto \phi$.*

Proof. See Appendix A. □

Example 3.9. The following example illustrates a transducer for the formula $\{a \cdot b\} \mapsto c$. The automaton for the SERE part is shown in Figure 3. The automaton dictates the transition relation of the transducer, which is:

$$\begin{aligned}\{(s, s') \quad | \quad & ((q_0 \in s \wedge \ell_r(s) = \{a\}) \Rightarrow q_1 \in s') \wedge \\ & ((q_0 \in s \wedge \ell_r(s) = \{a, b\}) \Rightarrow q_1 \in s') \wedge \\ & ((q_1 \in s \wedge \ell_r(s) = \{b\}) \Rightarrow q_2 \in s') \wedge \\ & ((q_1 \in s \wedge \ell_r(s) = \{a, b\}) \Rightarrow q_2 \in s') \wedge \\ & (q_2 \in s' \Rightarrow c \in s)\}\end{aligned}$$

For the path $\{a\}\{b, c\}\emptyset\emptyset$, on which the formula holds, there exists an execution of the transducer that accepts at the first state, namely: $\{a, q_0\}\{b, c, q_1\}\{q_2\}\emptyset$. For the path $\{a\}\emptyset\emptyset$, for which the formula holds as well since there is no match for the SERE, there exists the execution of the transducer $\{a, q_0\}\{q_1\}\emptyset$. In both cases the execution must continue until a valid end state is reached, i.e. one that does not have any state variables from the automaton.

3.3.6 Tail conjunction

The intuition behind the transducer for the tail conjunction $r \diamond\!\!\rightarrow \phi$ is a non-deterministically simulated automaton for the SERE r, combined with enforcing that ϕ must hold when the simulated automaton accepts. The simulation is different from the tail implication, because only one match needs to be captured, instead of all possible matches.

Let $A_r = (Q_r, \Sigma, \delta_r, q_0, F_r)$ be a finite, non-deterministic automaton that satisfies the following requirements: (i) at least one state in F_r is reachable from

every state, i.e. there are no rejecting states, (ii) there are no ε-transitions, (iii) $\mathcal{L}(A_r) = \mathcal{L}(r)$, and (iv) $\Sigma = 2^{AP(r)}$. With the help of A_r, we can construct a transducer

$$T_{r\diamond\!\!\rightarrow} = (Q, Q^{\text{in}}, q^{\text{out}}, I, F, \delta)$$

for the tail conjunction operator, where:

- $Q = Q_r$,
- $Q^{\text{in}} = AP(r) \cup \{q^\phi\}$, where q^ϕ is the input variable to which the output of T_ϕ is connected,
- $q^{\text{out}} = q_0$, the initial state of A_r,
- $I = 2^{Q \cup Q^{\text{in}}}$,
- $F = \{s \mid s \cap Q_r = \emptyset\}$, and
- $\delta = \{(s, s') \mid \bigwedge\limits_{v \in Q_r} [v \in s \Rightarrow \bigvee\limits_{(v,\sigma,v') \in \delta_r} \ell_r(s) = \sigma \wedge (v' \in s' \vee (v' \in F_r \wedge q^\phi \in s))]\}$.

The transducer for the entire formula $r \diamond\!\!\rightarrow \phi$ is obtained as the composition $T_{r\diamond\!\!\rightarrow\phi} = T_\phi \rhd_{q^\phi} T_{r\diamond\!\!\rightarrow}$, where T_ϕ is a transducer for ϕ. Intuitively, the transition relation takes care of the simulation of the automaton, except for the expression in the innermost parenthesis, which allow for the termination of the simulation if a match is found and ϕ holds. The state variables can be seen as a promise to find a match starting from that state of the automaton, and the final state constraint enforces that no such promise is left unfulfilled when the execution stops.

Lemma 3.10. *If T_ϕ is a transducer for ϕ, then $T_{r\diamond\!\!\rightarrow\phi} = T_\phi \rhd_{q^\phi} T_{r\diamond\!\!\rightarrow}$ is a transducer for the formula $r \diamond\!\!\rightarrow \phi$.*

Proof. See Appendix A. □

Example 3.11. As an example of a transducer for a tail conjunction, the following is a transducer for the formula $\{a{\cdot}b\} \diamond\!\!\rightarrow c$. The automaton for the SERE is the same

as in the example for the tail implication, presented in Figure 3. The transition relation for the transducer is:

$$\begin{aligned}\big\{(s,s') \mid \quad & \big(q_0 \in s \Rightarrow (\ell_r(s) = \{a,b\} \wedge q_1 \in s') \vee \\ & \qquad (\ell_r(s) = \{a\} \wedge q_1 \in s')\big) \\ \wedge\ & \big(q_1 \in s \Rightarrow (\ell_r(s) = \{a,b\} \wedge (q_2 \in s' \vee c \in s)) \vee \\ & \qquad (\ell_r(s) = \{b\} \wedge (q_2 \in s' \vee c \in s))\big) \\ \wedge\ & \big(q_2 \in s \Rightarrow \mathbf{false}\big)\big\}\end{aligned}$$

For the path $\{a\}\{b,c\}\emptyset$, for which the formula holds, there exists the following accepting execution of the transducer: $\{a,q_0\}\{b,c,q_1\}\emptyset$. Again, as with the tail implication example, the execution must continue until a valid end state is reached, i.e. one that does not contain any state variables from the automaton.

4 Observer implementation

Model checking with the transducers can be done in the following way:

> To detect informative bad prefixes for any PSL formula ϕ, a transducer for $\neg\phi$ (in negation normal form) is constructed. It is then converted to a NuSMV module including an invariant specification, and run together synchronously with the model to be verified. If a run exists where the output variable for the observer is true in the first state and the invariant is violated, that run violates the property ϕ.

Converting transducers to NuSMV modules that can be used as observers is straightforward. A bad prefix is found if the transducer accepts at the first state of an execution. A NuSMV module is used to check whether this is possible, but the execution of the module is not directly comparable to the execution of the transducer. More specifically, the NuSMV module will have executions that are not valid executions of the transducer, and bad prefixes are detected by checking if an

execution of the module is a valid execution of the transducer.

The executions of the module are such that they obey the transition relation and initial state constraints of the transducer, and the top level output variable is forced to be true in the initial state. Executions where the top level output variable is not true in the first state are not interesting since that would mean that $\neg\phi$ need not hold there, so they would not be counter-examples. The module then checks if a valid final state can be reached using an invariant specification, which means that the transducer has a corresponding run ending in a valid final state.

The state variables of a transducer are represented by local variables of a NuSMV module, input variables are represented by parameters to the module, the initial states are set with an `INIT`-block in the module, and the transition relation is enforced with a `TRANS`-block. The output variable is set to true in the initial state with an `INIT`-block, and then the reachability of a valid final state is checked for by adding a new special purpose unconstrained variable `fs` that represents a valid final state. The final state constraints are represented by an invariant constraint that allows `fs` to become true only in a valid final state. The reachability check can then be done by adding the invariant specification `INVARSPEC !fs` to the module, and running it synchronously together with the model to be checked. Converting SEREs to finite state automata for the construction is done in the usual way, e.g. like in [27].

Example 4.1. The transducer for the formula p **U** q is translated into the following NuSMV module:

```
MODULE observer(p,q)
VAR
  u : boolean;
INVAR
  fs -> (u <-> q)
TRANS
  u <-> (q | p & next(u))
INIT
  u
INVARSPEC !fs
```

The module would be generated to check for the formula $\neg p \mathbf{R} \neg q$, which is the negation of $p \mathbf{U} q$. Note that the `INIT`- and `INVARSPEC`-blocks are present because this is the module for the top-level transducer.

The actual implementation that was done for this work is a proof-of-concept, whose main purpose is to verify the feasibility of such an implementation and to allow experimentation with the algorithm. It is available online at `http://www.tcs.hut.fi/~tlauniai/psl-observer/`.

5 Experiments

To experiment with our algorithm, we ran two sets of benchmarks. All tests were done on a Debian Linux machine with an Intel Core Duo 1.86 GHz processor and 2 GiB RAM. Both benchmark sets were against the state-of-the-art PSL implementation that is presented in [8]. That implementation is also built on top of NuSMV. For the first comparison, the same set of benchmarks is used as in the paper [8], which includes both general (non-safety) and safety properties. Their

BDD-based algorithm, with syntactic optimisations turned on, is compared with the BDD-based invariant checking of NuSMV 2.4.3, combined with our transducer encoding based observer. Their approach combined with the simple bounded model checking (SBMC) approach [14] implemented in NuSMV is compared against the transducer encoding of this paper combined with SBMC LTL checking of NuSMV 2.4.3 where the invariant is expressed with a globally-operator. In order to provide a fair comparison, both SBMC algorithms were run with the completeness flag set (i.e. they only stop when they can either prove or disprove the property, see [14]). From the benchmark set, we filtered out instances for which our tool cannot guarantee a correct answer: namely the properties for which an informative bad prefix could not be detected. This left us with about 38% of the original instances, and 13% of the included properties were safety properties. There were no safety properties in the excluded part, i.e. all safety properties had a counter-example. These results are presented in Figure 4.

In the second set of benchmarks we used the real life models from [2], excluding the ones that are not compatible with the current implementation of NuSMV 2.4.3 due to modelling language grammar changes in new NuSMV versions. For these models we randomly generated PSL properties for which every counterexample is informative. This was done by syntactically limiting the properties to be safety properties. These models where then checked with both implementations as in the previous benchmark set. The results of this benchmark set are shown in Fig. 5.

The benchmarks show that our tool has a clear advantage over the state-of-the-art PSL model checker. While the real world tests with SBMC-checking are somewhat even, all the other benchmark sets, especially the BDD model checking based ones, show that our implementation is clearly faster in the majority of cases. It should be noted here that our approach does not benefit from any of the syntactic PSL simplifications described in [8] unlike the approach we compare against. Sum-

ming up, our approach is certainly viable for model checking PSL safety properties, as well as finding bugs with non-safety properties.

Some tests were also run with random generated models and properties, but with the exact setup used the runtime of those benchmarks seemed to depend only on the size of the model and the size of the formula, not the implementation they were checked with or the particular instance. That seems to imply that the checking of the property was trivial once the property and model were interpreted and encoded from the input file.

Unfortunately we do not know of a freely available implementation of the safety simple subset of PSL [1, 16, 15, 4], and therefore cannot run benchmarks against those. With regards to approaches based on the safety simple subset our main contribution is that we impose no syntactic restrictions on the model checked formulae unlike the safety simple subset that is quite restricted in its allowed syntax. In comparison to the explicit automata approach of [20] we can state that our symbolic encoding is exponentially more compact and also handles a larger set of properties, not only the LTL subset.

As a summary, the experiments show that for finding bugs by detecting finite, informative counter-examples to general (non-safety) properties, as well as for model checking safety properties, our tool is competitive compared to the state-of-the-art.

6 Conclusions

We have detailed a fast PSL model checking algorithm for safety properties. The approach uses transducers implemented symbolically, inspired by temporal testers of [17]. The formal semantics of PSL capturing the informative bad prefixes was defined in Section 2 and is based on the latest revision of PSL semantics [10, 11]. The transducers formalism used to construct the observers is detailed in Section 3.

We would like to stress that every PSL property can be expressed with the subset presented here, but only informative bad prefixes of the properties can be detected by our approach. Thus the subset we use is strictly larger than for example the PSL safety simple subset [1], which is used by many runtime monitoring implementations. For example, the formula $(p \mathbf{R} q) \mathbf{R} r$ is a safety property that is not syntactically in the safety simple subset without rewriting the formula. Because of the inclusion of regular expressions in the safety simple subset, many safety properties can be rewritten to it, but this makes the specified properties much harder to understand, and we are not aware of any automated tool that rewrites formulae into the safety simple subset. The experimental results show the approach to be a quite competitive bug finding tool and safety property model checker when compared to a state-of-the-art implementation of PSL model checking. Especially in combination with symbolic model checking with BDDs it avoids the use of costly algorithms used to find accepting cycles with BDDs and instead relies on simple and more efficient invariant checking.

There are interesting topics for further work. The approach presented in this work is a complete model checking method for many PSL properties used in practical specification work. For example, the LTL subset of PSL contains many syntactic subsets that result in formulae where every counterexample has an informative bad prefix. Similar careful characterization of all of PSL properties should result in larger syntactic subsets of PSL properties where our approach can fully replace general PSL model checking algorithms.

On the algorithmic side, the paper [20] describes a tool that analyzes an LTL specification used in model checking, and detects exactly those formulae for which our approach is a complete model checking approach. Namely, given a specification, the tool looks for an infinite counterexample word that does not have a finite informative bad prefix. If no such infinite counterexample can be found, the transducer

we generate for it is called fine [19], and for these LTL formulae our model checking approach can fully replace a generalized PSL model checking approach. The same approach should be extended to all of PSL in future work. This can clearly be done using the same basic approach as presented in [20].

Acknowledgements

The financial support of Academy of Finland (projects 126860, 128050, and 139402) and Technology Industries of Finland Centennial Foundation is gratefully acknowledged. In addition we would like to thank the anonymous reviewer of this paper for very detailed and constructive comments.

References

[1] Shoham Ben-David, Dana Fisman, and Sitvanit Ruah. The safety simple subset. In *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005*, volume 3875 of *Lecture Notes in Computer Science*, pages 14–29. Springer, 2005.

[2] Armin Biere, Keijo Heljanko, Tommi Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5), 2006.

[3] Roderick Bloem, Harold N. Gabow, and Fabio Somenzi. An algorithm for strongly connected component analysis in n log n symbolic steps. *Formal Methods in System Design*, 28(1):37–56, 2006.

[4] Marc Boule and Zeljko Zilic. Automata-based assertion-checker synthesis of PSL properties. *ACM Transactions on Design Automation of Electronic Systems*, 13(1), 2008.

[5] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Compututation*, 98(2):142–170, 1992.

[6] Doron Bustan, Dana Fisman, and John Havlicek. Automata construction for PSL. Technical report, The Weizmann Institute of Science, 2005.

[7] Edward Chang, Zohar Manna, and Amir Pnueli. The safety-progress classification. Technical report, Dept. of Computer Science, Stanford University, 1992.

[8] Alessandro Cimatti, Marco Roveri, and Stefano Tonetta. Symbolic compilation of PSL. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1737–1750, 2008.

[9] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

[10] Cindy Eisner and Dana Fisman. Formal Syntax and Semantics of IEEE Std 1850 Property Specification Language. Retrieved on 13th October, 2008, from `http://www.eda.org/ieee-1850/ieee-1850-issues/hm/att-0690/Final_Annex_B_08.pdf`.

[11] Cindy Eisner and Dana Fisman. Structural contradictions. In *Hardware and Software: Verification and Testing, 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27-30, 2008. Proceedings*, volume 5394 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2008.

[12] Bernd Finkbeiner and Lars Kuhtz. Monitor circuits for LTL with bounded and unbounded future. In *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers*, volume 5779 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2009.

[13] Ronald H. Hardin, Robert P. Kurshan, Sandeep K. Shukla, and Moshe Y. Vardi. A new heuristic for bad cycle detection using BDDs. *Formal Methods in System Design*, 18(2):131–140, 2001.

[14] Keijo Heljanko, Tommi Junttila, and Timo Latvala. Incremental and complete bounded model checking for full PLTL. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 98–111. Springer, 2005.

[15] Naiyong Jin and Chengjie Shen. Dynamic verifying the properties of the simple subset of PSL. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE 2007, June 5-8, 2007, Shanghai, China*, pages 229–240. IEEE Computer Society, 2007.

[16] Naiyong Jin, Chengjie Shen, Jun Chen, and Taoyong Ni. Engineering of an assertion-based $\text{PSL}^{\text{Simple}}$-Verilog dynamic verifier by alternating automata. In *Proceedings of the 1st International Workshop on Harnessing Theories for Tool Support in Software (TTSS 2007)*, volume 207 of *Electronic Notes in Theoretical Computer Science*, pages 153–169. Elsevier, 2008.

[17] Yonit Kesten, Amir Pnueli, and Li-on Raviv. Algorithmic verification of linear temporal logic specifications. In *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.

[18] Yonit Kesten, Amir Pnueli, Li-On Raviv, and Elad Shahar. Model Checking with Strong Fairness. *Formal Methods in System Design*, 28(1):57–84, 2006.

[19] Orna Kupferman and Moshe Y. Vardi. Model Checking of Safety Properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[20] Timo Latvala. Efficient Model Checking of Safety Properties. In *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, volume 2648 of *Lecture Notes in Computer Science*, pages 74–88. Springer, 2003.

[21] Tuomas Launiainen, Keijo Heljanko, and Tommi Junttila. Efficient model checking of PSL safety properties. In *Proceedings of the 10th International Conference on Application of Concurrency to System Design (ACSD'2010)*, pages 95–104, Braga, Portugal, June 2010.

[22] Kenneth L. McMillan. Applications of Craig interpolation to model checking. In *Applications and Theory of Petri Nets 2005, 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005, Proceedings*, volume 3536 of *Lecture Notes in Computer Science*, pages 15–16. Springer, 2005.

[23] Amir Pnueli and Aleksandr Zaks. On the merits of temporal testers. *25 Years of Model Checking*, pages 172–195, 2008.

[24] Fred B. Schneider. Decomposing properties into safety and liveness. Technical report, Cornell University, 1987.

[25] Viktor Schuppan and Armin Biere. Efficient reduction of finite state model checking to reachability analysis. *International Journal on Software Tools for Technology Transfer*, 5(2–3):185–204, 2004.

[26] Viktor Schuppan and Armin Biere. Shortest counterexamples for symbolic model checking of LTL with past. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 493–509. Springer, 2005.

[27] Michael Sipser. *Introduction to the Theory of Computation.* Course Technology, December 1996.

A Proof of Lemmas 3.6 to 3.10

This appendix presents the proofs for the important lemmas in this paper, i.e. the ones that prove the correctness of the transducer constructions.

A.1 Correctness of the transducer for the until operator

The following lemma proves the correctness of the transducer for the until operator.

Lemma 3.6. *If T_1 and T_2 are transducers for ϕ_1 and ϕ_2, respectively, then $T_{\phi_1 \mathbf{U} \phi_2} = T_1 \rhd_{q^{\text{left}}} (T_2 \rhd_{q^{\text{right}}} T_{\mathbf{U}})$ is a transducer for the formula $\phi_1 \mathbf{U} \phi_2$.*

Proof. This can be proven by backward induction from the last state of the execution.

- Base case: In the last state the final state constraint is equivalent to the semantics: since there is no next state in the execution, $\pi \models_n^{\text{strong}} \phi_1 \mathbf{U} \phi_2$ if and only if $q^{\text{right}} \in s_n$.
- Induction assumption: The lemma holds for states s_{n-k} with some limit for k.
- Induction step: The transition relation states that $q^{\mathbf{U}} \in s_{n-(k+1)} \Leftrightarrow (q^{\text{right}} \in s_{n-(k+1)} \vee (q^{\text{left}} \in s_{n-(k+1)} \wedge q^{\mathbf{U}} \in s_{n-k}))$. Clearly $q^{\text{right}} \in s_{n-(k+1)}$ implies $\pi \models_{n-(k+1)}^{\text{strong}} \phi_1 \mathbf{U} \phi_2$. Moreover, since $q^{\mathbf{U}} \in s_{n-k}$ if and only if $\pi \models_{n-k}^{\text{strong}} \phi_1 \mathbf{U} \phi_2$, $q^{\text{left}} \in s_{n-(k+1)} \wedge q^{\mathbf{U}} \in s_{n-k}$ implies both $\exists j > i : q^{\text{right}} \in s_j$ and $\forall l : n-(k+1) \leq l < j : q^{\text{left}} \in s_l$. Therefore $q^{\text{left}} \in s_{n-(k+1)} \wedge q^{\mathbf{U}} \in s_{n-k}$ implies $\pi \models_{n-(k+1)}^{\text{strong}} \phi_1 \mathbf{U} \phi_2$.

On the other hand, $\pi \models^{\text{strong}}_{n-(k+1)} \phi_1 \mathbf{U} \phi_2$ implies that either $\pi \models^{\text{strong}}_{n-(k+1)} \phi_2$ or $\exists j > i : q^{\text{right}} \in s_j$ and $\forall l : n-(l+1) \leq l < j : q^{\text{left}} \in s_l$. The latter implies that $\pi \models^{\text{strong}}_{n-k} \phi_1 \mathbf{U} \phi_2$, and therefore $q^{\mathbf{U}} \in s_{n-k}$. As a consequence, $\pi \models^{\text{strong}}_{n-(k+1)} \phi_1 \mathbf{U} \phi_2$ implies $q^{\text{right}} \in s_{n-(k+1)} \vee (q^{\text{left}} \in s_{n-(k+1)} \wedge q^{\mathbf{U}} \in s_{n-k})$. Now we have established that $q^{\mathbf{U}} \in s_{n-(k+1)} \Leftrightarrow q^{\text{right}} \in s_{n-(k+1)} \vee (q^{\text{left}} \in s_{n-(k+1)} \wedge q^{\mathbf{U}} \in s_{n-k})$ is equivalent to $\pi \models^{\text{strong}}_{n-(k+1)} \phi_1 \mathbf{U} \phi_2$, so the lemma holds for $k+1$, and by induction for any k.

□

A.2 Correctness of the transducer for the releases operator

The following lemma proves the correctness of the transducer for the releases operator.

Lemma 3.7. *If T_1 and T_2 are transducers for ϕ_1 and ϕ_2, respectively, then $T_{\phi_1 \mathbf{R} \phi_2} = T_1 \rhd_{q^{\text{left}}} (T_2 \rhd_{q^{\text{right}}} T_{\mathbf{R}})$ is a transducer for the formula $\phi_1 \mathbf{R} \phi_2$.*

Proof. This can be proven by backward induction from the last state of the execution.

- Base case: In the last state the final state constraint is equivalent to the semantics: since there is no next state in the execution, $\pi \models^{\text{strong}}_{n} \phi_1 \mathbf{R} \phi_2$ if and only if $q^{\text{left}} \in s_n \wedge q^{\text{right}} \in s_n$.
- Induction assumption: The lemma holds for states s_{n-k} with some limit for k.
- Induction step: The transition relation states that $q^{\mathbf{R}} \in s_{n-(k+1)} \Leftrightarrow (q^{\text{right}} \in s_{n-(k+1)} \wedge (q^{\text{left}} \in s_{n-(k+1)} \vee q^{\mathbf{R}} \in s_{n-k}))$. Dividing the right side of the equivalence, we see that $q^{\text{right}} \in s_{n-(k+1)} \wedge q^{\text{left}} \in s_{n-(k+1)}$ implies $\pi \models^{\text{strong}}_{n-(k+1)} \phi_1 \mathbf{R} \phi_2$. Additionally, since $q^{\mathbf{R}} \in s_{n-k}$ if and only if $\pi \models^{\text{strong}}_{n-k} \phi_1 \mathbf{R} \phi_2$, the

other part, $q^{\text{right}} \in s_{n-(k+1)} \wedge q^{\mathbf{R}} \in s_{n-k}$ implies both $\exists l, n-k \leq l \leq n : q^{\text{left}} \in s_l$ and $\forall j, n-(k+1) \leq j \leq n : q^{\text{right}} \in s_j$. Therefore $q^{\text{right}} \in s_{n-(k+1)} \wedge q^{\mathbf{R}} \in s_{n-k}$ implies $\pi \models^{\text{strong}}_{n-(k+1)} \phi_1 \mathbf{R} \phi_2$.

On the other hand, $\pi \models^{\text{strong}}_{n-(k+1)} \phi_1 \mathbf{R} \phi_2$ implies that either $\pi \models^{\text{strong}}_{n-(k+1)} \phi_1$ and $\pi \models^{\text{strong}}_{n-(k+1)} \phi_2$, or that $\exists l, n-k \leq l \leq n : q^{\text{left}} \in s_l$ and $\forall j : n-(l+1) \leq j \leq l : q^{\text{right}} \in s_l$. The latter implies that $\pi \models^{\text{strong}}_{n-k} \phi_1 \mathbf{R} \phi_2$, and therefore $q^{\mathbf{R}} \in s_{n-k}$. As a consequence, $\pi \models^{\text{strong}}_{n-(k+1)} \phi_1 \mathbf{R} \phi_2$ implies $q^{\text{right}} \in s_{n-(k+1)} \wedge (q^{\text{left}} \in s_{n-(k+1)} \vee q^{\mathbf{R}} \in s_{n-k})$.

Now we have established that $q^{\mathbf{R}} \in s_{n-(k+1)} \Leftrightarrow q^{\text{right}} \in s_{n-(k+1)} \wedge (q^{\text{left}} \in s_{n-(k+1)} \vee q^{\mathbf{U}} \in s_{n-k})$ is equivalent to $\pi \models^{\text{strong}}_{n-(k+1)} \phi_1 \mathbf{R} \phi_2$, so the lemma holds for $k+1$, and by induction for any k.

□

A.3 Correctness of the transducer for the tail implication operator

The following lemmas prove the correctness of the transducer for the tail implication operator. Lemma A.1 formalises the automation simulation, lemma A.2 states that the transducer cannot accept unless the condition that ϕ holds is satisfied, and lemma A.3 does the same with the condition that no matches are possible in any continuation of the execution. Lemma A.4 summarises these by stating that there are no bad executions for the transducer.

Lemma A.1. *If A_r has a sequence of transitions for an input $\sigma_i, \sigma_{i+1}, \ldots, \sigma_{j-1}$ that takes A_r from some state v_i to some state v_j, and $v_i \in s_i$, then for every execution $s_i, \ldots, s_j$, where $\forall l, i \leq l < j : \ell_r(s_l) = \sigma_l$, $v_j \in s_j$.*

Proof. Proof by induction over j:

- Base case: $i = j$, so $v_j \in s_j$ is equivalent to $v_i \in s_i$.

- Induction assumption: The above statement holds when $j \leq k$.
- Induction step: For every $(v, \sigma, v') \in \delta_r$ and $(s, s') \in \delta$, $(v \in s \wedge \ell_r(s) = \sigma) \Rightarrow v' \in s'$. In other words, if v is true in some transducer state s and $\ell_r(s) = \sigma$, then v' is true in every follower state s' of s. This means that if A_r has a transition that takes it from v to v' with the input $\ell_r(s)$, and v holds in s, then v' must hold in every follower state of s. The induction assumption states that $v_k \in s_k$. If A_r has a transition (v_k, σ_k, v_{k+1}), then $v_{k+1} \in s_{k+1}$. Therefore the above statement holds for $j \leq k+1$ as well, and by induction for any j.

□

Lemma A.2. *If $s_1, \ldots, s_{n+1}$ is an execution of $T_{r \mapsto \phi}$, $q_0 \in s_1$, and $\ell_r(s_1, \ldots, s_{n-1}) \in \mathcal{L}(r)$, then $F_r \cap s_n \neq \emptyset$ and therefore $q^\phi \in s_{n-1}$.*

Proof. Because $\ell_r(s_1, \ldots, s_{n-1}) \in \mathcal{L}(r)$, there exists a sequence of transitions (q_0, $\ell_r(s_1)$, q_1), (q_1, $\ell_r(s_2)$, q_2), …, (q_{n-2}, $\ell_r(s_{n-1})$, q_{n-1}), each in δ_r, for some q_1, q_2, … $\in Q_r$ and $q_{n-1} \in F_r$. This, combined with Lemma A.1 implies $q_{n-1} \in s_n$. Because of δ, this means that $q^\phi \in s_{n-1}$. □

Lemma A.3. *If $s_1, \ldots, s_n$ is an execution of $T_{r \mapsto \phi}$ that accepts at s_i, then $\ell_r(s_i, \ldots, s_{n-1}) \notin \mathcal{L}_{\text{pref}}(r)$.*

Proof. Suppose that $\ell_r(s_i, \ldots, s_{n-1}) \in \mathcal{L}_{\text{pref}}(r)$, i.e. $\ell_r(s_i, \ldots, s_{n-1})$ is a proper prefix of some word in $\mathcal{L}(r)$. Since $q_0 \in s_i$, and because of Lemma A.1, $q_j \in s_n$ for some $q_j \in Q_r$. This contradicts with s_n being a final state of $T_{r \mapsto \phi}$, so $\ell(s_i, \ldots, s_{n-1}) \notin \mathcal{L}_{\text{pref}}(r)$. □

Lemma A.4. *If $s_1, \ldots, s_i, \ldots, s_n$ is an execution of the transducer that accepts at s_i, then $\ell_r(s_i, \ldots, s_{n-1}) \notin \mathcal{L}_{\text{pref}}(r)$ and $\forall j, i < j < n$: if $s_i, \ldots, s_j \in \mathcal{L}(r)$ then $q^\phi \in s_j$.*

Proof. Follows from Lemmas A.1, A.2, and A.3. Lemmas A.1 and A.2 together prove that all matches for r are detected whenever q_0 is set to true, and Lemma A.3 proves that the suffix $s_i \dots s_{n-1}$ of the path after q_0 is true cannot belong to the prefix language of r. □

Lemma A.5 shows how to construct an accepting execution where the output variable is true in a single state, and lemma A.6 shows how to combine such executions. Lemma A.7 wraps up by stating that there always exists the desired execution that accepts at all the points where the tail implication property holds.

Lemma A.5. *If $\pi = p_1, \dots, p_{n-1}$ is an input sequence for $T_{r \mapsto \phi}$ and $\pi \models_k^{\text{strong}} r \mapsto \phi$ for some k, then there exists an accepting execution $s_1, \dots, s_n$ of $T_{r \mapsto \phi}$ for π such that $q_0 \in s_k$.*

Proof. The execution $s_0, s_1, \dots, s_n$ can be constructed in the following way:

- $s_i = p_i$ when $0 \leq i < k$, i.e. only atomic propositions hold in the states before s_k, all the state variables are false.
- $q_0 \in s_k$.
- $s_k \setminus \ell_r(s_k) = \{q_0\}$, i.e. q_0 is the only state variable that holds in s_k.
- $\bigwedge_{(v,\sigma,v') \in \delta_r} \Big((v \in s_i \wedge \ell_r(s_i) = \sigma) \Leftrightarrow v' \in s_{i+1} \Big)$ when $k \leq i < n$, i.e. the implication in the transition relation is replaced with equivalence after s_k.

For the above execution, if $q \in s_i$ for some $q \in Q_r$, then there exists an execution of A_r that takes it from q_0 to q with the input sequence $\ell_r(s_k, \dots, s_{i-1})$. This is proven by induction over i:

- Base case: $i = k$, and the execution of A_r is the empty sequence.
- Induction assumption: The above statement holds when $i \leq j$.

- Induction step: Let m be any $k < m \leq n$. A state variable $v' \in Q_r$ holds in the state s_m of the above execution if and only if there is a transition of A_r that takes it from v to v' with the input $\ell_r(s_{m-1})$, and $v \in s_{m-1}$. Thus, for every state variable in s_{j+1}, there is a transition of A_r with a corresponding state variable in s_j with the input $\ell_r(s_j)$. The induction assumption states that for every state variable in s_j a corresponding execution exists, and therefore an execution exists for every state variable in s_{j+1}, and by induction for every state variable in any state of the execution.

As a direct consequence, if $F_r \cap s_m \neq \emptyset$ for some m, then $\ell_r(s_k, \ldots, s_{m-1}) \in \mathcal{L}(r)$. Since $\pi \models_k^{\text{strong}} r \mapsto \phi$, q^ϕ must hold in s_{m-1}. Moreover, since $\ell_r(s_k, \ldots, s_n) \notin \mathcal{L}_{\text{pref}}(r)$, and since every state of A_r has an execution that leads to an accepting state, s_n cannot contain any variables from Q_r, so the final state constraint holds.

□

Lemma A.6. *If $s_1, \ldots, s_n$ and $s'_1, \ldots, s'_n$ are accepting executions of $T_{r \mapsto \phi}$ s.t. $\forall i, 1 \leq i \leq n : \ell_r(s_i) = \ell_r(s'_i)$, then $s_1 \cup s'_1, \ldots, s_n \cup s'_n$ is an accepting execution of $T_{r \mapsto \phi}$.*

Proof. The final state constraint obviously holds, since $s_n \cap Q_r = \emptyset$ and $s'_n \cap Q_r = \emptyset$. The transition relation holds since for any k, $1 \leq k < n$:

$$
\begin{aligned}
&\bigwedge_{(v,\sigma,v') \in \delta_r} \Big((v \in s_k \wedge \ell_r(s_k) = \sigma) \Rightarrow v' \in s_{k+1} \Big) \wedge \\
&\bigwedge_{(v,\sigma,v') \in \delta_r} \Big((v \in s'_k \wedge \ell_r(s'_k) = \sigma) \Rightarrow v' \in s'_{k+1} \Big) \\
\Rightarrow& \\
&\bigwedge_{(v,\sigma,v') \in \delta_r} \Big(((v \in s_k \wedge \ell_r(s_k) = \sigma) \Rightarrow v' \in s_{k+1}) \wedge \\
&\qquad ((v \in s'_k \wedge \ell_r(s'_k) = \sigma) \Rightarrow v' \in s'_{k+1}) \Big) \\
\Rightarrow&
\end{aligned}
$$

$$\bigwedge_{(v,\sigma,v')\in\delta_r} \Big((v \in s_k \wedge \ell_r(s_k) = \sigma) \vee (v \in s'_k \wedge \ell_r(s'_k) = \sigma) \Rightarrow (v' \in s_{k+1}) \vee (v' \in s'_{k+1}) \Big)$$

$\Rightarrow$

$$\bigwedge_{(v,\sigma,v')\in\delta_r} \Big((v \in s_k \vee v \in s'_k) \wedge (\ell_r(s_k) = \sigma) \Rightarrow (v' \in s_{k+1}) \vee (v' \in s'_{k+1}) \Big)$$

$\Rightarrow$

$$\bigwedge_{(v,\sigma,v')\in\delta_r} \Big((v \in (s_k \cup s'_k) \wedge \ell_r(s_k) = \sigma) \Rightarrow v' \in (s_{k+1} \cup s'_{k+1}) \Big)$$

and:

$$\Big(F_r \cap s_{k+1} \neq \emptyset \Rightarrow q^\phi \in s_k \Big) \wedge \Big(F_r \cap s'_{k+1} \neq \emptyset \Rightarrow q^\phi \in s'_k \Big)$$

$\Rightarrow$

$$\Big(F_r \cap s_{k+1} \neq \emptyset \vee F_r \cap s'_{k+1} \neq \emptyset \Big) \Rightarrow \Big(q^\phi \in s_k \vee q^\phi \in s'_k \Big)$$

$\Rightarrow$

$$\Big(F_r \cap (s_{k+1} \cup s'_{k+1}) \neq \emptyset \Big) \Rightarrow \Big(q^\phi \in (s_k \cup s'_k) \Big)$$

□

Lemma A.7. *Let $\pi = p_1, \ldots, p_{n-1}$ be an input sequence for $T_{r\mapsto\phi}$. There exists an accepting execution $s_1, \ldots, s_n$ of $T_{r\mapsto\phi}$ for π such that for every pair of indices i, j the following holds: if $p_i, \ldots, p_j \in \mathcal{L}(r)$ and $\pi \models^{\text{strong}}_j \phi$, then $q_0 \in s_i$.*

Proof. Follows from Lemmas A.5 and A.6. Lemma A.5 shows how to construct an execution that accepts at a single state, and Lemma A.6 shows that two executions that agree on the input sequence can be combined by taking the union of the corresponding states. This results in an execution where a variable is true in a state if it was true in either of the combined executions in that state. □

With the lemmas above we can prove that the presented transducer is correct for the tail implication operator.

Lemma 3.8. *If T_ϕ is a transducer for ϕ, then $T_{r\mapsto\phi} = T_\phi \rhd_{q^\phi} T_{r\mapsto}$ is a transducer for the formula $r \mapsto \phi$.*

Proof. Follows from Lemmas A.4 and A.7. □

A.4 Correctness of the transducer for the tail conjunction operator

The following lemmas prove the correctness of the transducer for the tail implication operator. Lemma A.8 formalises the automaton simulation, and lemma A.9 proves that there are no bad executions for the transducer.

Lemma A.8. *If $s_i, \ldots, s_j$ is an execution of the transducer, $q_i \in s_i$ for some $q_i \in Q_r$, and $\forall l, i \leq l < j : q^\phi \notin s_l$, then there is some $q_j \in s_j$ s.t. $q_j \in Q_r$ and the input sequence $\ell_r(s_i, \ldots, s_{j-1})$ takes A_r from q_i to q_j.*

Proof. Proof by induction over j:

- Base case: $i = j$ and $s_i = s_j$, so both claims hold trivially.
- Induction assumption: The claims hold when $j \leq k$.
- Induction step: There is an execution $s_i, \ldots, s_k$, $q_i \in s_i$, and $q_k \in s_k$. Because of the initial assumption, we also assume that $q^\phi \notin s_k$. Now we extend the execution with a single step to s_{k+1}. Note that if the transition relation does not permit such an extension, then the initial assumption that the execution exists is broken. Since $q^\phi \notin s_k$, the transition relation implies:

$$q_k \in s_k \Rightarrow \bigvee_{(q_k,\sigma,q')\in\delta_r} \ell_r(s_k) = \sigma \wedge q' \in s_{k+1}$$

  This means that there exists some $q' \in Q_r$ s.t. $q' \in s_{k+1}$ and $(q_k, \sigma, q') \in \delta_r$. Now both claims of Lemma A.8 hold for s_{k+1}, and by induction for any execution.

□

Lemma A.9. *If $s_1, \ldots, s_i, \ldots, s_n$ is an execution of the transducer that accepts at s_i, then there exists some j s.t. $i \leq j < n$, $q^\phi \in s_j$, and $\ell_r(s_i, \ldots, s_j) \in \mathcal{L}(r)$.*

Proof. Because of Lemma A.8 and the final state constraint of the transducer, there must be some s_j in the execution s.t. $q^\phi \in s_j$. Additionally, there is some $q_j \in s_j$ such that $\ell_r(s_i, \ldots, s_{j-1})$ takes A_r from q_0 to q_j. On the other hand, if there is no transition $(q_j, \ell_r(s_j), q') \in \delta_r$ for some $q' \in F_r$, then the transition relation at that point implies:

$$q_j \in s_j \Rightarrow \bigvee_{(q_j, \sigma, q') \in \delta_r} \ell_r(s_j) = \sigma \wedge q' \in s_{j+1}$$

That would mean that there is some q' in s_{j+1}, and because of Lemma A.8 the final state constraint would not be satisfied. That means that there must exist some transition from q_j to an accepting state of A_r with the input $\ell_r(s_j)$, which in turn means that $\ell_r(s_i, \ldots, s_j) \in \mathcal{L}(r)$. □

Lemma A.10 shows how to construct an accepting execution where the output variable is true in a single state, and lemma A.11 shows how to combine such executions. Lemma A.12 wraps up by stating that there always exists the desired execution that accepts at all the points where the tail conjunction property holds.

Lemma A.10. *If $\pi = p_1, \ldots, p_{n-1}$ is an input sequence for $T_{r \diamond\!\!\rightarrow \phi}$ such that $\pi \models_k^{\text{strong}} r \diamond\!\!\rightarrow \phi$ for some k, then there exists an accepting execution $s_1, \ldots, s_n$ of $T_{r \diamond\!\!\rightarrow \phi}$ for π such that $q_0 \in s_k$.*

Proof. Because $\pi \models_k^{\text{strong}} r \diamond\!\!\rightarrow \phi$, there exists some $j, k \leq j < n$ s.t. $\pi \models_j^{\text{strong}} \phi$ and $p_k, \ldots, p_j \in \mathcal{L}(r)$. The execution can then be constructed as follows:

- $s_i = p_i$ when $1 \leq i < k$, i.e. only atomic propositions hold in the states before s_k, all the state variables are false.
- $q_0 \in s_k$.

- $s_k \setminus \ell_r(s_k) = \{q_0\}$, i.e. the only state variable in s_k is q_0.
- Let $(q_0, p_k, q_{k+1}), (q_{k+1}, p_{k+1}, q_{k+2}), \ldots, (q_j, p_j, q_{j+1})$ be a sequence of transitions that takes A_r from q_0 to some $q_{j+1} \in F_r$ with the input $p_k, \ldots, p_j$. Such a sequence is guaranteed to exist, since $p_k, \ldots, p_j \in \mathcal{L}(r)$. For each s_i, $k \leq i \leq j$, $s_i \setminus \ell_r(s_i) = q_i$. In other words, the only state variable in each state from s_k to s_j is the state variable from the execution of A_r.
- For each s_i, $j < i \leq n$, $\ell_r(s_i) = s_i$, i.e. no state variables are true after s_j.

The final state constraint obviously holds for the above execution. The transition relation trivially holds for each pair of states (s_i, s_{i+1}) when $1 \leq i < k$, since no state variables are true, and therefore every implication in the transition relation is true.

Each pair of states (s_i, s_{i+1}), $k \leq i < j$ also satisfies the transition relation, since the state variables are taken from an execution of A_r, and therefore:

$$q_i \in s_i \Rightarrow (q_i, \sigma, q_{i+1}) \in \delta_r \wedge \ell_r(s) = \sigma \wedge q_{i+1} \in s_{i+1}$$

The rest of the implications are again trivially true, as no other state variables are true.

The pair of states (s_j, s_{j+1}) satisfies the transition relation, because there is a transition $(q_j, \ell_r(s_j), q_{j+1}) \in \delta_r$, and therefore:

$$q_j \in s_j \Rightarrow (q_j, \sigma, q_{j+1}) \in \delta_r \wedge \ell_r(s) = \sigma \wedge q_{j+1} \in F_r \wedge q^\phi \in s_j$$

Note that $q_{j+1} \notin s_{j+1}$.

For each pair (s_i, s_{i+1}), $j < i < n$, again no state variables are true, so the implications in the transition relation hold trivially. □

Lemma A.11. *If $s_1, \ldots, s_n$ and $s'_1, \ldots, s'_n$ are accepting executions of $T_{r\diamond\rightarrow\phi}$ s.t. $\forall i, 1 \leq i \leq n : \ell_r(s_i) = \ell_r(s'_i)$, then $s_1 \cup s'_1, \ldots, s_n \cup s'_n$ is an accepting execution of $T_{r\diamond\rightarrow\phi}$.*

Proof. The final state constraint obviously holds, since $s_n \cap Q_r = \emptyset$ and $s'_n \cap Q_r = \emptyset$.

The transition relation holds since for any k, $1 \leq k < n$:

$$\bigwedge_{v \in Q_r} \left(v \in s_k \Rightarrow \bigvee_{(v,\sigma,v') \in \delta_r} \ell_r(s_k) = \sigma \wedge (v' \in s_{k+1} \vee (v' \in F_r \wedge q^\phi \in s_k)) \right) \wedge$$
$$\bigwedge_{v \in Q_r} \left(v \in s'_k \Rightarrow \bigvee_{(v,\sigma,v') \in \delta_r} \ell_r(s'_k) = \sigma \wedge (v' \in s'_{k+1} \vee (v' \in F_r \wedge q^\phi \in s'_k)) \right)$$

$\Rightarrow$

$$\bigwedge_{v \in Q_r} \left(\left(v \in s_k \Rightarrow \bigvee_{(v,\sigma,v') \in \delta_r} \ell_r(s_k) = \sigma \wedge (v' \in s_{k+1} \vee (v' \in F_r \wedge q^\phi \in s_k)) \right) \wedge \right.$$
$$\left. \left(v \in s'_k \Rightarrow \bigvee_{(v,\sigma,v') \in \delta_r} \ell_r(s'_k) = \sigma \wedge (v' \in s'_{k+1} \vee (v' \in F_r \wedge q^\phi \in s'_k)) \right) \right)$$

$\Rightarrow$

$$\bigwedge_{v \in Q_r} \left((v \in s_k \vee v \in s'_k) \Rightarrow \right.$$
$$\left(\bigvee_{(v,\sigma,v') \in \delta_r} \ell_r(s_k) = \sigma \wedge (v' \in s_{k+1} \vee (v' \in F_r \wedge q^\phi \in s_k)) \right) \vee$$
$$\left. \left(\bigvee_{(v,\sigma,v') \in \delta_r} \ell_r(s'_k) = \sigma \wedge (v' \in s'_{k+1} \vee (v' \in F_r \wedge q^\phi \in s'_k)) \right) \right)$$

$\Rightarrow$

$$\bigwedge_{v \in Q_r} \left((v \in s_k \vee v \in s'_k) \Rightarrow \right.$$
$$\left(\bigvee_{(v,\sigma,v') \in \delta_r} \ell_r(s_k) = \sigma \wedge (v' \in s_{k+1} \vee (v' \in F_r \wedge q^\phi \in s_k)) \vee \right.$$
$$\left. \left. \ell_r(s'_k) = \sigma \wedge (v' \in s'_{k+1} \vee (v' \in F_r \wedge q^\phi \in s'_k)) \right) \right)$$

$$\Rightarrow$$

$$\bigwedge_{v \in Q_r} \Big((v \in s_k \vee v \in s'_k) \Rightarrow \Big(\bigvee_{(v,\sigma,v') \in \delta_r} \ell_r(s_k) = \sigma \wedge ((v' \in s_{k+1} \vee (v' \in F_r \wedge q^\phi \in s_k)) \vee (v' \in s'_{k+1} \vee (v' \in F_r \wedge q^\phi \in s'_k))) \Big) \Big)$$

$$\Rightarrow$$

$$\bigwedge_{v \in Q_r} \Big((v \in s_k \vee v \in s'_k) \Rightarrow \Big(\bigvee_{(v,\sigma,v') \in \delta_r} \ell_r(s_k) = \sigma \wedge ((v' \in s_{k+1} \vee v' \in s'_{k+1}) \vee (v' \in F_r \wedge q^\phi \in s_k) \vee (v' \in F_r \wedge q^\phi \in s'_k)) \Big) \Big)$$

$$\Rightarrow$$

$$\bigwedge_{v \in Q_r} \Big((v \in s_k \cup s'_k) \Rightarrow \Big(\bigvee_{(v,\sigma,v') \in \delta_r} \ell_r(s_k) = \sigma \wedge ((v' \in s_{k+1} \cup s'_{k+1}) \vee (v' \in F_r \wedge q^\phi \in s_k \cup s'_k)) \Big) \Big)$$

□

Lemma A.12. *Let $\pi = p_1, \ldots, p_{n-1}$ be an input sequence for $T_{r \diamond\!\!\rightarrow \phi}$. There exists an accepting execution $s_1, \ldots, s_n$ of $T_{r \diamond\!\!\rightarrow \phi}$ for π such that $\forall i, 1 \leq i < n : (\exists j, i \leq j < n : \ell_r(s_i, \ldots, s_j) \in \mathcal{L}(r)$ and $q^\phi \in s_j) \Rightarrow q_0 \in s_i$.*

Proof. Follows from Lemmas A.10 and A.11. □

With the lemmas above we can prove that the presented transducer is correct for the tail conjunction operator.

Lemma 3.10. *If T_ϕ is a transducer for ϕ, then $T_{r \diamond\!\!\rightarrow \phi} = T_\phi \triangleright_{q^\phi} T_{r \diamond\!\!\rightarrow}$ is a transducer for the formula $r \diamond\!\!\rightarrow \phi$.*

Proof. Follows from Lemmas A.9 and A.12. □

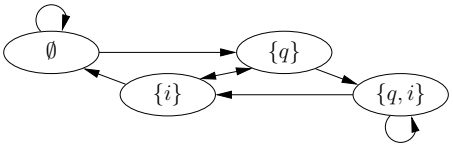

Figure 1: A graphical presentation of the states and the transition relation of a transducer for the formula $\mathbf{X}!\, i$

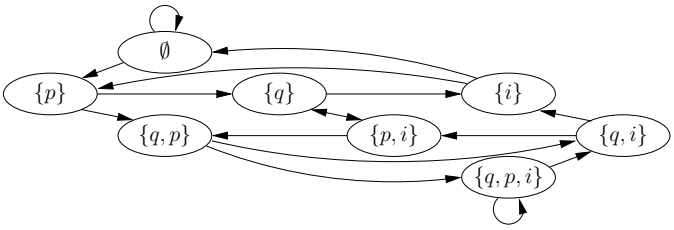

Figure 2: A graphical presentation of the states and the transition relation of a transducer for the formula $\mathbf{X}!\mathbf{X}!\, i$

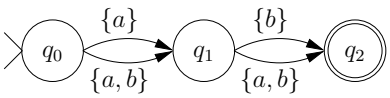

Figure 3: An automaton for the SERE $a \cdot b$

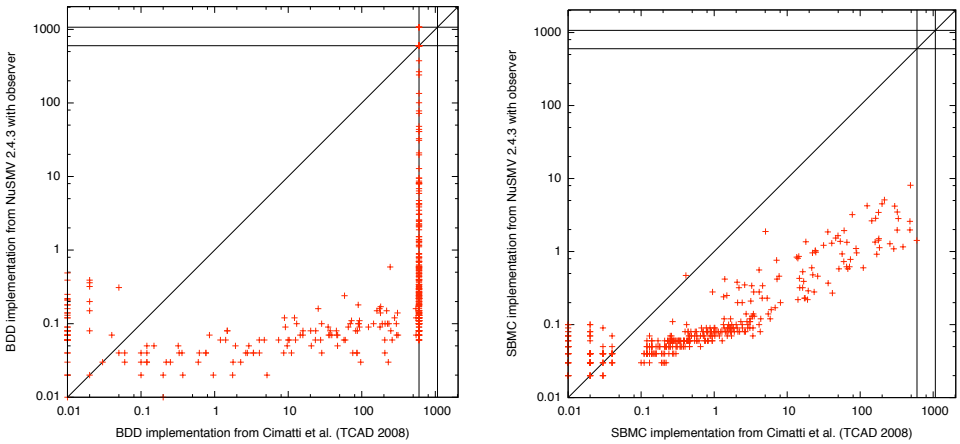

Figure 4: Run times (in seconds) of BDD-based implementations (left) and SBMC-based implementations (right) for the PSL benchmarks.

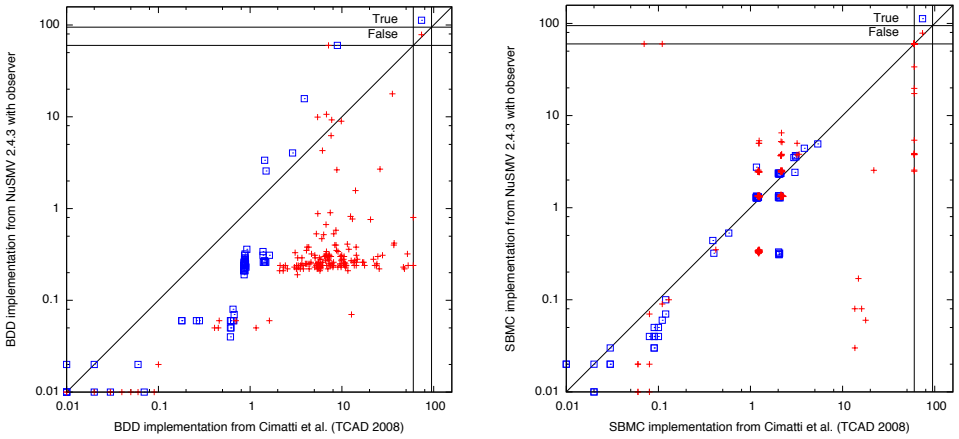

Figure 5: Run times (in seconds) of BDD-based implementations (left) and SBMC-based implementations (right) for the real life benchmarks.