Syntactic Optimizations for PSL Verification

Alessandro Cimatti[1], Marco Roveri[1], and Stefano Tonetta[2]

[1] ITC-irst Trento, Italy
{cimatti,roveri}@itc.it
[2] University of Lugano, Lugano, Switzerland
tonettas@lu.unisi.ch

Abstract. The IEEE standard Property Specification Language (PSL) allows to express all ω-regular properties mixing Linear Temporal Logic (LTL) with Sequential Extended Regular Expressions (SEREs), and is increasingly used in many phases of the hardware design cycle, from specification to verification.

In recent works, we propose a modular and symbolic PSL compilation that is extremely fast in conversion time and outperforms by several orders of magnitude translators based on the explicit construction and minimization of automata. Unfortunately, our approach creates rather redundant automata, which result in a penalty in verification time.

In this paper, we propose a set of syntactic simplifications that enable to significantly improve the verification time without paying the price of automata simplifications. A thorough experimental analysis over large sets of paradigmatic properties shows that our approach drastically reduces the overall verification time.

1 Introduction

The IEEE standard Property Specification Language PSL [1] is increasingly used in several phases of the design flows: it is a means to describe behavioral requirements, such as assumptions about the environment in which the design is expected to operate, internal behavioral requirements, and further constraints that arise during the design process from specification to verification.

The most important fragment of PSL combines Linear Temporal Logic (LTL) [2] with Sequential Extended Regular Expressions (SERE), a variant of classical regular expressions [1]. This combination results in ω-regular expressiveness, and enables to express many properties of practical interest in a compact and readable way.

The conversion from PSL to Nondeterministic Büchi Automata (NBAs) is an enabling factor for the the adaptation of standard verification tools, and has been recently investigated in several works (e.g. [3,4,5,6,7]).

[3] describes a classical approach based on Alternating Büchi Automata (ABA): the SEREs occurring in the PSL formula are first translated into minimum Nondeterministic Finite Automata (NFA); the NFAs are then combined bottom up and the overall PSL formula is translated into an ABA; the ABA is finally translated into an NBA by means of the Miyano-Hayashi (MH) construction [8]. [4] specializes this approach to SAT-based bounded model checking, exploiting the fact that alternating automata are weak. In [5], a symbolic encoding, based on MH, of the NBA corresponding to the ABA of

O. Grumberg and M. Huth (Eds.): TACAS 2007, LNCS 4424, pp. 505–518, 2007.

the PSL property is proposed. Both approaches try to limit the encoding size (delaying the explosion until search time), but are based on a library that tries to carry out some optimizations in order to minimize the ABA. However, the minimization of ABA is very expensive these approaches are often unable to carry out the conversion in acceptable time, even for PSL specifications of moderate size.

The works in [7] and in [6] independently propose direct encodings of PSL into symbolically represented NBA. Both approaches are compositional, and neither requires the generation of ABA. The former is based on the notion of transducer, while the latter proposes a reduction to Suffix Operator Normal Form (SONF). The experimental evaluation in [6] shows that the SONF construction is extremely fast in the construction of the NBA, thus enabling the verification in cases where the automaton construction blows up.

The results in [6], however, show that the verification times are often in favor of the approaches in [5] and [4]. In fact, the semantic simplifications implemented in the ABA library, though costly, are often able to construct optimized automata, that can then be verified more effectively. This is similar to what happens in LTL model checking, where semantic simplifications on the automaton recognizing the violation, though costly, can often pay off in overall verification time [9].

We notice that some specifications contain obvious forms of redundancy, since designers heavily rely on syntactic sugaring, and redundant specifications may enable reusability. A typical example is length matching between a fixed-length expression, and an expression containing stars. Thus, it is an important practical problem to devise means to deal with redundant specifications, without paying a price in performance. In principle, redundancies can be removed with automata minimization techniques; however, such reductions can be expensive and produce large intermediate automata.

In this paper, we propose a *syntactic* approach to improving PSL verification: rather than simplifying the automaton at a semantic level, we propose a number of syntactic rewriting rules on the formula. After the preprocessing, the SONF-based method result in more compact NBA, which in turn results in much faster verification. The rewrite rules are based on the following ideas. First, we try to minimize the size of the arguments to the SERE language intersection operators, given that they are associated with an exponential blow up. Second, whenever possible we convert the SEREs into LTL, in order to limit as much as possible Suffix operators, and to enable the use of specialized algorithms for LTL. Third, some Suffix Operator Subformulas resulting from the conversion into SONF can be eliminated by taking into account their structure. Finally, we also apply syntactic simplifications to the LTL component of the formula resulting from the conversion into SONF.

We experimentally evaluate our method on the test cases proposed in [6] on a large test suite with formulas identified and classified in [10] to be of practical relevance. The experiments show that the simplifications are computationally cheap, and substantially pay off in terms of verification time. The result is that overall the new method is vastly superior to [6] and to [5]. A final remark in favor of the proposed approach is that it is open and customizable with respect to typical patterns in the application at hand.

The paper is structured as follows. In Section 2 we present the syntax and semantics of PSL. In Section 3 we overview the approaches to PSL conversions into NBA, and

discuss the performance issues. In Sections 4 we describe the various classes of rewriting rules. In Section 5 we experimentally evaluate the impact of our optimizations. Finally, in Section 6 we draw some conclusions and discuss directions for future research.

2 The Property Specification Language PSL

PSL is a very rich language [1]. We consider the subset of PSL that combines Linear Temporal Logic [2] (LTL) and Sequential Extended Regular Expressions (SERE), a variant of classical regular expressions [1]. This subset provides ω-regular expressiveness [11], it is the mostly used in practice and constitutes *the core* of the PSL temporal layer [1]. We will not deal with PSL "clocked" expressions that are not part of the core since any clocked expression can be rewritten into an equivalent un-clocked one [1]. The same applies for the PSL "abort" operator that can be efficiently rewritten into pure LTL as shown in [12].

In the following, we assume as given a set $\mathcal{A}$ of atomic propositions. Let $\Sigma := 2^{\mathcal{A}}$. We denote a letter over the alphabet Σ by ℓ, a word from Σ by v or w, and the concatenation of v and w by vw. We denote with $|w|$ the length of word w. A finite word $w = \ell_0\ell_1\ldots\ell_n$ has length $n+1$, an infinite word has length ω. If $w = \ell_0\ell_1\ldots$, for $0 \leq i < |w|$, we use w^i to denote the letter ℓ_i, and we denote with $w^{i..}$ the suffix of w starting at w^i. When $i \leq j \leq |w|$, we denote with $w^{i..j}$ the finite sequence of letters starting from w^i and ending in w^j ($w^{i..j} := w^i w^{i+1}\ldots w^j$).

SEREs are the PSL version of regular expressions. In particular, they extend the standard regular expressions with language intersection. This allows for a greater succinctness, but it implies a possible exponential blow-up in the conversion to automata. Moreover, the atoms of SEREs are Boolean expressions enabling efficient determinization of automata. Formally,

Definition 1 (SEREs **syntax)**

- *if b is Boolean expression, then b is a* SERE*;*
- *if r is a* SERE*, then r***[*]** *is a* SERE*;*
- *if r_1 and r_2 are* SERE*s, then the following are* SERE*s*

$$r_1 \,;\, r_2 \qquad r_1 : r_2 \qquad r_1 \mid r_2 \qquad r_1 \,\&\, r_2 \qquad r_1 \,\&\&\, r_2.$$

SEREs can be concatenated with the operators **;** and **:**, the former for the consecutive concatenation of two sequences, the latter for one-state overlapping concatenation. The conjunction operators **&** and **&&** can be used to specify overlapping sequences, the latter for length-matching sequences. Disjunction can be specified using the **|** operator. The **[*]** operator specifies finite consecutive repetitions. We use r**[*n]** as an abbreviation of r **;** r **;** ... **;** r, where r is repeated n times.

The semantics of SEREs is formally defined over *finite* words using, as the base case, the semantics of Boolean expressions over letters in Σ, denoted with $\models_B$ hereafter.

Definition 2 (SEREs **semantics).** *Given a Boolean expression b, a* SERE *r, and a* finite word *w, we define the satisfaction relation $w \models r$ as follows:*

- $w \models b$ iff $|w| = 1$ and $w^0 \models_B b$;
- $w \models r_1 \,;\, r_2$ iff $\exists w_1, w_2$ s.t. $w = w_1 w_2$, $w_1 \models r_1$, $w_2 \models r_2$;
- $w \models r_1 : r_2$ iff $\exists w_1, w_2, \ell$ s.t. $w = w_1 \ell w_2$, $w_1 \ell \models r_1$, $\ell w_2 \models r_2$;
- $w \models r_1 \mid r_2$ iff $w \models r_1$ or $w \models r_2$;
- $w \models r_1 \,\&\, r_2$ iff $w \models r_1$ and $\exists w_1, w_2$ s.t. $w = w_1 w_2$, $w_1 \models r_2$, or $w \models r_2$ and $\exists w_1, w_2$ s.t. $w = w_1 w_2$, $w_1 \models r_1$;
- $w \models r_1 \,\&\&\, r_2$ iff $\exists w_1, w_2$ s.t. $w = w_1 w_2$, $w_1 \models r_1, w_1 \models r_2$;
- $w \models r[*]$ iff $|w| = 0$ or $\exists w_1, w_2$ s.t. $|w_1| \neq 0, w = w_1 w_2$, $w_1 \models r$, $w_2 \models r[*]$.

In the definition of the PSL syntax, for technical reasons, we introduce the "releases" operator (that is the dual of the "until" operator), and also we introduce the "suffix conjunction" connective as a dual of the suffix implication. Moreover, we consider only the strong version of the temporal operators (the weak operators can be rewritten in terms of the strong ones [1]) and the strong version of the SEREs (though our approach can be easily extended to deal also with the weak semantics).

Definition 3 (PSL **syntax**). *We define the* PSL *formulas over $\mathcal{A}$, as follows:*

- *if $p \in \mathcal{A}$, p is a* PSL *formula;*
- *if ϕ_1 and ϕ_2 are* PSL *formulas, then $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$ are* PSL *formulas;*
- *if ϕ_1 and ϕ_2 are* PSL *formulas, then $\mathbf{X}\,\phi_1$, $\phi_1\ \mathbf{U}\ \phi_2$, $\phi_1\ \mathbf{R}\ \phi_2$ are* PSL *formulas;*
- *if r is a* SERE *and ϕ is a* PSL *formulas, then $r \diamondsuit\!\!\rightarrow \phi$ and $r \mapsto \phi$ are* PSL *formulas;*
- *if r is a* SERE*, then r is a* PSL *formula.*

The $\mathbf{X}$ ("next-time"), the $\mathbf{U}$ ("until"), and the $\mathbf{R}$ ("releases") operators are called *temporal operators*. We call the $\diamondsuit\!\!\rightarrow$ ("suffix conjunction"), and the $\mapsto$ ("suffix implication"), *suffix operators*. Notice that, the r not occurring in the left side of a suffix operator is the *strong* version of a SERE ($r!$ in the PSL notation). In the following, we will consider such r as an abbreviation for $r \diamondsuit\!\!\rightarrow \mathit{True}$ [1,3]. We also use $\mathbf{G}\,\phi$ as an abbreviation for $\mathit{False}\ \mathbf{R}\ \phi$. LTL can be seen as a subset of PSL in which the suffix operators and the SEREs are suppressed.

We interpret PSL expressions over *infinite* words:

Definition 4 (PSL **semantics**). *Let $w \in \Sigma^\omega$.*

- *$w \models p$ iff $w^0 \models_B p$;*
- *$w \models \neg\phi$ iff $w \not\models \phi$;*
- *$w \models \phi \wedge \psi$ iff $w \models \phi$ and $w \models \psi$;*
- *$w \models \phi \vee \psi$ iff either $w \models \phi$ or $w \models \psi$;*
- *$w \models \mathbf{X}\,\phi$ iff $|w| > 1$ and $w^{1..} \models \phi$;*
- *$w \models \phi\ \mathbf{U}\ \psi$ iff, for some $j \geq 0$, $w^{j..} \models \psi$ and, for all $0 \leq k < j$, $w^{k..} \models \phi$;*
- *$w \models \phi\ \mathbf{R}\ \psi$ iff, for all $j \geq 0$, either $w^{j..} \models \psi$ or,for some $0 \leq k < j$, $w^{k..} \models \phi$;*
- *$w \models r \diamondsuit\!\!\rightarrow \phi$ iff, for some $j \geq 0$, $w^{0..j} \models r$ and $w^{j..} \models \phi$;*
- *$w \models r \mapsto \phi$ iff, for all $j \geq 0$, if $w^{0..j} \models r$, then $w^{j..} \models \phi$.*

Notice that we can build Boolean expressions by means of atomic formulas and Boolean connectives. The language of a PSL formula ϕ over the alphabet Σ is defined as the set $\mathcal{L}(\phi) := \{w \in \Sigma^\omega \mid w \models \phi\}$.

Example 1. Consider the PSL formula $\mathbf{G}\,(\{\{a \,;\, b[*] \,;\, c\} \,\&\&\, \{d[*] \,;\, e\}\} \mapsto \{f \,;\, g\})$. It encodes the property for which every sequence that matches both regular expressions $\{a \,;\, b[*] \,;\, c\}$ and $\{d[*] \,;\, e\}$ must be followed by $\{f \,;\, g\}$.

3 From PSL to NBA: Previous Approaches

In this section, we overview recent approaches to dealing with PSL verification. In the *monolithic* approaches, the first step is the conversion from PSL in a monolithic alternating Büchi automaton; during the conversion, semantic simplification steps (such as the elimination of unreachable states, and restricted forms of minimization by observational equivalence) are applied. The ABA is then converted into a symbolically represented NBA. In [5], this is done by means of a symbolic encoding of MH, and can be applied both to BDD-based and SAT-based verification. In [4], an encoding of the ABA that is specialized for bounded model checking is proposed.

The conversion proposed in [6] is based on the so called Suffix Operator Normal Form (SONF). The idea is to partition the translation, by first converting a PSL formula ϕ into an equisatisfiable formula in SONF, structured as follows

$$\overbrace{\bigwedge_i \phi_i}^{\Psi_{LTL}} \wedge \overbrace{\bigwedge_j \mathbf{G}\,(p_I^j \rightarrow (r_j \star\!\!\rightarrow p_F^j))}^{\Psi_{PSL}}$$

where ϕ_i are LTL formulas, r_j are SEREs, p_I^j and p_F^j are propositional atoms, and $\star\!\!\rightarrow$ is either $\mid\!\!\rightarrow$ or $\Diamond\!\!\rightarrow$. Formulae of the form $\mathbf{G}\,(p_I^j \rightarrow (r_j \star\!\!\rightarrow p_F^j)$ are called *Suffix Operator Subformulas* (SOS's).

The translation first converts the formula in NNF, and then "lifts out" the occurrences of suffix operators, by introducing fresh variables (intuitively, the p^j in the formula above), together with the corresponding SOS. For lack of space, we omit the details regarding the conversion of SOS into NBA. We only mention that the translation is specialized to exploit the structure of SOS (see [6] for details).

Example 2. The SONF of the PSL formula of Example 1 is $\mathbf{G}\,p_1 \wedge \mathbf{G}\,(p_1 \rightarrow \{\{a\,;\,b[*];\,c\}\ \&\&\ \{d[*];\,e\}\} \mid\!\!\rightarrow p_2) \wedge \mathbf{G}\,(p_2 \rightarrow \{f\,;\,g\} \Diamond\!\!\rightarrow p_3)$.

In [6], a substantial experimental evaluation is carried out, both on PSL satisfiability problems (denoted with LE for language emptiness) and on Model Checking (MC) problems. The modular approach results in dramatic improvements in PSL compilation time. However, on those problems where the ABA library is able to build an automaton within the time limit, the search time is typically in favor of the monolithic approach. This is mainly due to the fact that in certain examples the semantic simplifications are extremely effective. We notice that, the loss of efficiency in search is often compensated by the much faster compilation; yet, in the rest of the paper we show how to enhance our approach even further, by proposing a similar simplification mechanism.

4 Syntactic Optimizations for PSL

In this section, we describe an optimized approach, which extends the SONF-based conversion with the integration of the following simplifications. Before the SONF conversion, we apply two steps: (*i*) we simplify the SEREs in order to reduce the subformulas in the scope of SERE conjunction operators; (*ii*) we simplify occurrences of

$$
\begin{array}{rcl}
r \;\&\&\; (r_1 \mid r_2) & \Rightarrow & (r \;\&\&\; r_1) \mid (r \;\&\&\; r_2) \\
b_1 \;\&\&\; b_2 & \Rightarrow & b_1 \wedge b_2 \\
b \;\&\&\; \{r_1 \;\&\&\; r_2\} & \Rightarrow & \{b \;\&\&\; r_1\} \;\&\&\; r_2 \\
b \;\&\&\; \{r_1 \,;\, r_2\} & \Rightarrow & \begin{cases} \mathit{False} \text{ if } \varepsilon \notin \mathcal{L}(r_1), \varepsilon \notin \mathcal{L}(r_2) \\ b \;\&\&\; r_1 \text{ if } \varepsilon \notin \mathcal{L}(r_1), \varepsilon \in \mathcal{L}(r_2) \\ b \;\&\&\; r_2 \text{ if } \varepsilon \in \mathcal{L}(r_1), \varepsilon \notin \mathcal{L}(r_2) \\ b \;\&\&\; r_1 \mid b \;\&\&\; r_2 \text{ otherwise} \end{cases} \\
b \;\&\&\; \{r_1 : r_2\} & \Rightarrow & \{b \;\&\&\; r_1\} \;\&\&\; r_2 \\
b \;\&\&\; r[*] & \Rightarrow & b \;\&\&\; r \\
b[*] \;\&\&\; \{r_1 \,;\, r_2\} & \Rightarrow & \{b[*] \;\&\&\; r_1\} \,;\, \{b[*] \;\&\&\; r_2\} \\
b[*] \;\&\&\; \{r_1 : r_2\} & \Rightarrow & \{b[*] \;\&\&\; r_1\} : \{b[*] \;\&\&\; r_2\} \\
b[*] \;\&\&\; r[*] & \Rightarrow & \{b[*] \;\&\&\; r\}[*] \\
\{b_1 \,;\, r_1\} \;\&\&\; \{b_2 \,;\, r_2\} & \Rightarrow & \{b_1 \wedge b_2\} \,;\, \{r_1 \;\&\&\; r_2\} \\
\{b_1 : r_1\} \;\&\&\; \{b_2 : r_2\} & \Rightarrow & \{b_1 \wedge b_2\} : \{r_1 \;\&\&\; r_2\} \\
\{r_1 \,;\, b_1\} \;\&\&\; \{r_2 \,;\, b_2\} & \Rightarrow & \{r_1 \;\&\&\; r_2\} \,;\, \{b_1 \wedge b_2\} \\
\{r_1 : b_1\} \;\&\&\; \{r_2 : b_2\} & \Rightarrow & \{r_1 \;\&\&\; r_2\} : \{b_1 \wedge b_2\} \\
\{b_1[*] \,;\, r_1\} \;\&\&\; \{b_2 \,;\, r_2\} & \Rightarrow & \\
\multicolumn{3}{r}{\{r_1 \;\&\&\; \{b_2 \,;\, r_2\}\} \mid \{\{b_1 \wedge b_2\} \,;\, \{\{b_1[*] \,;\, r_1\} \;\&\&\; r_2\}\}} \\
\{b_1[*] \,;\, r_1\} \;\&\&\; \{b_2[*] \,;\, r_2\} & \Rightarrow & \\
\multicolumn{3}{r}{\{b_1 \wedge b_2\}[*] \,;\, \{\{r_1 \;\&\&\; \{b_2[*] \,;\, r_2\}\} \mid \{\{b_1[*] \,;\, r_1\} \;\&\&\; r_2\}\}} \\
\multicolumn{3}{r}{r_1[*] \;\&\&\; r_2[*] \Rightarrow^{*} \{r_1[*n_2] \;\&\&\; r_2[*n_1]\}[*]}
\end{array}
$$

*) where r_1 and r_2 have "fixed length", n_1 and n_2 are the least integers such that $n = (|r_1| \cdot n_2) = (|r_2| \cdot n_1)$.

Fig. 1. Rules for **&&**

suffix operators by converting as much as possible the regexps to which they are applied to into LTL. Then, after the conversion in SONF, we apply two other steps: (*iii*) we simplify the Suffix Operator Subformulas by means of rules that strengthen the ones in (*ii*) by exploiting the specific structure of SOSs; (*iv*) the LTL component is rewritten in order to minimize the overall automaton and reduce the number of resulting fairness constraints. In the rest of this section we describe the first three sets of rewriting rules, which regard SEREs and PSL formulas and are an original contributions of this paper. For lack of space, we do not report a detailed description of the LTL simplification rules, which follow [13,14].

In the following, we write $b, b_1, b_2, \ldots$ for boolean formulas, and $r, r_1, r_2, \ldots$ for SEREs. We notice that we can check if the empty word ε belongs to the language of r by parsing: if $r = \mathbf{[*0]}$, then *True*; if $r = b$, then *False*; if $r = r_1 \,;\, r_2$, then *True* if both r_1 and r_1 accept ε, *False* otherwise; if $r = r_1 : r_2$, then *False*; if $r = r_1\mathbf{[*]}$, then *True*; if $r = r_1$ **&&** r_2 or $r = r_1$ **&** r_2, then *True* if both r_1 and r_2 accept ε, *False* otherwise; if $r = r_1 \mid r_2$, then *True* if either r_1 or r_2 accept ε, *False* otherwise.

(i) Simplifying Regular Expressions. Step (*i*) of our simplification flow is implemented by the rules of Figures 1 and 2. For lack of space, Figure 2 only contains some of the rules for **&** ; other rules based on the commutativity and associativity of the operators are also omitted.

$$
\begin{array}{rl}
r \,\&\, (r_1 \mid r_2) & \Rightarrow (r \,\&\, r_1) \mid (r \,\&\, r_2) \\
b_1 \,\&\, b_2 & \Rightarrow b_1 \wedge b_2 \\
b \,\&\, \{r_1 \,\&\, r_2\} & \Rightarrow \{b \,\&\, r_1\} \,\&\, r_2 \\
b \,\&\, \{r_1 \,;\, r_2\} & \Rightarrow \begin{cases} b : \{r_1 \,;\, r_2\} & \text{if } \varepsilon \notin \mathcal{L}(r_1), \varepsilon \notin \mathcal{L}(r_2) \\ b : \{r_1 \,;\, r_2\} \mid b \,\&\&\, r_1 & \text{if } \varepsilon \notin \mathcal{L}(r_1), \varepsilon \in \mathcal{L}(r_2) \\ b : \{r_1 \,;\, r_2\} \mid b \,\&\&\, r_2 & \text{if } \varepsilon \in \mathcal{L}(r_1), \varepsilon \notin \mathcal{L}(r_2) \\ b : \{r_1 \,;\, r_2\} \mid b & \text{otherwise} \end{cases} \\
b \,\&\, \{r_1 : r_2\} & \Rightarrow b : \{r_1 : r_2\} \\
b \,\&\, r[*] & \Rightarrow b \mid \{b : r[*]\} \\
b[*] \,\&\, r & \Rightarrow r \mid \{b[*] \,\&\&\, r\} \,;\, b[*] \\
r_1[*] \,\&\, r_2 & \Rightarrow r_2 \mid r_1[*] \,\&\&\, \{r_2 \,;\, \top[*]\} \\
\{b_1 \,;\, r_1\} \,\&\, \{b_2 \,;\, r_2\} & \Rightarrow \{b_1 \wedge b_2\} \,;\, \{r_1 \,\&\, r_2\} \\
\{b_1 : r_1\} \,\&\, \{b_2 : r_2\} & \Rightarrow \{b_1 \wedge b_2\} : \{r_1 \,\&\, r_2\} \\
\{r_1 \,;\, b_1\} \,\&\, \{r_2 \,;\, b_2\} & \Rightarrow \{r_1 \,\&\, r_2\} \,;\, \{b_1 \wedge b_2\} \\
\{r_1 : b_1\} \,\&\, \{r_2 : b_2\} & \Rightarrow \{r_1 \,\&\, r_2\} : \{b_1 \wedge b_2\} \\
r_1[*] \,\&\, r_2[*] & \Rightarrow r_1[*] \mid r_2[*]
\end{array}
$$

Fig. 2. Rules for **&**

Example 3. The rewriting rules of Figure 1 apply to the SERE in the PSL formula of Example 1, as follows:

$$
\begin{array}{rl}
\{a \,;\, b[*] \,;\, c\} \,\&\&\, \{d[*] \,;\, e\} & \Rightarrow \{\{a \,;\, b[*]\} \,\&\&\, d[*]\} \,;\, c \wedge e \\
& \Rightarrow \{a \,\&\&\, d[*]\} \,;\, \{b[*] \,\&\&\, d[*]\} \,;\, c \wedge e \\
& \Rightarrow \{a \,\&\&\, d\} \,;\, \{b[*] \,\&\&\, d\}[*] \,;\, c \wedge e \\
& \Rightarrow a \wedge d \,;\, \{b \,\&\&\, d\}[*] \,;\, c \wedge e \\
& \Rightarrow a \wedge d \,;\, \{b \wedge d\}[*] \,;\, c \wedge e.
\end{array}
$$

(ii) Simplifying Suffix Operations In order to reduce a PSL formula to LTL "as much as possible", we define the rules in Figure 3. The rewritings are mostly effective on those expressions where iterations are applied to boolean expressions, as shown in the following example.

Example 4. Consider the formula of Example 1. After applying the rules of Figure 1 as in the Example 3, the formula becomes $\mathbf{G}\,(\{a \wedge d \,;\, \{b \wedge d\}[*] \,;\, c \wedge e\} \vdash\!\!\rightarrow \{f \,;\, g\})$. The rewriting rules of Figure 3 apply as follows:

$$
\begin{array}{l}
\mathbf{G}\,(\{a \wedge d \,;\, \{b \wedge d\}[*] \,;\, c \wedge e\} \vdash\!\!\rightarrow \{f \,;\, g\}) \Rightarrow \\
\mathbf{G}\,((a \wedge d) \rightarrow \{\{b \wedge d\}[*] \,;\, c \wedge e\} \vdash\!\!\rightarrow (f \wedge \mathbf{X}\,\{g\})) \Rightarrow \\
\mathbf{G}\,((a \wedge d) \rightarrow (\neg(b \wedge d)\ \mathbf{R}\ (\{c \wedge e\} \vdash\!\!\rightarrow (f \wedge \mathbf{X}\,g)) \Rightarrow \\
\mathbf{G}\,((a \wedge d) \rightarrow (\neg(b \wedge d)\ \mathbf{R}\ ((c \wedge e) \rightarrow (f \wedge \mathbf{X}\,g)).
\end{array}
$$

(iii) Rewriting Suffix Operator Subformulas After the simplifications described in previous sections, the SONF conversion is carried out [6], so that the occurrences of suffix operators have the fixed structure of SOS, and can be further rewritten. The aim is to

$$
\begin{array}{rll}
(\{\mathbf{[*0]}\} \diamond\!\!\rightarrow \phi) & \Rightarrow & \mathit{False} \\
(\{b\} \diamond\!\!\rightarrow \phi) & \Rightarrow & b \wedge \phi \\
\{r_1 : r_2\} \diamond\!\!\rightarrow \phi & \Rightarrow & \{r_1\} \diamond\!\!\rightarrow (\{r_2\} \diamond\!\!\rightarrow \phi) \\
\{r_1 ; r_2\} \diamond\!\!\rightarrow \phi & \Rightarrow^{*} & \{r_1\} \diamond\!\!\rightarrow \mathbf{X}\,(\{r_2\} \diamond\!\!\rightarrow \phi) \\
(\{r_1 \mid r_2\} \diamond\!\!\rightarrow \phi) & \Rightarrow & (\{r_1\} \diamond\!\!\rightarrow \phi) \vee (\{r_2\} \diamond\!\!\rightarrow \phi) \\
(\{r ; b\mathbf{[*]}\} \diamond\!\!\rightarrow \phi) & \Rightarrow^{**} & \{r\} \diamond\!\!\rightarrow ((\mathbf{X}\, b)\ \mathbf{U}\ \phi) \\
(\{b\mathbf{[*]} ; r\} \diamond\!\!\rightarrow \phi) & \Rightarrow^{**} & b\ \mathbf{U}\ (\{r\} \diamond\!\!\rightarrow \phi) \\
(\{\mathbf{[*0]}\} \vdash\!\!\rightarrow \phi) & \Rightarrow & \mathit{True} \\
(\{b\} \vdash\!\!\rightarrow \phi) & \Rightarrow & b \rightarrow \phi \\
\{r_1 : r_2\} \vdash\!\!\rightarrow \phi & \Rightarrow & \{r_1\} \vdash\!\!\rightarrow (\{r_2\} \vdash\!\!\rightarrow \phi) \\
\{r_1 ; r_2\} \vdash\!\!\rightarrow \phi & \Rightarrow^{*} & \{r_1\} \vdash\!\!\rightarrow \mathbf{X}\,(\{r_2\} \vdash\!\!\rightarrow \phi) \\
(\{r_1 \mid r_2\} \vdash\!\!\rightarrow \phi) & \Rightarrow & (\{r_1\} \vdash\!\!\rightarrow \phi) \wedge (\{r_2\} \vdash\!\!\rightarrow \phi) \\
(\{b\mathbf{[*]} ; r\} \vdash\!\!\rightarrow \phi) & \Rightarrow^{**} & \neg b\ \mathbf{R}\ (\{r\} \vdash\!\!\rightarrow \phi) \\
(\{r ; b\mathbf{[*]}\} \vdash\!\!\rightarrow \phi) & \Rightarrow^{**} & \{r\} \vdash\!\!\rightarrow ((\mathbf{X}\, \neg b)\ \mathbf{R}\ \phi)
\end{array}
$$

*) if $\varepsilon \notin \mathcal{L}(r_1)$ and $\varepsilon \notin \mathcal{L}(r_2)$

**) if $\varepsilon \notin \mathcal{L}(r)$

Fig. 3. Rules for suffix operators

$$
\mathbf{G}\,(P \rightarrow (\{r\mathbf{[*]}\} \vdash\!\!\rightarrow P')) \Rightarrow^{*} \mathbf{G}\,(P \rightarrow (\{r\} \vdash\!\!\rightarrow (P' \wedge \mathbf{X}\, P)))
$$

*) if $\varepsilon \notin \mathcal{L}(r)$

Fig. 4. Rules for SOS

apply the suffix operators to smaller SERE. This way, we partition further the automaton representation, and we enable the sharing of subformulas representations. The rule in Figure 4 push the occurrences of suffix implication inside the SEREs, while keeping the overall formula in SONF. Note that, in general, the transformation is not correct: it preserves the satisfiability only if the global formula is the result of the SONF-ization process described in [6] so that there is a fixed structure for SOS. Unfortunately, no similar transformation is possible for suffix conjunction.

5 Experimental Evaluation

The rewrite rules have been implemented within the NuSMV model checker [15]. We compared their effectiveness with the same experimental setting as [6]. [1] We compare three methods: MONO [5], FMCAD06 [6], and TACAS07 (the method presented in this paper). We preliminarly compare the methods in encoding. We use the test suite of 1000 properties proposed in [6]. The set of properties has been obtained by filling in, with randomly generated SEREs, typical patterns extracted from industrial case studies [10]. Then, we used both Boolean combinations and single and double implications

[1] All the experiments and files to reproduce the experimental analysis described in this paper can be downloaded from the url: `http://sra.itc.it/people/roveri/tacas07/tacas07.tar.gz`.

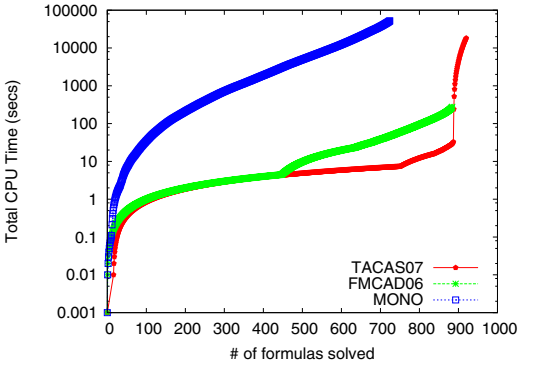

Fig. 5. Problem encoding on 1000 properties

between big conjunctions of typical properties. The latter cases model problems arising in requirements engineering setting, i.e. refinement and equivalence among specifications. For each of the methods (MONO, FMCAD06 and TACAS07), we report the time needed to construct the corresponding representation. All experiments were run on a 3GHz Intel CPU equipped with 4GB of memory running Linux; for each run, we used a timeout of 900 seconds and a memory limit of 1GB. Figure 5 reports the plot of the number of problems generated in a given amount of time (the samples are ordered by increasing computation time). The comparison between FMCAD06 and MONO, just as stated in [6], shows that the monolithic approach has a much harder time than FMCAD06 in completing the generation. The plots also show that the TACAS07 rewriting, in addition to causing negligible overhead in the simple cases, seems to pay off in the harder cases. There are several samples where the construction time is substantially reduced, and (by looking carefully at the data) we see that TACAS07 completes the 884 samples that FMCAD06 can solve one order of magnitude faster; in addition, we see that TACAS07 can solve 36 hard problems where FMCAD06 times out. The speed up typically occurs in examples where SERE automata have to be determinized both in MONO and FMCAD06, while for TACAS07 the rules manage to generate smaller SERE.

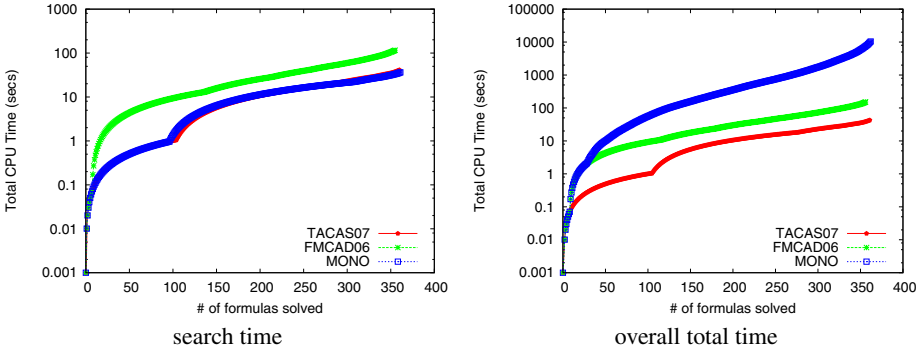

Fig. 6. Language emptiness using SBMC on 400 formulas

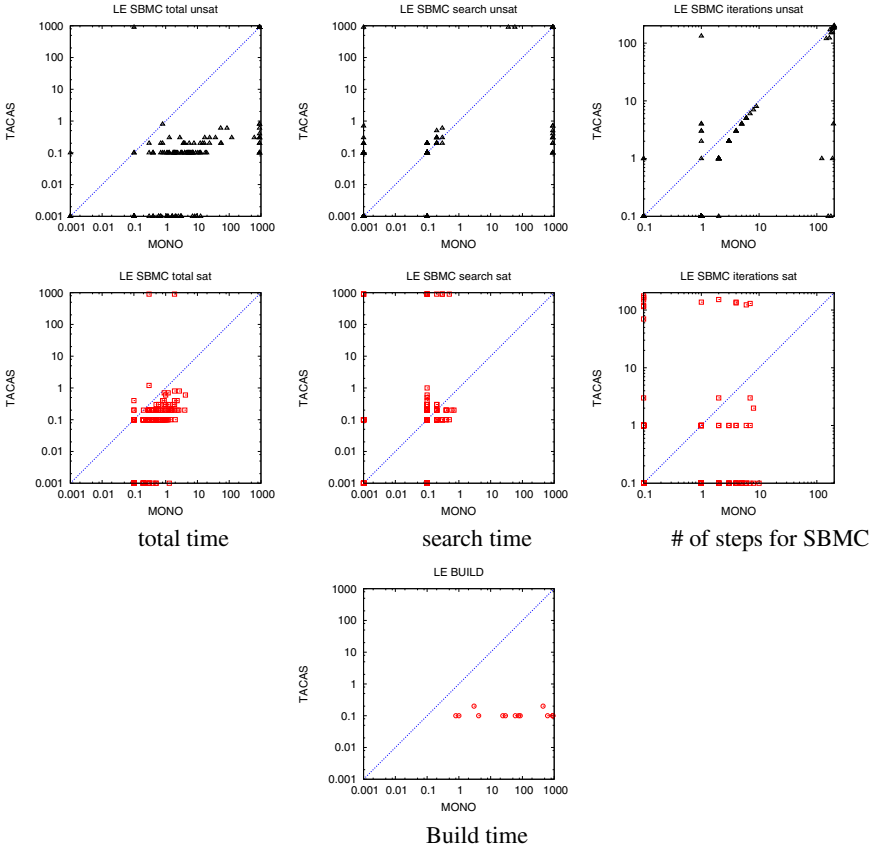

Fig. 7. LE — Experimental evaluation results on 400 formulas: MONO vs TACAS07

We then focus on the effect of the rewriting on the search, by considering, as in [6], a test suite of 400 selected problems for which the ABA library is able to complete the generation within the time limit. The test suite contains two kinds of problems, fair cycle detection (LE, for language emptiness), and model checking (MC). For LE, the problems are a subset of the 1000 problems used to test generation; for MC, the same PSL properties are applied to the Gigamax model taken from the standard NuSMV distribution. For each problem, each method takes in input a PSL formula (and, if MC, a model), and generates a file in NuSMV language, containing an LTL formula and possibly a model. Each file is solved with the SAT-based approach of Simple Bounded Model Checking (SBMC) [16], fixing a maximum length of 200 steps and enabling the check for completeness. For each method we compare solution time, and total time.

The overall results for language emptiness are collected in Figures 6, 7 and 8. Figure 6 plots the number of problem solved in a given amount of time, considering only the search time (on the left) and the search time plus the problem construction time (on

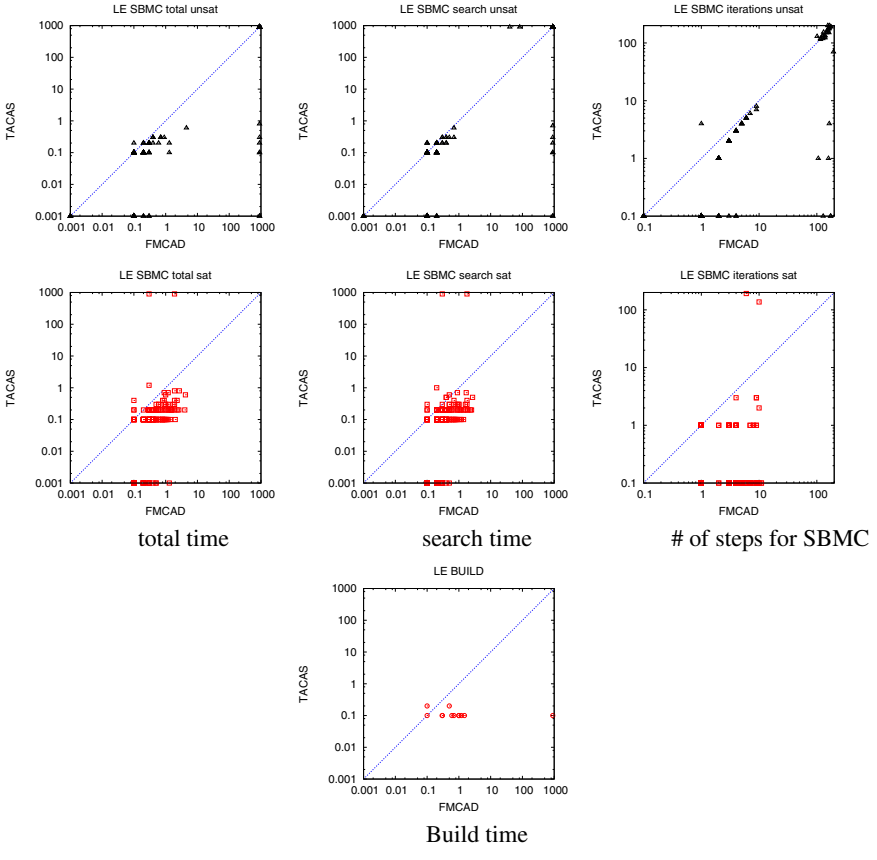

Fig. 8. LE — Experimental evaluation results on 400 formulas: FMCAD06 vs TACAS07

the right). We remark that, TACAS07 plot is under the MONO plot. The plot clearly shows that the search time for MONO and TACAS07 are comparable, i.e. the rewriting proposed in this paper are as effective as the semantic ones of MONO; the improvement with respect to FMCAD06 in terms of search time is also evident. When considering the total time, we notice that these advantages come without paying the price of the semantic simplification. In fact, this price is often so high that also FMCAD06 is superior to MONO. These claims are also confirmed by the scatter plots reported in Figure 7 (comparing MONO with TACAS07) and in Figure 8, where it is clear that TACAS07 is almost uniformly superior to FMCAD06. It is also interesting to notice that while MONO and TACAS07 have overall similar performance, they are not simplifying in the same way, and sometimes the semantic simplifications are unable to achieve as much reduction as rewriting.

The overall results for model checking are collected in Figures 9, 10 and 11. Figure 9 plots the number of model checking problems solved in a given amount of time, considering only the search time (on the left) and the search time plus the problem construction time (on the right). The plot of search time shows that the three methods,

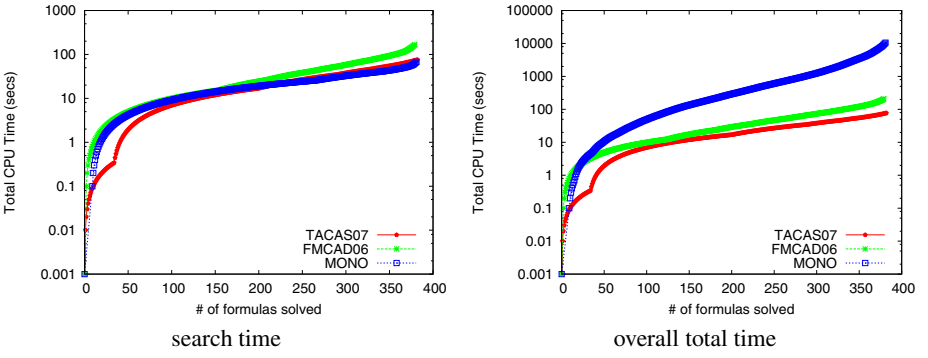

Fig. 9. Model checking using SBMC on 400 formulas

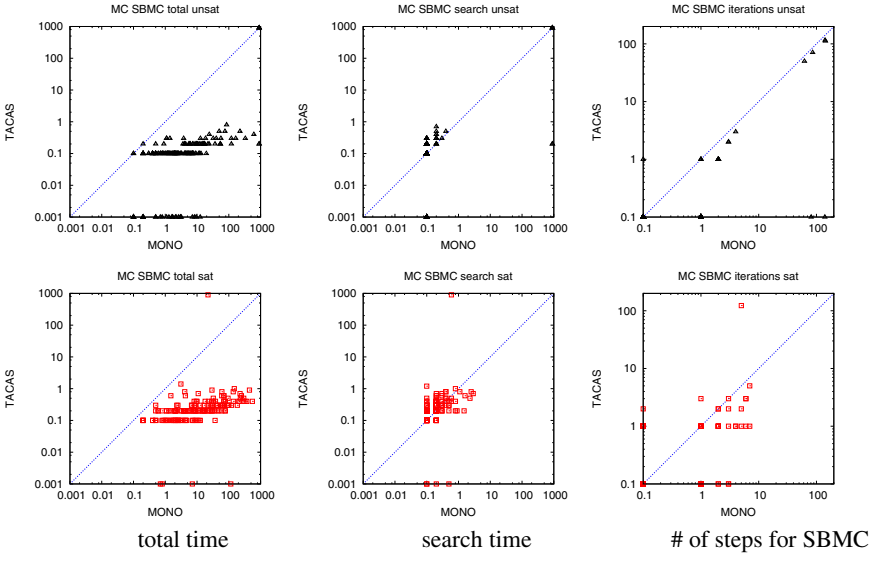

Fig. 10. MC — Experimental evaluation results on 400 formulas: MONO vs TACAS07

while tackling these model checking problems, are almost comparable; this is probably due to the presence of the model that here is predominant. However, if the total time is taken into account, it appears that MONO is outperformed by both FMCAD06 and TACAS07, and that TACAS07 is better than FMCAD06 as in the language emptiness case (again, here the difference is less evident because of the presence of the model). Also in this case the scatter plots in Figure 10 (comparing MONO with TACAS07) and in Figure 11 (comparing FMCAD06 with TACAS07) confirm that for the search TACAS07 is able to achieve substantial simplifications, although not exactly the same as MONO. The cost of semantic simplification is however substantial.

We restricted our experimental evaluation only to SBMC, even though in [6] the same experiments where carried out also using BDDs. Preliminary experiments showed

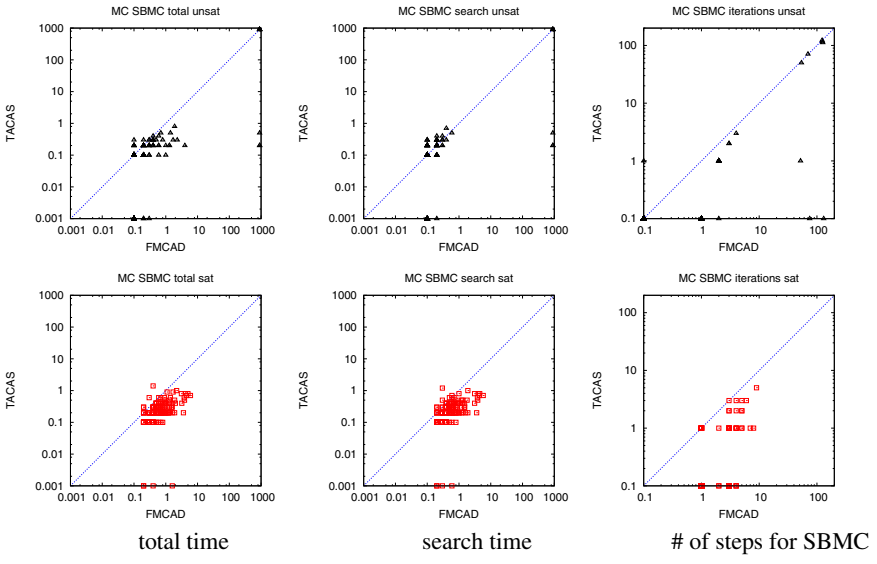

Fig. 11. MC — Experimental evaluation results on 400 formulas: FMCAD06 vs TACAS07

us that FMCAD06 and TACAS07 with the BDD engine are incomparable. The reason is that the optimizations we proposed produce a large number of fairness constraints so that the results are highly influenced by several factors (BDD variable ordering, order in which the fairness conditions are considered, algorithms for language emptiness). A fair comparison requires an improvement of language emptiness with multiple fairness conditions and a deep tuning of possible options. We plan to carry out this analysis later on to better support the new proposed approach.

Another relevant approach is the one by Heljanko et al. [4], in the following referred to as CAV06: basically, it takes in input an ABA and instead of using a symbolic MH for generating an NBA, a partitioning of the ABA is carried out by exploiting the fact that PSL will result in weak ABAs [3]. Given that the approach is substantially different, it would be worth to carry out a comparison with it. Since [4] implements its own format for reading in ABA, and we do not yet have a complete translator available, we leave the comparison to future work. We expect that the results would be biased by the fact that the approach implemented in [4] is not complete, so that we have to disable the completeness check. Since CAV06 must rely on the ABA library of MONO, it is easy to predict that it will inherit the same bottleneck in construction.

6 Conclusions and Future Work

In this paper, we proposed an approach based on syntactic rewriting to improve the verification times for PSL specifications. The approach improves on [6], greatly reducing the redundancies of the generated automata. Although the optimizations have negligible run-times, the benefit in verification and overall time is substantial.

In the future we plan to work on the problem of the analysis of requirements, trying to scale up on large sets of PSL formulas. In particular, we will concentrate on the definition of optimized algorithms for language emptiness, based on the structure of the modular automaton, on the definition of specialized BDD-based language emptiness algorithms. We also plan to investigate rewriting as a tool for better understanding the meaning of specifications.

References

1. IEEE: IEEE standard 1850 – Property Specification Language (PSL) (2005)
2. Pnueli, A.: The temporal logic of programs. In: Proceedings of 18th IEEE Symp. on Foundation of Computer Science. (1977) 46–57
3. Ben-David, S., Bloem, R., Fisman, D., Griesmayer, A., Pill, I., Ruah, S.: Automata Construction Algorithms Optimized for PSL. http://www.prosyd.org (2005) PROSYD deliverable D 3.2/4.
4. Heljanko, K., Junttila, T., Keinänen, M., Lange, M., Latvala, T.: Bounded Model Checking for Weak Alternating Büchi Automata. In: Proc. of the 18th Int. Conf. on Computer Aided Verification, CAV'06. Volume 4144 of LNCS., Seattle (USA) (2006) 95–108
5. Bloem, R., Cimatti, A., Pill, I., Roveri, M., Semprini, S.: Symbolic Implementation of Alternating Automata. In: Proc. of 11^{th} International Conference on Implementation and Application of Automata (CIAA06). Volume 4094 of LNCS. (2006) 208–218
6. Cimatti, A., Roveri, M., Semprini, S., Tonetta, S.: From PSL to NBA: a Modular Symbolic Encoding. In: Procs. of FMCAD06. (2006)
7. Pnueli, A., Zaks, A.: PSL Model Checking and Run-time Verification via Testers. In: Proc. of 14^{th} International Symposium on Formal Methods (FM'06). Volume 4085 of LNCS., Hamilton, Ontario, Canada (2006) 573–586
8. Miyano, S., Hayashi, T.: Alternating finite automata on omega-words. Theor. Comput. Sci. **32** (1984) 321–330
9. Sebastiani, R., Tonetta, S., Vardi, M.: Symbolic Systems, Explicit Properties: On Hybrid Approaches for LTL Symbolic Model Checking. In: Proceedings of the 16th International Conference on Computer-Aided Verification (CAV'05). (2005) 350–363
10. David, S.B., Orni, A.: Property-by-Example guide: a handbook of PSL/Sugar examples. http://www.prosyd.org (2005) PROSYD deliverable D 1.1/3.
11. Beer, I., Ben-David, S., Eisner, C., Fisman, D., Gringauze, A., Rodeh, Y.: The temporal logic sugar. In Berry, G., Comon, H., Finkel, A., eds.: Computer Aided Verification, 13th International Conference (CAV 2001). Volume 2102 of LNCS., Springer (2001) 363–367
12. Armoni, R., Bustan, D., Kupferman, O., Vardi, M.Y.: Resets vs. aborts in linear temporal logic. In: TACAS. (2003)
13. Somenzi, F., Bloem, R.: Efficient Büchi Automata from LTL Formulae. In: Proceedings of the 12th International Conference on Computer-Aided Verification. Volume 1855 of LNCS., Springer-Verlag (2000) 247–263
14. Etessami, K., Holtzmann, G.: Optimizing Büchi Automata. In: Proceedings of CONCUR'2000. Volume 1877 of LNCS. (2000) Springer.
15. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new Symbolic Model Verifier. In: Proc. of the 11th International Conference on Computer-Aided Verification. Volume 1633 of LNCS., Springer-Verlag (1999) 495 – 499
16. Heljanko, K., Junttila, T.A., Latvala, T.: Incremental and complete bounded model checking for full PLTL. In: Proc. of the 17th Int. Conf. on Computer Aided Verification (CAV'05). Volume 3576 of LNCS., Springer (2005) 98–111