Efficient Model Checking of PSL Safety Properties

Tuomas Launiainen
Department of Information and Computer Science
School of Science and Technology
Aalto University
PO Box 15400,
FI-00076 Aalto, Finland
Tuomas.Launiainen@tkk.fi

Keijo Heljanko
Department of Information and Computer Science
School of Science and Technology
Aalto University
PO Box 15400,
FI-00076 Aalto, Finland
Keijo.Heljanko@tkk.fi

Tommi Junttila
Department of Information and Computer Science
School of Science and Technology
Aalto University
PO Box 15400,
FI-00076 Aalto, Finland
Tommi.Junttila@tkk.fi

***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 formulas in question. The presented technique is inspired by the temporal testers approach of Pnueli and co-authors but is aimed at finite words instead of infinite words. 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.**

***Keywords*-PSL; safety properties; model checking; NuSMV**

I. 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 [1], that are restricted to safety 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. 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* (see Sect. II-C) for the PSL formulas in question. The semantics in our approach is based on [2] which coincides with the latest revision of the semantics of PSL [3]. Our approach extends to PSL the approach of [4] for finding so called informative bad prefixes for linear temporal logic (LTL) formulas. Thus our approach is sound for all PSL formulas 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 so called informative bad prefix ([4], see also Sect. II-C) for the PSL formula in question. On the technical level our approach is inspired by [5] but instead of general (non-safety) properties with temporal testers on infinite words, our approach is tailored for safety properties with *transducers* on finite words. The generated transducer is then translated into an observer NuSMV module and an invariant specification in a way that resembles the classical approach of encoding LTL model checking to an observer SMV module with fairness constraints [6] but again we are looking for finite words instead of infinite words as counterexamples.

There are a number of papers that encode smaller subsets of PSL safety properties than our encoding. The most widely known is the so called safety simple subset [7], [8], [9], [10]. 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 [11] tailored to be used in runtime monitoring of PSL in a simulation setting. The approach of [4] for encoding informative bad prefixes for LTL formulas has been implemented in the `scheck` tool [12] 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* formulas.

One could argue that using the liveness-to-safety reduction [13], [14], [15] 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. This is especially true with model checking techniques such as symbolic model checking with BDDs [16] 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., [17]. There is also a symbolic algorithm with a better theoretical worst-case complexity [18]. 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 [19]. 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 II describes the syntax and semantics of PSL, as well as introduces the central notion of informative bad prefixes. In Sect. III transducers are introduced and it is shown how one can construct a transducer for a PSL formula. In Sect. IV the implementation of model checking based on transducers encoded as NuSMV modules is described. Section V reports on the experiments and Sect. VI presents the conclusions. The proofs of presented Lemmas are omitted due to space restrictions but can be found in [20].

II. PSL SYNTAX AND SEMANTICS

This section formally defines the applied syntax and semantics of PSL and is based on the revised standard [3], [2], where 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 [3] as well as the Optional Branching Extension are not considered. The semantics applied in this paper is equivalent to the *strong semantics for finite paths* in [3], [2]. It 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 [3], [2]. 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, searching for counter-examples can be done by searching for paths that satisfy the negation of the property and using the strong semantics.

A. Syntax

Assume a non-empty set of atomic propositions AP. The syntax of Sequential Extended Regular Expressions (SERE) is defined by the grammar

$$r \;::=\; [*0] \mid p \mid \neg p \mid r_1[+] \mid r_1 \cdot r_2 \mid r_1 \circ r_2 \mid r_1 \cup r_2 \mid r_1 \cap r_2,$$

where p varies of over atomic propositions in AP and r_1 and r_2 are SEREs. 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. 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 expressed with the ones here. This syntax is similar to the one in [3].

The syntax of PSL formulas is defined by the grammar

$$\phi \;::=\; p \mid \neg\phi_1 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \,\mathbf{U}\, \phi_2 \mid \phi_1 \,\mathbf{R}\, \phi_2 \mid \mathbf{X}!\phi_1 \mid \mathbf{X}\phi_1 \mid r \mapsto \phi_1 \mid r \diamond\!\!\rightarrow \phi_1,$$

where ϕ_1 and ϕ_2 are PSL formulae, r is a SERE, and p varies over atomic propositions 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. 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 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, one letter overlapping, postfix 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 operator also appears in [19]. In addition to these operators, [3] also includes formulas of form $r!$ stating that a match must be found for the SERE r; we assume that these are rewritten to tail conjunctions with the equality $r! \equiv r \diamond\!\!\rightarrow \mathbf{true}$. Another, weaker version of the SERE match is presented in [3] as well but we do not present it here as it is equivalent to $r!$ under the applied strong semantics for finite paths.

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 with the full semantics of PSL from [3], [2]: $\neg(\phi \vee \psi) \equiv (\neg\phi \wedge \neg\psi)$, $\neg(\phi \wedge \psi) \equiv (\neg\phi \vee \neg\psi)$, $\neg(\mathbf{X}!\phi) \equiv (\mathbf{X}\neg\phi)$, $\neg(\mathbf{X}\phi) \equiv (\mathbf{X}!\neg\phi)$, $\neg(\phi \,\mathbf{U}\, \psi) \equiv (\neg\phi \;\mathbf{R}\; \neg\psi)$,

$\neg(\phi\,\mathbf{R}\,\psi) \equiv (\neg\phi\ \mathbf{U}\ \neg\psi)$, $\neg(r \mapsto \psi) \equiv (r \diamond\!\!\rightarrow \neg\psi)$, 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. *For these rewritten formulae* we use only the strong semantics from [3], [2], which is presented below.

B. Semantics

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. 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 $p \in AP$.
- $\mathcal{L}(r[+]) = \{\pi \mid \exists n \geq 1 : \pi = \pi_1\pi_2 \ldots \pi_n$ and $\forall i, 1 \leq i \leq n : \pi_i \in \mathcal{L}(r)\}$
- $\mathcal{L}(r_1 \cdot r_2) = \{\pi_1\pi_2 \mid \pi_1 \in \mathcal{L}(r_1)$ and $\pi_2 \in \mathcal{L}(r_2)\}$.
- $\mathcal{L}(r_1 \circ r_2) = \{\pi_1 s \pi_2 \mid \pi_1 s \in \mathcal{L}(r_1)$ and $s\pi_2 \in \mathcal{L}(r_2)\}$.
- $\mathcal{L}(r_1 \cup r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$.
- $\mathcal{L}(r_1 \cap r_2) = \mathcal{L}(r_1) \cap \mathcal{L}(r_2)$.

For example, the path $\{a,b\}\{a,d\}\{b\}$ is in $\mathcal{L}(a[+] \cdot (b \cup c))$ but the path $\{a,b\}\{a,d\}\{a\}$ is not. In addition to the base language, we define the prefix language $\mathcal{L}_{\text{pref}}(r) \subseteq S^\star$ for a SERE r 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.

For PSL formulae, we use the *strong semantics for finite paths* as given in [3], [2]. Assume a finite, non-empty path $\pi = s_1 \ldots s_n \in S^+$ for some $n \geq 1$. For each $1 \leq i \leq n$, define the relation $\models^f_i$ by the following inductive rules:

- $\pi \models^f_i p$ iff $p \in s_i$, where $p \in AP$.
- $\pi \models^f_i \neg p$ iff $p \notin s_i$, where $p \in AP$.
- $\pi \models^f_i \phi \wedge \psi$ iff $\pi \models^f_i \phi$ and $\pi \models^f_i \psi$.
- $\pi \models^f_i \phi \vee \psi$ iff $\pi \models^f_i \phi$ or $\pi \models^f_i \psi$.
- $\pi \models^f_i \mathbf{X}!\phi$ iff $i < n$ and $\pi \models^f_{i+1} \phi$.
- $\pi \models^f_i \mathbf{X}\phi$ iff $i < n$ and $\pi \models^f_{i+1} \phi$. (Note that this is exactly the same as $\mathbf{X}!\phi$.)
- $\pi \models^f_i \phi_1\,\mathbf{U}\,\phi_2$ iff $\exists j, i \leq j \leq n : (\pi \models^f_j \phi_2) \wedge (\forall k, i \leq k < j : \pi \models^f_k \phi_1)$.
- $\pi \models^f_i \phi_1\,\mathbf{R}\,\phi_2$ iff $\exists j, i \leq j \leq n : (\pi \models^f_j \phi_1) \wedge (\forall k, i \leq k \leq j : \pi \models^f_k \phi_2)$.
- $\pi \models^f_i r \mapsto \phi$ iff (i) $\forall j, i \leq j \leq n : (s_i \ldots s_j \in \mathcal{L}(r)) \Rightarrow (\pi \models^f_j \phi)$ and (ii) $s_i \ldots s_n \notin \mathcal{L}_{\text{pref}}(r)$.
- $\pi \models^f_i r \diamond\!\!\rightarrow \phi$ iff $\exists j, i \leq j \leq n : (s_i \ldots s_j \in \mathcal{L}(r)) \wedge (\pi \models^f_j \phi)$.

We use $\pi \models^f \phi$ to denote $\pi \models^f_1 \phi$ and say that "ϕ holds on π", or that "π satisfies ϕ", if $\pi \models^f \phi$. Since $\mathbf{X}!\phi$ and $\mathbf{X}\phi$ are equivalent in the strong semantics, we only use $\mathbf{X}!\phi$ from now on. Both are present in the syntax so that rewriting formulae to the negative normal form can be done even with the full semantics of PSL from [3], [2]. The semantics of the $\mathbf{U}$ and $\mathbf{R}$ operators may seem strange if compared to those in LTL. This is because we use the strong semantics for finite paths here.

C. Informative bad prefixes

As mentioned earlier, if a finite prefix π of an infinite or a finite path satisfies a property ϕ under the semantics presented above (i.e. $\pi \models^f \phi$), the whole path satisfies the property under any of the three semantics (weak, neutral, or strong) presented in [2], [3]. In the model checking context this means that if $\pi \models^f \neg\phi$ 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 [4], we formalize this by defining that a finite path $\pi \in S^\star$ is an *informative bad prefix* for a formula ϕ if $\pi \models^f \neg\phi$. For the LTL subset (i.e. PSL formulas without tail implications and tail conjunctions and thus without SEREs), the semantics here is equivalent to the definition of informativity in [4] and thus the definition of informative bad prefixes is also equivalent to the one in [4].

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 (taken from [4]), 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.

III. OBSERVERS

This work uses a custom formalism, similar to temporal testers in e.g. [5], [21], 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.

A. 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, and 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 Fig. 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. Every state in the transducer is an initial state, and the final states are the ones where the output is not true, namely $\emptyset$ and $\{i\}$. An example of an execution of the transducer is: $\{q\}, \{i\}, \{q\}, \{q, i\}, \{i\}$.

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: $F = \{s \in 2^{\{q,i\}} \mid q \notin s\}$, 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 $(q \in s) \Leftrightarrow (i \in s')$.

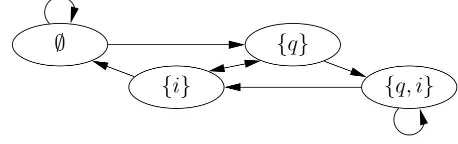

Figure 1. A graphical presentation of the states and the transition relation of a transducer for the formula $\mathbf{X}!i$

B. 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 $T_1 \triangleright_{q^{\text{in}}} T_2$, and defined as $(Q_\triangleright, Q_\triangleright^{\text{in}}, q_\triangleright^{\text{out}}, I_\triangleright, F_\triangleright, \delta_\triangleright)$, where:

- $Q_\triangleright = Q_1 \cup Q_2$,
- $Q_\triangleright^{\text{in}} = Q_1^{\text{in}} \cup (Q_2^{\text{in}} \setminus \{q^{\text{in}}\})$ and the plugged input variable cannot exist in Q_1^{in}, i.e. $q^{\text{in}} \notin Q_1^{\text{in}}$,
- $q_\triangleright^{\text{out}} = q_2^{\text{out}}$,
- $I_\triangleright = \{s_1 \cup s_2[q^{\text{in}}/q_1^{\text{out}}] \mid s_1 \in I_1, s_2 \in I_2,$ and $q_1^{\text{out}} \in s_1 \Leftrightarrow q^{\text{in}} \in s_2,$ and $\bigwedge_{v \in Q_1^{\text{in}} \cap Q_2^{\text{in}}} v \in s_1 \Leftrightarrow v \in s_2\}$,
- $F_\triangleright = \{s_1 \cup s_2[q^{\text{in}}/q_1^{\text{out}}] \mid s_1 \in F_1, s_2 \in F_2,$ and $q_1^{\text{out}} \in s_1 \Leftrightarrow q^{\text{in}} \in s_2,$ and $\bigwedge_{v \in Q_1^{\text{in}} \cap Q_2^{\text{in}}} v \in s_1 \Leftrightarrow v \in s_2\}$, and
- $\delta_\triangleright = \{(s_1 \cup s_2, s_1' \cup s_2')[q^{\text{in}}/q_1^{\text{out}}] \mid (s_1, s_1') \in \delta_1, (s_2, s_2') \in \delta_2,$ and $(q_1^{\text{out}} \in s_1 \Leftrightarrow q^{\text{in}} \in s_2) \wedge (q_1^{\text{out}} \in s_1' \Leftrightarrow q^{\text{in}} \in s_2'),$ and $\bigwedge_{v \in Q_1^{\text{in}} \cap Q_2^{\text{in}}} [(v \in s_1 \Leftrightarrow v \in s_2) \wedge (v \in s_1' \Leftrightarrow v \in s_2')]\}$.

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

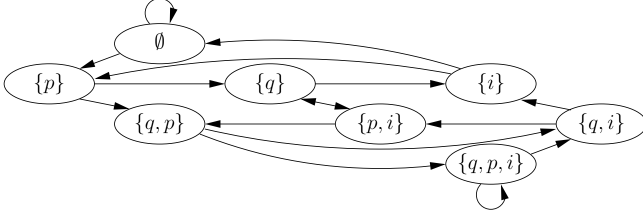

Figure 2. A graphical presentation of the states and the transition relation of a transducer for the formula $\mathbf{X}!\mathbf{X}!i$

a transducer that accepts when the formula $\mathbf{X}!\mathbf{X}!i$ 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 $T_{\mathbf{X}!\mathbf{X}!}$ is then the composition $T_{\mathbf{X}!} \triangleright_j T'_{\mathbf{X}!}$, which is presented in Fig. 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 $\mathbf{X}!\mathbf{X}!i$ holds. The output variable is p, every state is initial, and the final states are $\emptyset$ and $\{i\}$.

C. 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 the following hold:

- For every input $\pi = p_1, \ldots, p_n \in (2^D)^+$, where $D \subseteq AP$, there exists an execution $\gamma = s_1, \ldots, s_n$ of T such that T accepts at a state s_j of the execution iff $\pi \models^f_j \phi$, and
- there are no executions of T for π where T accepts at a state s_j and $\pi \not\models^f_j \phi$.

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 subformulae as explained below. The proofs of Lemmas are omitted due to space restrictions but can be found in [20].

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

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

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

3) The until operator: Recall the semantics of the until-operator: $\pi \models^f_i (\phi_1 \ \mathbf{U} \ \phi_2)$ iff $\exists j, i \leq j \leq n : (\pi \models^f_j \phi_2) \wedge (\forall k, i \leq k < j : \pi \models^f_k \phi_1)$. 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. 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:

- $Q = \{q^{\mathbf{U}}\}$,
- $Q^{\text{in}} = \{q^{\text{left}}, q^{\text{right}}\}$,
- $q^{\text{out}} = q^{\mathbf{U}}$,
- $I = 2^{Q \cup Q^{\text{in}}}$,
- $F = \{s \mid q^{\mathbf{U}} \in s \Leftrightarrow q^{\text{right}} \in s\}$, 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 $\phi_1 \ \mathbf{U} \ \phi_2$ is obtained as the composition $T_{\phi_1 \mathbf{U} \phi_2} = T_1 \triangleright_{q^{\text{left}}} (T_2 \triangleright_{q^{\text{right}}} T_{\mathbf{U}})$, where T_1 is a transducer for ϕ_1 and T_2 is a transducer for ϕ_2.

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

Proof: Omitted due to space constraints, see [20]. ∎

4) The releases operator: The semantics for the releases-operator is: $\pi \models^f_i (\phi_1 \ \mathbf{R} \ \phi_2)$ iff $\exists j, i \leq j \leq n : (\pi \models^f_j \phi_1) \wedge (\forall k, i \leq k \leq j : \pi \models^f_k \phi_2)$. The intuition behind the transducer for the releases-operator is that the formula $\phi_1 \ \mathbf{R} \ \phi_2$ 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:

- $Q = \{q^{\mathbf{R}}\}$,
- $Q^{\text{in}} = \{q^{\text{left}}, q^{\text{right}}\}$,
- $q^{\text{out}} = q^{\mathbf{R}}$,
- $I = 2^{Q \cup Q^{\text{in}}}$,
- $F = \{s \mid q^{\mathbf{R}} \in s \Leftrightarrow (q^{\text{left}} \in s \wedge q^{\text{right}} \in s)\}$, 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 3.5: 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: Refer to [20]. ∎

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. We also define a function $\ell : 2^{Q^{\text{in}} \cup Q} \rightarrow 2^{AP(r)}$ that maps the states of a transducer to the relevant atomic propositions that hold in the state by: $\ell(s) = s \cap AP(r)$. 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 $\mathbf{true}$ and $r \diamond\!\!\rightarrow \phi$ can be rewritten to $\mathbf{false}$.

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) $\Sigma = 2^{AP(r)}$. Using A_r, we can construct a transducer

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

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
- $\delta = \{(s, s') \mid \bigwedge_{(v,\sigma,v') \in \delta_r} ((v \in s \wedge \ell(s) = \sigma) \Rightarrow v' \in s')$ 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.

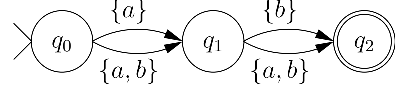

Figure 3. An automaton for the SERE $\{a \cdot b\}$

Lemma 3.6: 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 [20]. ∎

Example 3.7: The following example illustrates a transducer for the formula $\{a \cdot b\} \mapsto c$. The automaton for the SERE part is shown in Fig. 3. The automaton dictates the transition relation of the transducer, which is:

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

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.

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
- $\delta = \{(s, s') \mid \bigwedge_{v \in Q_r} [v \in s \Rightarrow \bigvee_{(v,\sigma,v') \in \delta_r} \ell(s) = \sigma \wedge (v' \in s' \vee (v' \in F_r \wedge q^{\phi} \in s))]\}$.

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 3.8: 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: Again, please refer to [20]. ∎

Example 3.9: 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 Fig. 3. The transition relation for the transducer is:

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

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.

IV. 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 $\neg\phi$ 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, but the output variable is forced to be true in the initial state as well. 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 [22].

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 $\neg p$ **R** $\neg q$, which is the negation of p **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/.

V. 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 [19]. That

implementation is also built on top of NuSMV. For the first comparison, the same set of benchmarks is used as in the paper [19], which includes both general (non-safety) and safety properties. Their BDD-based algorithm, with syntactic optimizations 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 [23] 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. 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 Fig. 4.

In the second set of benchmarks we used the real life models from [15], 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 [19] unlike the approach we compare against. Summing 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 [7], [8], [9], [10], 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 formulas unlike the safety simple subset that is quite restricted in its allowed syntax. In comparison to the explicit automata approach of [12] we can state that our symbolic encoding is exponentially more compact and also handles a larger set of properties, not only the LTL subset.

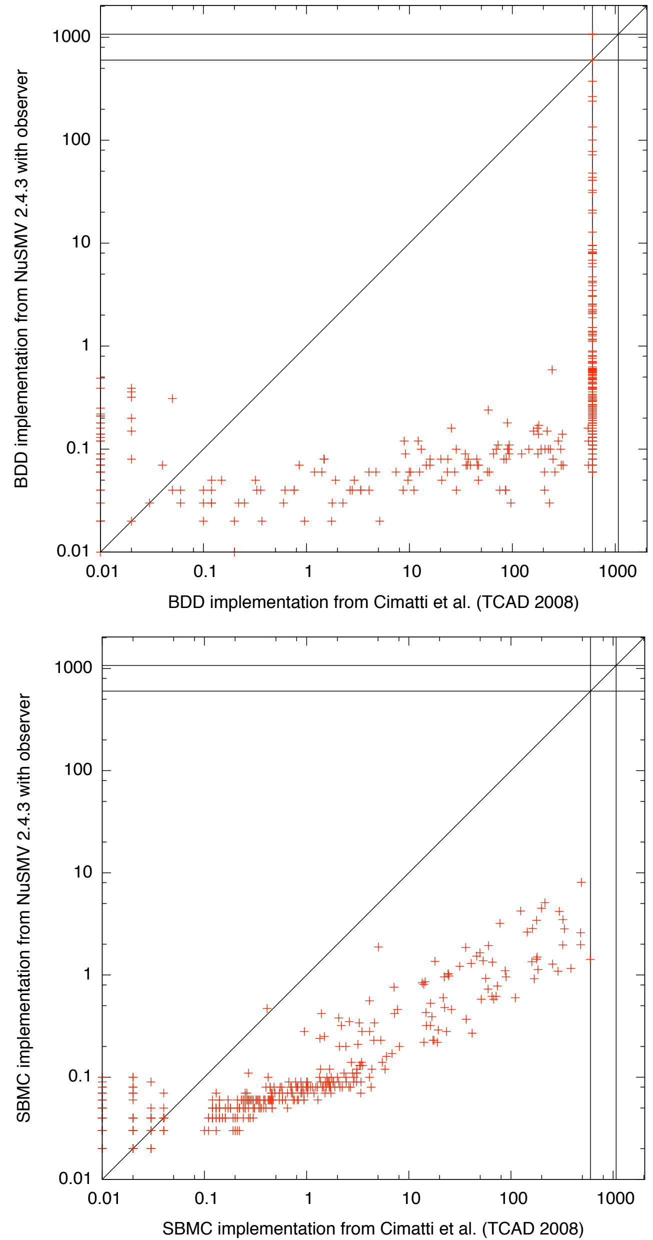

Figure 4. Run times (in seconds) of BDD-based implementations (above) and SBMC-based implementations (below) for the PSL benchmarks.

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

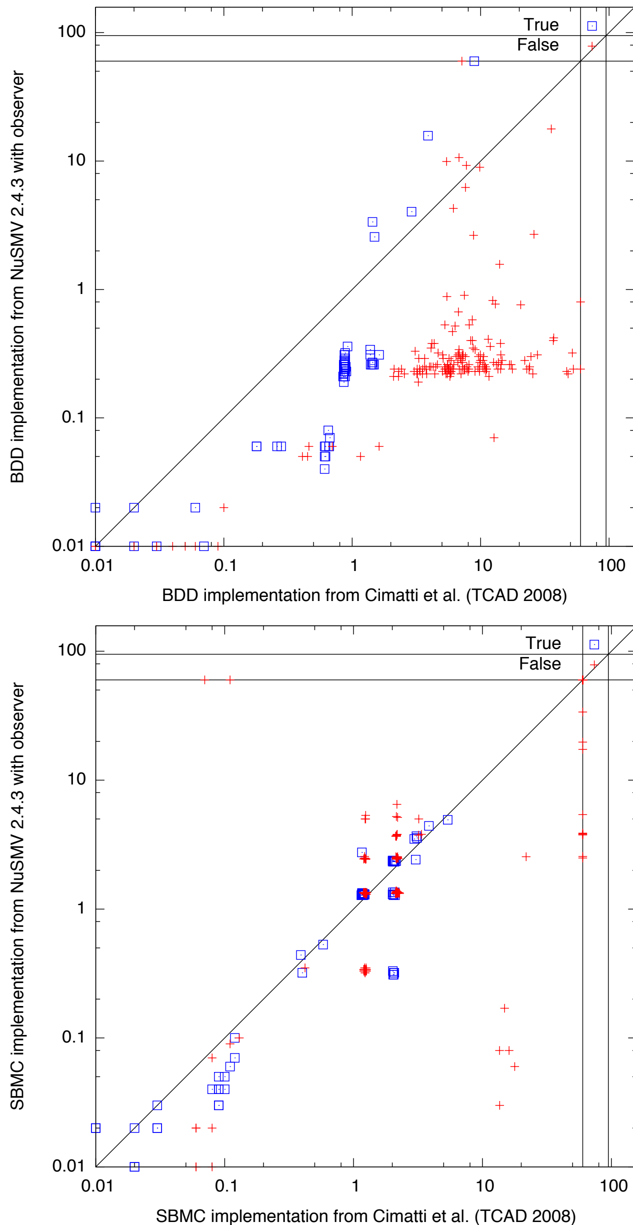

Figure 5. Run times (in seconds) of BDD-based implementations (above) and SBMC-based implementations (below) for the real life benchmarks.

properties, our tool is competitive compared to the state-of-the-art.

VI. Conclusions

We have detailed a fast PSL model checking algorithm for safety properties. The approach uses transducers implemented symbolically, inspired by temporal testers of [21]. The formal semantics of PSL capturing the informative bad prefixes was defined in Sect. II and is based on the latest revision of PSL semantics [3], [2]. The transducers formalism used to construct the observers is detailed in Sect. III. 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 [7], 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 formulas 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 formulas 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 [12] describes a tool that analyzes an LTL specification used in model checking, and detects exactly those formulas 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* [4], and for these LTL formulas 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 [12].

Acknowledgements

The financial support of Academy of Finland (projects 126860 and 128050) and Technology Industries of Finland Centennial Foundation is gratefully acknowledged.

References

[1] K. L. McMillan, “Applications of Craig interpolation to model checking,” in *Proc. ICATPN 2005*, ser. Lecture Notes in Computer Science, vol. 3536. Springer, 2005, pp. 15–16.

[2] C. Eisner and D. Fisman, "Structural contradictions," in *Proc. HVC 2008*, ser. Lecture Notes in Computer Science, vol. 5394. Springer, 2008, pp. 164–178.

[3] ——, "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.

[4] O. Kupferman and M. Y. Vardi, "Model Checking of Safety Properties," *Formal Methods in System Design*, vol. 19, no. 3, pp. 291–314, 2001.

[5] Y. Kesten, A. Pnueli, L.-O. Raviv, and E. Shahar, "Model Checking with Strong Fairness," *Formal Methods in System Design*, vol. 28, no. 1, pp. 57–84, 2006.

[6] E. M. Clarke, O. Grumberg, and K. Hamaguchi, "Another Look at LTL Model Checking," *Formal Methods in System Design*, vol. 10, no. 1, pp. 47–71, 1997.

[7] S. Ben-David, D. Fisman, and S. Ruah, "The safety simple subset," in *Proc. HVC 2005*, ser. Lecture Notes in Computer Science, vol. 3875. Springer, 2005, pp. 14–29.

[8] N. Jin, C. Shen, J. Chen, and T. Ni, "Engineering of an assertion-based $\text{PSL}^{\text{Simple}}$-Verilog dynamic verifier by alternating automata," in *Proc. TTSS 2007*, ser. Electronic Notes in Theoretical Computer Science, vol. 207. Elsevier, 2008, pp. 153–169.

[9] N. Jin and C. Shen, "Dynamic verifying the properties of the simple subset of PSL," in *Proc. TASE 2007*. IEEE Computer Society, 2007, pp. 229–240.

[10] M. Boule and Z. Zilic, "Automata-based assertion-checker synthesis of PSL properties," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 1, 2008.

[11] B. Finkbeiner and L. Kuhtz, "Monitor circuits for LTL with bounded and unbounded future," in *Proc. RV 2009*, ser. Lecture Notes in Computer Science, vol. 5779. Springer, 2009, pp. 60–75.

[12] T. Latvala, "Efficient Model Checking of Safety Properties," in *Proc. SPIN 2003*, ser. Lecture Notes in Computer Science, vol. 2648. Springer, 2003, pp. 74–88.

[13] V. Schuppan and A. Biere, "Efficient reduction of finite state model checking to reachability analysis," *International Journal on Software Tools for Technology Transfer*, vol. 5, no. 2–3, pp. 185–204, 2004.

[14] ——, "Shortest counterexamples for symbolic model checking of LTL with past," in *Proc. TACAS 2005*, ser. Lecture Notes in Computer Science, vol. 3440. Springer, 2005, pp. 493–509.

[15] A. Biere, K. Heljanko, T. Junttila, T. Latvala, and V. Schuppan, "Linear encodings of bounded LTL model checking," *Logical Methods in Computer Science*, vol. 2, no. 5, 2006.

[16] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Compututation*, vol. 98, no. 2, pp. 142–170, 1992.

[17] R. H. Hardin, R. P. Kurshan, S. K. Shukla, and M. Y. Vardi, "A new heuristic for bad cycle detection using BDDs," *Formal Methods in System Design*, vol. 18, no. 2, pp. 131–140, 2001.

[18] R. Bloem, H. N. Gabow, and F. Somenzi, "An algorithm for strongly connected component analysis in $n \log n$ symbolic steps," *Formal Methods in System Design*, vol. 28, no. 1, pp. 37–56, 2006.

[19] A. Cimatti, M. Roveri, and S. Tonetta, "Symbolic compilation of PSL," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1737–1750, 2008.

[20] T. Launiainen, "Model checking PSL safety properties," Master's thesis, Helsinki University of Technology, 2009, available online at: http://lib.tkk.fi/Reports/2009/isbn9789522480422.pdf.

[21] Y. Kesten, A. Pnueli, and L. Raviv, "Algorithmic verification of linear temporal logic specifications," in *Proc. ICALP 1998*, ser. Lecture Notes in Computer Science, vol. 1443. Springer, 1998, pp. 1–16.

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

[23] K. Heljanko, T. Junttila, and T. Latvala, "Incremental and complete bounded model checking for full PLTL," in *Proc. CAV 2005*, ser. Lecture Notes in Computer Science, vol. 3576. Springer, 2005, pp. 98–111.