On the Merits of Temporal Testers*

A. Pnueli and A. Zaks

New York University, New York,
{amir,zaks}@cs.nyu.edu

Abstract. The paper discusses the merits of *temporal testers*, which can serve as a compositional basis for automata construction corresponding to temporal formulas in the context of LTL, PSL, and MITL logics. Temporal testers can be viewed as (non-deterministic) transducers that, at any point, output a boolean value which is 1 iff the corresponding temporal formula holds starting at the current position.

The main advantage of testers, compared to acceptors (such as Büchi automata) is their compositionality. Namely, a tester for a compound formula can be constructed out of the testers for its sub-formulas. Besides providing the construction of testers for formulas specified in LTL, PSL, and MITL, the paper also presents a general overview of the tester methodology, and highlights some of the unique features and applications of transducers including compositional deductive verification of LTL properties.

1 Introduction

Automata theory plays a central role in formal methods. For example, the classical way of model checking an LTL property φ over a finite-state system S, represented by the automaton M_S, is based on the construction of an ω-automaton $\mathcal{A}_{\neg\varphi}$ that accepts all sequences that violate the property φ. Having both the system and its specification represented by automata, we may form the product automaton $M_S \times \mathcal{A}_{\neg\varphi}$ and check that it accepts the empty language, implying that there exists no computation of S which refutes φ [24]. For the working of this algorithm, it is sufficient that the automaton is a proper recognizer for the language $\mathcal{L}(\neg\varphi)$ specified by the formula $\neg\varphi$. It is no surprise that acceptors such as ω-automata is a formalism widely used by researchers and engineers alike.

However, with the advancements in the field of the formal verification, several drawbacks of acceptors became noticeable. First of all, modern model checkers may expect the automaton to be symbolic (BDD-based). Therefore, if one is to use the standard tableau-based construction, some encoding may be necessary. In addition, new temporal languages such as PSL [1] have been developed to address the need for formalizing more elaborate and intricate specifications. In particular, PSL has several features to support bottom-up construction of complex properties, where lower level properties are composed to construct more complex properties. Acceptors do not fit into this paradigm very well since they do not compose. That is, having constructed automata $\mathcal{A}_\varphi$ and $\mathcal{A}_\psi$

* This research was supported in part by the European community project Prosyd, ONR grant N00014-99-1-0131, and SRC grant 2004-TJ-1256.

O. Grumberg and H. Veith (Eds.): 25MC Festschrift, LNCS 5000, pp. 172–195, 2008.
© Springer-Verlag Berlin Heidelberg 2008

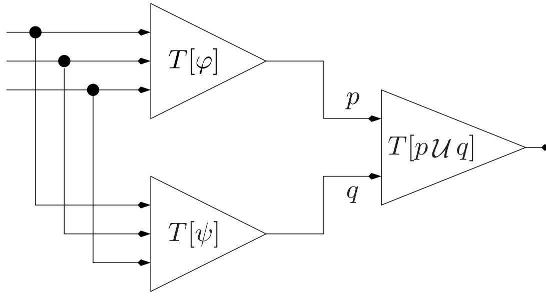


Fig. 1. Composition of transducers to form $T[\varphi\,\mathcal{U}\,\psi]$

for LTL formulas φ and ψ, there is no simple recipe for constructing the automaton for a compound formula that combines φ and ψ, such as $\varphi\mathcal{U}\psi$.

One remedy to this problem is to enhance ω-automata with universal non-determinism (i.e., alternating ω-automata) [6]. In this approach, there are no special requirements on the sub-automata, and any two acceptors can be composed using alternation. An orthogonal solution to the problem is to impose the responsibility of being composable on the sub-automata themselves. In particular, we suggest that an automaton not only tells whether the entire (infinite) input sequence is in the language, but does so for every suffix of the input word. We call such an automaton a *temporal tester*, which has been introduced first in [13]. More formally, a tester for a formula φ can be viewed as a *transducer* that keeps observing a state sequence σ and, at every position $j \geq 0$, outputs a boolean value which equals 1 iff $(\sigma, j) \models \varphi$.

While acceptors, such as the Büchi automata $\mathcal{A}_\varphi$, do not easily compose, temporal testers do. In Fig. 1, we show how transducers for the formulas φ, ψ, and $p\,\mathcal{U}\,q$ can be composed into a transducer for the formula $\varphi\,\mathcal{U}\,\psi$.
Below is a summary of several important features of temporal testers that make them very useful:

- The construction is compositional. Therefore, it is sufficient to specify testers for the basic temporal formulas. In case of LTL, we only need to consider the formulas $X!\,p$[1] and $p\,\mathcal{U}\,q$, where p and q are assertions (state formulas). Testers for more complex formulas can be derived by composition as in Fig. 1.
- The testers for the basic formulas are naturally symbolic. Thus, a general tester, which is a synchronous parallel composition (automata product) of symbolic modules can also be easily represented symbolically. As was shown in [21], the basic processes of model checking and run-time monitoring can be performed directly on the symbolic representation of the testers. There is no need for partial determinization to handle alternation nor conversion from explicit state representation.
- Extensions of an existing logic can be handled by constructing testers only for the newly introduced basic operators. This feature has been utilized to a great advantage when a compositional approach to the construction of transducers

[1] Inspired by the PSL notation, we write $X!\,p$ for "next p".

corresponding to LTL formulas [13] has been extended to handle the logics PSL [21] and MITL [16] which are extensions of LTL.

- In spite of the fact that transducers are more functionally complex than acceptors, the complexity of constructing a transducer (temporal tester) for an arbitrary LTL, PSL, or MITL formula is not worse than that of the lower-functionality acceptor. In its symbolic representation, the size of a tester is linear in the size of the formula. This implies that the worst-case state complexity is exponential for LTL and PSL formulas, which is an established lower bound.

Note that we can always regard a temporal tester as an acceptor. Therefore, it is interesting to compare automata construction using temporal testers to other techniques such as tableau construction for LTL [15] and alternating-automata based construction for PSL [8]. First, we note that the complexity of all of these techniques as well as that of the testers approach equally match the established lower bound. Of course, there is plenty of room for practical considerations and local improvements. Surprisingly, for LTL, a tableau-based approach [15] yields an automaton identical to the one induced by the transducer constructed according to [13]. Similarly, for PSL, the tester construction of [21] induces an acceptor almost identical to the one obtained in [8]. Actually, the two automata become exactly alike after several optimizations are applied to an alternating automata based approach, most of the optimizations become much more obvious once we realize how to build a proper transducer for these operators.

Nevertheless, the testers approach offers a conceptually new methodology, and while similarities are not accidental and rather striking, the differences are equally remarkable. Let us again compare temporal testers to tableau construction and alternation techniques, but now with an emphasis on the process itself rather than on the final result. The main building blocks of tableau construction are the expansion formulas, like $b_1 \mathcal{U} b_2 \iff b_2 \vee (b_1 \wedge X![b_1 \mathcal{U} b_2])$. Such expansion formulas, which exist for all the temporal operators, relate the value of an expression involving the operator at the current position to the values of its arguments in the current and next position and to the value of the expression in the next position. For past operators, the expansion formula relates the value of the expression and its arguments in the current position to their values in the previous position.

When constructing testers for an operator that has an expansion formula (such as all the LTL operators), one uses the expansion formula as the core for the transition relation of the tester. However, when developing testers for more complicated or simply ”unknown” (new) operators, the expansion formula approach may not always be an option. In such cases, one may use the intuition that treats a tester as a non-deterministic guesser, the correctness of whose output needs to be confirmed at a later stage. That was the approach successfully applied for handling PSL and MITL operators. And, while the tester construction for PSL produced expansion formulas as a nice side effect, there is no such result for MITL, where reliance on ”guessing” plays a crucial role. When comparing testers to an alternating automata, the main philosophical distinction is that an alternating automata is less structured than a non-deterministic acceptor, while testers, on the other hand, have more structure than classical acceptors.

The additional support provided by a transducer make them truly plug-and-play objects, which has several important practical implications. The most straightforward

illustration of this phenomenon is application of tester towards CTL* model checking [14]. The paper shows how to reduce CTL* model checking problem to that of CTL . Essentially, each path-quantifier free sub-formula is replaced by the corresponding LTL transducer. We could have performed a similar reduction using acceptors. However, using testers we have a freedom for each such sub-formula to leave the outer-most temporal operator intact and construct the tester for the remaining part. This results in a true CTL* to CTL reduction, where we may still have temporal operators in the final CTL formula. The ability to decompose an LTL formula using testers is also crucial for deductive verification, which we will discuss in a great detail in Section 11.

Another benefit of the plug-and-play nature of testers is the possibility to use different algorithms for different parts of the formula. For example, a user can manually build a highly optimized tester for a sub-formula, and the rest of the formula can be handled automatically. We can also combine testers with other techniques as was done in [7], where PSL operators are handled using the tester approach, but the rest of the formula uses an existing LTL to NBA transformation which, according to the experimental data, results in the fastest available implementation for PSL to NBA conversion.

2 Accellera PSL

In this section we introduce the property specification language PSL [1]. The construction of testers for PSL formulas will be presented in Section 8.

In this paper, we only consider a subset of PSL. For brevity, we omit the discussions of OBE (Optional Branching Extension) formulas that are based on CTL . Note that using testers we can obtain a model checking algorithm even for CTL* branching formulas by combining PSL testers with the work in [14]. In addition, we do not consider clocked formulas and formulas with *abort* operator. This is not a severe limitation since none of the above add any expressive power to PSL. One can find a rewriting scheme for the @ operator (clock operator) in [10] and for the *abort* operator in [22]. The rewriting rules produce a semantically equivalent formula not containing the operators, which is linear in the size of the original formula.

2.1 Syntax

The logic Accellera PSL is defined with respect to a non-empty set of atomic propositions P. Let B be the set of boolean expressions over P. We assume that the expressions $true$ and $false$ belong to B.

Definition 1 (Sequential Extended Regular Expressions (SEREs)) .

- *Every boolean expression $b \in B$ is a SERE.*
- *If r, r_1, and r_2 are SEREs, then the following are SEREs:*
  - $\{r\}$
  - $r_1 \; ; \; r_2$
  - $r_1 : r_2$
  - $r_1 \mid r_2$
  - $[*0]$
  - $r_1 \;\&\&\; r_2$
  - $r[*]$

Definition 2 (Formulas of the Foundation Language (FL formulas)) .

- *If r is a SERE, then both r and $r!$ are FL formulas.*
- *If φ and ψ are FL formulas, r is a SERE, and b is a boolean expression, then the following are FL formulas:*
  - (φ)
  - $\neg\varphi$
  - $\varphi \wedge \psi$
  - $\langle r \rangle \varphi$
  - $X!\varphi$
  - $[\varphi \mathcal{U} \psi]$
  - φ *abort* b
  - $r \mapsto \varphi$

Definition 3 (Accellera PSL Formulas) .

- *Every FL formula is an Accellera* PSL *formula.*

2.2 Semantics

The semantics of FL is defined with respect to finite and infinite words over $\Sigma = 2^P \cup \{\top, \bot\}$. We denote a letter from Σ by l and an empty, finite, or infinite word from Σ by u, v, or w (possibly with subscripts). We denote the length of word v as $|v|$. An empty word $v = \epsilon$ has length 0, a finite word $v = (l_0 l_1 l_2 \ldots l_k)$ has length $k+1$, and an infinite word has length ω. We use i, j, and k to denote non-negative integers. We denote the ith letter of v by v^{i-1} (since counting of letters starts at zero). We denote by $v^{i..}$ the suffix of v starting at v^i. That is, for every $i <|v|$, $v^{i..} = v^i v^{i+1} \cdots v^n$ or $v^{i..} = v^i v^{i+1} \cdots$. We denote by $v^{i..j}$ the finite sequence of letters starting from v^i and ending in v^j. That is, for $j \geq i$, $v^{i..j} = v^i v^{i+1} \cdots v^j$ and for $j < i, v^{i..j} = \epsilon$. We use l^ω to denote an infinite-length word, each letter of which is l.

We use $\bar{v}$ to denote the word obtained by replacing every $\top$ with a $\bot$ and vice versa. We call $\bar{v}$ the *complement* of v.

The semantics of FL *formulas* over *words* is defined inductively, using as the base case the semantics of *boolean expressions* over *letters* in Σ. The semantics of a boolean expression is assumed to be given as a relation $\Vert\!\!\models \subseteq \Sigma \times B$ relating letters in Σ with boolean expressions in B. If $(l, b) \in \Vert\!\!\models$, we say that the letter l satisfies the boolean expression b and denote it by $l \Vert\!\!\models b$. We assume the two special letters $\top$ and $\bot$ behave as follows: for every boolean expression b, $\top \Vert\!\!\models b$ and $\bot \Vert\!\!\not\models b$. We assume that, otherwise, the boolean relation $\Vert\!\!\models$ behaves in the usual manner. In particular, that for every letter $l \in 2^P$, atomic proposition $p \in P$ and boolean expressions $b, b_1, b_2 \in B$, (i) $l \Vert\!\!\models p$ iff $p \in l$, (ii) $l \Vert\!\!\models \neg b$ iff $l \Vert\!\!\not\models b$, and (iii) $l \Vert\!\!\models$ *true* and $l \Vert\!\!\not\models$ *false*. Finally, we assume that for every letter $l \in \Sigma$, $l \Vert\!\!\models b_1 \wedge b_2$ iff $l \Vert\!\!\models b_1$ and $l \Vert\!\!\models b_2$.

Semantics of SEREs. SEREs are defined over finite words from the alphabet Σ. The notation $v \models\!\!\equiv r$, where r is a SERE and v a finite word means that v *tightly models* r. The semantics of unclocked SEREs are defined as follows, where b denotes a boolean expression, and r, r_1, and r_2 denote unclocked SEREs.

- $v \models\!\!\equiv \{r\} \iff v \models\!\!\equiv r$
- $v \models\!\!\equiv b \iff |v| = 1 \wedge v^0 \Vert\!\!\models b$
- $v \models\!\!\equiv r_1 \,;\, r_2 \iff \exists v_1, v_2$ s.t. $v = v_1 v_2, v_1 \models\!\!\equiv r_1$ and $v_2 \models\!\!\equiv r_2$
- $v \models\!\!\equiv r_1 : r_2 \iff \exists v_1, v_2,$ and l s.t. $v = v_1 l v_2, v_1 l \models\!\!\equiv r_1$ and $l v_2 \models\!\!\equiv r_2$
- $v \models\!\!\equiv r_1 \mid r_2 \iff v \models\!\!\equiv r_1$ or $v \models\!\!\equiv r_2$
- $v \models\!\!\equiv r_1 \,\&\&\, r_2 \iff v \models\!\!\equiv r_1$ and $v \models\!\!\equiv r_2$
- $v \models\!\!\equiv [*0] \iff v = \epsilon$
- $v \models\!\!\equiv r[*] \iff v = \epsilon$ or $\exists v_1, v_2$ s.t. $v_1 \neq \epsilon, v = v_1 v_2$ and $v_1 \models\!\!\equiv r$ and $v_2 \models\!\!\equiv r[*]$

Semantics of FL. Let v be a finite or infinite word, b be a boolean expression, r be a SERE, and φ, ψ be FL formulas. We use $\models$ to define the semantics of FL formulas. If $v \models \varphi$ we say that v models (or satisfies) φ.

- $v \models (\varphi) \iff v \models \varphi$
- $v \models \neg\varphi \iff \bar{v} \not\models \varphi$
- $v \models \varphi \wedge \psi \iff v \models \varphi$ and $v \models \psi$
- $v \models b! \iff |v| > 0$ and $v^0 \models b$
- $v \models b \iff |v| = 0$ or $v^0 \models b$
- $v \models r! \iff \exists j < |v|$ s.t. $v^{0..j} \models r$
- $v \models r \iff \forall j < |v|,\ v^{0..j}\top^\omega \models r!$
- $v \models X!\varphi \iff |v| > 1$ and $v^{1..} \models \varphi$
- $v \models [\varphi \mathcal{U} \psi] \iff \exists k < |v|$ s.t. $v^{k..} \models \psi,$ and $\forall j < k,\ v^{j..} \models \varphi$
- $v \models [\varphi \mathcal{W} \psi] \iff \exists k < |v|$ s.t. $v^{k..} \models \psi,$ and $\forall j < min(k, |v|)\ v^{j..} \models \varphi$
- $v \models \varphi$ *abort* $b \iff v \models \varphi$ or $\exists j < |v|$ s.t. $v^j \models b$ and $v^{0..j-1}\top^\omega \models \varphi$
- $v \models \langle r \rangle \varphi \iff \exists j < |v|$ s.t. $\bar{v}^{0..j} \models r,\ v^{j..} \models \varphi$
- $v \models r \mapsto \varphi \iff \forall j < |v|$ s.t. $\bar{v}^{0..j} \models r,\ v^{j..} \models \varphi$

2.3 Associating a Regular Grammar with a SERE

Following [12], a grammar $\mathcal{G} = \langle \mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S} \rangle$ consists of the following components:

- $\mathcal{V}$: A finite set of *variables*.
- $\mathcal{T}$: A finite set of *terminals*. We assume that $\mathcal{V}$ and $\mathcal{T}$ are disjoint. In our framework, $\mathcal{T}$ consists of boolean expressions and a special terminal ϵ.
- $\mathcal{P}$: A finite set of *productions*. We only consider right-linear grammars, so all productions are of the form $V \rightarrow aW$ or $V \rightarrow a$, where a is a terminal, and V and W are variables.
- $\mathcal{S}$: A special variable called a *start symbol*.

We say that a grammar $\mathcal{G}$ is *associated* with a SERE r if, intuitively, they both define the same language. While this definition is not accurate, we show a precise construction of an associated grammar for a given SERE in Appendix A. For example, we associate the following grammar $\mathcal{G}$ with SERE $r = (a_1 b_1)[*]\ \&\&\ (a_2 b_2)[*]$

$$\begin{aligned} V_1 &\rightarrow \epsilon \mid (a_1 \wedge a_2) V_2 \\ V_2 &\rightarrow \quad\ (b_1 \wedge b_2) V_1 \end{aligned}$$

Theorem 1. *For every SERE r of length n, there exists an associated grammar $\mathcal{G}$ with the number of productions $O(2^n)$. If we restrict SERE's to the three traditional operators: concatenation (;), union (|), and Kleene closure ($[*]$), the number of productions becomes linear in the size of r.*

3 Signals, Their Temporal Logic and Timed Automata

In this section we presented the real-time logic MITL, for which we will present testers in Section 9. Most of the material in this section and in Section 9 is taken from [16].

3.1 Signals

Two basic semantic domains can be used to describe timed behaviors. *Time-event sequences* consist of instantaneous events separated by time durations while discrete-valued *signals* are functions from time to some discrete domain. The reader may consult the introduction to [5] for more details on the algebraic characterization of these domains. In this work we use Boolean signals as the semantic domain, which is the natural choice for MITL.

Let the time domain $\mathbb{T}$ be the set $\mathbb{R}_{\geq 0}$ of non-negative real numbers. A Boolean signal is a function $\xi : \mathbb{T} \rightarrow \mathbb{B}^n$. We use $\xi[t]$ for the value of the signal at time t and the notation $\sigma_1^{t_1} \cdot \sigma_2^{t_2} \cdots$ for a signal whose value is σ_1 at the interval $[0, t_1)$, σ_2 in the interval $[t_1, t_1 + t_2)$, etc. For the sake of simplicity we restrict ourselves to such left-closed right-open signal segments and to timed modalities that use only closed intervals. As a consequence we prohibit signal with punctual intervals which are meaningless in the algebraic definition of signals in [5].

3.2 Real-Time Temporal Logic

The syntax of MITL is defined by the grammar

$$\varphi := p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathcal{U}_{[a,b]} \varphi_2 \mid \varphi_1 \mathcal{U} \varphi_2$$

where p belongs to a set $P = \{p_1, \ldots, p_n\}$ of propositions and $b > a \geq 0$ are rational numbers (in fact, it is sufficient to consider integer constants). From the basic MITL operators one can derive other standard Boolean and temporal operators, in particular the time-constrained *eventually* and *always* operators:

$$\Diamond_{[a,b]} \varphi = \mathrm{T}\, \mathcal{U}_{[a,b]} \varphi \quad \text{and} \quad \Box_{[a,b]} \varphi = \neg \Diamond_{[a,b]} \neg\varphi$$

We interpret $\text{MITL}_{[a,b]}$ over n-dimensional Boolean signals and define the satisfiability relation similarly to LTL.

$$\begin{array}{ll}
(\xi, t) \models p & \leftrightarrow p[t] = \mathrm{T} \\
(\xi, t) \models \neg\varphi & \leftrightarrow (\xi, t) \not\models \varphi \\
(\xi, t) \models \varphi_1 \vee \varphi_2 & \leftrightarrow (\xi, t) \models \varphi_1 \text{ or } (\xi, t) \models \varphi_2 \\
(\xi, t) \models \varphi_1 \mathcal{U} \varphi_2 & \leftrightarrow \exists t' \geq t\ (\xi, t') \models \varphi_2 \text{ and } \forall t'' \in [t, t'], (\xi, t'') \models \varphi_1 \\
(\xi, t) \models \varphi_1 \mathcal{U}_{[a,b]} \varphi_2 & \leftrightarrow \exists t' \in [t + a, t + b]\ (\xi, t') \models \varphi_2 \text{ and } \forall t'' \in [t, t'], (\xi, t'') \models \varphi_1
\end{array}$$

Note that our definition of the semantics of the time-bounded $until$ operator differs slightly from definition in [3] which requires φ_1 to hold in the open interval (t', t). Hence our definition can be expressed in their terms as $\varphi_1 \wedge (\varphi_1 \mathcal{U}_{[a,b]} (\varphi_1 \wedge \varphi_2))$. A signal ξ satisfies the formula φ iff $(\xi, 0) \models \varphi$.

3.3 Timed Automata

We use a variant of timed automata which differs slightly from the classical definitions [2], [23] as it reads multi-dimensional *dense-time* Boolean signals, hence the alphabet

letters are associated with *states* rather than with *transitions*. We also extend the domain of clock values to include the special symbol $\perp$ indicating that the clock is currently *inactive*.[2]

The set of valuations of a set $\mathcal{C} = \{c_1, \ldots, c_n\}$ of clock variables, each denoted as $v = (v_1, \ldots, v_n)$, defines the clock space $\mathcal{H} = (\mathbb{R}_{\geq 0} \cup \{\perp\})^n$. A *configuration* of a timed automaton is a pair of the form (q, v) with q being a discrete state. For a clock valuation $v = (v_1, \ldots, v_n)$, $v + t$ is the valuation $(v'_1, \ldots, v'_n)$ such that $v'_i = v_i$ if $v_i = \perp$ and $v'_i = v_i + t$ otherwise. A *clock constraint* is a Boolean combination of conditions of the forms $c \geq d$ or $c > d$ for some integer d.

Definition 1 (Timed Automaton). *A timed automaton over signals is a tuple* $\mathcal{A} = (\Sigma, Q, \mathcal{C}, \lambda, I, \Delta, q_0, F)$ *where* Σ *is the input alphabet (*$\mathbb{B}^n$ *in this paper),* Q *is a finite set of discrete states and* $\mathcal{C}$ *is a set of clock variables. The labeling function* $\lambda : Q \rightarrow 2^{\Sigma}$ *associates a subset of the alphabet to every state while the staying condition (invariant)* I *assigns to every state* q *a subset* I_q *of* $\mathcal{H}$ *defined by a conjunction of inequalities of the form* $x \leq d$*, for some clock* x *and integer* d*. The transition relation* Δ *consists of elements of the form* (q, g, ρ, q') *where* q *and* q' *are discrete states, the transition guard* g *is a subset of* $\mathcal{H}$ *defined by a clock constraint and* ρ *is the update function, a transformation of* $\mathcal{H}$ *defined by a assignments of the form* $c := 0$ *or* $c := \perp$*. Finally* q_0 *is the initial state and* $F \subseteq Q$ *is the acceptance condition.*

The behavior of the automaton as it reads a signal ξ consists of an alternation between time progress periods where the automaton stays in a state q as long as $\xi[t] \in \lambda(q)$ and I_q holds, and discrete instantaneous transitions guarded by clock conditions. Formally, a *step* of the automaton is one of the following:

- A time step: $(q, v) \xrightarrow{\sigma^t} (q, v + t)$, $t \in \mathbb{R}_+$ such that $\sigma \in \lambda(q)$ and $v + t$ satisfies I_q (due to the structure of I_q this holds as well for every t', $0 \leq t' < t$).
- A discrete step: $(q, v) \xrightarrow{\delta} (q', v')$, for some transition $\delta = (q, g, \rho, q') \in \Delta$, such that v satisfies g and $v' = \rho(v)$

A *run* of the automaton starting from a configuration (q_0, v_0) is a finite or infinite sequence of alternating time and discrete steps of the form

$$\xi : \quad (q_0, v_0) \xrightarrow{\sigma_1^{t_1}} (q_0, v_0 + t_1) \xrightarrow{\delta_1} (q_1, v_1) \xrightarrow{\sigma_2^{t_2}} (q_1, v_1 + t_2) \xrightarrow{\delta_2} \cdots,$$

such the $\sum t_i$ diverges. A run is accepting if the set of time instants in which it visits states in F is unbounded. The signal carried by the run is $\sigma_1^{t_1} \cdot \sigma_2^{t_2} \cdots$ The language of the automaton consists of all signals carried by accepting runs.

4 Computational Model

In this section we present the computational model for describing software and hardware systems whose properties we wish to verify.

[2] This is a syntactic sugar since clock inactivity in a state can be encoded implicitly by the fact that in all paths emanating from the state, the clock is reset to zero before being tested [9].

4.1 Fair Discrete Systems with Finite Computations

As our computational model We take a *just discrete system* (JDS), which is a variant of *fair transition system* [19], and is a weaker version of the more general *fair discrete system* considered in [13]. Under this model, a system $\mathcal{D} : \langle V, \Theta, R, \mathcal{J}, F \rangle$ consists of the following components:

- V: A finite set of *system variables*. A *state* of the system $\mathcal{D}$ provides a type-consistent interpretation of the system variables V. For a state s and a system variable $v \in V$, we denote the value assigned to v by the state s by $s[v]$.
- Θ: The *initial condition*. This is an assertion (state formula) characterizing the initial states. A state is defined to be *initial* if it satisfies Θ.
- $R(V, V')$: The *transition relation*, which is an assertion that relates the values of the variables in V interpreted by a state s to the values of the variables V' in an R*-successor* state s'.
- $\mathcal{J}$: A set of *justice* (*weak fairness*) requirements. Each justice requirement is an assertion. An infinite computation must include infinitely many states satisfying the assertion.
- F: The *termination condition*, which is an assertion specifying the set of *final* states. Each finite computation must end in a final state.

A *computation* of a JDS $\mathcal{D}$ is a non-empty sequence of states $\sigma : s_0, s_1, s_2, ...$, satisfying the requirements:

- *Initiality*: s_0 is initial.
- *Consecution*: For each $i \in [0, |\sigma|)$, the state s_{i+1} is a R-successor of state s_i. That is, $\langle s_i, s_{i+1} \rangle \in R(V, V')$ where, for each $v \in V$, we interpret v as $s_i[v]$ and v' as $s_{i+1}[v]$.
- *Justice*: If σ is infinite, then for every $J \in \mathcal{J}$, σ contains infinitely many occurrences of J-states.
- *Termination*: If $\sigma = s_0, s_1, s_2, ..., s_k$ (i.e., σ is finite), then s_k must satisfy F.

Given two JDS's, $\mathcal{D}_1$ and $\mathcal{D}_2$, their *synchronous parallel composition*, $\mathcal{D}_1 \,|||\, \mathcal{D}_2$, is the JDS whose sets of variables and justice requirements are the unions of the corresponding sets in the two systems, whose initial and termination conditions are the conjunctions of the corresponding assertions, and whose transition relation is defined as the conjunction of the two transition relations. Thus, a step in an execution of the composed system is a joint step of the systems $\mathcal{D}_1$ and $\mathcal{D}_2$.

4.2 Interpretation of PSL Formulas over a JDS

We assume that the set of atomic propositions P is a subset of the variables V, so we can easily evaluate all the propositions at a given state of a JDS. We say that a letter $l \in 2^P$ *corresponds* to a state s if $p \in l$ iff $s[p] = 1$. Similarly, we define a correspondence between words and computations. We say, that a computation σ models (or satisfies) PSL formula φ, denoted $\sigma \models \varphi$, if the corresponding word v satisfies PSL formula φ.

5 Temporal Testers

One of the main problems in constructing a Büchi automaton for a PSL formula (or for that matter any ω-regular language) is that the conventional construction is not compositional. In particular, given Büchi automata $\mathcal{A}_\varphi$ and $\mathcal{A}_\psi$ for formulas φ and ψ, it is not trivial to build an automaton for $\varphi \mathcal{U} \psi$. Compositionality is an important consideration, especially in the context of PSL. It is expected that specifications are written in a modular way, and PSL has several language constructs to facilitate that. For example, any property can be given a name, and a more complex property can be built by simply using a named sub-property instead of an atomic proposition.

One way to achieve compositionality with Büchi automata is to use alternation [6]. Nothing special is required from the Büchi automata to be composed in such manner, but the presence of universal branching in the resulting automaton is undesirable. Though most model checkers can deal with existential non-determinism directly and efficiently, universal branching is usually preprocessed at exponential cost.

Our approach is based on the observation that while there is very little room to maneuver during the merging step of two Büchi automata, the construction process of the sub-components is wide open for a change. In particular, we suggest that each sub-component assumes the responsibility of being easily composed with other parts. The hope is that, by requiring that individual parts be more structured than the traditional Büchi automata, we can significantly simplify the composition process.

Recall that the main property of Büchi automata (as well as any other acceptor) is to correctly identify a language membership of a given sequence of letters, starting from the very first letter. It turns out that for composition it is also very useful to know whether a word is in the language starting from an arbitrary position i. We refer to this new class of objects as *testers*. Essentially, testers are transducers that at each step output whether the suffix of the input sequence is in the language. Of course, the suffix is not known by the time the decision has to be made, so the testers are inherently non-deterministic.

Formally, a *full tester* for a formula φ is a JDS T_φ, which has a distinguished boolean variable x_φ, such that:

- **Soundness:** For every computation $\sigma : s_0, s_1, s_2, ...$ of T_φ , $s_i[x_\varphi] = 1$ iff $(\sigma, i) \models \varphi$
- **Completeness:** For every sequence of states $\sigma : s_0, s_1, s_2, ...$, there is a corresponding computation of T_φ $\sigma' : s'_0, s'_1, s'_2, ...$ such that for each i, s_i and s'_i agree on the interpretation of φ-variables.

Intuitively, the second condition requires that a tester must be able to correctly interpret x_φ for an arbitrary input sequence. Otherwise, the first condition can be trivially satisfied by a JDS that has no computations.

5.1 Positive and Negative Testers

For many applications, such as model checking, a full tester can be too powerful. Indeed, everywhere where an acceptor suffices, we can use a full tester, but in such case

we are really only interested in the very first output value and only when the value is $true$. While we still need intermediate output values for compositionality, we can relax the soundness condition to concentrate on the positive values of x_φ. Another way to look at the problem is the fact that a full tester for a formula φ not only implicitly defines an acceptor for φ itself, but also for $\neg\varphi$. An undesirable consequence of this fact is that a full tester for a safety property such as $\Box p$ will have a non trivial justice requirement since it is also a full tester for $\Diamond \neg p$. To address this issue, we define positive and negative testers. Formally, a *positive tester* for a formula φ is a JDS T^+_φ, which has a distinguished boolean variable x_φ, such that:

- **Soundness:** For every computation $\sigma : s_0, s_1, s_2, ...$ of T^+_φ, if $s_i[x_\varphi] = 1$ then $(\sigma, i) \models \varphi$
- **Completeness:** For every sequence of states $\sigma : s_0, s_1, s_2, ...$, there exists a corresponding T^+_φ-computation $\sigma' : s'_0, s'_1, s'_2, ...$ such that for each i, s_i and s'_i agree on the interpretation of φ-variables, and $s_i[x_\varphi] = 1$ iff $(\sigma, i) \models \varphi$.

The definition of a *negative tester* is fully analogous.

Theorem 1. *A full tester is also a proper positive tester and a negative tester.*

Theorem 2. *If T^+_φ a positive tester and T^-_φ is a negative tester for a formula φ that may only share φ-variables, then a full tester can be defined as the composition $T^+_\varphi \parallel\!| T^-_\varphi$, whose transition relation is augmented with the following conjunct that defines the output variable x_φ:*

$$(x^+_\varphi \rightarrow x_\varphi) \wedge (x_\varphi \rightarrow x^-_\varphi)$$

From now on, we may refer to a full tester as simply a tester.

6 LTL Testers

We continue the presentation of testers by considering the two basic LTL operators $X!$ (next) and $\mathcal{U}$ (until), which are also part of the PSL logic (being an extension of LTL). First, we show how to build testers for the two *basic formulas* $X!\,b$ and $b_1 \,\mathcal{U}\, b_2$, where b, b_1, and b_2 are boolean formulas. Then, we demonstrate the compositionality of the testers by easily extending the construction to cover full LTL. Note that our construction for LTL operators is very similar to the one presented in [13].

6.1 A Tester for $\varphi = X!\,b$

Let $T_\varphi = \langle V_\varphi, \Theta_\varphi, R_\varphi, \mathcal{J}_\varphi, F_\varphi \rangle$ be the tester we wish to construct. The components of T_φ are defined as follows:

$$T(X!\,b) : \begin{cases} V_\varphi : \mathit{Vars}(b) \cup \{x_\varphi\} \\ \Theta_\varphi : 1 \\ R_\varphi(V, V') : x_\varphi = b' \\ \mathcal{J}_\varphi : \emptyset \\ F_\varphi : \neg x_\varphi \end{cases}$$

The set *Vars*(b) contains all the propositions on which the boolean formula b depends.

It almost immediately follows from the construction that $T(X!\,b)$ is indeed a good tester for $X!\,b$. The soundness of the $T(X!\,b)$ is guaranteed by the transition relation with the exception that we still have a freedom to incorrectly interpret x_φ at the very last state. This case is handled separately by insisting that every final state must interpret x_φ as false. The completeness follows from the fact that we do not restrict the *Vars*(b) variables in any way by the transition relation, and we can always interpret x_φ properly, by either matching b' or setting it to false in the last state.

6.2 A Tester for $\varphi = b_1 \mathcal{U}\, b_2$

The components of T_φ are defined as follows:

$$T(b_1 \mathcal{U}\, b_2) : \begin{cases} V_\varphi : \mathit{Vars}(b_1, b_2) \cup \{x_\varphi\} \\ \Theta_\varphi : 1 \\ R_\varphi(V, V') : x_\varphi = [b_2 \vee (b_1 \wedge x'_\varphi)] \\ \mathcal{J}_\varphi : \neg x_\varphi \vee b_2 \\ F_\varphi : x_\varphi \leftrightarrow b_2 \end{cases}$$

Unlike the previous tester, $T(b_1 \mathcal{U}\, b_2)$ has a non-empty justice set. A technical reason is that the transition relation allows x_φ to be continuously set to true without having a single state that actually satisfies b_2. The situation is ruled out by the justice requirement. Another way to look at the problem is that R_φ represents an expansion formula for the $\mathcal{U}$ (until) operator, namely $b_1 \mathcal{U}\, b_2 \leftrightarrow b_2 \vee (b_1 \wedge X![b_1 \mathcal{U}\, b_2])$. In general, starting with an expansion formula is a good first step when building a tester. However, the expansion formula alone is usually not sufficient for a proper tester. Indeed, consider the operator $\mathcal{W}$ (weak until, unless), which has exactly the same expansion formula, namely $b_1 \mathcal{W}\, b_2 \leftrightarrow b_2 \vee (b_1 \wedge X![b_1 \mathcal{W}\, b_2])$. We use justice to differentiate between the two operators. Note that the justice is only needed to confirm true output values. Therefore, a negative tester T^-_φ for $\varphi = b_1 \mathcal{U}\, b_2$ is simpler (no justice) and can be formally defined as:

$$T^-(b_1 \mathcal{U}\, b_2) : \begin{cases} V_\varphi : \mathit{Vars}(b_1, b_2) \cup \{x_\varphi\} \\ \Theta_\varphi : 1 \\ R_\varphi(V, V') : b_2 \vee (b_1 \wedge x'_\varphi) \rightarrow x_\varphi \\ \mathcal{J}_\varphi : \emptyset \\ F_\varphi : b_2 \rightarrow x_\varphi \end{cases}$$

7 Tester Composition

In Fig. 2, we present a recursive algorithm that builds a tester for an arbitrary LTL formula φ. In Example 1, we illustrate the algorithm by applying the tester construction for the formula $\varphi = \mathit{true}\, \mathcal{U}\big(X![b_1 \mathcal{U}\, b_2] \vee (b_3 \mathcal{U}[b_1 \mathcal{U}\, b_2])\big)$.

- **Base Case**: If φ is a basic formula (i.e., $\varphi = X!\,b$ or $\varphi = b_1\,\mathcal{U}\,b_2$), use construction from Section 6. For the trivial case, when the formula φ does not contain any temporal operators, we can use a tester for $\mathit{false}\,\mathcal{U}\,\varphi$.
- **Induction Step**: Let ψ be an innermost basic sub-formula of φ, then $T_\varphi = T_{\varphi[\psi/x_\psi]} \;|||\; T_\psi$, where $\varphi[\psi/x_\psi]$ denotes the formula φ in which each occurrence of the sub-formula ψ is replaced with x_ψ.

Fig. 2. Tester construction for an arbitrary LTL formula φ

Example 1. *A tester for* $\varphi = \mathit{true}\,\mathcal{U}\big(X![b_1\,\mathcal{U}\,b_2] \vee \neg(b_3\,\mathcal{U}[b_1\,\mathcal{U}\,b_2])\big)$

We start by identifying $b_1\,\mathcal{U}\,b_2$ to be the innermost basic sub-formula and building the corresponding tester, $T_{b_1\,\mathcal{U}\,b_2}$. Assume that z is the output variable of the tester $T_{b_1\,\mathcal{U}\,b_2}$. Let $\alpha = \varphi[b_1\,\mathcal{U}\,b_2/z]$; after the substitution $\alpha = \mathit{true}\,\mathcal{U}\big(X!\,z \vee \neg(b_3\,\mathcal{U}\,z)\big)$. Note that we performed the substitution twice, but there is no need for two testers, which can result in significant savings. We proceed in similar fashion and build two more testers $T_{X!\,z}$ and $T_{b_3\,\mathcal{U}\,z}$ with the output variables x and y. After the substitutions, we obtain $\beta = \mathit{true}\,\mathcal{U}[x \vee \neg y]$. Since $x \vee \neg y$ is just a boolean expression, the formula satisfies the condition of the base case, and we can finish the construction with one more step. The final result can be expressed as:

$$T_\varphi = T_\beta \,|||\, T_{X!\,z} \,|||\, T_{b_3\,\mathcal{U}\,z} \,|||\, T_{b_1\,\mathcal{U}\,b_2}.$$

7.1 Composition Rules for Positive and Negative Testers

Definition 4 (Polarity of a sub-formula ψ) *Given a formula φ, the* polarity *of a sub-formula ψ with respect to φ is positive if the number of negations enclosing ψ in φ is even and negative otherwise.*

To build a positive tester T^+_φ, we optimize the induction step in Fig. 2 as follows:

- If sub-formula ψ has a positive polarity, then $T^+_\varphi = T^+_{\varphi[\psi/x_\psi]} \;|||\; T^+_\psi$
- If sub-formula ψ has a negative polarity, then $T^+_\varphi = T^+_{\varphi[\psi/x_\psi]} \;|||\; T^-_\psi$
- Otherwise, if sub-formula appears with both positive and negative polarity, then $T^+_\varphi = T^+_{\varphi[\psi/x_\psi]} \;|||\; T_\psi$

The algorithm for building a negative tester is fully symmetric. To illustrate this construction consider the formula φ presented in Example 1. A positive tester is given by:

$$T^+_\varphi = T^+_\beta \,|||\, T^+_{X!\,z} \,|||\, T^-_{b_3\,\mathcal{U}\,z} \,|||\, T_{b_1\,\mathcal{U}\,b_2}.$$

A negative tester is given by:

$$T^-_\varphi = T^-_\beta \,|||\, T^-_{X!\,z} \,|||\, T^+_{b_3\,\mathcal{U}\,z} \,|||\, T_{b_1\,\mathcal{U}\,b_2}.$$

Also note that while we assumed that φ is an LTL formula, the algorithms described in this section are applicable for PSL and MITL as well. The only extension necessary is the ability to deal with additional basic formulas.

8 PSL Testers

As we have mentioned before, to handle the full PSL it is enough to handle all the basic PSL formulas. More complicated formulas can be handled via tester composition according to the algorithm in Fig. 2. There are only two additional PSL basic formulas that we need to consider, namely $\varphi = \langle r \rangle b$ and $\varphi = r$, where r is a SERE and b is a boolean expression. All other PSL temporal operators can be expressed using those two and the LTL operators, $X!$ and $\mathcal{U}$. For example, $r! \equiv \langle r \rangle true$, and $r \mapsto b \equiv \neg(\langle r \rangle \neg\varphi)$.

8.1 A Tester for $\varphi = \langle r \rangle b$

Let $T_\varphi = \langle V_\varphi, \Theta_\varphi, R_\varphi, \mathcal{J}_\varphi, F_\varphi \rangle$ be the tester we wish to construct. Assume that x_φ is the output variable. Let $\mathcal{G} = \langle \mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S} \rangle$ be a grammar associated with r. With no loss of generality, we assume that $\mathcal{G}$ has variables $V_1, \ldots, V_n$ with V_1 being the start symbol. In addition, each variable V_i, has derivations of the form:

$$V_i \rightarrow \alpha_1 \mid \cdots \mid \alpha_m \mid \beta_1 V_1 \mid \cdots \mid \beta_n V_n$$

where $\alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_n$ are boolean expressions. The case that variable V_i does not have a particular derivation $V_i \rightarrow \beta_j V_j$ or $V_i \rightarrow \alpha_k$, is covered by having $\beta_j = \mathit{false}$, and similarly, $\alpha_k = \mathit{false}$. Note that by insisting on this specific form, which does not allow ϵ productions, we cannot express whether an empty string is in the language. However since, by definition of the $\langle \cdot \rangle$ operator, a prefix that satisfies r must be non-empty, we do not need to consider this. The tester T_φ is given by:

$$T_\varphi : \begin{cases} V_\varphi : \mathit{Vars}(r, b) \cup \{x_\varphi\} \cup \{X_1, \ldots, X_n, Y_1, \ldots, Y_n\} \\ \Theta_\varphi : 1 \\ R_\varphi : \text{Each derivation } V_i \rightarrow \alpha_1 \mid \cdots \mid \alpha_m \mid \beta_1 V_1 \mid \cdots \mid \beta_n V_n \\ \qquad \text{contributes to } \rho \text{ the conjunct} \\ \qquad X_i = (\alpha_1 \wedge b) \vee \cdots \vee (\alpha_m \wedge b) \vee (\beta_1 \wedge X_1') \vee \cdots \vee (\beta_n \wedge X_n') \\ \qquad \text{and the conjunct} \\ \qquad Y_i \rightarrow (\alpha_1 \wedge b) \vee \cdots \vee (\alpha_m \wedge b) \vee (\beta_1 \wedge Y_1') \vee \cdots \vee (\beta_n \wedge Y_n') \\ \qquad \text{the output variable is constrained by the conjunct} \\ \qquad x_\varphi = X_1 \\ \mathcal{J}_\varphi : \{\neg Y_1 \wedge \cdots \wedge \neg Y_n, \quad X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n\} \\ F_\varphi : \text{Each derivation } V_i \rightarrow \alpha_1 \mid \cdots \mid \alpha_m \mid \beta_1 V_1 \mid \cdots \mid \beta_n V_n \\ \qquad \text{contributes to } F \text{ the conjunct} \\ \qquad X_i = (\alpha_1 \wedge b) \vee \cdots \vee (\alpha_m \wedge b) \end{cases}$$

Example 2. *A Tester for* $\varphi = \langle \{pq\}[*] \rangle b$.

To illustrate the construction, consider the formula $\langle \{pq\}[*] \rangle b$. Following the algorithm from Appendix A and removing ϵ productions, the associated right-linear grammar for the SERE $\{pq\}[*]$ is given by

$$\begin{aligned} V_1 &\rightarrow \quad pV_2 \\ V_2 &\rightarrow q \mid qV_1 \end{aligned}$$

Consequently, a tester for $\langle\{pq\}[*]\rangle b$ is given by

$$T(\langle\{pq\}[*]\rangle b): \begin{cases} V_\varphi : \{p, q, b, x_\varphi\} \cup \{X_1, X_2, Y_1, Y_2\} \\ \Theta_\varphi : 1 \\ R_\varphi(V, V') : \begin{pmatrix} (X_1 = (p \wedge X_2')) & \wedge \\ (X_2 = (q \wedge b) \vee (q \wedge X_1')) & \wedge \\ (Y_1 \rightarrow (p \wedge Y_2')) & \wedge \\ (Y_2 \rightarrow (q \wedge b) \vee (q \wedge Y_1')) & \wedge \\ x_\varphi = X_1 & \end{pmatrix} \\ \mathcal{J}_\varphi : \{\neg Y_1 \wedge \neg Y_2, \quad X_1 = Y_1 \wedge X_2 = Y_2\} \\ F_\varphi : (X_1 = \mathit{false}) \wedge (X_2 = q \wedge b) \end{cases}$$

The variables $\{X_1, \ldots, X_n, Y_1, \ldots, Y_n\}$ are expected to check that the rest of the sequence from now on has a prefix satisfying the SERE r. Thus, the subsequence $s_j, \ldots, s_k, \ldots \models \langle r\rangle b$ iff there exists a generation sequence $V^j = V_1, V^{j+1}, \ldots, V^k$, such that for each $i, j \leq i < k$, there exists a grammar rule $V^i \rightarrow \beta V^{i+1}$, where $s_i \Vdash \beta, V^k \rightarrow \alpha$, and $s_k \Vdash (\alpha \wedge b)$.

The generation sequence is represented in a run of the tester by a sequence of true valuations for the variables $Z^j = Z_1, Z^{j+1}, \ldots, Z^k$ where $Z^i \in \{X^i, Y^i\}$ for each $i \in [j..k]$. An important element in this checking is to make sure that any such generation sequence is finite. This is accomplished through the double representation of each V_i by X_i and Y_i. The justice requirement $(X_1 = Y_1) \wedge \cdots \wedge (X_n = Y_n)$ guarantees that that any true X_i is eventually copied into Y_i. The justice requirement $\neg Y_1 \wedge \cdots \wedge \neg Y_n$ guarantees that all true Y_i's are eventually falsified. Together, they guarantee that there exists no infinite generation sequence. The double representation approach was first introduced in [20].

8.2 A Tester for $\varphi = r$

We start the construction exactly the same way as we did for $\varphi = \langle r\rangle b$, in Section 8.1. Let $T_\varphi = \langle V_\varphi, \Theta_\varphi, R_\varphi, \mathcal{J}_\varphi, F_\varphi\rangle$ be the tester we wish to construct. Assume that x_φ is the output variable. Let $\mathcal{G} = \langle \mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S}\rangle$ be a grammar associated with r.

The tester T_φ is given by:

$$T(r): \begin{cases} V_\varphi : \mathit{Vars}(r) \cup \{x_\varphi\} \cup \{X_1, \ldots, X_n, Y_1, \ldots, Y_n\} \\ \Theta_\varphi : 1 \\ R_\varphi(V, V') : \text{Each derivation } V_i \rightarrow \alpha_1 \mid \cdots \mid \alpha_m \mid \beta_1 V_1 \mid \cdots \mid \beta_n V_n \\ \quad \text{contributes to } \rho \text{ the conjunct} \\ \quad X_i = \alpha_1 \vee \cdots \vee \alpha_m \vee (\beta_1 \wedge X_1') \vee \cdots \vee (\beta_n \wedge X_n') \\ \quad \text{and the conjunct} \\ \quad \alpha_1 \vee \cdots \vee \alpha_m \vee (\beta_1 \wedge Y_1') \vee \cdots \vee (\beta_n \wedge Y_n') \rightarrow Y_i \\ \quad \text{the output variable is constrained by the conjunct} \\ \quad x_\varphi = X_1 \\ \mathcal{J}_\varphi : \{Y_1 \wedge \cdots \wedge Y_n, \quad X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n\} \\ F_\varphi : \text{Each derivation } V_i \rightarrow \alpha_1 \mid \cdots \mid \alpha_m \mid \beta_1 V_1 \mid \cdots \mid \beta_n V_n \\ \quad \text{contributes to } F \text{ the conjunct} \\ \quad X_i = \alpha_1 \vee \cdots \vee \alpha_m \vee \beta_1 \vee \cdots \vee \beta_n \end{cases}$$

The variables $\{X_1, \ldots, X_n, Y_1, \ldots, Y_n\}$ are expected to check that the rest of the sequence from now on has a prefix that does not violate SERE r. We follow a similar approach as for the tester $\varphi = \langle r \rangle b$. However, now we are more concerned with false values of the variables $X_1 \ldots X_n$. The duality comes from the fact that, now, we are trying to prevent postponing the violation of the formula r forever.

8.3 Complexity of the Construction

Theorem 2. *For every* PSL *formula* φ *of length* n*, there exists a tester with* $O(2^n)$ *variables. If we restrict SERE's to three traditional operators: concatenation* (;)*, union* (|)*, and Kleene closure* ($[*]$)*, the number of variables is linear in the size of* φ*.*

To justify the result, we can just count the fresh variables introduced at each step of the tester construction. There is only linear number of sub-formulas, so there is a linear number of output variables. The only other variables introduced are the ones that are used to handle SERE's. According to Theorem 1, the associated grammars contain at most $O(2^n)$ non-terminals ($O(n)$ - for the restricted case). We conclude by observing that testers for the formulas $\varphi = \langle r \rangle b$ and $\varphi = r$ introduce exactly two variables, X_i and Y_i, for each non-terminal V_i.

9 MITL Testers

In this section we show how to build for every MITL formula φ a *timed tester*, which is a timed automaton T_φ that accepts a language defined by the formula $\varphi \wedge \Box(x_\varphi \equiv \varphi)$. For untimed operators, we can use LTL testers defined in Section 5. In addition, we can use tester composition algorithm described in Section 7. Therefore, in order to handle an arbitrary MITL formula, we just need to build one additional tester for $\varphi = p\,\mathcal{U}_{[a,b]}\,q$. Much of the material in this section is taken from [16].

Our construction for timed until would follow the lines for the untimed case, based on generating predictions for x_φ and aborting when actual values of the signals p and q show they were wrong. However, working on dense time we have the problem that a-priori, the set of potential predictions even for a bounded period of time includes signals with an arbitrary number of switchings between true and false, and such predictions cannot be memorized by a finite-state timed device. An analogous problem exists with untimed case also, since, there, we also make an unbounded number of prediction that should be a checked sometime in the futures. We have resolved the problem for the untimed case based on the observation that our guesses essentially form a finite number of equivalence classes, so we only need to memorize finitely many things. For example, consider untimed until, $\varphi = p\,\mathcal{U}\,q$. Assume, that at the current state p is true and q is false, and we guess that x_φ is true, meaning φ holds at the current position. Moreover, at the next state again $p \wedge \neg q$ is true, and we again predict that x_φ is true. There is no need to distinguish the prediction done at the current and the previous state, and it is enough to remember and verify just one of them. The solution for the timed case is completely different and based on the observation that predictions that switch too frequently cannot be correct. The following lemma, taken from [16] formalizes this observation:

Lemma 1. *Let x be a boolean signal satisfying $\square(x \equiv p\,\mathcal{U}_{[a,b]}\, q)$ for some arbitrary signals p and q. Then, for any factorization $x = v \cdot 1^{r_1} \cdot 0^{r_2} \cdot 1^{r_3} \cdot 0^{r_4} \cdot w$ we have $r_2 + r_3 > \min\{a, b-a\}$.*

Proof: The following observations concerning the constraints on the values of x, p and q at every t follow from the definitions:

1. If x holds at t, p must hold in all the interval $[t, t+a]$;
2. If q holds at $t+b$ and p holds throughout $[t, t+b]$ then x holds during $[t, t+b-a]$.

Let $[t_1, t_2)$ and $[t_2, t_3)$ be the corresponding intervals for 0^{r_2} and 1^{r_3}, respectively (see Fig. 3), and let us show that $(t_2 - t_1) < a$ implies that $(t_3, t_2) \geq b-a$. Since $(t_2 - t_1) < a$ and $x(t_1 - \epsilon) = 1$, observation 1 implies that $p = 1$ throughout the interval $[t_1, t_2]$. As $x(t) = 1$ for all $t \in [t_2, t_3)$, it follows that $p(t) = 1$ for all $t \in [t_1, t_3)$. This implies that q must start holding at $t_2 + b$ and not before that, because otherwise this will imply that x holds inside the interior of $[t_1, t_2]$ contrary to our assumptions. It follows by observation 2 that x holds continuously in $[t_2, t_2 + b - a]$. Consequently, $t_3 \geq t_2 + b - a$, implying that $(t_3 - t_2) \geq b - a$ ▄

The importance of this property is that it bounds the variability of any reasonable prediction and constrains the relation between the logical and metrical length of the candidate signals. Let $d = \min\{a, b-a\}$ and $m = b/d$. Each 01 part of x has metric length of at least d, and an acceptable prediction of the form $(01)^m \cdot 0$ has a metric length beyond b. Therefore, its initial part can be forgotten and the remaining part has at most $2m$ predictions left unverified.

In addition to the above lemma, we are going to use the following equivalence:

$$p\,\mathcal{U}_{[a,a+b]}\, q \quad \equiv \quad \Diamond_a \left[\boxminus_{[0,a]}\, p \wedge (p\,\mathcal{U}_{[0,b]}\, q)\right]$$

where $\Diamond_a$ is a "shift by a" operator, a shorthand for $\Diamond_{[a,a]}$, and $\boxminus_{[0,a]}\, p$ is a past analog of the $\square_{[a,b]}$operator. Note that $\Diamond_{[a,a]p}$ is not a proper MITL formula since operator $\Diamond_{[a,b]}$ requires that $a < b$. The formula $\boxminus_{[0,a]}\, p$ is also not in the language, but it can be added if needed. We define that the formula $\boxminus_{[0,a]}\, p$ is satisfied iff p has been continuously true for the last a time units.

Thus, assuming that we have constructed testers $T[\Diamond_a\, p]$, $T[\boxminus_{[0,a]}\, p]$, and $T[p\,\mathcal{U}_{[0,b]}\, q]$ for the corresponding formulas, then $T[p\,\mathcal{U}_{[a,a+b]}\, q]$ can be given by the three ways synchronous parallel composition:

$$T[p\,\mathcal{U}_{[0,b]}\, q] \quad ||| \quad T[\boxminus_{[0,a]}\, p] \quad ||| \quad T[\Diamond_a (x_{p\,\mathcal{U}_{[0,b]}\, q} \wedge x_{\boxminus_{[0,a]}\, p})].$$

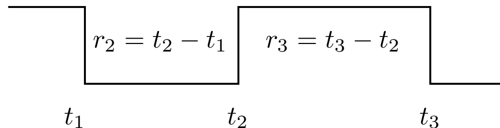


Fig. 3. A signal u satisfying $\square(x \equiv p\,\mathcal{U}_{[a,b]}\, q)$

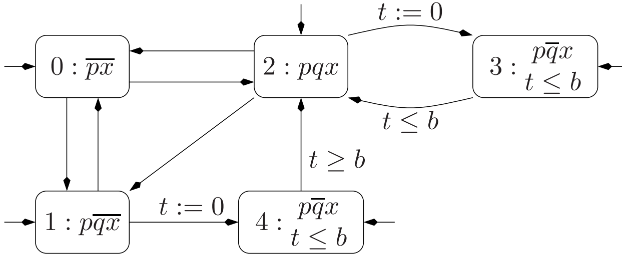


Fig. 4. A tester for $p\,\mathcal{U}_{[0,b]}\,q$

Next, we are going to present the three remaining testers. By convention, each node in Fig. 4, Fig. 5, and Fig. 6 has an implicit self-loop; the self-loops are omitted for sake of clarity. We also assume that all the clocks are initially set to 0. In addition, note that unlike the untimed case, for the timed testers shown in this paper the validity of predictions is always resolved in finite time. Therefore, we do not need any conditions at infinity.

9.1 A Tester for $p\,\mathcal{U}_{[0,b]}\,q$

In Fig. 4, we present a tester for this formula.

9.2 A Tester for $\boxminus_{[0,a]}\,p$

In Fig. 5, we present a tester for the formula $\boxminus_{[0,a]}\,p$.

9.3 A Tester for $\Diamond_a\,p$

In general, it is impossible to construct a tester for the formula $\Diamond_a\,p$ with a bounded number of clocks. However, if we know that the signal p has bounded variability, then such a construction is possible. We assume in the following that p has no more than k changes for each period of length a. This holds in our case since operator $\Diamond_a$ is only used as an auxiliary construct to handle $\mathcal{U}_{[a,a+b]}$ operator to which Lemma 1 applies. The tester for $\Diamond_a\,p$ can be given by the following parallel composition:

$$U \;|||\; P \;|||\; ON[0] \;|||\; OFF[0] \;|||\; \cdots \;|||\; ON[k{-}1] \;|||\; OFF[k{-}1]$$

In Fig. 6, we present the automata U, P, and generic $ON[i]$ and $OFF[i]$.

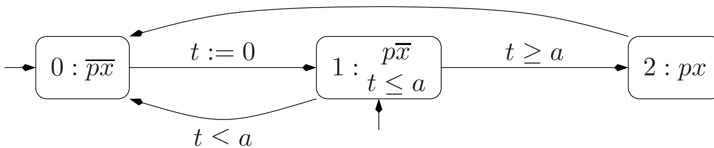


Fig. 5. A tester for $\boxminus_{[0,a]}\,p$

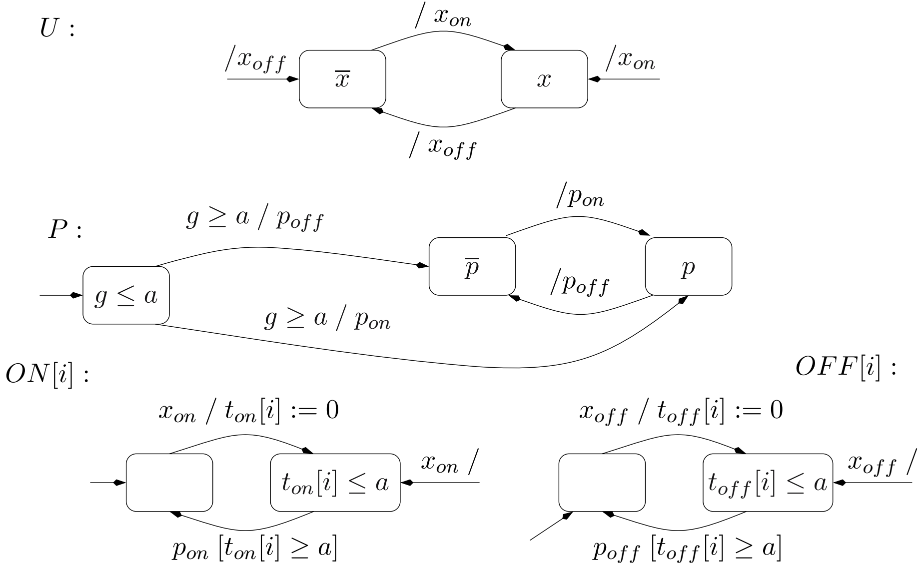


Fig. 6. The automata U, P, $ON[i]$, and $OFF[i]$

10 Using Testers for Model Checking

One of the main advantages of our construction is that all the steps, as well as the final result – the tester itself, can be represented symbolically. That is particularly handy if one is to use symbolic model checking [4]. Assume that the formula under consideration is φ, and $T_\varphi = \langle V_\varphi, \Theta_\varphi, R_\varphi, \mathcal{J}_\varphi, F_\varphi \rangle$ is the corresponding tester. Let JDS $\mathcal{D}$ represent the system we wish to model check.

We are going to use traditional automata theoretic approach based on synchronous composition, as in [4]. We perform the following steps:

- Compose $\mathcal{D}$ with T_φ^+ to obtain $\mathcal{D} \,|||\, T_\varphi^+$.
- Check if $\mathcal{D} \,|||\, T_\varphi^+$ has a (fair) computation, such that $s_0[x_\varphi] = 0$.
  $\mathcal{D} \,|||\, T_\varphi^+$ has such a computation iff $\mathcal{D}$ does not satisfy φ.

As can be seen, a tester can be used anywhere instead of an automaton. Indeed, we can always obtain an automaton from a tester by restricting the initial state to interpret x_φ as $true$.

11 LTL Deductive Verification

Another important application of testers is deductive verification, which is ultimately the only approach towards verification of infinite state systems. A complete deductive proof system for linear-time temporal logic (LTL) has been presented in [17] and further elaborated in [18] and [19]. The approach first defines deductive proof rules for

special form formulas, the most important of which are formulas of the form $p \Rightarrow \Box q$, $p \Rightarrow \Diamond q$, and $\Box \Diamond p \Rightarrow \Box \Diamond q$, where p and q are arbitrary *past formulas*, where $\varphi \Rightarrow \psi$ is a shorthand for $\Box(\neg\varphi \vee \psi)$. To deal with arbitrary formulas, [17] invokes a general canonic-form theorem, according to which every (quantifier-free) LTL formula is equivalent to a conjunction of formulas of the form $\Box \Diamond p_i \Rightarrow \Box \Diamond q_i$, for some past formulas p_i and q_i. While this approach is theoretically adequate, it is not a practically acceptable solution to the verification of arbitrary LTL formulas. This is because the best known algorithms for converting an arbitrary LTL formula into canonic form are at least doubly exponential (e.g., [11] which is actually non-elementary).

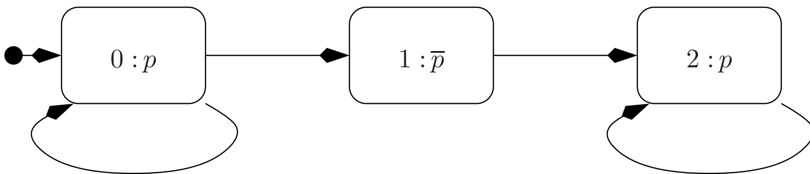
The new tester-based approach which has been first introduced in [14], is based on a successive elimination of temporal operators from a given formula φ until we hit a special form, to which we can apply the predefined rules. Elimination of the temporal operators is based on the construction of temporal testers, as presented in Section 5. Let φ be a an arbitrary LTL or even PSL formula containing one or more occurrences of the sub-formula ψ. In Fig. 7, we present the rule that reduces the proof of φ with respect to some system $\mathcal{D}$ to the proof of $\varphi[\psi/x_\psi]$ over $\mathcal{D} \mathbin{|||} T[\varphi]$, where $T[\varphi]$ is a temporal tester for φ, x_φ is the output variable of $T[\varphi]$, and $\varphi[\psi/x_\psi]$ denotes the formula φ in which each occurrence of the sub-formula ψ is replaced with x_ψ.

For an arbitrary PSL formula φ and FDS $\mathcal{D}$,

$$\frac{\mathcal{D} \mathbin{|||} T_\varphi \models \varphi[\psi/x_\psi]}{\mathcal{D} \quad \models \varphi}$$

Fig. 7.

We are going to illustrate application of the rule from Fig. 7 on the following example:



The property whose validity we wish to establish is $\Diamond \Box p$. First, we construct a tester for $\Box p$ and compose it with our system. The transition relation of the new system $\mathcal{D} \mathbin{|||} T[\varphi]$ is presented in Fig. 8.

The justice requirement associated with $\mathcal{D} \mathbin{|||} T[\varphi]$ is $x_\Box \vee \neg p$, and all just states are depicted using double ovals. The new property under consideration $\varphi[\Box p/x_\Box]$, which after the substitution is simply $\Diamond x_\Box$. This is one of the special form formulas, and we can apply the deductive proof from Fig. 9. Strictly speaking our original formula also has a specialized rule, but for the sake of example we ignored it. However, this is not totally artificial since the rule we are going to apply is simpler. Of course, our system after composition is more complex. Nevertheless, one can argue that the additional state

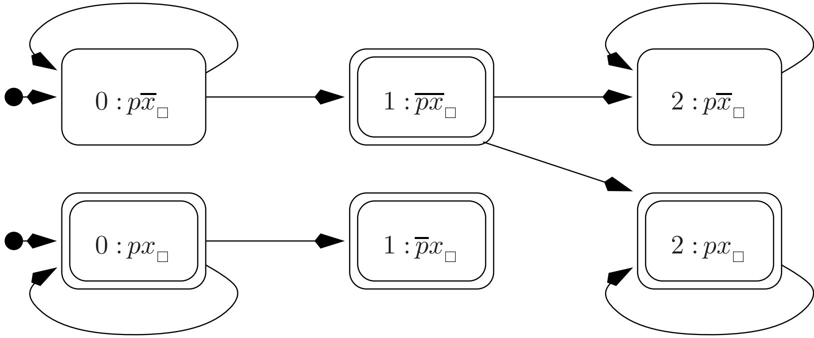


Fig. 8. System $\mathcal{D} \,|||\, T[\varphi]$

For an FDS $\mathcal{D}$ with transition relation ρ and justice set $\mathcal{J} = \{J_1, \ldots, J_m\}$,
assertions $p, q, \varphi_1, \ldots, \varphi_m$,
well-founded domain $(\mathcal{A}, \succ)$ and a ranking function $\delta : \Sigma \mapsto \mathcal{A}$

W1. $\quad p \quad \Rightarrow q \vee \bigvee_{j=1}^{m} \varphi_j$

W2. For $i = 1, \ldots, m$

$$\varphi_i \wedge \rho \Rightarrow q' \vee (\neg J_i' \wedge \varphi_i' \wedge \delta = \delta') \vee \left(\bigvee_{j=1}^{m} \varphi_j' \wedge (\delta \succ \delta') \right)$$

$$p \Rightarrow \diamondsuit\, q$$

Fig. 9. Well-founded eventuality under justice

variable $x_\square$ makes thing easier since it essentially provides CTL like statification for the formula $\square\, p$, where all fair paths out of a state with $x_\square$ set to true must satisfy $\square\, p$.

To apply the rule from Fig. 9, we need to define a well-founded domain $(\mathcal{A}, \succ)$, a ranking function δ, and a set of intermediate assertions $\varphi_1, \ldots, \varphi_m$. The function δ is intended to measure the distance of the current state to a state satisfying the goal q. Premise W1 states that every p-state satisfies q or one of $\varphi_1, \ldots, \varphi_m$. Premise W2 states that for every i, $1 \leq i \leq m$, a φ_i-state with rank $\delta = u$ is followed by either a q-state or a φ_i-state that does not satisfy J_i and has the same rank u, or by a φ_j-state $(1 \leq j \leq m)$ with a smaller rank (i.e., $u \succ \delta$). The rule claims that if premise W1, and the set of m premises W2 are $\mathcal{D}$-valid, then for all (fair) computations a p-state is eventually followed by q-state. In our case, we take p to be true, $\delta = 2 -$ state id (e.g., δ for the state labeled with $[0 :\ px_\square]$ is $2 - 0 = 2$), $\varphi_1 = \overline{x}_\square$.

References

1. Accellera Organization, Inc. Property Specification Language Reference Manual, Version 1.01 (2003), `http://www.accellera.org/`
2. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)

3. Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. In: Symposium on Principles of Distributed Computing, pp. 139–152 (1991)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2000)
5. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. Journal of the ACM 49(2), 172–206 (2002)
6. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. Journal of ACM 28(1), 114–133 (1981)
7. Cimatti, A., Roveri, M., Semprini, S., Tonetta, S.: From PSL to NBA: a modular symbolic encoding, pp. 125–133 (2006)
8. Bustan, D., Fisman, D., Havlicek, J.: Automata Construction for PSL (2005), http://www.wisdom.weizmann.ac.il/~dana/publicat/automta_constructionTR.pdf
9. Daws, C., Yovine, S.: Reducing the number of clock variables of timed automata, pp. 73–81
10. Eisner, C., Fisman, D., Havlicek, J., Gordon, M., McIsaac, A., Van Campenhout, D.: Formal Syntax and Semantics of PSL (2003), http://www.wisdom.weizmann.ac.il/ dana/publicat/formal_semantics_standalone.pdf
11. Gabbay, D.: The declarative past and imperative future. In: Banieqbal, B., Barringer, H., Pnueli, A. (eds.) Temporal Logic in Specification, vol. 398, pp. 407–448 (1987)
12. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley, Reading (1979)
13. Kesten, Y., Pnueli, A., Raviv, L.: Algorithmic verification of linear temporal logic specifications. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 1–16. Springer, Heidelberg (1998)
14. Kesten, Y., Pnueli, A.: A compositional approach to CTL* verification. Theoretical Computer Science 331, 397–428 (2005)
15. Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: POPL 1985: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 97–107. ACM Press, New York (1985)
16. Maler, O., Nickovic, D., Pnueli, A.: From MITL to timed automata. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 274–289. Springer, Heidelberg (2006)
17. Manna, Z., Pnueli, A.: Completing the temporal picture. Theoretical Computer Science 83(1), 97–130 (1991)
18. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification, New York (1991)
19. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, New York (1995)
20. Miyano, S., Hayashi, T.: Alternating finite automata on ω-words. Theoretical Computer Science 32, 321–330 (1984)
21. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 573–586. Springer, Heidelberg (2006)
22. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. Technical Report, Dept. of Computer Science, New York University (2006)
23. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic Model Checking for Real-Time Systems. In: 7th. Symposium of Logics in Computer Science, Santa-Cruz, California, pp. 394–406. IEEE Computer Scienty Press (1992)
24. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. Proc. First IEEE Symp. Logic in Comp. Sci., 332–344 (1986)

A Associating a Regular Grammar with a SERE

Let b be a boolean expression, r', r, r_1, r_2 be SEREs, and $\mathcal{G}', \mathcal{G}, \mathcal{G}_1, \mathcal{G}_2$ the corresponding grammars. Our algorithm is recursive and we assume that $\mathcal{G}, \mathcal{G}_1$, and $\mathcal{G}_2$ have already been properly constructed. Our goal is to build $\mathcal{G}' = \langle \mathcal{V}', \mathcal{T}', \mathcal{P}', \mathcal{S}' \rangle$ for the SERE r'.

- $r' = b$
  - $\mathcal{V}' = \{V\}$
  - $\mathcal{T}' = \{b\}$
  - $\mathcal{P}' = \{V \rightarrow b\}$
  - $\mathcal{S}' = V$
- $r' = r_1 \,;\, r_2$
  - $\mathcal{V}' = \mathcal{V}_1 \cup \mathcal{V}_2$
  - $\mathcal{T}' = \mathcal{T}_1 \cup \mathcal{T}_2$
  - $$\mathcal{P}' = \begin{array}{ll} \{V \rightarrow aW \mid V \rightarrow aW \in \mathcal{P}_1\} & \cup \\ \{V \rightarrow a\mathcal{S}_2 \mid V \rightarrow a \in \mathcal{P}_1, a \neq \epsilon\} & \cup \\ \{V \rightarrow a\mathcal{S}_2 \mid V \rightarrow aW \in \mathcal{P}_1, W \rightarrow \epsilon \in \mathcal{P}_1\} & \cup \\ \mathcal{P}_2 & \end{array}$$
  - $\mathcal{S}' = \mathcal{S}_1$
- $r' = r_1 : r_2$
  - $\mathcal{V}' = \mathcal{V}_1 \cup \mathcal{V}_2$
  - $\mathcal{T}' = \mathcal{T}_1 \cup \mathcal{T}_2$
  - $$\mathcal{P}' = \begin{array}{ll} \{V \rightarrow aW \mid V \rightarrow aW \in \mathcal{P}_1\} & \cup \\ \{V \rightarrow a \wedge b \mid V \rightarrow a \in \mathcal{P}_1, \mathcal{S}_2 \rightarrow b \in \mathcal{P}_2\} & \cup \\ \{V \rightarrow (a \wedge b)W \mid V \rightarrow a \in \mathcal{P}_1, \mathcal{S}_2 \rightarrow bW \in \mathcal{P}_2\} & \cup \\ \mathcal{P}_2 & \end{array}$$

    $$\text{where } a \wedge b = \begin{cases} \epsilon, & \text{if } a = b = \epsilon \\ a, & \text{if } b = \epsilon \\ b, & \text{if } a = \epsilon \\ a \wedge b, & \text{otherwise} \end{cases}$$
  - $\mathcal{S}' = \mathcal{S}_1$
- $r' = r_1 \mid r_2$
  - $\mathcal{V}' = \{\mathcal{S}'\} \cup \mathcal{V}_1 \cup \mathcal{V}_2$
  - $\mathcal{T}' = \mathcal{T}_1 \cup \mathcal{T}_2$
  - $$\mathcal{P}' = \begin{array}{ll} \{\mathcal{S}' \rightarrow aW \mid \mathcal{S}_1 \rightarrow aW \in \mathcal{P}_1\} & \cup \\ \{\mathcal{S}' \rightarrow aW \mid \mathcal{S}_2 \rightarrow aW \in \mathcal{P}_1\} & \cup \\ \mathcal{P}_1 & \cup \\ \mathcal{P}_2 & \end{array}$$
  - $\mathcal{S}' = \mathcal{S}'$

- $r' = r_1 \;\&\&\; r_2$
  - $\mathcal{V}' = \mathcal{V}_1 \times \mathcal{V}_2$
  - $\mathcal{T}' = \mathcal{T}_1 \cup \mathcal{T}_2$
  - $\mathcal{P}' = \begin{array}{l} \{(V,X) \to a \wedge b(W,Y) \mid V \to aW \in \mathcal{P}_1, X \to bY \in \mathcal{P}_2\} \cup \\ \{(V,X) \to a \wedge b \mid V \to a \in \mathcal{P}_1, X \to b \in \mathcal{P}_2\} \end{array}$
  - $\mathcal{S}' = (\mathcal{S}_1, \mathcal{S}_2)$
- $r' = [*0]$
  - $\mathcal{V}' = \{V\}$
  - $\mathcal{T}' = \{b\}$
  - $\mathcal{P}' = \{V \to \epsilon\}$
  - $\mathcal{S}' = V$
- $r' = r[*]$
  - $\mathcal{V}' = \mathcal{V}$
  - $\mathcal{T}' = \mathcal{T}$
  - $\mathcal{P}' = \begin{array}{ll} \{\mathcal{S} \to \epsilon\} & \cup \\ \{V \to a\mathcal{S} \mid V \to a \in \mathcal{P}, a \neq \epsilon\} & \cup \\ \{V \to a\mathcal{S} \mid V \to aW \in \mathcal{P}, W \to \epsilon \in \mathcal{P}\} & \end{array}$
  - $\mathcal{S}' = \mathcal{S}$