Temporal Logic Made Practical

Cindy Eisner and Dana Fisman

Abstract In the late seventies, Pnueli suggested that functional properties of reactive systems be formally expressed in temporal logic. In order that model checking such a logic be possible, it must have sufficient expressive power, its semantics must be formally defined in a rigorous way, and the complexity of model checking it must be well understood and reasonable. In order to allow widespread adoption in the industry, there is an additional requirement: functional specification must be made easy, allowing common properties to be expressed intuitively and succinctly. But while adding syntax is simple, defining semantics without breaking properties of the existing semantics is a different story. This chapter is about the various extensions to temporal logic included in the IEEE standards PSL and SVA, their motivation, and the subtle semantic issues encountered in their definition.

1 Introduction

In his seminal 1977 paper [63], Pnueli first suggested that functional properties of reactive systems be formally expressed in temporal logic (Chapter 2). While the proposal was widely accepted in academia, the exact nature of the temporal logic was a topic of long debate. In particular, the nature of time was widely discussed – whether it should be linear, where each point has a unique next future, or branching, where each point may have multiple next futures. The choice has implications on expressivity as well as the complexity of model checking – see [70] for a survey. Another focus of debate surrounded the question of how to augment LTL, which has the expressive power of star-free ω-regular languages, to a temporal logic that has

Cindy Eisner
IBM Research - Haifa, Haifa 31905 Israel, e-mail: eisner@il.ibm.com

Dana Fisman
Dept. of Computer & Inf. Science, U. of Pennsylvania, USA, e-mail: fisman@seas.upenn.edu

the power of full ω-regular languages. Proposals included automata connectives, second order quantification and grammar operators – see [71] for a full exposition.

With the invention of symbolic model checking in the early 1990's [58], the industrial applicability of model checking began to be recognized, and in 1998, the Accellera Formal Verification Technical Committee began an effort to standardize a temporal logic for use in the hardware industry. Requirements were gathered from industrial users of hardware model checking as to what kinds of things they needed to be able to express more easily than they could in existing temporal logics. This effort eventually resulted in two leading IEEE standards, PSL (IEEE Std 1850 [44, 45]) and SVA (IEEE Std 1800 [46, 47]).

Both PSL and SVA are based on the linear paradigm, and in particular on LTL [63], while PSL includes as well an optional branching extension based on CTL [17] (see also Chapter 2 for a discussion of LTL and CTL). The approach of both to extending the expressive power of LTL to that of ω-regular languages is to add regular expressions and the *suffix implication* operator, which gives implication a temporal flavor by making the consequent dependent on the end of the regular expression used as an antecedent. They both also include a number of specialized operators to allow natural specification of sampling abstractions (e.g. hardware clocks) and truncated paths (e.g. hardware resets). Finally, both provide local variables, which can be seen as a mechanism for both declaring quantified variables and constraining their behavior. Local variables are another way to extend the expressive power of LTL, and also provide much succinctness.

When extending a temporal logic with a new operator, it is important not to break properties of the existing semantics or those of the extension. For instance, if a common property such as distributivity of union over intersection is broken, two formulas that are intuitively equivalent may no longer be so, and even worse, a tool relying on this property may produce an incorrect result. Even if we are willing to review and fix existing algorithms, breaking properties of the existing semantics or its extensions might compromise a user's intuition, making the logic effectively unusable.

Extending a logic without breaking existing properties of the semantics is not trivial, and many early attempts failed. For example, an early attempt at defining the semantics of a clock operator broke the fixed point characterization of LTL's strong until operator in terms of the next operator, and an early attempt at defining local variables broke the distributivity of union over intersection.

In this chapter we will examine the major issues raised by the extension of LTL in order to meet the needs of industry. Since syntax is not the interesting part of the story, we will use a mathematical syntax that allows us to illustrate the semantic issues addressed by both standards without emphasizing either one. In each section we present only a fragment of the semantics relevant to the discussion at hand. The full syntax and formal semantics of PSL and SVA can be found in the corresponding standards [44, 45, 46, 47]. Introductions to PSL and SVA aimed at users can be found in [19] and [16], respectively.

Let P be a set of atomic propositions and b a Boolean expression (propositional formula) over P. The syntax of LTL is given by:

$$f ::= b \mid \neg f \mid f \wedge f \mid f \vee f \mid \mathrm{X}f \mid \mathrm{G}f \mid \mathrm{F}f \mid [f\ \mathrm{U}\ f] \mid [f\ \mathrm{W}\ f]$$

Let f, f_1, f_2 be LTL formulas over P, and w an infinite word over $\Sigma = 2^P$. We use $\ell \Vdash b$ to denote that a letter $\ell \in \Sigma$ satisfies a given Boolean expression b. The semantics of LTL is given by:

- $w \models b \iff w(0) \Vdash b$
- $w \models \neg f \iff w \not\models f$
- $w \models f_1 \wedge f_2 \iff w \models f_1$ and $w \models f_2$
- $w \models \mathrm{X}\, f \iff w(1..) \models f$
- $w \models [f_1\ \mathrm{U}\ f_2] \iff \exists k$ such that $w(k..) \models f_2$ and $\forall j < k$ we have $w(j..) \models f_1$

$f_1 \vee f_2 \equiv \neg(\neg f_1 \wedge \neg f_2) \qquad \mathrm{F}f \equiv [\mathrm{TRUE}\ \mathrm{U}\ f] \qquad \mathrm{G}f \equiv \neg \mathrm{F} \neg f \qquad [f_1\ \mathrm{W}\ f_2] \equiv [f_1\ \mathrm{U}\ f_2] \vee \mathrm{G}f_1$

Fig. 1 The semantics of LTL [63]

The current standards are not necessarily the final word; standards keep evolving to meet the growing and changing needs of the industry. In Section 7 we touch briefly on some directions for future research.

2 From LTL to Regular Expression Based Temporal Logic

The temporal logics PSL and SVA are *linear* temporal logics. In contrast to branching time logics, they follow the view that every point in time has a unique future, and more precisely a unique next time point (see Chapter 2 for more on linear and branching time logics). At the core of PSL and SVA lies the temporal logic LTL standing for *linear time logic*, as proposed by Pnueli [63].

Formulas of LTL are phrased with respect to a set of atomic propositions P, and are interpreted with respect to an infinite word over the alphabet $\Sigma = 2^P$, whose letters represent points in time. We use the notation $[\![\varphi]\!]$ to denote the set of words on which formula φ holds. We number the letters of word w starting from 0, like this: $w = \ell_0\ell_1\ell_2\cdots$, and we use $w(k)$ to denote the $k+1^{st}$ letter of w (since counting starts at 0). We use $w(j..k)$ to denote the finite subword of w starting at $w(j)$ and ending at $w(k)$. We use $w(j..)$ to denote the suffix of w starting at $w(j)$. In addition to the usual propositional operators $\neg$, $\wedge$ and $\vee$, there are temporal operators X (read "next"), G (read "globally"), F (read "eventually"), U (read "strong until", or simply "until") and W (read "weak until") as shown in Figure 1. If φ is a formula of LTL, the formula $\mathrm{X}\varphi$ holds if φ holds at the next point in time. The formula $\mathrm{G}\varphi$ holds if φ holds now and at every point in the future. The formula $\mathrm{F}\varphi$ holds if φ holds now or at some point in the future. The formula $[\varphi\ \mathrm{U}\ \psi]$ holds if ψ holds now or at some point in the future, and in addition φ holds at every point in time until ψ holds. The formula $[\varphi\ \mathrm{W}\ \psi]$ holds if either $[\varphi\ \mathrm{U}\ \psi]$ holds, or φ holds forever. The operators G, F and W can be derived from the other operators as shown in Figure 1.

Using LTL we can express many interesting properties. For instance the property $\mathrm{G}(\mathit{send} \rightarrow \mathrm{XX}\,\mathit{received})$ states that signal *received* should be asserted exactly two time points after signal *send* is asserted. The property $\mathrm{G}(\mathit{send} \rightarrow [\mathit{busy}\ \mathrm{U}\ \mathit{received}])$ states that signal *received* should be asserted some time after signal *send* is asserted, and in the meantime signal *busy* should be asserted.

One of the most important properties of a formalism is its *expressive power*. Besides LTL, other known formalisms for reasoning on infinite words are first order logic over the naturals, monadic second order logic of one successor, automata on infinite words and ω-regular expressions. In terms of expressive power we can roughly classify these into two sets: those recognizing all the ω-regular languages, and those recognizing a strict subset of those, referred to as the *star-free ω-regular* languages.

It follows from a sequence of results of [32, 48, 69] that LTL belongs to the latter (see also [27]). For example, as shown by Wolper [75], the property "p holds on every even position" is not expressible in LTL. Note that simply adding regular expressions, without the ω operator, will not achieve the expressive power of ω-regular languages. However, we can achieve that expressive power through the addition of regular expressions plus the suffix implication operator, as discussed in the next sub-section.

Before continuing, we note that in regular expressions as used in PSL and SVA, the syntactic atoms are Boolean expressions, more than one of which may hold on a given letter of the alphabet. This is in contrast to standard regular expressions, in which the syntactic atoms are mutually exclusive. As in standard regular expressions, we use the notation $\mathcal{L}(r)$ to denote the set of words recognized by the regular expression

2.1 Adding expressive power – suffix implication

Suffix implication, denoted $\mapsto$, and also known as *triggers*, is an operator taking two arguments, a regular expression r and a temporal formula φ. Suffix implication gives implication a temporal flavor by making the consequent dependent on the end of the regular expression used as an antecedent. Intuitively, $r \mapsto \varphi$ holds on a word w if for every prefix of w recognized by r, the suffix of w, starting at the letter on which that prefix ends, satisfies φ.

The suffix implication operator, first proposed in the context of temporal logic in [8], is reminiscent of the modality $[\alpha]\varphi$ of dynamic logic [28, 39, 64]. It is different than the other attempts at combining temporal and dynamic logic ([36, 38, 40, 72]) in that it borrows the dynamic modalities but remains state-based as in temporal logic rather than action-based as in dynamic logic.

Augmenting LTL with the suffix implication operator $\mapsto$, the semantics of which are shown in Figure 2, increases the expressive power to that of ω-regular languages, as stated in the following theorem.

Let r be a regular expression, φ a formula of LTL extended with suffix implication and its dual, and w an infinite word over $\Sigma = 2^P$.

- $w \models r \ \mapsto \ \varphi \iff \forall j$ if $w(0..j) \in \mathcal{L}(r)$ then $w(j..) \models \varphi$
- $w \models r \ \diamond\!\!\rightarrow \varphi \iff \exists j$ s.t. $w(0..j) \in \mathcal{L}(r)$ and $w(j..) \models \varphi$
- All other LTL operators are as usual.

Fig. 2 Semantics of LTL extended with suffix implication [6, 7, 8, 44, 45, 46, 47] and its dual

Theorem 1 ([6, 52]). *Let L be an ω-regular language. Then there exists a formula φ of* LTL *extended with suffix implication such that $[\![\varphi]\!] = L$.*

Using suffix implication, we can express the property that p holds on every even position as follows:

$$((\text{TRUE}, \text{TRUE})^*) \mapsto p \tag{1}$$

Although suffix implication extends the expressive power to that of ω-regular languages, expressive power is not the only issue. In order to make suffix implication really useful to users, we must provide syntactic support for commonly needed patterns. These are the subjects of the following sub-sections.

2.2 Adding Succintness

2.2.1 Counting

The User's Point of View Consider the property expressing that if a request is acknowledged (signal *ack* is asserted four to six cycles after *req* is asserted, and in the meantime signal *busy* is asserted), then signal *busy* should remain asserted until *done* holds. This property can be expressed as follows in LTL:

$$\begin{aligned} &\text{G}\,(req \rightarrow \text{X}(busy \rightarrow \text{X}(busy \rightarrow \text{X}(busy \rightarrow \text{X}((ack \rightarrow [busy\ \text{U}\ done]) \wedge \\ &(busy \rightarrow \text{X}((ack \rightarrow [busy\ \text{U}\ done]) \wedge (busy \rightarrow \text{X}(ack \rightarrow [busy\ \text{U}\ done]))))))))) \end{aligned} \tag{2}$$

However, increase "four to six cycles" and the property soon becomes unwieldy. Thus the user would like there to be an easier way.

Bare suffix implication is not much help. We could use three separate formulas, or we could use the regular expression union operator $\cup$ (distinguished from U, denoting the strong until operator) to write:

$$\begin{aligned} \text{G}\,((req \cdot\ &((busy \cdot busy \cdot busy) \cup (busy \cdot busy \cdot busy \cdot busy) \cup \\ &(busy \cdot busy \cdot busy \cdot busy \cdot busy)) \cdot ack) \mapsto [busy\ \text{U}\ done]) \end{aligned} \tag{3}$$

However, add the ability to count and Formulas 2 and 3 can be expressed much more clearly. Let $\cdot$ and * denote concatenation and Kleene star, respectively, let the

repetition operator $r[^*k]$ abbreviate r concatenated to itself k times, and let $r[^*i..j]$ abbreviate $\bigcup_{k=i}^{j} r[^*k]$. Then we can write simply:

$$\mathrm{G}\ ((req \cdot busy[^*3..5] \cdot ack) \mapsto [busy\ \mathrm{U}\ done]) \tag{4}$$

The user would like counting operators, so that Formula 2 can be expressed simply as Formula 4. Other desirable counting operators are the goto repetition operator $b[\rightarrow k]$, abbreviating $(\neg b^* \cdot b)[^*k]$ and discussed in the next sub-section, and the non-consecutive repetition operator, $b[= k]$, abbreviating $(\neg b^* \cdot b)[^*k] \cdot \neg b^*$.

Semantic Issues Counting operators are simple syntactic sugar, and as such do not add expressive power. However, they do affect succintness.

Theorem 2 ([33, 50]). *Regular expressions with counting operators are doubly-exponential more succinct than* DFA*s and exponentially more succinct than standard regular expressions and* NFA*s.*

2.2.2 First match

The User's Point of View Consider the properties "the first occurence of *ack* after every *req* is followed by *gnt*". In LTL this would be

$$\mathrm{G}\ (req \rightarrow \mathrm{X}[\neg ack\ \mathrm{U}\ (ack \wedge \mathrm{X}gnt)]) \tag{5}$$

The user would like a more direct way to express this. Using the *goto* operator, $b[\rightarrow]$, abbreviating $\neg b^* b$ and supported by both PSL and SVA, achieves this as follows:

$$\mathrm{G}\ ((req \cdot ack[\rightarrow]) \mapsto (gnt)) \tag{6}$$

Semantic Issues The goto operator is a kind of counting operator, and like the repetition operators of the previous sub-section, designating the ith occurrence of a Boolean expression is achieved through syntactic sugaring: $b[\rightarrow i]$ abbreviates $(\neg b^* b)[^*i]$. SVA includes as well the *first match* operator, denoted here by $\mathrm{FM}(r)$, designating the first occurrence of a regular expression. First match adds succintness, but does not add expressive power. On the other hand, while intuitive, its semantics is not easily derived from other operators, so it is more than syntactic sugar.

Formally, $\mathcal{L}(\mathrm{FM}(r)) = \{w \in \mathcal{L}(r) \mid w = uv \text{ and } u \in \mathcal{L}(r) \text{ implies that } v = \varepsilon\}$. The origin of first match is in [60], which provides also the *fail* operator, capturing the set of shortest words w such that w is not in $\mathcal{L}(r)$. Formally, $\mathcal{L}(\mathrm{FAIL}(r)) = \{w \notin \mathcal{L}(r) \mid w = uv \text{ and } u \notin \mathcal{L}(r) \text{ implies that } v = \varepsilon\}$. Intuitively, $\mathcal{L}(\mathrm{FM}(r))$ is the set of shortest words in $\mathcal{L}(r)$ ("good prefixes") and $\mathcal{L}(\mathrm{FAIL}(r))$ is the set of shortest words with no extension in $\mathcal{L}(r)$ ("bad prefixes"). These operators raise issues related to the distinction between bad prefixes and informative prefixes and the related complexity issues – see Section 4.3.

2.2.3 Intersection

The User's Point of View Consider the property asserting that if a print request is issued (*glbl_prnt* is asserted), then all three printers should issue either a success or error message (assertion of $succ_i$ or err_i, or in short, assertion of se_i) before signal *prnt_done* is asserted. If we try to formulate it using only the standard operators, we see that we need to account for all the possible orders in which the printers will issue the corresponding message.

The user would like an easier way. Using the intersection operator and the non-consecutive repetition operator $b[=k] \equiv (\neg b^* \cdot b)[^*k] \cdot \neg b^*$, the property can be stated as follows:

$$\mathrm{G}\ (glbl\_prnt \rightarrow ((se_1[=1] \cap se_2[=1] \cap se_3[=1]) \cdot prnt\_done))$$

Semantic Issues The semantics of intersection is straightforward, and is given by $\mathcal{L}(r_1 \cap r_2) = \mathcal{L}(r_1) \cap \mathcal{L}(r_2)$. It is well known that regular sets are closed under intersection [43], thus there is no increase in expressive power. However, there is an increase in succinctness, as stated by the following theorem.

Theorem 3 ([33, 34]). *Regular expressions with intersection are doubly-exponential more succinct than standard regular expressions and* DFA*s and exponentially more succinct than* NFA*s.*

2.2.4 Fusion

The User's Point of View Let a *send transaction* be a sequence of cycles where *processing* holds for some number (possibly zero) of cycles, then *sending* holds for some number (possibly zero) of cycles and then *sent* holds, and consider the property that if a granted send-request (assertion of *send* followed by assertion of *gnt*) is followed by a successful send transaction (that starts at the same cycle as the grant), then signal *ok* should be asserted at the same time that *sent* is asserted.

Our formula must distinguish between three different ways in which a send transaction might begin, so we get something like this:

$$\begin{aligned}\mathrm{G}\ ((send \cdot ((gnt \wedge sent) \cup ((gnt \wedge sending) \cdot sending^* \cdot sent) \cup \\ ((gnt \wedge processing) \cdot processing^* \cdot sending^* \cdot sent))) \mapsto (ok))\end{aligned} \tag{7}$$

The user would like a more straightforward way. Using the *fusion* operator, a kind of overlapping concatenation denoted $\circ$, we can formulate this more succinctly as follows:

$$\mathrm{G}\ (((send \cdot gnt) \circ (processing^* \cdot sending^* \cdot sent)) \mapsto (ok)) \tag{8}$$

Semantic Issues The fusion of languages U and V over Σ, denoted $U \circ V$, is the set $\{u\ell v \mid \ell \in \Sigma, u\ell \in U, \text{ and } \ell v \in V\}$. Like the counting operators and intersection, fusion does not add expressive power, but does add succintness.

Theorem 4 ([42]). *Regular expressions with fusion are exponentially more succinct than standard regular expressions and* DFA*s.*

Henceforth we refer to regular expressions augmented with intersection and fusion as *semi-extended regular expressions (*SERE*s)* (note that the term *extended regular expression* is sometimes used for regular expressions with complementation).

2.2.5 Past operators

The User's Point of View Consider the following property: "signal *dt_complete* should be asserted if and only if a data transfer has just completed", where a data transfer is an assertion of signal *data_start* followed by some number of assertions of *data* followed by *data_end*. One direction of the implication is easy:

$$\mathrm{G}\ (data\_start \cdot data^* \cdot data\_end) \mapsto dt\_complete \tag{9}$$

However, the other direction is extremely difficult to express without past operators, as it involves negation of a regular expression and thus determinization. The user would like to be able to say simply:

$$\mathrm{G}\ (dt\_complete \leftrightarrow ended(data\_start \cdot data^* \cdot data\_end)) \tag{10}$$

where, for SERE r, $ended(r)$ holds in position j of word w iff there exists $i \leq j$ such that $w(i..j) \in \mathcal{L}(r)$. Both PSL and SVA supply the $ended()$ operator, which can be used wherever a Boolean expression can be used.

Semantic Issues Adding past operators to LTL (for every LTL operator its dual past operator) does not increase the expressive power:

Theorem 5 ([55]). LTL *with (future and) past operators has the same expressive power as* LTL *(with just future operators).*

It is easy to see by automata construction that the analogous result holds for LTL augmented with suffix implication and the past operator $ended()$. Past operators do, however, add succinctness [53] (see also Chapter 2).

2.3 Distinguishing between weak and strong regular expressions

The User's Point of View Recall Formula 4, repeated below as Formula 11:

$$\mathrm{G}\ ((req \cdot busy[^*3..5] \cdot ack) \mapsto [busy\ \mathrm{U}\ done]) \tag{11}$$

As long as regular expressions are available, many users prefer to avoid LTL completely. Can we use regular expressions to formulate the right hand side of Formula 11 as well? The regular expression $(busy^* \cdot done)$ seems a good candidate to

represent the right hand side. But in LTL, it is possible to distinguish between the formula $[busy \text{ U } done]$, using the *strong* until operator, requiring *done* to eventually hold, and the formula $[busy \text{ W } done]$, using the *weak* until operator, which holds also if *done* never holds and *busy* holds forever.

Thus the user would like a syntax distinguishing between *strong* and *weak* versions of a regular expression. Using $r!$ to denote a strong regular expression, Formula 4 can be rephrased as follows:

$$\text{G } (req \cdot busy[^*3..5] \cdot ack) \mapsto (busy^* \cdot done)! \tag{12}$$

Similarly, the weak version of Formula 11:

$$\text{G } (req \cdot busy[^*3..5] \cdot ack) \mapsto [busy \text{ W } done] \tag{13}$$

can be rephrased as follows, where r (vs. $r!$) denotes a weak regular expression:

$$\text{G } (req \cdot busy[^*3..5] \cdot ack) \mapsto (busy^* \cdot done) \tag{14}$$

Semantic Issues The intended semantics of a strong regular expression is clear. An infinite word satisfies a strong regular expression if it has a prefix in the language of the regular expression. For a weak regular expression, roughly speaking, we would like to say that we can get stuck in an infinite loop of a starred subexpression. Intuitively, this seems to require that every prefix of the observed word has some extension in the language of the regular expression.

While initially pleasing, this does not give the desired semantics. To see why, recall that we want the weak regular expression $(busy^* \cdot done)$ to be equivalent to $[busy \text{ W } done]$ and the weak regular expression $(busy^* \cdot \text{FALSE})$ to be equivalent to $[busy \text{ W FALSE}]$. However, using the definition that every prefix has some extension in the language of the regular expression works for $(busy^* \cdot done)$ but not for $(busy^* \cdot \text{FALSE})$, whose language is empty over the alphabet 2^P. Note that FALSE can be replaced with a complicated unsatisfiable formula, and so the situation is not necessarily easy to identify syntactically.

The $\top, \bot$ approach, proposed in [25], shown in Figure 3 and adopted by PSL 1850-2005 [44] and SVA [46, 47], addresses this issue (see also Section 4.2). The idea is that $\top$ and $\bot$ are special letters, with the following properties: $\top$ satisfies any Boolean expression, including FALSE, and $\bot$ satisfies no Boolean expression, including TRUE. The inductive definition of the temporal logic makes the word $\top^\omega$ satisfy every temporal logic formula, whereas $\bot^\omega$ satisfies no temporal formula. This way, we can talk about extension without the need to worry about unsatisfiable eventualities of a regular expression.

Let r be a regular expression (but not a SERE, see below). Under the $\top, \bot$ semantics, $r!$ and r are related in the same way as other pairs of strong and weak LTL operators [22] . In analogy to the *safety* and *liveness* components of a formula [2, 3], [22] first defines the *weak* and *strong* component of a formula as the topological closure and interior, respectively, over the extended alphabet $\Sigma \cup \{\top, \bot\}$, and shows that such a definition is equivalent to the definition of Figure 4. It then states the

Let r be a regular expression, let $\top$ be a special letter such that $\top \Vdash b$ for any Boolean expression b (including FALSE), and let w be an infinite word over $\Sigma = 2^P \cup \{\top, \bot\}$.

- $w \models r! \iff \exists j$ s.t. $w(0..j) \in \mathcal{L}(r)$
- $w \models r \iff \forall j\ w(0..j)\top^\omega \models r!$

Fig. 3 Semantics of strong and weak regular expressions [25] adopted by PSL 1850-2005 [44] and SVA [46, 47]

Let $\top$ and $\bot$ be special letters such that $\top \Vdash b$ for any Boolean expression b (including FALSE), and $\bot \nVdash b$ for any b (including TRUE). For some alphabet Γ, let Γ^∞ denote $\Gamma^* \cup \Gamma^\omega$. Let φ be a formula in LTL extended with weak and strong regular expressions. Then its weak and strong components, denoted $weak(\varphi)$ and $strong(\varphi)$ respectively, are defined as follows.

- $weak(\varphi) = \{w \in (\Sigma \cup \{\top, \bot\})^\infty \mid \forall \text{ finite } u \preceq w : u\top^\omega \in [\![\varphi]\!]\}$
- $strong(\varphi) = \{w \in (\Sigma \cup \{\top, \bot\})^\infty \mid \exists \text{ finite } u \preceq w : u\bot^\omega \in [\![\varphi]\!]\}$

Fig. 4 Weak and Strong Components [22]

following characterization.

Theorem 6 ([22]). *Let φ be a formula of* LTL *extended with weak and strong regular expressions in positive normal form. Let φ_w be the formula obtained by weakening all operators (replacing $r!$ with r,* X! *with* X *and* U *with* W*). Let φ_s be the formula obtained by strengthening all operators. Then,*

- $[\![\varphi_w]\!] = weak(\varphi)$
- $[\![\varphi_s]\!] = strong(\varphi)$

It is shown in [21, 22] that for the semantics of PSL 1850-2005 [44] and of SVA [46, 47], the characterization does not hold for SEREs, i.e., when intersection and fusion are included, because of the presence of *structural contradictions*. Structural contradictions are SEREs that are unsatisfiable (i.e., whose language is empty) due to their structure. That is, for any replacement of the propositions in a SERE, the language of the SERE remains empty. For example, $p \cap (p \cdot q)$ is a structural contradiction, while $(p \cdot q) \cap (p \cdot \neg q)$ is a contradiction, but not a structural one. A semantics fixing this problem was proposed in [21] and was adopted by PSL 1850-2010 [45]. It defines, in addition to the traditional $\mathcal{L}(r)$, the language of finite proper prefixes of a SERE, denoted $\mathcal{F}(r)$, and the loop language of a SERE, denoted $\mathcal{I}(r)$, shown in Figure 5. Then the truncated semantics of strong and weak SEREs are as shown in Figure 6. The topological characterization for this semantics appears in [23].

It is instructive to illustrate the difference between the weak/strong components and the safety/liveness components. Often they coincide, but not always. For example, the safety component of $[p$ U FALSE$]$ is simply FALSE, while its weak component is $[p$ W FALSE$]$, i.e., G p.

A formula φ is semantically weak if $[\![\varphi_w]\!] = [\![\varphi]\!]$ (where φ_w is defined in Theorem 6). The notion of semantic weakness captures exactly the set of "good" safety

Let ε be the empty word, let λ denote the empty regular expression, let b be a Boolean expression and let r, r_1 and r_2 be SEREs.

$\mathcal{L}(\lambda) = \{\varepsilon\}$	$\mathcal{L}(r_1 \cdot r_2) = \mathcal{L}(r_1) \cdot \mathcal{L}(r_2)$	$\mathcal{L}(r_1 \cup r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$
$\mathcal{L}(b) = \{\ell \in \widehat{\Sigma} \mid \ell \Vdash b\}$	$\mathcal{L}(r_1 \circ r_2) = \mathcal{L}(r_1) \circ \mathcal{L}(r_2)$	$\mathcal{L}(r_1 \cap r_2) = \mathcal{L}(r_1) \cap \mathcal{L}(r_2)$
	$\mathcal{L}(r^+) = \mathcal{L}(r)^+$	
$\mathcal{F}(\lambda) = \emptyset$	$\mathcal{F}(r_1 \cdot r_2) = \mathcal{F}(r_1) \cup (\mathcal{L}(r_1) \cdot \mathcal{F}(r_2))$	$\mathcal{F}(r_1 \cup r_2) = \mathcal{F}(r_1) \cup \mathcal{F}(r_2)$
$\mathcal{F}(b) = \varepsilon$	$\mathcal{F}(r_1 \circ r_2) = \mathcal{F}(r_1) \cup (\mathcal{L}(r_1) \circ \mathcal{F}(r_2))$	$\mathcal{F}(r_1 \cap r_2) = \mathcal{F}(r_1) \cap \mathcal{F}(r_2)$
	$\mathcal{F}(r^+) = \mathcal{L}(r)^* \cdot \mathcal{F}(r)$	
$\mathcal{I}(\lambda) = \emptyset$	$\mathcal{I}(r_1 \cdot r_2) = \mathcal{I}(r_1) \cup (\mathcal{L}(r_1) \cdot \mathcal{I}(r_2))$	$\mathcal{I}(r_1 \cup r_2) = \mathcal{I}(r_1) \cup \mathcal{I}(r_2)$
$\mathcal{I}(b) = \emptyset$	$\mathcal{I}(r_1 \circ r_2) = \mathcal{I}(r_1) \cup (\mathcal{L}(r_1) \circ \mathcal{I}(r_2))$	$\mathcal{I}(r_1 \cap r_2) = \mathcal{I}(r_1) \cap \mathcal{I}(r_2)$
	$\mathcal{I}(r^+) = (\mathcal{L}(r)^* \cdot \mathcal{I}(r)) \cup (\mathcal{L}(r) \setminus \{\varepsilon\})^\omega$	

Fig. 5 The language $\mathcal{L}(r)$, the language of proper prefixes $\mathcal{F}(r)$ and the loop language $\mathcal{I}(r)$ of a SERE r [21, 23, 45].

- $w \models r! \iff \exists j < |w|$ s.t. $w(0..j) \in \mathcal{L}(r)$

  $w \models r \iff$ either $w \models r!$ or $w \in \mathcal{I}(r) \cup \mathcal{F}(r) \cup \{\varepsilon\}$

Fig. 6 The truncated semantics of strong and weak SEREs, as proposed by [21] and adopted by PSL 1850-2010 [45]; see also [23].

properties – those that are computationally easy to verify, as shown by Theorem 19 in Section 4.3.

Defining the semantics is not enough, of course. We also need to supply an implementation. Automata construction for LTL augmented with strong regular expressions via suffix implication is given in [9, 13, 14]. Implementation for the enhancement with weak regular expressions as well is given in [9, 13]. Special treatments of subsets thereof are given in [10, 11].

3 Clocks and Sampling

In this section, we present the clock operator (@). We begin with hardware clocks as a motivating example, and in Section 3.2 present an example where the @ operator is used as a time sampling abstraction, separate from the notion of a hardware clock.

3.1 Hardware clocks

Synchronous hardware designs are based on a notion of discrete time, in which a *clock signal* causes memory elements (flip-flops or latches) to transition from one state to the next. The time from one transition until just before the next is termed a *clock cycle*. In the early days of hardware model checking, designs typically had a single clock, and tools that built the model from the source code (written in some

Hardware Description Language, or HDL) would abstract away the clock cycle, assuming that a clock cycle corresponded to a single step in time of the model. Thus, the following LTL formula:

$$\mathrm{G}(p \rightarrow \mathrm{X}\, q) \tag{15}$$

was used to express the property "globally, if p then *at the next clock cycle*, q".

Modern hardware designs, however, are typically based on multiple clocks. In such a design, for instance, some memory elements may be clocked with *clka*, while others are clocked with *clkb*. In such a case, clock cycles cannot be considered to be atomic – the clock cycles of *clka* and *clkb* might overlap, and the nature of their interaction affects the behavior of the design in important ways. Thus tools that build the model cannot abstract away a clock cycle, and it becomes necessary for the formula to mention the clock signal explicitly. Another complication is that clocking is often done on the *edge* of a clock, resulting in what is termed an *edge-triggered* design. A *positive edge* means a point in time when the clock has just risen from 0 to 1, and a *negative edge* means a point in time when the clock has just fallen from 1 to 0.

The User's Point of View Consider the property "globally, if p at a positive edge of clock *clka*, then at the next positive edge of *clka*, q". In LTL, this is:

$$\begin{aligned} &\mathrm{G}\ (\neg clka \rightarrow \mathrm{X}((clka \wedge p) \rightarrow \\ &\quad \mathrm{X}[(\neg(\neg clka \wedge\ \mathrm{X}\ clka))\ \mathrm{W}\ (\neg clka \wedge\ \mathrm{X}\ (clka \wedge q))]) \end{aligned} \tag{16}$$

Using suffix implication and the goto operator (see Section 2.2.2), we can make it slightly more readable, as follows:

$$\mathrm{G}\ (((clka \wedge p)[\rightarrow] \cdot \neg clka[\rightarrow] \cdot clka[\rightarrow]) \mapsto q) \tag{17}$$

However, taking the clock signal into consideration in each formula quickly becomes unwieldy. Thus the user would like a clock operator, allowing the behavior relative to the clock to be expressed more directly as, for instance:

$$\mathrm{G}(p \rightarrow \mathrm{X}\, q)@(posedge\ clka) \tag{18}$$

Semantic Issues Intuitively, the purpose of the clock operator is to define a projection onto those letters where the clock holds (or transitions, in the case of edge-triggered designs). For example, consider the *trace* shown in Figure 7. The hardware designer understands the x-axis as time; the value of each signal is indicated by a *waveform* – low indicates a value of 0, and high indicates a value of 1. Formally, we view each point in time as a letter from the alphabet 2^P, mapping the atomic propositions p, q, etc. to either 0 or 1. The projection of the trace onto those points in time where *clka* has a positive edge results in a trace consisting of three points in time, shown shaded in the figure. Formula 18 holds on the full trace because Formula 15 holds on the projection.

When we turn to formalize this intuition, we are confronted with two issues. First, hardware clocks do not "accumulate". That is, we want a nested clock operator

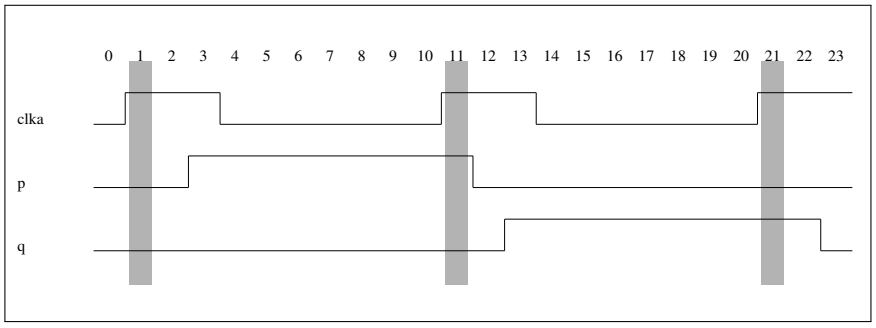

Fig. 7 A trace, showing time on the x-axis. The shaded points in time show the projection of the trace onto the positive edge of *clka*.

to have the effect of "changing the projection", rather than further projecting the projected word. This means that while projection is a useful intuition, we cannot simply define that $w \models \varphi@c \iff w' \models \varphi$, where w' is the projection of w onto those letters where c holds, because doing so would mean that we lose the ability to later "change the projection" of the original word w.

Second, we must define what happens if the clock "stops ticking". For example, on an infinite word w such that c holds on $w(0)$ but on no other letter, do we want $(\mathrm{X}\ \varphi)@c$ to hold or not? Thus, the addition of clocks introduces problems similar to those of defining LTL semantics for finite words. And not only may the projection of an infinite word be finite, it may be empty as well.

Early attempts to deal with these issues [6, 18] defined two kinds of clocks, strong and weak, which behaved differently on finite and empty words. Doing so resulted in semantics that suffered from various weaknesses, for instance that a liveness formula may hold on some word w, but not on an extension ww', or that the formula $(\mathrm{F}\ p) \wedge (\mathrm{G}\ q)$ cannot be satisfactorily clocked for a finite word.

The semantics of [26], adopted by PSL and SVA and shown in Figure 8, solves these problems. In it, the issues of finite and empty words are cleanly separated from the clock operator, whose only role is to define a projection, and nesting of clock operators has the effect of changing the projection. The issue of the semantics on a finite or empty projection is solved by taking the strength from the formula itself, and to this end, there are strong and weak versions of every operator and of Boolean expressions. While the idea of strong and weak clocks has disappeared from PSL and SVA, the issues raised by a clock that stops ticking are manifested in the concept of a truncated path, and dealt with by the truncated semantics discussed in Section 4.

Note that the semantics are defined for $\varphi@c$, where c is a Boolean expression. Edge-triggered designs are supported by defining that *posedge clk* is equivalent to $\mathit{ended}(\neg \mathit{clk}^* \cdot \mathit{clk})$ and similarly that *negedge clk* is equivalent to $\mathit{ended}(\mathit{clk}^* \cdot \neg \mathit{clk})$ (see Section 2.2.5).

In the following, let $\models$ denote the traditional semantics of LTL, augmented in the obvious way to support strong and weak next operators and strong and weak Boolean expressions on finite and empty as well as infinite words.

Let f, f_1 and f_2 be formulas in LTL extended with the clock operator @, let b and c be Boolean expressions, and let w be a word over $\Sigma = 2^P$. For finite word w, let $w \overset{c}{\models}$ tick denote that $w \in \mathcal{L}(\neg c^* \cdot c)$.

- $w \overset{c}{\models} b \iff \forall j < |w|$ s.t. $w(0..j) \overset{c}{\models}$ tick $, w(j) \Vdash b$
- $w \overset{c}{\models} b! \iff \exists j < |w|$ s.t. $w(0..j) \overset{c}{\models}$ tick and $w(j) \Vdash b$
- $w \overset{c}{\models} \neg f \iff w \overset{c}{\not\models} f$
- $w \overset{c}{\models} f_1 \wedge f_2 \iff w \overset{c}{\models} f_1$ and $w \overset{c}{\models} f_2$
- $w \overset{c}{\models} \mathrm{X}!\, f \iff \exists j < k < |w|$ s.t. $w(0..j) \overset{c}{\models}$ tick and $w(j+1..k) \overset{c}{\models}$ tick and $w(k..) \overset{c}{\models} f$
- $w \overset{c}{\models} [f_1 \ \mathrm{U}\ f_2] \iff \exists k < |w|$ s.t. $w(k) \Vdash c$ and $w(k..) \overset{c}{\models} f_2$ and $\forall j < k$ s.t. $w(j) \Vdash c$, $w(j..) \overset{c}{\models} f_1$
- $w \overset{c}{\models} f@c_1 \iff w \overset{c_1}{\models} f$

Fig. 8 The strengthless clock [26] adopted by PSL and SVA, where the "initial" clock context is defined to be $c =$ TRUE. The semantics of clocked regular expressions are in the same spirit, but not shown for brevity. The interested reader is referred to [44, 45, 46, 47].

Theorem 7 ([26]). [1] *Let f be a formula in* LTL *augmented with strong and weak next operators and strong and weak propositions, let c be a Boolean expression and w an infinite, finite, or empty word. Let $w|_c$ denote the word obtained from w after leaving only the letters that satisfy c, and let $\overset{c}{\models}$ be as defined in Figure 8. Then*

$$w \overset{c}{\models} f \quad \textit{if and only if} \quad w|_c \models f$$

Note that Theorem 7 refers to unclocked formulas, thus shows that intuition regarding a projection holds for singly clocked formulas. Recall that for multiply clocked formulas, we wanted (and achieved) something different than a projection.

The clock operator does not add expressive power.

Theorem 8 ([26]). *The logic consisting of* LTL *plus the @ operator is as expressive as* LTL.

The proof of Theorem 8 is by rewriting, thus direct treatment of the @ operator is not required of a tool. However, it can be advantageous to do so, as shown by [56].

In LTL, $[f \ \mathrm{U}\ g]$ can be defined as a least solution of the equation $S = g \vee (f \wedge \mathrm{X}!\, S)$. In the semantics of Figure 8, there is a fixed point characterization if f and g are themselves unclocked, because $[f \ \mathrm{U}\ g]@c \equiv (\mathrm{TRUE}! \wedge g) \vee (f \wedge \mathrm{X}![f \ \mathrm{U}\ g])@c$. But if f and g contain clock operators, this equivalence no longer holds [26].

As shown in [29], this can be fixed by adding an *alignment operator*, $\mathrm{X}!^0$, that moves to the closest clock tick. The semantics of the alignment operator, adopted by PSL and SVA and shown in Figure 9, gives the following fixed point characterization of until under a clocked semantics.

Theorem 9 ([29]). *Let f and g be formulas of* LTL *augmented with the clock operator. Then:*

[1] Theorems 7 and 8 were stated and proved in [26]. Later, automated proofs of these theorems were obtained after an embedding of the semantics of PSL into HOL [35].

Let f be a formula in LTL extended with the clock operator @, and let w be a word over $\Sigma = 2^P$. For finite word w, let *w is m clock ticks*, for $m > 0$, denote that $w \in \mathcal{L}((\neg c^* \cdot c)[^*m])$. Then, for $i \geq 0$:

- $w \models^c \mathrm{X}!^i f \iff \exists j < |w|$ s.t. $w(0..j)$ is $i+1$ clock ticks of c and $w(j..) \models^c f$

Fig. 9 Generalizing the *next* operator [29], adopted by PSL1850-2010 [45] and SVA1800-2009 [47]

$[f \ \mathrm{U} \ g]$ *is a least solution of the equation* $S = \mathrm{X}!^0 \ (g \vee (f \wedge \mathrm{X}! \, S)$
$[f \ \mathrm{W} \ g]$ *is a greatest solution of the equation* $S = \mathrm{X}^0 \ \ (g \vee (f \wedge \mathrm{X} \ \ S)$

3.2 Using a clock as a time sampling abstraction

The clock operator is a time sampling abstraction, and as such has uses other than representing a hardware clock. For example, consider the property that consecutive writes (signal *write* is asserted) cannot both be high priority writes (signal *high* is asserted). Without the clock operator, this can be expressed as follows

$$\mathrm{G}((write \wedge high) \rightarrow \ \mathrm{X} \ [\neg write \ \mathrm{W} \ (write \wedge \neg high)]) \tag{19}$$

However, that is quite cryptic. Using the clock operator, we can express it more simply as:

$$\mathrm{G}(high \rightarrow \ \mathrm{X} \ \neg high) @ write \tag{20}$$

4 Hardware Resets and Other Sources of Truncated Paths

A *path* of a model M with transition relation R is a finite or infinite sequence of states $(s_0, s_1, \cdots s_n)$ or $(s_0, s_1, \cdots)$, such that each successive pair of states (s_i, s_{i+1}) is an element of R. A path of model M is maximal if either it is infinite, or the last state of the path has no successor in M.

Traditionally, the semantics of temporal logic is defined over infinite words corresponding to infinite paths in the model (see for example Chapter 2), and indeed hardware is a reactive system, whose paths are intrinsically infinite. It turns out, however, that hardware designers do find themselves needing to reason over finite paths, specifically, over *truncated paths* – paths that are finite, but not necessarily maximal.

The need to reason over truncated paths arises in several different contexts. Incomplete methods of verification, such as bounded model checking or dynamic verification, naturally reason over truncated paths. A hardware reset can also be understood as truncating a path, because anything that comes after should not affect the truth value of the property on the infinite path. Finally, hardware clocks are related

to truncated paths, because a clock that stops ticking gives that the projection of an infinite path may be finite. Early definitions of a clock operator struggled with this issue, which was solved by the truncated semantics, leaving the clock operator to define a projection without worrying about whether or not the clock ticks infinitely often.

In this section, we start with a presentation of hardware resets in Section 4.1. In Section 4.2 we move to a discussion of truncated paths arising from other sources, and then show that hardware resets are a particular case of truncated paths. Section 4.3 discusses the surprising relation of truncated paths to the classification of safety formulas.

4.1 Hardware Resets

The User's Point of View Consider the property that if a high priority request is received (signal *hi_rq* is asserted) then one of the next two grants (assertion of signal *gnt*) will be to the high priority destination (signal *hi_dt* is asserted). In LTL, this can be expressed as:

$$\mathrm{G}(hi\_rq \rightarrow [\neg gnt \ \mathrm{W} \ (gnt \wedge ((hi\_dt) \vee \ \mathrm{X}[\neg gnt \ \mathrm{W} \ (gnt \wedge hi\_dt)]))]) \tag{21}$$

although the user will usually prefer to use the suffix implication operator to obtain the much simpler form:

$$\mathrm{G}(hi\_rq \mapsto (gnt[\rightarrow 1..2] \circ hi\_dt)) \tag{22}$$

Now, if the reset signal (*rst*) is to be taken into consideration, such that when signal *rst* is asserted, all outstanding requirements are cancelled, and reckoning begins again at the next deassertion of *rst*, then Formula 21 must be modified as follows:

$$\begin{aligned} &\mathrm{G}((hi\_rq \wedge \neg rst) \rightarrow \\ &\quad [\neg gnt \ \mathrm{W} \ (rst \vee (gnt \wedge (hi\_dt \vee \ \mathrm{X}[\neg gnt \ \mathrm{W} \ (rst \vee (gnt \wedge hi\_dt))]))))]) \end{aligned} \tag{23}$$

This is quite cumbersome, and while modifying Formula 22 in a similar manner is less verbose:

$$\mathrm{G}((hi\_rq \circ gnt[\rightarrow 1..2]) \cap ((\neg rst)^*)) \mapsto hi\_dt \tag{24}$$

it requires moving part of the formula from the right to the left hand side of the suffix implication, which can be problematical if the original LTL version had used strong until. The user would like an easier way, so that Formulas 21 and 22 can be reset simply like this:

$$\mathrm{G}(hi\_rq \rightarrow [\neg gnt \ \mathrm{W} \ (gnt \wedge (hi\_dt \vee \ \mathrm{X}[\neg gnt \ \mathrm{W} \ (gnt \wedge hi\_dt)]))]) \ \mathrm{RESET} \ rst \tag{25}$$

$$\mathrm{G}(hi\_rq \mapsto (gnt[\rightarrow 1..2] \circ hi\_dt)) \ \mathrm{RESET} \ rst \tag{26}$$

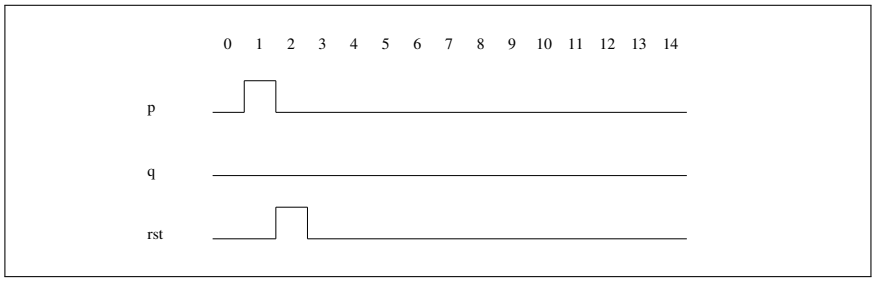

Fig. 10 Weak until vs. reset. Formula 27 does not hold, but Formula 28 does.

Semantic Issues Intuitively, the desired semantics of the reset operator is that up until the reset condition, "nothing has yet gone wrong". So, informally, we want $w \models \varphi$ RESET b to hold iff φ holds on w truncated at the first occurrence of b. Before continuing, it is instructive to compare the desired semantics of the reset operator with the semantics of the weak until operator. Consider

$$[(p \rightarrow \mathrm{XXX}\, q)\ \mathrm{W}\ rst] \tag{27}$$

and

$$(\mathrm{G}(p \rightarrow \mathrm{XXX}q))\ \mathrm{RESET}\ rst \tag{28}$$

on a word w such that p holds on (and only on) $w(1)$, q holds on no letter, and *rst* holds on (and only on) $w(2)$. This is illustrated in Figure 10. Then Formula 27 does not hold on w, but we want that Formula 28 does hold on w. Thus the *rst* in $(\mathrm{G}\ \varphi)$ RESET *rst* can be thought of as "canceling future obligations of φ", as opposed to $[\varphi\ \mathrm{W}\ rst]$, in which the "obligations" of φ may extend beyond the first occurrence of *rst*.

It is tempting to define that φ RESET b holds on w if φ holds on w or if there exist words u and v and letter ℓ such that b holds on ℓ, $w = u\ell v$, and u has an extension on which φ holds. However, such a definition does not give us what we want for reasons similar to those discussed in Section 2.3 regarding the definition of weak regular expressions. For example, we would like $[busy\ \mathrm{U}\ \mathrm{FALSE}]$ RESET b to hold on a word where b holds on some letter and *busy* holds on every letter up until the letter where b holds, but a definition that looks for an extension on which $[busy\ \mathrm{U}\ \mathrm{FALSE}]$ holds does not give us that. Furthermore, the complexity of model checking using such a definition is non-elementary [4].

The solution of [6] was to add two contexts, the *accept* condition and the *reject* condition. The resulting semantics is shown in Figure 11, and following [4], we term these the *reset* semantics. A formula φ holds in a model if $\langle w, \mathrm{FALSE}, \mathrm{FALSE}\rangle \models \varphi$ for every word in the model. The complexity of the reset semantics is no different than that of LTL, as stated by Theorem 10.

Theorem 10 ([4]). *The satisfiability and model-checking problems for* LTL *plus the reset operator under the reset semantics are PSPACE-complete.*

Let φ and ψ be formulas of LTL extended with the RESET operator, let a and r be Boolean expressions, and let w be a word over $\Sigma = 2^P$.

- $\langle w,a,r\rangle \models p \iff w(0) \Vdash a \vee (p \wedge \neg r)$
- $\langle w,a,r\rangle \models \neg\varphi \iff \langle w,r,a\rangle \not\models \varphi$
- $\langle w,a,r\rangle \models \varphi \wedge \psi \iff \langle w,a,r\rangle \models \varphi$ and $\langle w,a,r\rangle \models \psi$
- $\langle w,a,r\rangle \models \mathrm{X}\ \varphi \iff w(0) \Vdash a$ or both $w(0) \not\Vdash r$ and $\langle w(1..),a,r\rangle \models \varphi$
- $\langle w,a,r\rangle \models [\varphi\ \mathrm{U}\ \psi] \iff \exists k < |w|$ s.t. $\langle w(k..),a,r\rangle \models \psi$, and $\forall j < k,\ \langle w(j..),a,r\rangle \models \varphi$
- $\langle w,a,r\rangle \models \varphi\ \mathrm{RESET}\ b \iff \langle w,a \vee (b \wedge \neg r),r\rangle \models \varphi$

Fig. 11 The reset semantics [6, 4]

The reset operator does not add expressive power. Automata construction for the reset operator, as well as rewrite rules for translating LTL plus the reset operator into LTL are given by [4].

Theorem 11 ([4]). *The logic consisting of* LTL *plus the reset operator is as expressive as* LTL.

More insight into the reset semantics can be found in the next sub-section, which shows two equivalent statements of the reset semantics.

4.2 Other Sources of Truncated Paths

The User's Point of View Model checking is a tool, but not the only tool. Users want a specification language that is equally relevant to every verification method in their toolbox. Consider the property "every request must receive a grant, and once asserted, the request signal must stay asserted until it receives its grant". Typically we would specify this in LTL as follows:

$$\mathrm{G}(\mathit{request} \rightarrow \mathrm{X}\ [\mathit{request}\ \mathrm{U}\ \mathit{grant}]) \tag{29}$$

However, when using an incomplete method of verification, such as bounded model checking or simulation, the user has a decision to make. Is it possible that the verification run will end in between a request and its grant, so that

$$\mathrm{G}(\mathit{request} \rightarrow \mathrm{X}\ [\mathit{request}\ \mathrm{W}\ \mathit{grant}]) \tag{30}$$

should be used instead?

The answer does not depend on the design under verification, but it does depend on properties of the verification method. For instance, some simulation tests are designed to continue until correct output can be confirmed, and then the answer is that Formula 29 should be used. On the other hand, some tests have no "opinion" on the correct length of a test, and they may end at any time. In such a case, Formula 29 might result in a false negative, and Formula 30 should be used instead.

Let φ and ψ be LTL formulas, let b be a Boolean expression, and let w be a word over $\Sigma = 2^P$. Allow "overflow" of indices: for $k \geq |w|$, let $w(k..) = \varepsilon$.

holds weakly: For w over 2^P such that $|w| \geq 0$,

- $w \models^{-} b \iff |w| = 0$ or $w(0) \Vdash b$
- $w \models^{-} \neg\varphi \iff w \not\models^{+} \varphi$
- $w \models^{-} \varphi \wedge \psi \iff w \models^{-} \varphi$ and $w \models^{-} \psi$
- $w \models^{-} \mathrm{X!}\, \varphi \iff w(1..) \models^{-} \varphi$
- $w \models^{-} [\varphi\ \mathrm{U}\ \psi] \iff \exists k$ such that $w(k..) \models^{-} \psi$ and for every $j < k,\ w(j..) \models^{-} \varphi$

holds neutrally: For w over 2^P such that $|w| > 0$,

- $w \models b \iff w(0) \Vdash b$
- $w \models \neg\varphi \iff w \not\models \varphi$
- $w \models \varphi \wedge \psi \iff w \models \varphi$ and $w \models \psi$
- $w \models \mathrm{X!}\, \varphi \iff |w| > 1$ and $w(1..) \models \varphi$
- $w \models [\varphi\ \mathrm{U}\ \psi] \iff \exists k < |w|$ such that $w(k..) \models \psi$ and for every $j < k,\ w(j..) \models \varphi$

holds strongly: For w over 2^P such that $|w| \geq 0$,

- $w \models^{+} b \iff |w| > 0$ and $w(0) \Vdash b$
- $w \models^{+} \neg\varphi \iff w \not\models^{-} \varphi$
- $w \models^{+} \varphi \wedge \psi \iff w \models^{+} \varphi$ and $w \models^{+} \psi$
- $w \models^{+} \mathrm{X!}\, \varphi \iff w(1..) \models^{+} \varphi$
- $w \models^{+} [\varphi\ \mathrm{U}\ \psi] \iff \exists k$ such that $w(k..) \models^{+} \psi$ and for every $j < k,\ w(j..) \models^{+} \varphi$

Fig. 12 The truncated semantics of LTL [24]

The answer might depend on properties of the verification method, but designers want the specification to specify the design, and be reusable no matter the method of verification. Thus it is desirable for a specification language to provide a means to distinguish between properties of the design and properties of the verification.

Semantic Issues Distinguishing between properties of the design and properties of the verification can be achieved by defining separate *views* of the same formula, and distinguishing between maximal and non-maximal, or *truncated*, words [24]. The formula itself describes the design, while each view is useful in a different verification method. The views differ only on finite words, and only in cases in which there is doubt whether the formula holds on the original, possibly unknown, untruncated word. For instance, consider the formula F p on a truncated word such that p does not hold on any letter, or the formula G q on a truncated word such that q holds on every letter. In both cases it is impossible to know whether or not the formula holds on the original, untruncated, word.

In such cases, in which there is doubt as to whether the formula holds on the original word, the truncated semantics, shown in Figure 12, define that it holds in the

Let φ be a formula of LTL extended with the RESET operator, let b be a Boolean expression, and let w be a word over $\Sigma = 2^P$.

- $w \models^{-} \varphi$ RESET $b \iff$ either $w \models^{-} \varphi$ or $\exists j < |w|$ s.t. $w(j) \Vdash b$ and $w(0..j-1) \models^{-} \varphi$
- $w \models \varphi$ RESET $b \iff$ either $w \models \varphi$ or $\exists j < |w|$ s.t. $w(j) \Vdash b$ and $w(0..j-1) \models^{-} \varphi$
- $w \models^{+} \varphi$ RESET $b \iff$ either $w \models^{+} \varphi$ or $\exists j < |w|$ s.t. $w(j) \Vdash b$ and $w(0..j-1) \models^{-} \varphi$
- All other LTL operators are as usual.

Fig. 13 The truncated semantics of LTL extended with the RESET operator [24]

weak view, does not hold in the *strong view*, and holds in the *neutral view* iff it holds in the traditional LTL semantics on finite words (this is equivalent to considering the word to be maximal rather than truncated). Note that Figure 12 does not show three separate semantics, but rather a single semantics, in which the view is a context. Thus $w \models^{=} \varphi$, $w \models \varphi$ and $w \models^{\pm} \varphi$ should be understood as shorthand for $\langle w, \text{weak} \rangle \models \varphi$, $\langle w, \text{neutral} \rangle \models \varphi$ and $\langle w, \text{strong} \rangle \models \varphi$, respectively.

Let us now return to the example represented by Formulas 29 and 30. The specification should describe the design, thus Formula 29 is the formula that should be used. In the case of a test that was designed to continue until correct output can be confirmed, Formula 29 can be checked under the neutral view. In the case that the test may end at any time, Formula 29 can be checked under the weak view. Note that most formulas do not hold under the strong view on finite words, because for any φ, $\mathrm{G}\varphi$ does not hold under the strong view on such a word. Thus the main purpose of the strong view is to serve as a dual to the weak.

As shown in [24], the truncated semantics has the property that the strong view is stronger than the neutral view, which is in turn stronger than the weak view, as stated by Theorem 12 below. It also supports the intuition that φ holds weakly on w if up till now nothing "has gone wrong", and thus holds as well on any prefix of w, and that φ holds strongly on w if "all future obligations have been met", and thus holds as well on any extension of w. This is stated by Theorem 13 below.

Theorem 12 (Strength relation theorem [24]). *Let w be a non-empty word. Then:*

$$\bullet\ w \models^{+} \varphi \Longrightarrow w \models \varphi \qquad \bullet\ w \models \varphi \Longrightarrow w \models^{-} \varphi$$

Theorem 13 (Prefix/extension theorem [24]).

$$\bullet\ v \models^{+} \varphi \Longleftrightarrow \forall w \succeq v : w \models^{+} \varphi \qquad \bullet\ v \models^{-} \varphi \Longleftrightarrow \forall u \preceq v : u \models^{-} \varphi$$

The reset operator, discussed in detail in the previous subsection, is easily stated in the truncated semantics, as a reset can be understood as truncating an infinite word, and moving to the weak view. Figure 13 shows the formal statement of the truncated semantics of the reset operator.

The reset semantics and the truncated semantics accomplish their goal by restating the semantics of every operator. This complicates things considerably in a temporal logic that has many core operators (as is the case in PSL and SVA). Using ideas

Let φ be a formula of LTL extended with the RESET operator, let b be a Boolean expression. Let $\top$ and $\bot$ be special letters such that $\top \Vdash b$ and $\bot \nVdash b$ for any b (including TRUE and FALSE). Let w be a word over $\Sigma = 2^P \cup \{\top, \bot\}$. Let $\overline{w}$ denote the word obtained from w by switching every $\top$ with $\bot$ and vice versa.

- $w \models \neg\varphi \iff \overline{w} \not\models \varphi$
- $w \models \varphi$ RESET $b \iff$ either $w \models \varphi$ or $\exists j < |w|$ such that $w(j) \Vdash b$ and $w(0..j-1)\top^\omega \models \varphi$
- All other LTL operators are as usual.

Fig. 14 $\top,\bot$ approach to the truncated semantics [25], adopted by PSL1850-2005 [44] and SVA [46, 47]

from [22] discussed in Section 2.3 one can restate the semantics more succinctly without requiring a restatement of the semantics of each operator. This approach, makes use of the special letters $\top$ and $\bot$. Recall that $\top$ satisfies all Boolean expressions including FALSE while $\bot$ satisfies no Boolean expression, not even TRUE. The resulting semantics, referred to as the $\top,\bot$ approach, is given in Figure 14.

Theorem 14 ([24, 25]). *Let φ be a formula of* LTL *extended with the reset operator. Let w be a word over $\Sigma = 2^P$. Then $\langle w, \text{FALSE}, \text{FALSE}\rangle \models \varphi$ in the reset semantics iff $w \models \varphi$ in the truncated semantics iff $w \models \varphi$ in the $\top,\bot$ approach to the truncated semantics.*

As mentioned in Section 2.3, the $\top, \bot$ approach breaks when regular expression intersection and fusion are included [21, 22]. Thus the semantics of PSL 1850-2010 [45] uses a version of the truncated semantics that supports SEREs, defined in [21], characterized in [23], and shown in Figures 5 and 6.

4.3 The truncated semantics and classification of safety formulas

A *bad prefix* of a safety formula φ is a finite word w all of whose infinite extensions violate φ. An *informative prefix*, defined syntactically in [51] in order to classify safety properties, is, loosely speaking, a bad prefix that provides enough information to "explain" why φ does not hold. For instance, an informative prefix of $\varphi = \text{X!X!}p$ would be a word of length at least 3 where p does not hold on the third letter. The formula $\psi = \text{FG}p \wedge \text{FG}\neg p$ has no informative prefixes, since no finite prefix is long enough to "explain" why it fails. This, despite the fact that ψ is a contradiction, thus every finite word is a bad prefix of it.

The motivation for the classification of safety formulas is complexity. For general LTL properties, the automata theoretic approach to model checking [54, 67, 73] builds the product of the design under verification and a Büchi automaton for the negation of the property and checks for emptiness. For safety properties it suffices to work with automata on finite words (see Chapter 2), and an automaton on finite words recognizing bad prefixes can also be used by simulation, simply by means

of translating it into a checker coded in some Hardware Description Language [1]. Model checking with a finite automaton rather than a Büchi automaton should be easier since it replaces a search for fair cycles with an invariant check. However, the complexity results tell a different story. For an arbitrary LTL formula φ, the equivalent Büchi automaton is of size exponential in $|\varphi|$, while for a safety LTL formula φ, the size of an automaton recognizing its bad prefixes is doubly exponential in $|\varphi|$.

Theorem 15 ([73]). *Let φ be an* LTL *formula of size n. Then there exists a nondeterministic Büchi automaton of size $2^{O(n)}$ recognizing $[\![\varphi]\!]$.*

Theorem 16 ([51]). *Let φ be an* LTL *formula of size n. The size of an* NFA *recognizing the bad prefixes of φ is $2^{2^{O(n)}}$ and $2^{2^{\Omega(\sqrt{n})}}$.*

As shown by [51], if we are willing to limit ourselves to recognizing all informative, rather than bad, prefixes, we return to a single exponent.

Theorem 17 ([51]). *Let φ be a safety* LTL *formula of size n. There exists an* NFA *of size $2^{O(n)}$ recognizing all the informative prefixes of φ.*

The question then becomes whether we can use the automaton for informative prefixes instead of the one for bad prefixes. As noted in [51], it suffices to recognize a single bad prefix for each violating word. Thus they classify safety properties according to whether all, some, or none of its bad prefixes are informative. A property is *intentionally safe* if all of its bad prefixes are informative. A property φ is *accidentally safe* if every computation that violates it has an informative prefix. A property is *pathologically safe* if there is a computation that violates φ and has no informative prefix. It follows from Theorem 17 that for non-pathological formulas, an automaton of exponential (rather than doubly exponential) size exists that recognizes at least one bad prefix for each violating word, matching our intuition that (non-pathological) safety formulas are easier to check than general LTL formulas.

The motivation for the classification of safety formulas was very different from the motivation for the truncated semantics, but it turns out that the two are related in an interesting way. While [51] defines an informative prefix syntactically, the truncated semantics provide a semantic definition:

Theorem 18 ([24]). *Let φ be an* LTL *formula and w a finite non-empty word. Then w is an informative prefix for φ iff $w \not\models^{-} \varphi$.*

Furthermore, the notion of semantic weakness discussed in Section 2.3 and captured by the truncated semantics provides a semantic characterization of the set of safety properties that are computationally easy to verify:

Theorem 19 ([23, 22]). *Let φ be an* LTL *formula. Then φ is semantically weak iff it is non-pathologically safe.*

Let b be a Boolean expression and r a regular expression. Then RLTL^{lv} consists of the formulas defined by the following grammar:

$$\varphi ::= b \mid \varphi \wedge \varphi \mid \mathrm{X}\ \varphi \mid (b \wedge \varphi) \vee (\neg b \wedge \varphi) \mid [(b \wedge \varphi)\ \mathrm{W}\ (\neg b \wedge \varphi)] \mid r \mapsto \varphi$$

Fig. 15 RLTL^{lv}, a syntactic subset of LTL extended with suffix implication (see Section 2.1)

5 The Simple Subset

In Section 4.3 we saw a subset of safety formulas that are computationally easy to verify, relative to all of LTL. Finding a syntactic subset for which simulation and model checking is guaranteed to be even more efficient was the intention of the *simple subset* of PSL, defined in [44, 45]. Figure 15 shows RLTL^{lv}, a syntactic subset of LTL extended with suffix implication (see Section 2.1), defined in [10] and subsuming the fragment of the simple subset that uses only weak operators. The *lv* of RLTL^{lv} stands for *linear violation*, and as we shall see below, formulas of RLTL^{lv} can be model checked using an NFA that is linear in the size of the formula, rather than exponential as for arbitrary non-pathologically safe formulas.

Intuitively, time flows from left to right through formulas in the simple subset, and syntactically every temporal binary operator has only one non-temporal operand. In this respect, the simple subset shares much in common with the subset of RCTL formulas that can be model checked using invariance checking [8] (RCTL is an extension of CTL with suffix implication). It is also reminiscent of the syntactic subset LTL^{det} of LTL described by [57], in which every formula that can be expressed in both LTL and ACTL has an equivalent. In fact, the restriction of PSL's simple subset to LTL formulas is a subset of LTL^{det}.

While the motivation in [57] was expressiveness, it was also shown there that formulas in LTL^{det} have a 1-weak Büchi automaton of linear size, where a 1-weak Büchi automaton is a Büchi automaton in which every strongly connected component is a singleton. The structure of 1-weak Büchi automata make them more efficiently checkable than general Büchi automata [12, 61], making formulas in the simple subset more efficiently checkable than formulas not in the subset.

Theorem 20 ([57]). *Let φ be an* LTL^{det} *formula of size n. There exists a 1-weak Büchi automaton of size $O(n)$ recognizing $[\![\varphi]\!]$.*

The analogous result for RLTL^{lv} provides the promised efficiency for both simulation and model checking:

Theorem 21 ([10]). *Let φ be an* RLTL^{lv} *formula of size n. There exists an* NFA *of size $O(n)$ recognizing the informative prefixes of φ.*

6 Quantified and Local Variables

Both PSL and SVA provide quantified and local variables. Quantified variables, discussed in Section 6.1, are variables in the familiar sense in logic. Local variables, discussed in Section 6.2, are different – the intuition behind them borrows from programming languages. Intuitively, local variables are a mechanism for both declaring quantified variables and constraining their behavior.

6.1 Quantified Variables

The User's Point of View Suppose that we want to check that every read request returns correct data, in a design that uses tagged requests. In such a design, every request gets an identifying number (the *tag*), which is used to identify the corresponding data. Thus we want to check that if there is a read request to address a, tagged with tag t, and the value of the data at address a is d, then the next time that data for tag t appears on the data bus, the value of the data is d. Conceptually, the formula we want to check is that for every value of a, t and d,

$$\begin{aligned} &\mathrm{G}(((addr = a) \wedge (tag = t) \wedge (mem(a) = d)) \rightarrow \\ &\quad [\neg(data\_valid \wedge (tag = t))\ \mathrm{W}\ (data\_valid \wedge (tag = t) \wedge (data = d))]) \end{aligned} \tag{31}$$

If an address is 32 bits wide, a tag is 8 bits wide, and data is 128 bits wide, then without quantifiers, we will need to write 2^{168} LTL formulas. Thus the user wants quantified variables, allowing the 2^{168} formulas to be expressed more succinctly in a single formula.

Semantic Issues Formula 31 requires quantification over *rigid* variables — variables whose valuation stays constant over time. Clearly, adding quantification over such variables does not increase the expressive power (though as discussed above it adds much succinctness). Both PSL and SVA support quantification over rigid variables, allowing the 2^{168} formulas referred to above to be expressed as a single formula, as follows:

$$\begin{aligned} &\forall a \in \mathbb{B}^{32}\ \forall t \in \mathbb{B}^{8}\ \forall d \in \mathbb{B}^{128} \\ &\mathrm{G}(((addr = a) \wedge (tag = t) \wedge (mem(a) = d)) \rightarrow \\ &\quad [\neg(data\_valid \wedge (tag = t))\ \mathrm{W}\ (data\_valid \wedge (tag = t) \wedge (data = d))]) \end{aligned} \tag{32}$$

Adding quantification over *flexible* variables — variables that may have different values at different time points — increases the expressive power of LTL to that of ω-regular languages. While LTL cannot count (a fact known through [32, 48, 59]), LTL extended with universal/existential quantification over flexible variables, henceforth referred to as QLTL, can. Indeed, the following QLTL formula:

$$\forall t.(t \wedge \mathrm{G}(t \leftrightarrow \mathrm{X}\neg t)) \rightarrow \mathrm{G}(t \rightarrow p) \tag{33}$$

expresses that p holds at every even position, a property expressible in PSL and SVA using regular expressions, but not expressible in LTL [75] (see also Chapter 2). Note that here t is a variable standing for an atomic proposition whose behavior over time is unknown, whereas rigid variables play the role of constants rather than atomic propositions.

QLTL was introduced in [66]. It was shown in [68] that its expressive power is ω-regular and the complexity of the satisfiability problem is non-elementary. In [49] it was argued that QLTL is also important for reasoning about abstraction refinement methods, and a complete axiomatic proof system for this logic was provided.[2] The QLTL property expressed by Formula 33 can practically be checked by a model checker for plain LTL by introducing an unconstrained auxiliary variable t and checking the formula $(t \wedge \mathrm{G}(t \leftrightarrow \mathrm{X}\ \neg t)) \rightarrow (\mathrm{G}(t \rightarrow p))$. Obviously this is not a general solution, for example if quantifications are nested with alternating quantifiers.

6.2 Local Variables

The User's Point of View Consider now the property that every request must receive a unique grant, where n is the maximum number of requests that may be outstanding (i.e., that have not yet received a grant). Designate a request by r and a grant by g, and assume for simplicity that requests and grants are mutually exclusive. Then for $n = 1$, the property can easily be expressed as

$$\mathrm{G}\ (r \rightarrow \mathrm{X}[\neg r\ \mathrm{U}\ g]) \tag{34}$$

For $n > 1$, we can turn to the power of regular expressions. If we can express the set of shortest counterexamples by a regular expression, then we can assert that that regular expression never occurs. As previously, let r stand for request and g for grant, and let e stand for "else", that is, $\neg r \wedge \neg g$. Then the set of shortest counterexamples for $n = 2$ is the language of the following regular expression:

$$e^* \cdot r \cdot (e \cup (g \cdot e^* \cdot r) \cup (r \cdot e^* \cdot g))^* \cdot r \cdot e^* \cdot r \tag{35}$$

and for $n = 3$ it is the language of this:

$$e^* \cdot r \cdot ((g \cdot e^* \cdot r) \cup e \cup (r \cdot (e \cup (r \cdot e^* \cdot g))^* \cdot g))^* \cdot r \cdot (e \cup (r \cdot e^* \cdot g))^* \cdot r \cdot e^* \cdot r \tag{36}$$

As n grows, the set of shortest counterexamples (and thus our property) turns out to be surprisingly difficult to express. In contrast, it is very easy to construct a non-deterministic finite automaton for a particular n. The user would like the same ease of expression inside the specification language. For instance, she would like to

[2] While [49] considers a logic with past operators, [31] enhances the result to a logic without them.

be able to say the following:

$$\text{NEW}(v{=}0)\ \ (((r \vee g)[\rightarrow],\ v{=}(r\ ?\ \ v{+}{+}\ :\ v{-}\,{-}))^*\ \ \mapsto\ \ ((v \geq 0) \wedge (v \leq n))) \qquad (37)$$

where $b[\rightarrow]$ abbreviates $\neg b^* \cdot b$ and = stands for assignment. Formula 37 declares a *local variable* v, controlled from within the regular expression. Its initial value is 0, and it is incremented every time there is a request (r) and decremented every time there is a grant (g). The formula holds iff v stays in the range $0 \leq v \leq n$.

Semantic Issues Intuitively, local variables can be seen as a mechanism for both declaring quantified variables and constraining their behavior. For instance, consider the following formula using local variables:

$$\text{NEW}(i,j)\ \ ((a, i{=}0, j{=}0) \cdot (b, i{+}{+})^* \cdot (c, j{+}{+})^* \cdot (i{=}{=}j)) \mapsto d \qquad (38)$$

It states that a sequence consisting of an a followed by some number of b's followed by the same number of c's should be followed by a d. It uses the local variables i and j to make sure the same number of b's and c's are observed. If i and j are unbounded, local variables may increase the expressive power to that of context-free languages. In practice, however only bounded local variables are considered. Restricting attention to bounded local variables, there is no increase in expressiveness over ω-regular languages, but there is an increase in succinctness. Use of local variables was advocated in [62, 65].

When we try to formulate the semantics of a logic with local variables we confront the issue that for a given word w and a given regular expression r we may want more than one value of a local variable in each letter of w. For instance, consider Formula 38 and a word where a holds on the first letter and only on the first letter, and both b and c hold on the second to fifth letters and only on the second to fifth letters. What is the value of i and j in the fifth letter? It could be $i = 2, j = 2$ by matching the b's on the second and third letters and the c's on the fourth and fifth letters; it could also be $i = 3, j = 1$ by matching b's on the second to fourth letters and c on the fifth letter. Several other options are possible as well. For the formula to hold d should hold in both third and fifth letters — the locations where i and j may be equal.

Another issue one confronts when formulating the semantics has to do with the intersection operator. Suppose we want a SERE whose language includes only words where the number of a's between s and e equals the number of b's between s and e. Consider the following formula:

$$\text{NEW}(i)\ \ ((s, i{=}0)\ \cdot\ ((\neg a^* \cdot (a, i{+}{+}))^*\ \ \cap\ \ (\neg b^* \cdot (b, i{+}{+}))^*)\ \cdot e) \qquad (39)$$

Using simple intersection will require the a's and the b's to hold on the same cycles on the words being intersected, which is not what we want.

For this reason, the idea in the first formal definition for a logic augmented with local variables, given in [15, 46] was to control local variables only at the beginning and end of the word. Regular expression semantics is defined with respect to a word

Let V be a set of local variables and let $\mathrm{Z} \subseteq V$. Let D be the domain of the local variables and let $\gamma_1, \gamma_2 \in D^V$ be assignments to the local variables. Let $\gamma_1 \overset{\mathrm{Z}}{\sim} \gamma_2$ (read "γ_1 agrees with γ_2 relative to Z") denote that for every $z \in \mathrm{Z}$ we have that the value of variable z in valuation γ_1 is the same as its value in γ_2. For an expression e, and an enhanced letter $\mathtt{a}$, the notation $e[\mathtt{a}|_{\sigma\gamma}]$ provides the value of e with respect to the σ and γ components of $\mathtt{a}$. The notation $\mathtt{v} \models_{\mathrm{Z}} r$ means that the good enhanced word $\mathtt{v}$ models tightly r with respect to controlled variables Z.

- $\mathtt{v} \models_{\mathrm{Z}} b, x{=}e \iff |\mathtt{v}| = 1$ and $b[\mathtt{v}(0)|_{\sigma\gamma}] = \mathrm{T}$ and $\mathtt{v}(0)|_{\gamma'} \overset{\mathrm{Z}}{\sim} \hat{\gamma}$ where $\hat{\gamma}$ results from $\mathtt{v}(0)|_{\gamma}$ by assiging $e[\mathtt{v}(0)|_{\sigma\gamma}]$ to x
- $\mathtt{v} \models_{\mathrm{Z}} r_1 \cdot r_2 \iff \exists \mathtt{v}_1, \mathtt{v}_2$ such that $\mathtt{v} = \mathtt{v}_1\mathtt{v}_2$ and $\mathtt{v}_1 \models_{\mathrm{Z}} r_1$ and $\mathtt{v}_2 \models_{\mathrm{Z}} r_2$
- $\mathtt{v} \models_{\mathrm{Z}} r_1 \cup r_2 \iff \mathtt{v} \models_{\mathrm{Z}} r_1$ or $\mathtt{v} \models_{\mathrm{Z}} r_2$
- $\mathtt{v} \models_{\mathrm{Z}} r_1 \cap r_2 \iff \mathtt{v} \models_{\mathrm{Z}} r_1$ and $\mathtt{v} \models_{\mathrm{Z}} r_2$
- $\mathtt{v} \models_{\mathrm{Z}} \{\mathrm{NEW}(\mathrm{Y})\ r\} \iff \mathtt{v} \models_{\mathrm{Z}\cup\mathrm{Y}} r$
- $\mathtt{v} \models_{\mathrm{Z}} \{\mathrm{FREE}(\mathrm{Y})\ r\} \iff \mathtt{v} \models_{\mathrm{Z}\setminus\mathrm{Y}} r$

Fig. 16 Parts of the PSL semantics for SEREs with local variables as per [20, 45]

w and two contexts L_0 and L_1 standing for the values of local variables at the beginning and end of the word, respectively. It was shown in [41] that this semantics breaks distributivity of union over intersection.

This drawback was addressed in the definition proposed in [20] and adopted by PSL. Parts of the semantics of SEREs with local variables as per [20] are given in Figure 16. It is argued in [20] that any semantics that try to automatically divide the responsibility for the set of controlled variables between the SERE operands of intersection and union, will break distributivity or another basic algebraic property. Instead, the approach was to define the semantics with respect to (1) *good enhanced words* and (2) a set of *controlled local variables*.

The semantics without local variables is defined with respect to words over $\Sigma = 2^P$ where P is the set of atomic propositions. The enhanced alphabet is $\Sigma \times \Gamma \times \Gamma$ where $\Gamma = D^V$, V is the set of local variables and D is their domain. Let $\langle \sigma, \gamma, \gamma' \rangle$ be an enhanced letter. The component σ provides a valuation to the atomic propositions; the component γ provides a valuation to the local variables before assignments are executed; and the component γ' provides a valuation to the local variables after assignments are executed. An enhanced word is *good* if the component γ of a given letter is equivalent to the component γ' of the previous letter (if such a letter exists). Thus, in comparison to [15], local variables are watched at every letter of the word rather than just at the beginning and end of a word.

To deal with the problem imposed by the intersection operator, the semantics is defined with respect to a *set of controlled variables* Z which is a subset of the local variables V corresponding to the variables whose valuation should be observed. A controlled variable should change its value according to an assignment given to it, if such was made, and retain its value otherwise. The values of uncontrolled variables do not play a role in the semantics. The control over which variables should be

Let b be a Boolean expression. Let Y be a sequence of local variables and E a sequence of expressions of the same length as Y. The grammar below defines the formulae φ that compose the practical subset, denoted PSL^{pract}, where the R operator is a dual to strong until: φ R $\psi \equiv \neg[\neg\varphi$ U $\neg\psi]$.

$$r ::= \; b \mid r \cdot r \mid r \cup r \mid r^{+}$$

$$\mathcal{R} ::= b \mid (b, \mathrm{Y} := \mathrm{E}) \mid \mathcal{R} \cdot \mathcal{R} \mid \mathcal{R} \cup \mathcal{R} \mid \mathcal{R}^{+} \mid (\textsc{new}(\mathrm{Y})\ \mathcal{R}) \mid (\textsc{free}(\mathrm{Y})\ \mathcal{R})$$

$$\psi ::= r! \mid \neg r! \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathrm{X}!\psi \mid [\psi\ \mathrm{U}\ \psi] \mid [\psi\ \mathrm{R}\ \psi] \mid r \models \psi$$

$$\varphi ::= \neg\mathcal{R}! \mid \psi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathrm{X}!\varphi \mid [\varphi\ \mathrm{U}\ \psi] \mid [\psi\ \mathrm{R}\ \varphi] \mid \mathcal{R} \models \psi \mid (\textsc{new}(\mathrm{Y})\ \varphi) \mid (\textsc{free}(\mathrm{Y})\ \varphi)$$

Fig. 17 The Practical Subset, PSL^{pract} [5]

controlled is given to the user by means of two operators $\textsc{new}(x)$ and $\textsc{free}(x)$ which add and remove, respectively, a variable from the set of controlled variables.

Theorem 22 ([20]). *The following properties hold for* SEREs *extended with local variables using the semantics of [20]:*

- *The operators $\cup$ and $\cap$ are commutative, and $\cup$, $\cap$ and $\cdot$ are associative.*
- *The operator $\cup$ distributes over $\cap$ and $\cap$ distributes over $\cup$.*
- $r \equiv (r \cup r) \equiv (r \cap r) \equiv (\lambda \cdot r) \equiv (r \cdot \lambda) \equiv (\textsc{false} \cup r) \equiv ((\textsc{free}(V)\ \textsc{true})^* \cap r)$

The addition of local variables increases the complexity of the model checking problem from PSPACE to EXPSPACE [15]. This holds for both approaches to the semantics. Automata construction for the semantics of the first approach was given in [15] and for the second approach in [20]. The lower bound appears in [15].

Theorem 23 ([15, 20]). *Let V be a set of local variables and P be a set of atomic propositions. The satisfiability and model checking problems for formulas of* LTL *extended with suffix implication, local variables and the intersection operator using the semantics of [15] or [20] are EXPSPACE-complete with respect to $|V| \cdot |P|$.*

The source of the increase in complexity lies in the need to track different values for the same local variable on the same time instance of the same run. This issue was analyzed in [5]. It was shown there that in a significant syntactic fragment of PSL this situation will not occur. Given an alternating Büchi automaton for the formula at hand, roughly speaking, we may have more than one value for a certain local variable on a certain time point of the same state (a *local variable conflict*) if there are assignments after a universal branch that loops back. It was shown in [5] that on the subset given in Figure 17, referred to as the *practical subset*, there will be no conflicts, and the complexity of model checking goes back to PSPACE.

Theorem 24 ([5]). *The space complexity of the verification problem of any formula φ in PSL^{pract} is polynomial in $|\varphi|$.*

7 Summary and Open Issues

We have seen that both PSL and SVA extend the expressive power of LTL to that of ω-regular languages by adding regular expressions and the *suffix implication* operator. They both also include a number of specialized operators to allow natural specification of sampling abstractions and truncated paths. Finally, both provide local variables, which can be seen as a mechanism for both declaring quantified variables and constraining their behavior. But neither PSL nor SVA is perfect, nor necessarily done. Below we mention a few directions in which we think there might be room for improvement.

7.1 One-to-one correspondence

The User's Point of View Consider the property that every request must receive a unique grant. This property is a common one, and most hardware designers need to express something of this sort.

Semantic Issues It is well known that the property $\{w \mid |w|_a = |w|_b\}$, where $|w|_\ell$ denotes the number of occurrences of the letter ℓ in word w, is not regular (see for example [43]), and cannot be expressed in any of the logics that we have considered in this chapter. On the one hand, this seems to be a non-issue. In every piece of hardware that obeys the desired property, there can be only a finite number n of requests outstanding (i.e., that have not yet received a grant), and Formulas 34-37 state the property for particular n's. Also, if the requests are tagged with unique identifiers, then the problem goes away.

However, from the user's point of view, the expressive power is not enough. She does not want a formula for a specific n, but rather one stating that there exists such an n. Therein lies an interesting paradox. Although ω-regular expressive power is sufficient to describe the behavior of any particular design, the wish list of the user includes more. This is because we often want a specification to describe not a particular implementation, but a family of implementations, for instance all implementations in which every request receives a unique grant.

7.2 Triggering Procedural Code From Within a Formula

The User's Point of View Temporal logic is such a powerful tool, wouldn't it be nice to harness that power in order to "trigger" procedural code? A simple motivation for this would be to print out debugging information when the calculation "reaches" a particular point in the formula. Another application would be to drive simulation inputs or to collect coverage information.

Such an ability exists in SVA [46, 47], in which a subroutine call is allowed wherever a local variable assignment is allowed, and in PSL 1850-2010 [45], where a *procedural block* is similarly allowed.

Semantic Issues Triggering of procedural code from within a formula can be viewed as an extension of local variables as discussed in Section 6. However, whereas the semantics of local variables are well defined, and given by the formal semantics of SVA and PSL, the semantics of triggering procedural code are not formally defined at all in either specification language.

Doing so is non-trivial, not only because it is difficult to write down the precise, mathematically well-behaved semantics, but also because first it must be decided what they are. For example, what side effects are allowed in the procedural code (i.e., how does the procedural code interact with the design under verification)? And if a property consists of the disjunction of two sub-properties, one of which has already been determined to hold, must we continue to trigger procedural code attached to the other sub-property if it holds as well? And if a regular expression is "matched" in two different ways at the same cycle, do we trigger the relevant procedural code once or twice? Note that any decision on such issues limits the implementation of a tool checking properties in the specification language.

7.3 Separation of Concerns

The User's Point of View Consider the property "If p occurs followed eventually by q, then v will not occur between p and q, will not occur at q, and furthermore will occur one cycle after q". This can be expressed in LTL as follows:

$$\mathrm{G}(p \rightarrow ([\neg v \ \mathrm{W} \ (q \wedge \neg v)] \wedge [\neg q \ \mathrm{W} \ (q \wedge \mathrm{X} v)])) \tag{40}$$

However, this formulation blurs the distinction between cause and effect: in the English language description of the formula it is quite clear that the "responsibility" for the property lies with signal v, but in an LTL formula all variables are created equal, so to speak.

Semantic Issues The hardware specification language ITL [74] (distinct from the logic of the same name described in [37]) enforces a "separation of concerns", by providing a syntactic distinction between the antecedent and the consequent of every formula. For this example, separating cause from effect clearly is possible in PSL and SVA as well. For example, the following pair of formulas is together equivalent to Formula 40 while restricting the p's and q's to be on the left and the v's to be on the right of a suffix implication operator:

$$\mathrm{G}((p \cdot \neg q^*) \mapsto \neg v) \tag{41}$$

$$\mathrm{G}((p \cdot \neg q^* \cdot q) \mapsto (\neg v \cdot v)) \tag{42}$$

However, this approach is quite far from the philosophy of ITL, in which the separation of cause and effect is enforced by the syntax of the language. It would be interesting to see if there is an elegant way to incorporate the separation of concerns into PSL and SVA without breaking useful features of these languages.

Acknowledgements

Thank you to Gadi Aleksandrowicz, Shoham Ben-David, Alexander Ivrii, Avigail Orni, Sitvanit Ruah, Moshe Vardi and anonymous reviewers of this chapter for useful comments.

References

1. Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. FoCs: Automatic generation of simulation checkers from formal specifications. In *Proceedings CAV 2000*, volume 1855 of *LNCS*, pages 538–542. Springer, 2000.
2. B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.
3. B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.
4. R. Armoni, D. Bustan, O. Kupferman, and M. Vardi. Resets vs. aborts in linear temporal logic. In *Proceedings TACAS 2003*, volume 2619 of *LNCS*, pages 65–80. Springer, 2003.
5. R. Armoni, D. Fisman, and N. Jin. SVA and PSL local variables - a practical approach. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2013.
6. R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification language. In *Proceedings TACAS 2002*, volume 2280 of *LNCS*, pages 296–311. Springer, 2002.
7. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic Sugar. In *Proc. 13th International Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 363–367. Springer, 2001.
8. I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *Proc. 10th International Conference on Computer Aided Verification (CAV 1998)*, volume 1427 of *LNCS*, pages 184–194. Springer, 1998.
9. S. Ben-David, R. Bloem, D. Fisman, A. Griesmayer, I. Pill, and S. Ruah. Automata construction algorithms optimized for PSL (Deliverable 3.2/4). Technical report, Prosyd, 2005.
10. S. Ben-David, D. Fisman, and S. Ruah. The safety simple subset. In *Proc. 1st International Haifa Verification Conference*, volume 3875 of *LNCS*, pages 14–29. Springer, 2005.
11. S. Ben-David, D. Fisman, and S. Ruah. Embedding finite automata within regular expressions. *Theor. Comput. Sci.*, 404(3):202–218, 2008.
12. R. Bloem, K. Ravi, and F. Somenzi. Efficient decision procedures for model checking of linear time logic properties. In *Proceedings CAV 1999*, pages 222–235, 1999.
13. D. Bustan, D. Fisman, and J. Havlicek. Automata construction for PSL. Technical Report MCS05-04, The Weizmann Institute of Science, May 2005.
14. D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M. Vardi. Regular vacuity. In *Proc. 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *LNCS*, pages 191–206. Springer, 2005.

15. D. Bustan and J. Havlicek. Some complexity results for SystemVerilog assertions. In *Proceedings CAV 2006*, pages 205–218, 2006.
16. E. Cerny, S. Dudani, J. Havlicek, and D. Korchemny. *The Power of Assertions in SystemVerilog*. Springer, 2010.
17. E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs Workshop*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.
18. C. Eisner and D. Fisman. Sugar 2.0 proposal presented to the Accellera Formal Verification Technical Committee, March 2002. At http://www.haifa.il.ibm.com/projects/verification/sugar/Sugar_2.0_Accellera.ps.
19. C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, 2006.
20. C. Eisner and D. Fisman. Augmenting a regular expression-based temporal logic with local variables. In *FMCAD '08: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, pages 1–8, Piscataway, NJ, USA, 2008. IEEE Press.
21. C. Eisner and D. Fisman. Structural contradictions. In *Haifa Verification Conference*, pages 164–178, 2008.
22. C. Eisner, D. Fisman, and J. Havlicek. A topological characterization of weakness. In *Proceedings PODC 2005*, pages 1–8. ACM, 2005.
23. C. Eisner, D. Fisman, and J. Havlicek. Safety and liveness, weakness and strength, and the underlying topological relations. *Transactions on Computational Logic*, 15(2), 2014.
24. C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proceedings CAV 2003*, volume 2725 of *LNCS*, pages 27–39. Springer, 2003.
25. C. Eisner, D. Fisman, J. Havlicek, and J. Mårtensson. The $\top, \bot$ approach for truncated semantics. Technical Report 2006.01, Accellera, May 2006.
26. C. Eisner, D. Fisman, J. Havlicek, A. McIsaac, and D. Van Campenhout. The definition of a temporal clock operator. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 2003)*, volume 2719 of *LNCS*, pages 857–870. Springer, 2003.
27. E. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B*, chapter 16, pages 995–1072. Elsevier Science Publishers and The MIT Press, 1994.
28. M. Fischer and R. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and Systems Sciences*, 18:194–211, 1979.
29. D. Fisman. On the characterization of until as a fixed point under clocked semantics. In *Proc. Haifa Verification Conference*, volume 4899 of *LNCS*, pages 19–33. Springer, 2007.
30. H. Foster, A. Krolnik, and D. Lacey. *Assertion Based Design, 2nd Edition*. Kluwer Academic Publishers, 2004.
31. T. French and M. Reynolds. A sound and complete proof system for QPTL. In P. Balbiani, N.-Y. Suzuki, F. Wolter, and M. Zakharyaschev, editors, *Advances in Modal Logic*, pages 127–148. King's College Publications, 2002.
32. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, New York, NY, USA, 1980. ACM.
33. W. Gelade. Succinctness of regular expressions with interleaving, intersection and counting. In *MFCS '08: Proceedings of the 33rd international symposium on Mathematical Foundations of Computer Science*, pages 363–374, Berlin, Heidelberg, 2008. Springer-Verlag.
34. W. Gelade and F. Neven. Succinctness of the complement and intersection of regular expressions. In *Proceedings STACS 2008*, volume 1 of *Leibniz International Proceedings in Informatics*, pages 325–336, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.
35. M. Gordon. Validating the PSL/Sugar semantics using automated reasoning. *Formal Asp. Comput.*, 15(4):406–421, 2003.
36. T. Hafer and W. Thomas. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In *ICALP '87: Proceedings of the 14th International Colloquium, on Automata, Languages and Programming*, pages 269–279, London, UK, 1987. Springer-Verlag.

37. J. Y. Halpern, Z. Manna, and B. C. Moszkowski. A hardware semantics based on temporal intervals. In J. Díaz, editor, *ICALP*, volume 154 of *Lecture Notes in Computer Science*, pages 278–291. Springer, 1983.
38. D. Harel, D. Kozen, and R. Parikh. Process logic: Expressiveness, decidability, completeness. *J. Comput. Syst. Sci.*, 25(2):144–170, 1982.
39. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000.
40. D. Harel and D. Peleg. Process logic with regular formulas. *Theor. Comput. Sci.*, 38:307–322, 1985.
41. J. Havlicek, K. Shultz, R. Armoni, S. Dudani, and E. Cerny. Notes on the semantics of local variables in Accellera SystemVerilog 3.1 concurrent assertions. Technical Report 2004.01, Accellera, May 2004.
42. M. Holzer and S. Jakobi. State complexity of chop operations on unary and finite languages. In M. Kutrib, N. Moreira, and R. Reis, editors, *DCFS*, volume 7386 of *Lecture Notes in Computer Science*, pages 169–182. Springer, 2012.
43. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
44. IEEE Standard for Property Specification Language (PSL), Annex B. IEEE Std 1850™-2005.
45. IEEE Standard for Property Specification Language (PSL), Annex B. IEEE Std 1850™-2010.
46. IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language, Annex E. IEEE Std 1800™-2005.
47. IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language, Annex F. IEEE Std 1800™-2009.
48. J. Kamp. *Tense logic and the theory of order*. PhD thesis, UCLA, 1968.
49. Y. Kesten and A. Pnueli. A complete proof system for QPTL. In *In Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 2–12. IEEE Computer Society Press, 1995.
50. P. Kilpeläinen and R. Tuhkanen. Regular expressions with numerical occurrence indicators - preliminary results. In *SPLST*, pages 163–173, 2003.
51. O. Kupferman and M. Y. Vardi. Model checking of safety properties. In *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 172–183, 1999.
52. M. Lange. Linear time logics around PSL: Complexity, expressiveness, and a little bit of succinctness. In *Proc. CONCUR 2007*, volume 4703 of *LNCS*, pages 90–104, 2007.
53. F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal logic with forgettable past. In *LICS*, pages 383–392, 2002.
54. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL*, pages 97–107, 1985.
55. O. Lichtenstein, A. Pnueli, and L. D. Zuck. The glory of the past. In *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218, 1985.
56. J. Long, A. Seawright, and P. Kavalipati. Multi-clock SVA synthesis without re-writing. In *ASP-DAC*, pages 648–653, 2009.
57. M. Maidl. The common fragment of CTL and LTL. In *Proceedings FOCS 2000*, pages 643–652. IEEE Computer Society, 2000.
58. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
59. R. McNaughton and S. A. Papert. *Counter-Free Automata (M.I.T. research monograph no. 65)*. The MIT Press, 1971.
60. M. Morley. Semantics of Temporal e. In *Proc. Banff'99 Higher Order Workshop (Formal Methods in Computation)*, 1999. University of Glasgow, Dept. of Computing Science Technical Report.
61. D. E. Muller, A. Saoudi, and P. E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Proceedings, Symposium on Logic in Computer Science (LICS)*, pages 422–427. IEEE Computer Society, 1988.
62. M. T. Oliveira and A. J. Hu. High-level specification and automatic generation of IP interface monitors. In *DAC*, pages 129–134, 2002.

63. A. Pnueli. The temporal logics of programs. In *Proc. of the Annual IEEE Symposium on Foundations of Computer Science (FOCS 1977)*, pages 46–57. IEEE Computer Society, 1977.
64. V. R. Pratt. Semantical considerations on Floyd-Hoare logic. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1976.
65. A. Seawright and F. Brewer. High-level symbolic construction technique for high performance sequential synthesis. In *DAC*, pages 424–428, 1993.
66. A. P. Sistla. *Theoretical issues in the design and verification of distributed systems*. PhD thesis, Harvard University, Cambridge, MA, USA, 1983.
67. A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.
68. A. P. Sistla, M. Y. Vardi, and P. L. Wolper. The Complementation Problem for Büchi Automata, with Applications to Temporal Logic. *Theoretical Computer Science*, 49:217–237, 1987.
69. W. Thomas. Star-free regular sets of ω-sequences. *Information and Control*, 42(2):148–156, 1979.
70. M. Vardi. Branching vs. linear time: Final showdown. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*. Springer, 2001.
71. M. Y. Vardi. From Church and Prior to PSL. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 150–171. Springer, 2008.
72. M. Y. Vardi and P. Wolper. Yet another process logic (preliminary version). In *Logic of Programs*, pages 501–512, 1983.
73. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings, Symposium on Logic in Computer Science (LICS)*, pages 332–344. IEEE Computer Society, 1986.
74. K. Winkelmann. Capturing timing diagrams in assertion languages. At http://www.onespin-solutions.com/downloads/WhitePaper_ITL_Tidal.pdf.
75. P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983.

Index