The Temporal Logic Sugar

Ilan Beer[1], Shoham Ben-David[1], Cindy Eisner[1], Dana Fisman[1,2], Anna Gringauze[1], and Yoav Rodeh[1,2]

[1] IBM Haifa Research Laboratory
[2] Weizmann Institute of Science, Rehovot, Israel
danaf@il.ibm.com

1 Introduction

Since the introduction of temporal logic for the specification of computer programs [5], usability has been an issue, because a difficult-to-use formalism is a barrier to the wide adoption of formal methods. Our solution is Sugar, the temporal logic used by the RuleBase formal verification tool [2]. Sugar adds the power of regular expressions to CTL [4], as well as an extensive set of operators which provide *syntactic sugar*. That is, while these operators do not add expressive power, they allow properties to be expressed more succinctly than in the basic language. Experience shows that Sugar allows hardware engineers to easily and intuitively specify their designs. The full language is used for model checking, and a significant portion can be model checked on-the-fly [3]. The automatic generation of simulation checkers from the same portion of Sugar is described in [1]. While previous papers have described various features of the language, this paper presents the first complete description of Sugar.

2 The Basic Language

We use boolean expressions to describe states in the model, and Sugar Extended Regular Expressions to describe sequences of states, and define them as follows:

Definition 1. (Boolean Expression).

1. *Every atomic proposition is a boolean expression.*
2. *If b, b_1, and b_2 are boolean expressions, then so are $\neg b$ and $b_1 \wedge b_2$.*

Definition 2. (Sugar Extended Regular Expressions (SEREs)).

1. *Every boolean expression is a SERE.*
2. *If r, r_1, and r_2 are SEREs, then so are the following: i) r_1, r_2 ii) $r_1 \sim r_2$ iii) $r_1 || r_2$ iv) $r_1 \,\&\&\, r_2$ v) $r[*]$*

A comma denotes concatenation, $\sim$ denotes overlapping concatenation, where the last state of r_1 coincides with the first state of r_2, $||$ denotes disjunction, $\&\&$ denotes conjunction, and $[*]$ is used to specify 0 or more repetitions.

There are two ways to use SEREs in Sugar formulas. The first is to link two SEREs in order to form Sugar formulas of the linear fragment, as defined in Definition 3 below. A second way is to link a single SERE with a general Sugar formula, as defined in Definition 4 below.

M. Kerckhove (Ed.): Scale-Space 2001, LNCS 2106, pp. 363–367, 2001.

Definition 3. (Sugar Formulas of the Linear Fragment[1]). *If r_1 and r_2 are SEREs, then $\{r_1\} \mapsto \{r_2\}!$ and $\{r_1\} \mapsto \{r_2\}$ are Sugar formulas of the linear fragment.*

The $\{r_1\} \mapsto \{r_2\}!$ and $\{r_1\} \mapsto \{r_2\}$ constructs are known as *strong suffix implication* and *weak suffix implication*, respectively. Strong suffix implications are liveness formulas, indicating that every sequence of states on which r_1 holds must be followed by a sequence of states on which r_2 holds. Weak suffix implications are safety formulas, indicating that every sequence of states on which r_1 holds may not be followed by a sequence of states which contradicts r_2. For instance, the Sugar formula $\{[*], p, q\} \mapsto \{s[*], t\}!$ requires that every sequence of two states such that p is valid in the first and q is valid in the second, must be followed by a sequence of states in which s is valid for some number of states, and then t is valid in the final state of the sequence. The weak form of this formula does not require that the second sequence "reach its end": a sequence matching $\{p, q\}$ must be followed either by a sequence in which s holds forever, or by a sequence in which s holds for some number of states, and then t holds.

Definition 4. (Sugar Formulas).

1. *Every boolean expression is a Sugar formula.*
2. *Every Sugar formula of the linear fragment is a Sugar formula.*
3. *If f, f_1, and f_2 are Sugar formulas and r is a SERE, then the following are Sugar formulas: i) $\neg f$ ii) $f_1 \wedge f_2$ iii) EXf iv) $E[f_1 U f_2]$ v) EGf vi) $\{r\}(f)$*

The operators $\neg$, $\wedge$, EX, EU, and EG have the usual meaning. The construct $\{r\}(f)$ (*suffix implication*) holds for a state s if, for all finite sequences starting from s on which r holds, formula f holds on the final state in the sequence r.

3 Syntactic Sugar

Because the basic language can be verbose, Sugar adds syntactic sugar: additional operators which allow many properties to be expressed succinctly in an intuitive manner. We will now illustrate the advantages of the syntactic sugar with a few examples[2].

The *next_event* Operators. These operators are a conceptual extension of the AX operator. While AX refers to the next state, *next_event* refers to the next state in which a boolean expression is valid. For instance, the following:

$$AG(hi\_pri\_req \rightarrow next\_event\_f(gnt)[1..2](dst = hi\_pri)) \quad (1)$$

[1] Note that Sugar formulas of the linear fragment are not closed under the boolean operators. The result of a boolean operation on two Sugar formulas of the linear fragment is a general Sugar formula as described in Definition 4.

[2] The abbreviations presented here and in Appendix B are given as an explanatory semantics and do not imply the actual implementation.

expresses the requirement that whenever *hi_pri_req* is asserted, one of the next two assertions of signal *gnt* must have *dst* equal to *hi_pri*.

The *next_event* operator, and its variant $next\_event\_f(b)[1..2](f)$ are defined in terms of the weak until (AW) operator as follows: $A[f_1\ W\ f_2]$ is equivalent to $\neg E[\neg f_2\ U\ \neg f_1 \wedge \neg f_2]$, $next\_event(b)(f)$ is equivalent to $A[\neg b\ W\ b \wedge f]$, and $next\_event(b)[1..2](f)$ is equivalent to $next\_event(b)(f \vee AX next\_event(b)(f))$. Thus, Formula 1 can be expressed in CTL with the addition of the AW operator as follows:

$$AG(hi\_pri\_req \rightarrow A[\neg gnt\ W\ ((gnt \wedge dst = hi\_pri) \vee (gnt \wedge AX A[\neg gnt\ W\ (gnt \wedge dst = hi\_pri)]))]) \quad (2)$$

The *within* Operators. The *within* operators ease the expression of requirements such as the following: "every transaction must complete, and within every transaction, a full data transfer must occur", which is expressible in Sugar as:

$$AG\ within!(tr\_strt, tr\_end)\{\mathbf{true}[*], dat\_strt, \mathbf{true}[*], dat\_end\} \quad (3)$$

$within!(r_1, b)\{r_2\}$ is equivalent to $\{r_1\} \mapsto \{r_2\ \&\&\ \{\neg b[*]\}, \neg b[*], b\}!$. Thus, Formula 3 can be expressed (albeit somewhat cryptically) in CTL as follows:

$$AG(tr\_strt \rightarrow A[\neg dat\_strt \wedge \neg tr\_end\ U\ dat\_strt \wedge A[\neg tr\_end\ U\ dat\_end \wedge A[\neg tr\_end\ U\ tr\_end]]]) \quad (4)$$

Counters. Counters are used to describe sequences of events that would otherwise be tedious to specify. For example, i consecutive occurrences of sequence r can be expressed as $r[i]$, and i non-consecutive occurrences of boolean expression b can be expressed as $b[= i]$. Formally, $r[0]$ is equivalent to $\mathbf{false}[*]$, while $r[i]$ is equivalent to i concatenations of r, and $b[= i]$ is equivalent to $\{\neg b[*], b\}[i], \neg b[*]$. The utility of the $b[= i]$ construct is illustrated in the following Sugar formula:

$$AG(\{go, \{get[= 8]\}\&\&\{kill[= 0]\}\} \mapsto \{\mathbf{true}, \{put[= 8]\}\&\&\{end[= 0]\}\}) \quad (5)$$

which expresses the requirement that a sequence beginning with the assertion of signal *go*, and containing eight not necessarily consecutive assertions of signal *get*, during which signal *kill* is not asserted, must be followed by a sequence containing eight assertions of signal *put* before signal *end* can be asserted. The equivalent CTL formula is both non-intuitive and tedious. The CTL formula expressing the same requirement but for sequences of only two gets and puts illustrates this point:

$$AG\neg(go \wedge EX\ E[\neg get \wedge \neg kill\ U\ get \wedge \neg kill \wedge EX\ E[\neg get \wedge \neg kill\ U\ get \wedge \neg kill \wedge E[\neg put\ U\ end] \vee E[\neg put \wedge \neg end\ U\ (put \wedge \neg end \wedge EX E[\neg put\ U\ end])]]]) \quad (6)$$

Formulas 1, 3 and 5 can also be expressed in LTL [6]. However, the equivalent LTL formulas are not any less daunting to code or decipher than the CTL versions, while the Sugar version expresses the requirements succinctly, in a manner accessible to the non-logician.

References

1. Y. Abarbanel and I. Beer et al. FoCs - automatic generation of simulation checkers from formal specifications. In *CAV '00*, LNCS 1855. Springer-Verlag, 2000.
2. I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: an industry-oriented formal verification tool. In *DAC '96*, pages 655–660, June 1996.
3. I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *CAV '98*, LNCS 1427, pages 184–194. Springer-Verlag, 1998.
4. E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logics of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.
5. A. Pnueli. The temporal logic of programs. In *Proc.* 18^{th} *Annual IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.
6. A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

A Formal Semantics

The semantics of a Sugar formula are defined with respect to a model M. A model is a quintuple (S, S_0, R, P, L), where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is the transition relation, P is a non-empty set of atomic propositions, and L is the valuation, a function $L : S \longrightarrow 2^P$, mapping each state with a set of atomic propositions true in that state. R is total with respect to its first argument. A computation path π of a model M is an infinite sequence of states $\pi = (\pi_0, \pi_1, \pi_2, \cdots)$ such that $R(\pi_i, \pi_{i+1})$ for every i. We will denote by $\pi^{i,j}$ a finite sequence of states starting from π_i and ending in π_j.

The semantics of SEREs are defined over the alphabet 2^P. Thus, a letter is a subset of the set of atomic propositions P. We will denote a letter from 2^P by ℓ and a finite word over 2^P by w. The concatenation of w_1 and w_2 is denoted by $w_1 w_2$. The empty word is denoted by ϵ, so that $w\epsilon = \epsilon w = w$. The notation $w \in \mathcal{L}(r)$, where r is a SERE, means that w is in the language of r. The semantics of SEREs are defined as follows:

1. $w \in \mathcal{L}(p) \Longleftrightarrow$ there exists an ℓ s.t. $w = \ell$ and $p \in \ell$
2. $w \in \mathcal{L}(\neg b) \Longleftrightarrow w \notin \mathcal{L}(b)$
3. $w \in \mathcal{L}(b_1 \wedge b_2) \Longleftrightarrow w \in \mathcal{L}(b_1)$ and $w \in \mathcal{L}(b_2)$
4. $w \in \mathcal{L}(r_1, r_2) \Longleftrightarrow$ there exist w_1 and w_2 s.t. $w = w_1 w_2$ and $w_1 \in \mathcal{L}(r_1)$ and $w_2 \in \mathcal{L}(r_2)$
5. $w \in \mathcal{L}(r_1 \sim r_2) \Longleftrightarrow$ there exist w_1, w_2, and ℓ s.t. $w = w_1 \ell w_2$ and $w_1 \ell \in \mathcal{L}(r_1)$ and $\ell w_2 \in \mathcal{L}(r_2)$
6. $w \in \mathcal{L}(r_1 || r_2) \Longleftrightarrow w \in \mathcal{L}(r_1)$ or $w \in \mathcal{L}(r_2)$
7. $w \in \mathcal{L}(r_1 \,\&\&\, r_2) \Longleftrightarrow w \in \mathcal{L}(r_1)$ and $w \in \mathcal{L}(r_2)$
8. $w \in \mathcal{L}(r[*]) \Longleftrightarrow$ either $w = \epsilon$ or there exist $w_1, w_2, \ldots, w_j$ s.t. $w = w_1 w_2 \ldots w_j$ and, for all i, $1 \leq i \leq j$, $w_i \in \mathcal{L}(r)$

Recall that every state $s \in S$ in a model $M = (S, S_0, R, P, L)$ is associated with a set of atomic propositions by the valuation L. We define $\hat{L}$, an extension of the valuation function L as follows: $\hat{L}(\pi_i, \pi_{i+1}, \ldots \pi_j) = L(\pi_i)L(\pi_{i+1}) \ldots L(\pi_j)$. Thus we have a mapping from states in M to letters of 2^P, and from finite sequences of states in M to words over 2^P.

We now turn to the semantics of Sugar formulas. The notation $M, s \models f$ means that formula f holds in state s of model M. The notation $M \models f$ is equivalent to $\forall s \in S_0\ M, s \models f$, in other words, f is valid for all initial states of M. We use p, p_1 and p_2 to denote atomic propositions, b, b_1 and b_2 to denote boolean expressions, r, r_1 and r_2 to denote SEREs, and f, f_1 and f_2 to denote Sugar formulas. The semantics of a Sugar formula are defined as follows:

1. $M, s \models p \Longleftrightarrow p \in L(s)$
2. $M, s \models \neg f \Longleftrightarrow M, s \not\models f$
3. $M, s \models f_1 \wedge f_2 \Longleftrightarrow M, s \models f_1$ and $M, s \models f_2$
4. $M, s \models r_1 \vdash\!\!\!\rightarrow\ r_2! \Longleftrightarrow$ for all paths π s.t. $\pi_0 = s$, for all j s.t. $\hat{L}(\pi^{0,j}) \in \mathcal{L}(r_1)$, there exists a k s.t. $\hat{L}(\pi^{j,k}) \in \mathcal{L}(r_2)$
5. $M, s \models r_1 \vdash\!\!\!\rightarrow\ r_2 \Longleftrightarrow$ for all paths π s.t. $\pi_0 = s$, for all j s.t. $\hat{L}(\pi^{0,j}) \in \mathcal{L}(r_1)$, either there exists a k s.t. $\hat{L}(\pi^{j,k}) \in \mathcal{L}(r_2)$, or for all k, there exists a word w (not necessarily a computation path in M) s.t. $\hat{L}(\pi^{j,k})w \in \mathcal{L}(r_2)$
6. $M, s \models EX\ f \Longleftrightarrow$ for some path π s.t. $\pi_0 = s$, $M, \pi_1 \models f$
7. $M, s \models E[f_1 U f_2] \Longleftrightarrow$ for some path π s.t. $\pi_0 = s$, there exists k s.t. $M, \pi_k \models f_2$ and for all j s.t. $j < k$, $M, \pi_j \models f_1$
8. $M, s \models EG\ f \Longleftrightarrow$ for some path π s.t. $\pi_0 = s$, for all $j \geq 0$, $M, \pi_j \models f$
9. $M, s \models \{r\}(f) \Longleftrightarrow$ for all paths π s.t. $\pi_0 = s$, for all j s.t. $\hat{L}(\pi^{0,j}) \in \mathcal{L}(r)$, $M, \pi_j \models f$

B The Full Syntactic Sugar

Additional boolean operators

1. $b_1 \vee b_2 = \neg(\neg b_1 \wedge \neg b_2)$
2. $b_1 \rightarrow b_2 = \neg b_1 \vee b_2$
2. $b_1 \rightarrow b_2 = \neg b_1 \vee b_2$
4. $b_1 \oplus b_2 = (b_1 \wedge \neg b_2) \vee (\neg b_1 \wedge b_2)$

Additional SERE operators (where i, j, and k are integer constants s.t. $i \geq 0$, $j \geq i$, $k > 0$)

1. $r[+] = r, r[*]$
2. $r[i] = \begin{cases} \mathbf{false}[*] & \text{if } i = 0 \\ \overbrace{r, r, \ldots, r}^{i \text{ times}} & \text{otherwise} \end{cases}$
3. $r[i..j] = r[i]||r[i+1]|| \ldots ||r[j]$
4. $r[i..] = r[i], r[*]$
5. $r[..i] = r[0]||r[1]||r[2]|| \ldots ||r[i]$
6. $[+] = \mathbf{true}[+]$
7. $[*] = \mathbf{true}[*]$
8. $[i] = \mathbf{true}[i]$
9. $[i..j] = \mathbf{true}[i..j]$
10. $[i..] = \mathbf{true}[i..]$
11. $[..i] = \mathbf{true}[..i]$
12. $b[= i] = \{\neg b[*], b\}[i], \neg b[*]$
13. $b[> i] = b[= i+1], [*]$
14. $b[< k] = b[= 0]||b[= 1]|| \ldots ||b[= (k-1)]$
15. $b[\geq i] = b[> i]||b[= i]$
16. $b[\leq i] = b[< i]||b[= i]$
17. $b[> i, < j] = b[> i]\ \&\&\ b[< j]$
18. $b[\geq i, < j] = b[\geq i]\ \&\&\ b[< j]$
19. $b[> i, \leq j] = b[> i]\ \&\&\ b[\leq j]$
20. $b[\geq i, \leq j] = b[\geq i]\ \&\&\ b[\leq j]$

Additional linear operators

1. $always\{r\} = \{\mathbf{true}[*]\} \vdash\!\!\!\dashv \{r\}$
2. $never\{r\} = \{\mathbf{true}[*], r\} \vdash\!\!\!\dashv \{\mathbf{false}\}$
3. $eventually\{r\} = \{\mathbf{true}\} \vdash\!\!\!\dashv \{\mathbf{true}[*], r\}!$
4. $within!(r_1, b)\{r_2\} = \{r_1\} \vdash\!\!\!\dashv \{r_2\ \&\&\ b[= 0], \neg b[*], b\}!$
5. $within(r_1, b)\{r_2\} = \{r_1\} \vdash\!\!\!\dashv \{r_2\ \&\&\ b[= 0]\}$
6. $within!\_(r_1, b)\{r_2\} = \{r_1\} \vdash\!\!\!\dashv \{\{r_2\ \&\&\ b[= 0], \neg b[*], b\}||\{r_2\ \&\&\ \{\neg b[*], b\}\}\}!$
7. $within\_(r_1, b)\{r_2\} = \{r_1\} \vdash\!\!\!\dashv \{r_2\ \&\&\ \{\neg b[*], \mathbf{true}\}\}$
8. $whilenot!(b)\{r\} = within!(\mathbf{true}, b)\{r\}$
9. $whilenot(b)\{r\} = within(\mathbf{true}, b)\{r\}$
10. $whilenot!\_(b)\{r\} = within!\_(\mathbf{true}, b)\{r\}$
11. $whilenot\_(b)\{r\} = within\_(\mathbf{true}, b)\{r\}$
12. $\{r_1\} \Vdash\!\!\!\dashv \{r_2\}! = \{r_1\} \vdash\!\!\!\dashv \{\mathbf{true}, r_2\}!$
13. $\{r_1\} \Vdash\!\!\!\dashv \{r_2\} = \{r_1\} \vdash\!\!\!\dashv \{\mathbf{true}, r_2\}$

Additional branching operators (where i and j are integers s.t. $i > 0$ and $j \geq i$)

1. $EF\ f = E[\mathbf{true} U f]$
2. $AXf = \neg EX\neg f$
3. $AGf = \neg E[\mathbf{true}\ U\ \neg f]$
4. $A[f_1\ U\ f_2] = \neg(E[\neg f_2\ U\ \neg f_1 \wedge \neg f_2] \vee EG\neg f_2)$
5. $AFf = A[\mathbf{true}\ U\ f]$
6. $E[f_1\ W\ f_2] = E[f_1\ U\ f_2] \vee EGf_1$
7. $A[f_1\ W\ f_2] = \neg E[\neg f_2\ U\ \neg f_1 \wedge \neg f_2]$
8. $AX[i]f = \overbrace{AXAX \ldots AX}^{i \text{ times}} f$
9. $ABG[i..j]f = AX[i]f \wedge AX[i+1]f \wedge \ldots \wedge AX[j]f$
10. $ABF[i..j]f = AX[i](\overbrace{f \vee AX(f \vee AX(\ldots f \vee AX(}^{j-i \text{ times}} f)\ldots)))$
11. $f_1\ until!\ f_2 = A[f_1\ U\ f_2]$
12. $f_1\ until\ f_2 = A[f_1\ W\ f_2]$
13. $f_1\ until!\_\ f_2 = A[f_1\ U\ f_1 \wedge f_2]$
14. $f_1\ until\_\ f_2 = A[f_1\ W\ f_1 \wedge f_2]$
15. $f_1\ before!\ f_2 = A[\neg f_2\ U\ f_1]$
16. $f_1\ before\ f_2 = A[\neg f_2\ W\ f_1]$
17. $f_1\ before!\_\ f_2 = A[\neg f_2\ U\ f_1 \wedge \neg f_2]$
18. $f_1\ before\_\ f_2 = A[\neg f_2\ W\ f_1 \wedge \neg f_2]$
19. $next\_event!(b)(f) = A[\neg b\ U\ b \wedge f]$
20. $next\_event(b)(f) = A[\neg b\ W\ b \wedge f]$
21. $next\_event!(b)[i](f) = \overbrace{(AXnext\_event!(b)(AXnext\_event!(b) \ldots (AXnext\_event!(b)}^{i-1 \text{ times}}(f)\ldots)))$
22. $next\_event(b)[i](f) = \overbrace{(AXnext\_event(b)(AXnext\_event(b) \ldots (AXnext\_event(b)}^{i-1 \text{ times}}(f)\ldots)))$
23. $next\_event!(b)[i..j](f) = next\_event!(b)[i](f) \wedge \ldots \wedge next\_event!(b)[j](f)$
24. $next\_event(b)[i..j](f) = next\_event(b)[i](f) \wedge \ldots \wedge next\_event(b)[j](f)$
25. $next\_event\_f!(b)[i..j](f) = \overbrace{(f \vee AXnext\_event!(b)(f \vee AXnext\_event!(b) \ldots (f \vee AXnext\_event!(b)}^{j-i \text{ times}}(f)\ldots)))$
26. $next\_event\_f(b)[i..j](f) = \overbrace{(f \vee AXnext\_event(b)(f \vee AXnext\_event(b) \ldots (f \vee AXnext\_event(b)}^{j-i \text{ times}}(f)\ldots)))$