
Spot's Temporal Logic Formulas

Alexandre Duret-Lutz <adl@lrde.epita.fr>
compiled on February 23, 2023, for Spot 2.10.4

1. Reasoning with Infinite Sequences	2
1.1. Finite and Infinite Sequences	2
1.2. Usage in Model Checking	2
2. Temporal Syntax & Semantics	3
2.1. Boolean Constants	3
2.1.1. Semantics	3
2.2. Atomic Propositions	3
2.2.1. Examples	4
2.2.2. Semantics	4
2.3. Boolean Operators (for Temporal Formulas)	4
2.3.1. Semantics	5
2.3.2. Trivial Identities (Occur Automatically)	5
2.4. Temporal Operators	5
2.4.1. Semantics	6
2.4.2. Syntactic Sugar	6
2.4.3. Trivial Identities (Occur Automatically)	6
2.5. SERE Operators	7
2.5.1. Semantics	7
2.5.2. Syntactic Sugar	8
2.5.3. Trivial Identities (Occur Automatically)	9
2.6. SERE-LTL Binding Operators	10
2.6.1. Semantics	10
2.6.2. Syntactic Sugar	11
2.6.3. Trivial Identities (Occur Automatically)	11
3. Grammar	12
3.1. Operator precedence	12
4. Properties	14
4.1. Pure Eventualities and Purely Universal Formulas	14
4.2. Syntactic Hierarchy Classes	15
5. Rewritings	17
5.1. Unabbreviations	17
5.2. LTL simplifier	17
5.3. Negative normal form	18
5.4. Simplifications	18
5.4.1. Basic Simplifications	19
5.4.2. Simplifications for Eventual and Universal Formulas	23
5.4.3. Simplifications Based on Implications	24

A. Defining LTL with only one of \cup, \cap, \vee, or \wedge	27
B. Syntactic Implications	29
Bibliography	29

1. Reasoning with Infinite Sequences

1.1. Finite and Infinite Sequences

Let $N = \{0, 1, 2, \dots\}$ denote the set of natural numbers and $\omega \notin N$ the first transfinite ordinal. We extend the $<$ relation from N to $N \cup \{\omega\}$ with $\forall n \in N, n < \omega$. Similarly let us extend the addition and subtraction with $\forall n \in N, \omega + n = \omega - n = \omega + \omega = \omega$.

For any set A , and any number $n \in N \cup \{\omega\}$, a *sequence* of length n is a function $\sigma : \{0, 1, \dots, n-1\} \rightarrow A$ that associates each index $i < n$ to an element $\sigma(i) \in A$. The sequence of length 0 is a particular sequence called the *empty word* and denoted ε . We denote A^n the set of all sequences of length n on A (in particular A^ω is the set of infinite sequences on A), and $A^* = \bigcup_{n \in N} A^n$ denotes the set of all finite sequences. The length of any sequence σ is noted $|\sigma|$, with $|\sigma| \in N \cup \{\omega\}$.

For any sequence σ , we denote $\sigma^{i..j}$ the finite subsequence built using letters from $\sigma(i)$ to $\sigma(j)$. If σ is infinite, we denote $\sigma^{i..}$ the suffix of σ starting at letter $\sigma(i)$.

1.2. Usage in Model Checking

The temporal formulas described in this document, should be interpreted on behaviors (or executions, or scenarios) of the system to verify. In model checking we want to ensure that a formula (the property to verify) holds on all possible behaviors of the system.

If we model the system as some sort of giant automaton (e.g., a Kripke structure) where each state represent a configuration of the system, a behavior of the system can be represented by an infinite sequence of configurations. Each configuration can be described by an affectation of some proposition variables that we will call *atomic propositions*. For instance $r = 1, y = 0, g = 0$ describes the configuration of a traffic light with only the red light turned on.

Let AP be a set of atomic propositions, for instance $AP = \{r, y, g\}$. A configuration of the model is a function $\rho : AP \rightarrow B$ (or $\rho \in B^{AP}$) that associates a truth value ($B = \{0, 1\}$) to each atomic proposition.

A behavior of the model is an infinite sequence σ of such configurations. In other words: $\sigma \in (B^{AP})^\omega$.

When a formula φ holds on an *infinite* sequence σ , we write $\sigma\varphi$ (read as σ is a model of φ).

When a formula φ holds on an *finite* sequence σ , we write $\sigma\varphi$.

2. Temporal Syntax & Semantics

2.1. Boolean Constants

The two Boolean constants are '1' and '0'. They can also be input as 'true' or 'false' (case insensitive) for compatibility with the output of other tools, but Spot will always use '1' and '0' in its output.

2.1.1. Semantics

$\sigma 0$

$\sigma 1$

2.2. Atomic Propositions

Atomic propositions in Spot are strings of characters. There are no restrictions on the characters that appear in the strings, however because some of the characters may also be used to denote operators you may have to represent the strings differently if they include these characters.

1. Any string of characters represented between double quotes is an atomic proposition.
2. Any sequence of alphanumeric characters (including '_') that is not a reserved keyword and that starts with a character that is not an uppercase 'F', 'G', or 'X', is also an atomic proposition. In this case the double quotes are not necessary.
3. Any sequence of alphanumeric character that starts with 'F', 'G', or 'X', has a digit in second position, and anything afterwards, is also an atomic proposition, and the double quotes are not necessary.

Here is the list of reserved keywords:

- 'true', 'false' (both are case insensitive)
- 'F', 'G', 'M', 'R', 'U', 'V', 'W', 'X', 'xor'

The only way to use an atomic proposition that has the name of a reserved keyword, or one that starts with a digit, is to use double quotes.

The reason we deal with leading 'F', 'G', and 'X' specifically in rule 2 is that these are unary LTL operators and we want to be able to write compact formulas like 'GFa' instead of the equivalent 'G(F(a))' or 'G F a'. If you want to name an atomic proposition 'GFa', you will have to quote it as "GFa".

The exception done by rule 3 when these letters are followed by a digit is meant to allow 'X0', 'X1', 'X2', ... to be used as atomic propositions. With only rule 2, 'X0' would be interpreted as 'X(0)', that is, the LTL operator X applied to the constant *false*, but there is really little reason to use such a construction in a formula (the same is true for 'F' and 'G', and also when applied to '1'). On the other hand, having numbered versions of a variable is pretty common, so it makes sense to favor this interpretation.

If you are typing in formulas by hand, we suggest you name all your atomic propositions in lower case, to avoid clashes with the uppercase operators.

If you are writing a tool that produces formula that will be feed to Spot and if you cannot control the atomic propositions that will be used, we suggest that you always output atomic propositions between double quotes to avoid any unintended misinterpretation.

2.2.1. Examples

- `"a<=b+c"` is an atomic proposition. Double quotes can therefore be used to embed constructs specific to the underlying formalism, and still regard the resulting construction as an atomic proposition.
- `'light_on'` is an atomic proposition.
- `'Fab'` is not an atomic proposition, this is actually equivalent to the formula `'F(ab)'` where the temporal operator `F` is applied to the atomic proposition `'ab'`.
- `'FINISHED'` is not an atomic proposition for the same reason; it actually stands for `'F(INISHED)'`
- `'F100ZX'` is an atomic proposition by rule 3.
- `'FX100'` is not an atomic proposition, it is equivalent to the formula `'F(X100)'`, where `'X100'` is an atomic proposition by rule 3.

2.2.2. Semantics

For any atomic proposition a , we have

$$\sigma a \iff \sigma(0)(a) = 1$$

In other words a holds if and only if it is true in the first configuration of σ .

2.3. Boolean Operators (for Temporal Formulas)

Two temporal formulas f and g can be combined using the following Boolean operators:

operation	preferred syntax	other supported syntaxes			UTF8 characters supported	
					preferred	others
negation	<code>! f</code>	<code>~ f</code>			<code>¬ U+00AC</code>	
disjunction	<code>f g</code>	<code>f g</code>	<code>f \ / g</code>	<code>f + g</code>	<code>∨ U+2228</code>	<code>∪ U+222A</code>
conjunction	<code>f & g</code>	<code>f && g</code>	<code>f /\ g</code>	<code>f * g</code> ¹	<code>∧ U+2227</code>	<code>∩ U+2229</code>
implication	<code>f -> g</code>	<code>f => g</code>	<code>f --> g</code>		<code>→ U+2192</code>	<code>→ U+27F6, ⇒ U+21D2 U+27F9</code>
exclusion	<code>f xor g</code>	<code>f ^ g</code>			<code>⊕ U+2295</code>	
equivalence	<code>f <-> g</code>	<code>f <=> g</code>	<code>f <--> g</code>		<code>↔ U+2194</code>	<code>⇔ U+21D4</code>

Additionally, an atomic proposition a can be negated using the syntax `'a=0'`, which is equivalent to `'! a'`. Also `'a=1'` is equivalent to just `'a'`. These two syntaxes help us read formulas written using Wring's syntax.

When using UTF-8 input, a `'=0'` that follow a single-letter atomic proposition may be replaced by a combining overline U+0305 or a combining overbar U+0304. When instructed to emit UTF-8, Spot will output `'ā'` using a combining overline instead of `'¬a'` for any single-letter atomic proposition.

When a formula is built using only Boolean constants (section 2.1), atomic proposition (section 2.2), and the above operators, we say that the formula is a *Boolean formula*.

¹The `*`-form of the conjunction operator (allowing better compatibility with Wring and VIS) may only used in temporal formulas. Boolean expressions that occur inside SERE (see Section 2.5) may not use this form because the `*` symbol is used as the Kleen star.

2.3.1. Semantics

$$\begin{aligned}
\sigma ! f &\iff (\sigma f) \\
\sigma f \& g &\iff (\sigma f) \wedge (\sigma g) \\
\sigma f \mid g &\iff (\sigma f) \vee (\sigma g) \\
\sigma f \rightarrow g &\iff (\sigma f) \vee (\sigma g) \\
\sigma f \text{ xor } g &\iff ((\sigma f) \wedge (\sigma g)) \vee ((\sigma f) \wedge (\sigma g)) \\
\sigma f \leftrightarrow g &\iff ((\sigma f) \wedge (\sigma g)) \vee ((\sigma f) \wedge (\sigma g))
\end{aligned}$$

2.3.2. Trivial Identities (Occur Automatically)

Trivial identities are applied every time an expression is constructed. This means for instance that there is not way to construct the expression ' $! ! a$ ' in Spot, such an attempt will always yield the expression ' a '.

$$\begin{array}{lll}
! 0 \equiv 1 & 1 \rightarrow f \equiv f & f \rightarrow 1 \equiv 1 \\
! 1 \equiv 0 & 0 \rightarrow f \equiv 1 & f \rightarrow 0 \equiv ! f \\
! ! f \equiv f & & f \rightarrow f \equiv 1
\end{array}$$

The next set of rules apply to operators that are commutative, so these identities are also valid with the two arguments swapped.

$$\begin{array}{llll}
0 \& f \equiv 0 & 0 \mid f \equiv f & 0 \text{ xor } f \equiv f & 0 \leftrightarrow f \equiv ! f \\
1 \& f \equiv f & 1 \mid f \equiv 1 & 1 \text{ xor } f \equiv ! f & 1 \leftrightarrow f \equiv f \\
f \& f \equiv f & f \mid f \equiv f & f \text{ xor } f \equiv 0 & f \leftrightarrow f \equiv 1
\end{array}$$

The ' $\&$ ' and ' \mid ' operators are associative, so they are actually implemented as n -ary operators (i.e., not binary): this allows us to reorder all arguments in a unique way (e.g. alphabetically). For instance the two expressions ' $a\&c\&b\&!d$ ' and ' $c\&!d\&b\&a$ ' are actually represented as the operator ' $\&$ ' applied to the arguments $\{a, b, c, !d\}$. Because these two expressions have the same internal representation, they are actually considered equal for the purpose of the above identities. For instance ' $(a\&c\&b\&!d) \rightarrow (c\&!d\&b\&a)$ ' will be rewritten to ' 1 ' automatically.

2.4. Temporal Operators

Given two temporal formulas f , and g , the following temporal operators can be used to construct another temporal formula.

operator	preferred syntax	other supported syntaxes	UTF8 characters supported	
			preferred	others
(Weak) Next	$X f$	$() f$	○ U+25CB	○ U+25EF
Strong Next	$X[!] f$		⊗ U+24CD	
Eventually	$F f$	$\langle \rangle f$	◇ U+25C7	◇ U+22C4 U+2662
Always	$G f$	$[] f$	□ U+25A1	□ U+2B1C U+25FB
(Strong) Until	$f U g$			
Weak Until	$f W g$			
(Weak) Release	$f R g$	$f V g$		
Strong Release	$f M g$			

2.4.1. Semantics

$$\begin{aligned}
\sigma X f &\iff \sigma^1 \dots f \\
\sigma X[!] f &\iff \sigma^1 \dots f \\
\sigma F f &\iff \exists i \in N, \sigma^i \dots f \\
\sigma G f &\iff \forall i \in N, \sigma^i \dots f \\
\sigma f U g &\iff \exists j \in N, \begin{cases} \forall i < j, \sigma^i \dots f \\ \sigma^j \dots g \end{cases} \\
\sigma f W g &\iff (\sigma f U g) \vee (\sigma G f) \\
\sigma f M g &\iff \exists j \in N, \begin{cases} \forall i \leq j, \sigma^i \dots f \\ \sigma^j \dots g \end{cases} \\
\sigma f R g &\iff (\sigma f M g) \vee (\sigma G g)
\end{aligned}$$

Note that the semantics of X (weak next) and $X[!]$ (strong next) are identical in LTL formulas. The two operators make sense only to build LTLf formulas (i.e., LTL with finite semantics), for which support is being progressively introduced in Spot.

Appendix A explains how to rewrite the above LTL operators using only X and one operator chosen among U , W , M , and R . This could be useful to understand the operators R , M , and W if you are only familiar with X and U .

2.4.2. Syntactic Sugar

The syntax on the left is equivalent to the syntax on the right. Some of rewritings taken from the syntax of TSLF [?] are performed from left to right when parsing a formula. They express the fact that some formula f has to be true in n steps, or at some or all times between n and m steps.

$$\begin{aligned}
X[n] f &\equiv \underbrace{X X \dots X}_n f \\
&\quad n \text{ occurrences of } X \\
F[n:m] f &\equiv \underbrace{X X \dots X}_n (f \mid \underbrace{X(f \mid X(\dots \mid X f))}_{m-n \text{ occ. of } X}) \\
G[n:m] f &\equiv \underbrace{X X \dots X}_n (f \& \underbrace{X(f \& X(\dots \& X f))}_{m-n \text{ occ. of } X}) \\
X[n!] f &\equiv \underbrace{X[!] X[!] \dots X[!]}_n f \\
&\quad n \text{ occurrences of } X[!] \\
F[n:m!] f &\equiv \underbrace{X[!] X[!] \dots X[!]}_n (f \mid \underbrace{X[!](f \mid X[!](\dots \mid X[!] f))}_{m-n \text{ occ. of } X[!]}) \\
G[n:m!] f &\equiv \underbrace{X[!] X[!] \dots X[!]}_n (f \& \underbrace{X[!](f \& X[!](\dots \& X[!] f))}_{m-n \text{ occ. of } X[!]})
\end{aligned}
\qquad
\begin{aligned}
F[n:] f &\equiv X[n] F f \\
G[n:] f &\equiv X[n] G f \\
F[n:] f &\equiv X[n!] F f \\
G[n:] f &\equiv X[n!] G f
\end{aligned}$$

2.4.3. Trivial Identities (Occur Automatically)

$$\begin{array}{lll}
X[!] 0 \equiv 0 & F 0 \equiv 0 & G 0 \equiv 0 \\
X 1 \equiv 1 & F 1 \equiv 1 & G 1 \equiv 1 \\
FF f \equiv F f & & GG f \equiv G f
\end{array}$$

$f \cup 1 \equiv 1$	$f \vee 1 \equiv 1$	$f \cdot 0 \equiv 0$	$f \cdot 1 \equiv 1$
$0 \cup f \equiv f$	$0 \vee f \equiv f$	$0 \cdot f \equiv 0$	$f \cdot 0 \equiv 0$
$f \cup 0 \equiv 0$	$1 \vee f \equiv 1$	$1 \cdot f \equiv f$	$1 \cdot f \equiv f$
$f \cup f \equiv f$	$f \vee f \equiv f$	$f \cdot f \equiv f$	$f \cdot f \equiv f$

2.5. SERE Operators

The “SERE” acronym will be translated to different word depending on the source. It can mean either “*Sequential Extended Regular Expression*” [? ?], “*Sugar Extended Regular Expression*” [?], or “*Semi-Extended Regular Expression*” [?]. In any case, the intent is the same: regular expressions with traditional operations (union ‘|’, concatenation ‘;’, Kleen star ‘[*]’) are extended with operators such as intersection ‘&&’, and fusion ‘:’.

Any Boolean formula (section 2.3) is a SERE. SERE can be further combined with the following operators, where f and g denote arbitrary SERE.

operation	preferred syntax	other supported syntaxes			UTF8 characters supported	
					preferred	others
empty word	[*0]					
union	$f g$	$f g$	$f \setminus / g$	$f + g$		\vee U+2228 \cup U+222A
intersection	$f \&\& g$	$f \wedge g$			\cap U+2229	\wedge U+2227
NLM intersection ²	$f \& g$					
concatenation	$f ; g$					
fusion	$f : g$					
bounded ;-iter.	$f[*i..j]$	$f[*i:j]$	$f[*i \text{ to } j]$	$f[*i,j]$		
unbounded ;-iter.	$f[*i..]$	$f[*i:]$	$f[*i \text{ to}]$	$f[*i,]$		
bounded :-iter.	$f[:*i..j]$	$f[:*i:j]$	$f[:*i \text{ to } j]$	$f[:*i,j]$		
unbounded :-iter.	$f[:*i..]$	$f[:*i:]$	$f[:*i \text{ to}]$	$f[:*i,]$		
first match	<code>first_match(f)</code>					

The character ‘\$’ or the string ‘inf’ can also be used as value for j in the above operators to denote an unbounded range.³ For instance ‘ $a[*i, \$]$ ’, ‘ $a[*i:inf]$ ’ and ‘ $a[*i..]$ ’ all represent the same SERE.

2.5.1. Semantics

The following semantics assume that f and g are two SEREs, while a is an atomic proposition.

$$\begin{aligned}
\sigma 0 & \\
\sigma 1 & \iff |\sigma| = 1 \\
\sigma [*0] & \iff |\sigma| = 0 \\
\sigma a & \iff \sigma(0)(a) = 1 \wedge |\sigma| = 1 \\
\sigma f | g & \iff (\sigma f) \vee (\sigma g) \\
\sigma f \&\& g & \iff (\sigma f) \wedge (\sigma g) \\
\sigma f \& g & \iff \exists k \in N, \begin{cases} \text{either} & (\sigma f) \wedge (\sigma^{0..k-1} g) \\ \text{or} & (\sigma^{0..k-1} f) \wedge (\sigma g) \end{cases} \\
\sigma f ; g & \iff \exists k \in N, (\sigma^{0..k-1} f) \wedge (\sigma^k g)
\end{aligned}$$

²Non-Length-Matching interesction.

³SVA uses ‘\$’ while PSL uses ‘inf’.

$$\begin{aligned}
\sigma f : g &\iff \exists k \in N, (\sigma^{0..k} f) \wedge (\sigma^{k..} g) \\
\sigma f[*i..j] &\iff \begin{cases} \text{either} & i = 0 \wedge \sigma = \varepsilon \\ \text{or} & i = 0 \wedge j > 0 \wedge (\exists k \in N, (\sigma^{0..k-1} f) \wedge (\sigma^{k..} f[*0..j-1])) \\ \text{or} & i > 0 \wedge j > 0 \wedge (\exists k \in N, (\sigma^{0..k-1} f) \wedge (\sigma^{k..} f[*i-1..j-1])) \end{cases} \\
\sigma f[*i..] &\iff \begin{cases} \text{either} & i = 0 \wedge \sigma = \varepsilon \\ \text{or} & i = 0 \wedge (\exists k \in N, (\sigma^{0..k-1} f) \wedge (\sigma^{k..} f[*0..])) \\ \text{or} & i > 0 \wedge (\exists k \in N, (\sigma^{0..k-1} f) \wedge (\sigma^{k..} f[*i-1..])) \end{cases} \\
\sigma f[:*i..j] &\iff \begin{cases} \text{either} & i = 0 \wedge j = 0 \wedge \sigma 1 \\ \text{or} & i = 0 \wedge j > 0 \wedge (\exists k \in N, (\sigma^{0..k} f) \wedge (\sigma^{k..} f[:*0..j-1])) \\ \text{or} & i > 0 \wedge j > 0 \wedge (\exists k \in N, (\sigma^{0..k} f) \wedge (\sigma^{k..} f[:*i-1..j-1])) \end{cases} \\
\sigma f[:*i..] &\iff \begin{cases} \text{either} & i = 0 \wedge \sigma 1 \\ \text{or} & i = 0 \wedge (\exists k \in N, (\sigma^{0..k} f) \wedge (\sigma^{k..} f[:*0..])) \\ \text{or} & i > 0 \wedge (\exists k \in N, (\sigma^{0..k} f) \wedge (\sigma^{k..} f[:*i-1..])) \end{cases} \\
\sigma \text{first_match}(f) &\iff (\sigma f) \wedge (\forall k < |\sigma|, \sigma^{0..k} f)
\end{aligned}$$

Notes:

- The semantics of $\&\&$ and $\&$ coincide if both operands are Boolean formulas.
- The SERE $f : g$ will never hold on $[*0]$, regardless of the value of f and g . For instance $a[*] : b[*]$ is actually equivalent to $a[*] ; \{a \&\& b\} ; b[*]$.
- The $[:*i..]$ and $[:*i..j]$ operators are iterations of the $:$ operator just like The $[*i..]$ and $[*i..j]$ are iterations of the $;$ operator. More graphically:

$$\begin{aligned}
f[*i..j] &= \underbrace{f ; f ; \dots ; f}_{\text{between } i \text{ and } j \text{ copies of } f} & f[:*i..j] &= \underbrace{f : f : \dots : f}_{\text{between } i \text{ and } j \text{ copies of } f}
\end{aligned}$$

with the convention that

$$f[*0..0] = [*0] \qquad f[:*0..0] = 1$$

- The $[:*i..]$ and $[:*i..j]$ operators are not defined in PSL. While the bounded iteration can be seen as syntactic sugar on $:$, the unbounded version really is a new operator.

$[:*1..]$, for which we define the $[:+]$ syntactic sugar below, actually corresponds to the \oplus operator introduced by ?]. With this simple addition, it is possible to define a subset of PSL that expresses exactly the stutter-invariant ω -regular languages.

- The `first_match` operator does not exist in PSL. It comes from SystemVerilog Assertions (SVA) [?]. One intuition behind `first_match(f)` is that the DFA for `first_match(f)` can be obtained from the DFA for f by removing all transitions leaving accepting states.

2.5.2. Syntactic Sugar

The syntax on the left is equivalent to the syntax on the right. These rewritings are performed from left to right when parsing a formula, and *some* are performed from right to left when writing it for output. b must be a Boolean formula.

$$\begin{aligned}
b[->i..j] &\equiv \{\{!b\}[*0..]; b\}[*i..j] & b[=i..j] &\equiv \{\{!b\}[*0..]; b\}[*i..j]; \{!b\}[*0..] \\
b[->i..] &\equiv \{\{!b\}[*0..]; b\}[*i..] & b[=i..] &\equiv \{\{!b\}[*0..]; b\}[*i..]; \{!b\}[*0..] \text{ if } i > 0 \\
b[=0..] &\equiv 1[*0..]
\end{aligned}$$

$$\begin{aligned}
f* &\equiv f[*0..] \\
f[*] &\equiv f[*0..] & f[:*] &\equiv f[:*0..] & f[=] &\equiv f[=0..] & f[->] &\equiv f[->1..1] \\
f[*..] &\equiv f[*0..] & f[:*..] &\equiv f[:*0..] & f[=..] &\equiv f[=0..] & f[->..] &\equiv f[->1..] \\
f[*..j] &\equiv f[*0..j] & f[:*..j] &\equiv f[:*0..j] & f[=..j] &\equiv f[=0..j] & f[->..j] &\equiv f[->1..j] \\
f[*k] &\equiv f[*k..k] & f[:*k] &\equiv f[:*k..k] & f[=k] &\equiv f[=k..k] & f[->k] &\equiv f[->k..k] \\
f[+] &\equiv f[*1..] & f[:+] &\equiv f[:*1..]
\end{aligned}$$

$$[*k] \equiv 1[*k..k] \qquad [*] \equiv 1[*0..] \qquad [+] \equiv 1[*1..]$$

The following adds input support for the SVA concatenation (or delay) operator $[?]$. The simplest equivalence are that $f \## 0 g$, $f \## 1 g$, $f \## 2 g$ mean respectively $f : g$, $f ; g$, and $f ; 1 ; g$, but the delay can be a range, and f can be omitted.

$$\begin{aligned}
f \## [i..j] g &\equiv f ; 1[*i-1..j-1] ; g \quad \text{if } i > 0 \\
f \## [0..j] g &\equiv f : (1[*0..j] ; g) \quad \text{if } \varepsilon f \\
f \## [0..j] g &\equiv (f ; 1[*0..j]) : g \quad \text{if } \varepsilon f \wedge \varepsilon g \\
f \## [0..j] g &\equiv (f : g) \mid (f ; 1[*0..j-1] ; g) \quad \text{if } \varepsilon f \wedge \varepsilon g \\
f \## [i..] g &\equiv f ; 1[*i-1..] ; g \quad \text{if } i > 0 \\
f \## [0..] g &\equiv f : (1[*] ; g) \quad \text{if } \varepsilon f \\
f \## [0..] g &\equiv (f ; 1[*]) : g \quad \text{if } \varepsilon f \wedge \varepsilon g \\
f \## [0..] g &\equiv (f : g) \mid (f ; 1[*] ; g) \quad \text{if } \varepsilon f \wedge \varepsilon g
\end{aligned}$$

$$\begin{aligned}
\## [i..j] g &\equiv 1[*i..j] ; g & \## [i..] g &\equiv 1[*i..] ; g \\
f \## i g &\equiv f \## [i..i] g & \## i g &\equiv 1[*i] ; g \\
f \## [+] g &\equiv f \## [1..] g & \## [+] g &\equiv \## [1..] g \\
f \## [*] g &\equiv f \## [0..] g & \## [*] g &\equiv \## [0..] g \\
f \## [..j] g &\equiv f \## [0..j] g \} & \## [..j] g &\equiv 1 \## [0..j] g \} \\
f \## [..] g &\equiv f \## [0..] f g \} & \## [..] g &\equiv 1 \## [0..] g \}
\end{aligned}$$

2.5.3. Trivial Identities (Occur Automatically)

The following identities also hold if j or l are missing (assuming they are then equal to ∞). f can be any SERE, while b , b_1 , b_2 are assumed to be Boolean formulas.

$$\begin{array}{ll}
0[*0..j] \equiv [*0] & 0[*i..j] \equiv 0 \text{ if } i > 0 \\
[*0][*i..j] \equiv [*0] & f[*i..j][*k..l] \equiv f[*ik..jl] \text{ if } i(k+1) \leq jk+1 \\
f[*0] \equiv [*0] & f[*1] \equiv f \\
b[*0..j] \equiv 1 & b[*i..j] \equiv b \text{ if } i > 0 \\
[*0][*:0..j] \equiv 1 & [*0][*:i..j] \equiv 0 \text{ if } i > 0 \\
f[:*0] \equiv 1 & f[:*i..j][:*k..l] \equiv f[:*ik..jl] \text{ if } i(k+1) \leq jk+1 \\
\text{first_match}(b) \equiv b & f[:*1] \equiv f \text{ if } \varepsilon f \\
& \text{first_match}(f) \equiv [*0] \text{ if } \varepsilon f \\
& \text{first_match}(\text{first_match}(f)) \equiv \text{first_match}(f)
\end{array}$$

The following rules are all valid with the two arguments swapped.

$$\begin{array}{llllll}
0 \& f \equiv 0 & 0 \&\& f \equiv 0 & 0 \mid f \equiv f & 0 : f \equiv 0 & 0 ; f \equiv 0 \\
1 \& f \equiv \begin{cases} 1 & \text{if } \varepsilon f \\ f & \text{if } \varepsilon f \end{cases} & 1 \&\& b \equiv b & 1 \mid b \equiv 1 & 1 : f \equiv f \text{ if } \varepsilon f & \\
[*] \& f \equiv f & [*] \mid f \equiv 1[*] & & & [*] ; f \equiv [*] \text{ if } \varepsilon f & \\
[*0] \& f \equiv f & [*0] \&\& f \equiv \begin{cases} [*0] & \text{if } \varepsilon f \\ 0 & \text{if } \varepsilon f \end{cases} & [*0] : f \equiv 0 & & [*0] ; f \equiv f \\
f \& f \equiv f & f \&\& f \equiv f & f \mid f \equiv f & f : f \equiv f[:*2] & f ; f \equiv f[*2] \\
b_1 \& b_2 \equiv b_1 \&\& b_2 & & & b_1 : b_2 \equiv b_1 \&\& b_2 \\
f[*i..j] ; f \equiv f[*i+1..j+1] & & f[*i..j] ; f[*k..l] \equiv f[*i+k..j+l] & & & \\
f[:*i..j] : f \equiv f[:*i+1..j+1] & & f[:*i..j] : f[:*k..l] \equiv f[:*i+k..j+l] & & &
\end{array}$$

2.6. SERE-LTL Binding Operators

The following operators combine a SERE r with a PSL formula f to form another PSL formula.

operation	preferred syntax	other supported syntaxes
(universal) suffix implication	$\{r\}[] \rightarrow f$	$\{r\} \mid \rightarrow f$ $\{r\}(f)$
existential suffix implication	$\{r\} \langle \rightarrow \rangle f$	
weak closure	$\{r\}$	
negated weak closure	$!\{r\}$	

For technical reasons, the negated weak closure is actually implemented as an operator, even if it is syntactically and semantically equal to the combination of $!$ and $\{r\}$.

UTF-8 input may combine one box or diamond character from section 2.4 with one arrow character from section 2.3 to replace the operators $[] \rightarrow$, $\langle \rightarrow \rangle$, as well as the operators $[] \Rightarrow$ and $\langle \Rightarrow \rangle$ that will be defined in 2.6.2. Additionally, $\mid \rightarrow$ may be replaced by \mapsto U+21A6, and $\mid \Rightarrow$ by U+2907.

2.6.1. Semantics

The following semantics assume that r is a SERE, while f is a PSL formula.

$$\begin{aligned}
\sigma\{r\} \langle \rangle \rightarrow f &\iff \exists k \geq 0, (\sigma^{0..k} r) \wedge (\sigma^k f) \\
\sigma\{r\} [] \rightarrow f &\iff \forall k \geq 0, (\sigma^{0..k} r) \rightarrow (\sigma^k f) \\
\sigma\{r\} &\iff (\exists k \geq 0, \sigma^{0..k} r) \vee (\forall k \geq 0, \exists \pi \in (B^{\text{AP}})^*, (\sigma^{0..k} \prec \pi) \wedge (\pi r)) \\
\sigma!\{r\} &\iff (\forall k \geq 0, \sigma^{0..k} r) \wedge (\exists k \geq 0, \forall \pi \in (B^{\text{AP}})^*, (\sigma^{0..k} \prec \pi) \rightarrow (\pi r))
\end{aligned}$$

The \prec symbol should be read as “is a prefix of”. So the semantic for ‘ $\sigma\{r\}$ ’ is that either there is a (non-empty) finite prefix of σ that is a model of r , or any prefix of σ can be extended into a finite sequence π that is a model of r . An infinite sequence $a; a; a; a; \dots$ is therefore a model of the formula ‘ $\{a[+] ; ! a\}$ ’ even though it never sees ‘ $! a$ ’. The same sequence is not a model of ‘ $\{a[+] ; ! a ; (a[*] \&\& (a[*] ; ! a ; a[*]))\}$ ’ because this SERE does not accept any word.

2.6.2. Syntactic Sugar

The syntax on the left is equivalent to the syntax on the right. These rewritings are performed from left to right when parsing a formula. Except the one marked with $\overset{+}{\equiv}$, the opposite rewritings are also preformed on output to ease reading.

$$\begin{aligned}
\{r\} \langle \rangle \Rightarrow f &\equiv \{r ; 1\} \langle \rangle \rightarrow f & \{r\} [] \Rightarrow f &\equiv \{r ; 1\} [] \rightarrow f \\
\{r\} ! &\equiv \{r\} \langle \rangle \rightarrow 1 & \{r\} \mid \Rightarrow f &\overset{+}{\equiv} \{r ; 1\} [] \rightarrow f
\end{aligned}$$

$[] \Rightarrow$ and $\mid \Rightarrow$ are synonyms in the same way as $[] \rightarrow$ and $\mid \rightarrow$ are.

The $\{r\} !$ operator is a *strong closure* operator.

2.6.3. Trivial Identities (Occur Automatically)

For any PSL formula f , any SERE r , and any Boolean formula b , the following rewritings are systematically performed (from left to right).

$$\begin{array}{llll}
\{0\} [] \rightarrow f \equiv 1 & \{0\} \langle \rangle \rightarrow f \equiv 0 & \{0\} \equiv 0 & !\{0\} \equiv 1 \\
\{1\} [] \rightarrow f \equiv f & \{1\} \langle \rangle \rightarrow f \equiv f & \{1\} \equiv 1 & !\{1\} \equiv 0 \\
\{[*0]\} [] \rightarrow f \equiv 1 & \{[*0]\} \langle \rangle \rightarrow f \equiv 0 & \{[*0]\} \equiv 0 & !\{[*0]\} \equiv 1 \\
\{b\} [] \rightarrow f \equiv (! b) \mid f & \{b\} \langle \rangle \rightarrow f \equiv b \& f & \{b\} \equiv b & !\{b\} \equiv ! b \\
\{r\} [] \rightarrow 1 \equiv 1 & \{r\} \langle \rangle \rightarrow 0 \equiv 0 & &
\end{array}$$

3. Grammar

For simplicity, this grammar gives only one rule for each operator, even if the operator has multiple synonyms (like '|', '||', and '\/').

```

constant ::= 0 | 1
atomic_prop ::= see secn 2.2

bformula ::= constant
           | ( bformula )
           | bformula xor bformula
           | atomic_prop
           | ! bformula
           | bformula <-> bformula
           | atomic_prop=0
           | bformula & bformula
           | bformula -> bformula
           | atomic_prop=1
           | bformula | bformula


sere ::= bformula
       | { sere }
       | ( sere )
       | sere | sere
       | sere & sere
       | sere && sere
       | sere ; sere
       | sere : sere
       | [*i . .j]
       | [+]
       | sere [*i . .j]
       | sere [+]
       | sere [:*i . .j]
       | sere [:+]
       | sere [=i . .j]
       | sere [->i . .j]
       | ##i sere
       | ##[i . .j] sere
       | sere ##i sere
       | sere ##[i . .j] sere
       | first_match( sere )

tformula ::= bformula
           | X tformula
           | { sere } [] -> tformula
           | ( tformula )
           | X[!] tformula
           | { sere } [] => tformula
           | ! tformula
           | X[i . .j] tformula
           | { sere } <-> tformula
           | tformula & tformula
           | X[i . .j!] tformula
           | { sere } <=> tformula
           | tformula | tformula
           | F tformula
           | { sere }
           | tformula -> tformula
           | F[i . .j] tformula
           | { sere } !
           | tformula xor tformula
           | F[i . .j!] tformula
           | tformula <-> tformula
           | G tformula
           | tformula U tformula
           | G[i . .j] tformula
           | tformula W tformula
           | G[i . .j!] tformula
           | tformula R tformula
           | tformula M tformula

```

3.1. Operator precedence

The following operator precedence describes the current parser of Spot. It has not always been this way. Especially, all operators were left associative until version 0.9, when we changed the associativity of ->, <->, U, R, W, and M to get closer to the PSL standard [? ?].

assoc.	operators	priority
right	[] ->, [] =>, <>->, <>=>	lowest
left	;	
left	:	
left	##i, ##[i..j]	
right	->, <->	
left	xor	
left		
left	&, &&	
right	U, W, M, R	
	F, G, F[i..j], G[i..j]	
	X, X[i..j]	
	[*i..j], [+], [:*i..j], [:+], [=i..j], [->i..j]	
	!	
	=0, =1	
		highest

Beware that not all tools agree on the associativity of these operators. For instance Spin, ltl2ba (same parser as spin), Wring, psl2ba, Modella, and NuSMV all have U and R as left-associative, while Goal (hence Büchi store), LTL2AUT, and LTL2Büchi (from JavaPathFinder) have U and R as right-associative. Vis and LBTT have these two operators as non-associative (parentheses required). Similarly the tools do not agree on the associativity of -> and <->: some tools handle both operators as left-associative, or both right-associative, other have only -> as right-associative.

4. Properties

When Spot builds a formula (represented by an AST with shared subtrees) it computes a set of properties for each node. These properties can be queried from any `spot::formula` instance using the following methods:

<code>is_boolean()</code>	Whether the formula uses only Boolean operators.
<code>is_sugar_free_boolean()</code>	Whether the formula uses only <code>&</code> , <code> </code> , and <code>!</code> operators. (Especially, no <code>-></code> or <code><-></code> are allowed.)
<code>is_in_nenoform()</code>	Whether the formula is in negative normal form. See section 5.3.
<code>is_X_free()</code>	Whether the formula avoids the <code>X</code> operator.
<code>is_ltl_formula()</code>	Whether the formula uses only LTL operators. (Boolean operators are also allowed.)
<code>is_psl_formula()</code>	Whether the formula uses only PSL operators. (Boolean and LTL operators are also allowed.)
<code>is_sere_formula()</code>	Whether the formula uses only SERE operators. (Boolean operators are also allowed, provided no SERE operator is negated.)
<code>is_finite()</code>	Whether a SERE describes a finite language (no unbounded stars), or an LTL formula uses no temporal operator but <code>X</code> .
<code>is_eventual()</code>	Whether the formula is a pure eventuality.
<code>is_universal()</code>	Whether the formula is purely universal.
<code>is_syntactic_safety()</code>	Whether the formula is a syntactic safety property.
<code>is_syntactic_guarantee()</code>	Whether the formula is a syntactic guarantee property.
<code>is_syntactic_obligation()</code>	Whether the formula is a syntactic obligation property.
<code>is_syntactic_recurrence()</code>	Whether the formula is a syntactic recurrence property.
<code>is_syntactic_persistence()</code>	Whether the formula is a syntactic persistence property.
<code>is_marked()</code>	Whether the formula contains a special “marked” version of the <code><-></code> or <code>!{r}</code> operators. ³
<code>accepts_eword()</code>	Whether the formula accepts <code>[*0]</code> . (This can only be true for a SERE formula.)
<code>has_lbt_atomic_props()</code>	Whether the atomic propositions of the formula are all of the form “ <code>pnn</code> ” where <code>nn</code> is a string of digits. This is required when converting formula into LBT’s format. ⁴

4.1. Pure Eventualities and Purely Universal Formulas

These two syntactic classes of formulas were introduced by ?] to simplify LTL formulas. We shall present the associated simplification rules in Section 5.4.2, for now we only define these two classes.

Pure eventual formulas describe properties that are left-append closed, i.e., any accepted (infinite) sequence can be prefixed by a finite sequence and remain accepted. From an LTL standpoint, if φ is a left-append closed formula, then $F \varphi \equiv \varphi$.

Purely universal formulas describe properties that are suffix-closed, i.e., if you remove any finite prefix of an accepted (infinite) sequence, it remains accepted. From an LTL standpoint if φ is a suffix-closed formula, then $G \varphi \equiv \varphi$.

³These “marked” operators are used when translating recurring `<->` or `!{r}` operators. They are rendered as `<->+` and `!+{r}` and obey the same simplification rules and properties as their unmarked counterpart (except for the `is_marked()` property).

⁴<http://www.tcs.hut.fi/Software/maria/tools/lbt/>

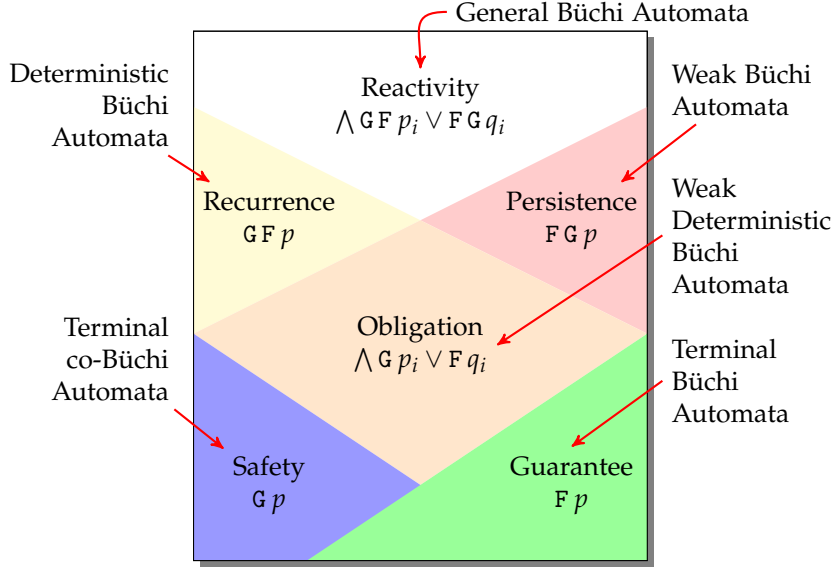


Figure 4.1.: The temporal hierarchy of [?] with their associated classes of automata [?]. The formulas associated to each class are more than canonical examples: they show the normal forms under which any LTL formula of the class can be rewritten, assuming that p, p_i, q, q_i denote subformulas involving only Boolean operators, X , and past temporal operators (Spot does not support the latter).

Let φ denote any arbitrary formula and φ_E (resp. φ_U) denote any instance of a pure eventuality (resp. a purely universal) formula. We have the following grammar rules:

$$\begin{aligned}
\varphi_E ::= & 0 \mid 1 \mid X \varphi_E \mid X[!] \varphi_E \mid F \varphi \mid G \varphi_E \mid \varphi_E \& \varphi_E \mid (\varphi_E \mid \varphi_E) \mid ! \varphi_U \\
& \mid \varphi \cup \varphi_E \mid 1 \cup \varphi \mid \varphi_E R \varphi_E \mid \varphi_E W \varphi_E \mid \varphi_E M \varphi_E \mid \varphi M 1 \\
\varphi_U ::= & 0 \mid 1 \mid X \varphi_U \mid X[!] \varphi_U \mid F \varphi_U \mid G \varphi \mid \varphi_U \& \varphi_U \mid (\varphi_U \mid \varphi_U) \mid ! \varphi_E \\
& \mid \varphi_U \cup \varphi_U \mid \varphi R \varphi_U \mid 0 R \varphi \mid \varphi_U W \varphi_U \mid \varphi W 0 \mid \varphi_U M \varphi_U
\end{aligned}$$

4.2. Syntactic Hierarchy Classes

The hierarchy of linear temporal properties was introduced by [?] and is illustrated on Fig. 4.1. In the case of the LTL subset of the hierarchy, a first syntactic characterization of the classes was presented by [?], but other presentations have been done including negation [?] and weak until [?].

The following grammar rules extend the aforementioned work slightly by dealing with PSL operators. These are the rules used by Spot to decide upon construction to which class a formula belongs (see the methods `is_syntactic_safety()`, `is_syntactic_guarantee()`, `is_syntactic_obligation()`, `is_syntactic_recurrence()`, and `is_syntactic_persistence()` listed on page 14).

The symbols $\varphi_G, \varphi_S, \varphi_O, \varphi_P, \varphi_R$ denote any formula belonging respectively to the Guarantee, Safety, Obligation, Persistence, or Recurrence classes. Additionally φ_B denotes a finite LTL formula (the unnamed class at the intersection of Safety and Guarantee formulas, at the bottom of Fig. 4.1). v denotes any variable, r any SERE, r_F any bounded SERE (no loops), and r_I any unbounded SERE.

$$\begin{aligned}
\varphi_B &::= 0 \mid 1 \mid v \mid !\varphi_B \mid \varphi_B \& \varphi_B \mid (\varphi_B \mid \varphi_B) \mid \varphi_B \leftrightarrow \varphi_B \mid \varphi_B \text{ xor } \varphi_B \mid \varphi_B \rightarrow \varphi_B \mid \text{X } \varphi_B \\
&\quad \mid \{r_F\} \mid !\{r_F\} \\
\varphi_G &::= \varphi_B \mid !\varphi_S \mid \varphi_G \& \varphi_G \mid (\varphi_G \mid \varphi_G) \mid \varphi_S \rightarrow \varphi_G \mid \text{X } \varphi_G \mid \text{F } \varphi_G \mid \varphi_G \cup \varphi_G \mid \varphi_G \text{M } \varphi_G \\
&\quad \mid !\{r\} \mid \{r\} \leftrightarrow \varphi_G \mid \{r_F\} \square \rightarrow \varphi_G \\
\varphi_S &::= \varphi_B \mid !\varphi_G \mid \varphi_S \& \varphi_S \mid (\varphi_S \mid \varphi_S) \mid \varphi_G \rightarrow \varphi_S \mid \text{X } \varphi_S \mid \text{G } \varphi_S \mid \varphi_S \text{R } \varphi_S \mid \varphi_S \text{W } \varphi_S \\
&\quad \mid \{r\} \mid \{r_F\} \leftrightarrow \varphi_S \mid \{r\} \square \rightarrow \varphi_S \\
\varphi_O &::= \varphi_G \mid \varphi_S \mid !\varphi_O \mid \varphi_O \& \varphi_O \mid (\varphi_O \mid \varphi_O) \mid \varphi_O \leftrightarrow \varphi_O \mid \varphi_O \text{ xor } \varphi_O \mid \varphi_O \rightarrow \varphi_O \\
&\quad \mid \text{X } \varphi_O \mid \varphi_O \cup \varphi_G \mid \varphi_O \text{R } \varphi_S \mid \varphi_S \text{W } \varphi_O \mid \varphi_G \text{M } \varphi_O \\
&\quad \mid \{r\} \mid !\{r\} \mid \{r_F\} \leftrightarrow \varphi_O \mid \{r_I\} \leftrightarrow \varphi_G \mid \{r_F\} \square \rightarrow \varphi_O \mid \{r_I\} \square \rightarrow \varphi_S \\
\varphi_P &::= \varphi_O \mid !\varphi_R \mid \varphi_P \& \varphi_P \mid (\varphi_P \mid \varphi_P) \mid \varphi_P \leftrightarrow \varphi_P \mid \varphi_P \text{ xor } \varphi_P \mid \varphi_P \rightarrow \varphi_P \\
&\quad \mid \text{X } \varphi_P \mid \text{F } \varphi_P \mid \varphi_P \cup \varphi_P \mid \varphi_P \text{R } \varphi_S \mid \varphi_S \text{W } \varphi_P \mid \varphi_P \text{M } \varphi_P \\
&\quad \mid \{r\} \leftrightarrow \varphi_P \mid \{r_F\} \square \rightarrow \varphi_P \mid \{r_I\} \square \rightarrow \varphi_S \\
\varphi_R &::= \varphi_O \mid !\varphi_P \mid \varphi_R \& \varphi_R \mid (\varphi_R \mid \varphi_R) \mid \varphi_R \leftrightarrow \varphi_R \mid \varphi_R \text{ xor } \varphi_R \mid \varphi_R \rightarrow \varphi_R \\
&\quad \mid \text{X } \varphi_R \mid \text{G } \varphi_R \mid \varphi_R \cup \varphi_G \mid \varphi_R \text{R } \varphi_R \mid \varphi_R \text{W } \varphi_R \mid \varphi_G \text{M } \varphi_R \\
&\quad \mid \{r\} \square \rightarrow \varphi_R \mid \{r_F\} \leftrightarrow \varphi_R \mid \{r_I\} \leftrightarrow \varphi_G
\end{aligned}$$

It should be noted that a formula can belong to a class of the temporal hierarchy even if it does not syntactically appear so. For instance the formula $(\text{G}(q \mid \text{F G } p) \& \text{G}(r \mid \text{F G } ! p)) \mid \text{G } q \mid \text{G } r$ is not syntactically safe, yet it is a safety formula equivalent to $\text{G } q \mid \text{G } r$. Such a formula is usually said *pathologically safe*.

5. Rewritings

5.1. Unabbreviations

The ‘unabbreviate()’ function can apply the following rewriting rules when passed a string denoting the list of rules to apply. For instance passing the string “~ei” will rewrite all occurrences of xor, <-> and ->.

“i”	$f \rightarrow g \equiv (!f) \mid g$	
“e”	$f \leftrightarrow g \equiv (f \& g) \mid ((!g) \& (!f))$	
“~e”	$f \text{ xor } g \equiv (f \& !g) \mid (g \& !f)$	
“~” without “e”	$f \text{ xor } g \equiv !(f \leftrightarrow g)$	
“F”	$F e \equiv e$	when e is a pure eventuality
“F”	$F f \equiv 1 \cup f$	
“G”	$G u \equiv u$	when u is purely universal
“G” without “R”	$G f \equiv 0 \cap f$	
“GR” without “W”	$G f \equiv f \cap 0$	
“GRW”	$G f \equiv !F!f$	
“M”	$f M e \equiv F(f \& e)$	when e is a pure eventuality
“M”	$f M g \equiv g \cup (g \& f)$	
“R”	$f R u \equiv u$	when u is purely universal
“R” without “W”	$f R g \equiv g \cap (f \& g)$	
“RW”	$f R g \equiv g \cup ((f \& g) \mid Gg)$	
“W”	$f W u \equiv G(f \mid u)$	when u is purely universal
“W” without “R”	$f W g \equiv g \cap (g \mid f)$	
“WR”	$f W g \equiv f \cup (g \mid Gf)$	

Among all the possible rewritings (see Appendix A) the default rules for R, W and M, those were chosen because they are easier to translate in a tableau construction [?, Fig. 11].

Besides the ‘unabbreviate()’ function, there is also a class ‘unabbreviator()’ that implements the same functionality, but maintains a cache of abbreviated subformulas. This is preferable if you plan to abbreviate many formulas sharing identical subformulas.

5.2. LTL simplifier

The LTL rewritings described in the next three sections are all implemented in the ‘tl_simplifier’ class defined in spot/tl/simplify.hh. This class implements several caches in order to quickly rewrite formulas that have already been rewritten previously. For this reason, it is suggested that you reuse your instance of ‘tl_simplifier’ as much as possible. If you write an algorithm that will simplify LTL formulas, we suggest you accept an optional ‘tl_simplifier’ argument, so that you can benefit from an existing instance.

The ‘tl_simplifier’ takes an optional ‘tl_simplifier_options’ argument, making it possible to tune the various rewritings that can be performed by this class. These options cannot be changed afterwards (because changing these options would invalidate the results stored in the caches).

5.3. Negative normal form

This is implemented by the `tl_simplifier::negative_normal_form` method.

A formula in negative normal form can only have negation operators (!) in front of atomic properties, and does not use any of the xor, -> and <-> operators. The following rewriting arrange any PSL formula into negative normal form.

$$\begin{array}{lll}
 !Xf \equiv X!f & !(f \cup g) \equiv (!f) R (!g) & !(f \& g) \equiv (!f) \mid (!g) \\
 !Ff \equiv G!f & !(f R g) \equiv (!f) \cup (!g) & !(f \mid g) \equiv (!f) \& (!g) \\
 !Gf \equiv F!f & !(f W g) \equiv (!f) M (!g) & !(\{r\} [] \rightarrow f) \equiv \{r\} <-> !f \\
 !(\{r\}) \equiv !\{r\} & !(f M g) \equiv (!f) W (!g) & !(\{r\} <-> f) \equiv \{r\} [] \rightarrow !f
 \end{array}$$

Recall that the negated weak closure $!\{r\}$ is actually implemented as a specific operator, so it is not actually prefixed by the ! operator.

$$\begin{array}{lll}
 f \text{ xor } g \equiv ((!f) \& g) \mid (f \& !g) & !(f \text{ xor } g) \equiv ((!f) \& (!g)) \mid (f \& g) & !(f \& g) \equiv (!f) \mid (!g) \\
 f <-> g \equiv ((!f) \& (!g)) \mid (f \& g) & !(f <-> g) \equiv ((!f) \& g) \mid (f \& !g) & !(f \mid g) \equiv (!f) \& (!g) \\
 f \rightarrow g \equiv (!f) \mid g & !(f \rightarrow g) \equiv f \& !g &
 \end{array}$$

Note that the above rules include the “unabbreviation” of operators “<->”, “->”, and “xor”, correspondings to the rules “ei” of function `unabbreviate()` as described in Section 5.1. Therefore it is never necessary to apply these abbreviations before or after `tl_simplifier::negative_normal_form`.

If the option `nenofrm_stop_on_boolean` is set, the above recursive rewritings are not applied to Boolean subformulas. For instance calling `tl_simplifier::negative_normal_form` on $!FG(a \text{ xor } b)$ will produce $GF(((!a) \& (!b)) \mid (a \& b))$ if `nenofrm_stop_on_boolean` is unset, while it will produce $GF(! (a \text{ xor } b))$ if `nenofrm_stop_on_boolean` is set.

5.4. Simplifications

The `tl_simplifier::simplify` method performs several kinds of simplifications, depending on which `tl_simplifier_options` was set.

The goals in most of these simplification are to:

- remove useless terms and operator.
- move the X operators to the front of the formula (e.g., XGf is better than the equivalent GXf). This is because LTL translators will usually want to rewrite LTL formulas in a kind of disjunctive form: $\bigvee_i (\beta_i \wedge X\psi_i)$ where β_i s are Boolean formulas and ψ_i s are LTL formulas. Moving X to the front therefore simplifies the translation.
- move the F operators to the front of the formula (e.g., $F(f \mid g)$ is better than the equivalent $(Ff) \mid (Fg)$), but not before X (XFf is better than FXf). Because Ff incurs some indeterminism, it is best to factorize these terms to limit the sources of indeterminism.

Rewritings defined with \equiv are applied only when `tl_simplifier_options::favor_event_univ` is true: they try to lift subformulas that are both eventual and universal *higher* in the syntax tree. Conversely, rules defined with \equiv are applied only when `favor_event_univ` is false: they try to *lower* subformulas that are both eventual and universal.

Currently all these simplifications assume LTL semantics, so they make no differences between X and X[!]. For simplicity, they are only listed with X.

5.4.1. Basic Simplifications

These simplifications are enabled with `tl_simplifier_options::reduce_basics'`. A couple of them may enlarge the size of the formula: they are denoted using $\stackrel{*}{\equiv}$ instead of \equiv , and they can be disabled by setting the `tl_simplifier_options::reduce_size_strictly'` option to `true`.

Basic Simplifications for Temporal Operators

The following are simplification rules for unary operators (applied from left to right, as usual). The terms $\text{dnf}(f)$ and $\text{cnf}(f)$ denote respectively the disjunctive and conjunctive normal forms of f , handling non-Boolean sub-formulas as if they were atomic propositions.

$$\begin{array}{lll}
XFGf \equiv FGf & F(fUg) \equiv Fg & G(fRg) \equiv Gg \\
XGFf \equiv GFf & F(fMg) \equiv F(f \& g) & G(fWg) \equiv G(f \mid g) \\
FXf \equiv XFf & FG(f \& Xg) \equiv FG(f \& g) & GF(f \mid Xg) \equiv GF(f \mid g) \\
GXf \equiv XGf & FG(f \& Gg) \equiv FG(f \& g) & GF(f \mid Fg) \equiv GF(f \mid g) \\
X0 \equiv 0 & FG(f \mid Gg) \equiv F(Gf \mid Gg) & GF(f \& Fg) \equiv G(Ff \& Fg) \\
GFf \stackrel{*}{\equiv} GF(\text{dnf}(f)) & FG(f \& Fg) \stackrel{*}{\equiv} FGf \& GFg & GF(f \& Gg) \stackrel{*}{\equiv} GFf \& FGg \\
FGf \stackrel{*}{\equiv} FG(\text{cnf}(f)) & FG(f \mid Fg) \equiv FGf \mid GFg & GF(f \mid Gg) \equiv GFf \mid FGg
\end{array}$$

$$G(f_1 \mid \dots \mid f_n \mid GF(g_1) \mid \dots \mid GF(g_m)) \equiv G(f_1 \mid \dots \mid f_n) \mid GF(g_1 \mid \dots \mid g_m)$$

Here are the basic rewriting rules for binary operators (excluding \mid and $\&$ which are considered in Spot as n -ary operators). b denotes any Boolean formula.

$$\begin{array}{ll}
1Uf \equiv Ff & fW0 \equiv Gf \\
fM1 \equiv Ff & 0Rf \equiv Gf \\
(Xf)U(Xg) \equiv X(fUg) & (Xf)W(Xg) \equiv X(fWg) \\
(Xf)M(Xg) \equiv X(fMg) & (Xf)R(Xg) \equiv X(fRg) \\
(Xf)Ub \stackrel{*}{\equiv} b \mid X(bMf) & (Xf)Wb \stackrel{*}{\equiv} b \mid X(fRb) \\
(Xf)Mb \stackrel{*}{\equiv} b \& X(bUf) & (Xf)Rb \stackrel{*}{\equiv} b \& X(fWb) \\
fU(Gf) \equiv Gf & fW(Gf) \equiv Gf \\
fM(Ff) \equiv Ff & fR(Ff) \equiv Ff \\
fU(g \mid G(f)) \equiv fWg & fW(g \mid G(f)) \equiv fWg \\
fM(g \& F(f)) \equiv fMg & fR(g \& F(f)) \equiv fMg \\
fU(g \& f) \equiv gMf & fW(g \& f) \equiv gRf \\
fM(g \mid f) \equiv gUf & fR(g \mid f) \equiv gWf
\end{array}$$

Here are the basic rewriting rules for n -ary operators (& and |):

$$\begin{array}{ll}
(FGf) \& (FGg) \equiv FG(f \& g) & (GFf) | (GFg) \equiv GF(f | g) \\
(Xf) \& (Xg) \equiv X(f \& g) & (Xf) | (Xg) \equiv X(f | g) \\
(Xf) \& (FGg) \equiv X(f \& FGg) & (Xf) | (GFg) \equiv X(f | GFg) \\
(Gf) \& (Gg) \equiv G(f \& g) & (Ff) | (Fg) \equiv F(f | g) \\
(f_1 U f_2) \& (f_3 U f_2) \equiv (f_1 \& f_3) U f_2 & (f_1 U f_2) | (f_1 U f_3) \equiv f_1 U (f_2 | f_3) \\
(f_1 U f_2) \& (f_3 W f_2) \equiv (f_1 \& f_3) U f_2 & (f_1 U f_2) | (f_1 W f_3) \equiv f_1 W (f_2 | f_3) \\
(f_1 W f_2) \& (f_3 W f_2) \equiv (f_1 \& f_3) W f_2 & (f_1 W f_2) | (f_1 W f_3) \equiv f_1 W (f_2 | f_3) \\
(f_1 R f_2) \& (f_1 R f_3) \equiv f_1 R (f_2 \& f_3) & (f_1 R f_2) | (f_3 R f_2) \equiv (f_1 | f_3) R f_2 \\
(f_1 R f_2) \& (f_1 M f_3) \equiv f_1 M (f_2 \& f_3) & (f_1 R f_2) | (f_3 M f_2) \equiv (f_1 | f_3) R f_3 \\
(f_1 M f_2) \& (f_1 M f_3) \equiv f_1 M (f_2 \& f_3) & (f_1 M f_2) | (f_3 M f_2) \equiv (f_1 | f_3) M f_3 \\
(Fg) \& (f Ug) \equiv f Ug & (Gf) | (f Ug) \equiv f W g \\
(Fg) \& (f W g) \equiv f Ug & (Gf) | (f W g) \equiv f W g \\
(Ff) \& (f R g) \equiv f M g & (Gg) | (f R g) \equiv f R g \\
(Ff) \& (f M g) \equiv f M g & (Gg) | (f M g) \equiv f R g \\
f \& ((Xf) W g) \equiv g R f & f | ((Xf) R g) \equiv g W f \\
f \& ((Xf) U g) \equiv g M f & f | ((Xf) M g) \equiv g U f \\
f \& (g | X(g R f)) \equiv g R f & f | (g \& X(g W f)) \equiv g W f \\
f \& (g | X(g M f)) \equiv g M f & f | (g \& X(g U f)) \equiv g U f
\end{array}$$

The above rules are applied even if more terms are presents in the operator's arguments. For instance $FG(a) \& G(b) \& FG(c) \& X(d)$ will be rewritten as $X(d \& FG(a \& c)) \& G(b)$.

The following more complicated rules are generalizations of $f \& XGf \equiv Gf$ and $f | XFf \equiv Ff$:

$$\begin{array}{l}
f \& X(G(f \& g \& \dots) \& h \& \dots) \equiv G(f) \& X(G(g \& \dots) \& h \& \dots) \\
f | X(F(f) | h | \dots) \equiv F(f) | X(h | \dots)
\end{array}$$

The latter rule for $f | X(F(f) | h \dots)$ is only applied if all F-formulas can be removed from the argument of X with the rewriting. For instance $a | b | c | X(F(a | b) | F(c) | Gd)$ will be rewritten to $F(a | b | c) | XGd$ but $b | c | X(F(a | b) | F(c) | Gd)$ will only become $b | c | X(F(a | b | c) | Gd)$.

Finally the following rule is applied only when no other terms are present in the OR arguments:

$$F(f_1) | \dots | F(f_n) | GF(g) \equiv F(f_1 | \dots | f_n | GF(g))$$

Basic Simplifications for SERE Operators

The following rules, mostly taken from [?] are not complete yet. We only show those that are implemented.

The following simplification rules are used for the n -ary operators $\&\&$, $\&$, and $|$. The patterns are of course commutative. b or b_i denote any Boolean formula while r or r_i denote any SERE.

$$\begin{aligned}
b \&\& r[*i..j] &\equiv \begin{cases} b \&\& r & \text{if } i \leq 1 \leq j \\ 0 & \text{else} \end{cases} & b \&\& r \stackrel{*}{\equiv} \begin{cases} b \mid \{b : r\} & \text{if } \varepsilon r_i \\ b : r & \text{if } \varepsilon \not\vdash_i \end{cases} \\
b \&\& \{r_1 : \dots : r_n\} &\equiv b \&\& r_1 \&\& \dots \&\& r_n \\
b \&\& \{r_1 ; \dots ; r_n\} &\equiv \begin{cases} b \&\& r_i & \text{if } \exists ! i, \varepsilon \not\vdash_i \\ b \&\& (r_1 \mid \dots \mid r_n) & \text{if } \forall i, \varepsilon r_i \\ 0 & \text{else} \end{cases} \\
\{b_1 ; r_1\} \&\& \{b_2 ; r_2\} &\equiv \{b_1 \&\& b_2\} ; \{r_1 \&\& r_2\} & \{r_1 ; b_1\} \&\& \{r_2 ; b_2\} &\equiv \{r_1 \&\& r_2\} ; \{b_1 \&\& b_2\} \\
\{b_1 : r_1\} \&\& \{b_2 : r_2\} &\equiv \{b_1 \&\& b_2\} : \{r_1 \&\& r_2\} & \{r_1 : b_1\} \&\& \{r_2 : b_2\} &\equiv \{r_1 \&\& r_2\} : \{b_1 \&\& b_2\} \\
\{b_1 ; r_1\} \&\& \{b_2 ; r_2\} &\equiv \{b_1 \&\& b_2\} ; \{r_1 \&\& r_2\} \\
\{b_1 : r_1\} \&\& \{b_2 : r_2\} &\equiv \{b_1 \&\& b_2\} : \{r_1 \&\& r_2\} \quad \text{if } \varepsilon r_1 \wedge \varepsilon r_2
\end{aligned}$$

$$\begin{aligned}
\text{first_match}(b[*i..j]) &\equiv b[*i] \\
\text{first_match}(r[*i..j]) &\equiv \text{first_match}(r[*i]) \\
\text{first_match}(r[:*i..j]) &\equiv \text{first_match}(r[:*i]) \\
\text{first_match}(r_1 ; r_2[*i..j]) &\equiv \text{first_match}(r_1 ; r_2[*i]) \\
\text{first_match}(r_1 ; r_2[:*i..j]) &\equiv \text{first_match}(r_1 ; r_2[:*i]) \\
\text{first_match}(r_1 : r_2[*i..j]) &\equiv \text{first_match}(r_1 : r_2[*\max(i, 1)]) \\
\text{first_match}(r_1 : r_2[:*i..j]) &\equiv \text{first_match}(r_1 : r_2[:*i]) \\
\text{first_match}(b ; r) &\equiv b ; \text{first_match}(r) \\
\text{first_match}(b[*i..j] ; r) &\equiv b[*i] ; \text{first_match}(b[*0..j-i] ; r) \\
\text{first_match}(b[*i..j] : r) &\equiv b[*i-1] ; \text{first_match}(b[*1..j-i+1] : r) \quad \text{if } i > 1 \\
\text{first_match}(r_1 ; r_2) &\equiv \text{first_match}(r_1) \quad \text{if } \varepsilon r_2 \\
\text{first_match}(\text{first_match}(r_1) ; r_2) &\equiv \text{first_match}(r_1) ; \text{first_match}(r_2) \\
\text{first_match}(\text{first_match}(r_1) : r_2) &\equiv \text{first_match}(r_1) : \text{first_match}(1 : r_2)
\end{aligned}$$

Starred subformulas are rewritten in Star Normal Form [?] with:

$$r[*] \equiv r^\circ[*]$$

where r° is recursively defined as follows:

$$\begin{aligned}
r^\circ &= r \text{ if } \varepsilon \not\vdash \\
[*0]^\circ &= 0 & (r_1 ; r_2)^\circ &= r_1^\circ \mid r_2^\circ \text{ if } \varepsilon r_1 \text{ and } \varepsilon r_2 \\
r[*i..j]^\circ &= r^\circ \text{ if } i = 0 \text{ or } \varepsilon r & (r_1 \&\& r_2)^\circ &= r_1^\circ \mid r_2^\circ \text{ if } \varepsilon r_1 \text{ and } \varepsilon r_2 \\
(r_1 \mid r_2)^\circ &= r_1^\circ \mid r_2^\circ & (r_1 \&\& r_2)^\circ &= r_1 \&\& r_2
\end{aligned}$$

Note: the original SNF definition [?] does not include ‘&’ and ‘&&’ operators, and it guarantees that $\forall r, \varepsilon \not\vdash r^\circ$ because r° is stripping all the stars and empty words that occur in r . For instance $\{a[*] ; b[*] ; \{[*0] \mid c\}\}^\circ[*] = \{a \mid b \mid c\}[*]$. Our extended definition still respects this property in presence of ‘&’ operators, but unfortunately not when the ‘&&’ operator is used.

We extend the above definition to bounded repetitions with:

$$\begin{aligned}
r[*i..j] &\equiv r^\square[*0..j] \quad \text{if } \varepsilon r[*i..j], \varepsilon \not\vdash^\square, \text{ and } j > 1 \\
r[*i..j] &\equiv r^\square[*1..j] \quad \text{if } \varepsilon r[*i..j], \varepsilon r^\square \text{ and } j > 1 \\
r[*i..j] &\equiv r \quad \text{if } \varepsilon r \text{ and } j = 1
\end{aligned}$$

where r^\square is recursively defined as follows:

$$\begin{aligned}
r^\square &= r \text{ if } \varepsilon \neq \text{ } \\
[*0]^\square &= 0 & (r_1 ; r_2)^\square &= r_1 ; r_2 \\
r[*i..j]^\square &= r^\square[*\max(1,i)..j] \text{ if } i = 0 \text{ or } \varepsilon & (r_1 \& r_2)^\square &= r_1^\square \mid r_2^\square \text{ if } \varepsilon r_1 \text{ and } \varepsilon r_2 \\
(r_1 \mid r_2)^\square &= r_1^\square \mid r_2^\square & (r_1 \&\& r_2)^\square &= r_1 \&\& r_2
\end{aligned}$$

The differences between \square and \circ are in the handling of $r[*i..j]$ and in the handling of $r_1 ; r_2$.

Basic Simplifications SERE-LTL Binding Operators

The following rewritings are applied to the operators $[] \rightarrow$ and $\langle \rangle \rightarrow$. They assume that b , denote a Boolean formula.

As noted at the beginning for section 5.4.1, rewritings denoted with $\stackrel{*}{\equiv}$ can be disabled by setting the `tl_simplifier_options::reduce_size_strictly'` option to true.

$$\begin{aligned}
\{[*]\}[] \rightarrow f &\equiv G f \\
\{b[*]\}[] \rightarrow f &\equiv f W ! b \\
\{b[+]\}[] \rightarrow f &\equiv f W ! b \\
\{r[*0..j]\}[] \rightarrow f &\equiv \{r[*1..j]\}[] \rightarrow f \\
\{r[*i..j]\}[] \rightarrow f &\stackrel{*}{\equiv} \{r\}[] \rightarrow X(\{r\}[] \rightarrow X(\dots [] \rightarrow X(r[*1..j-i+1]))) \text{ if } i \geq 1 \text{ and } \varepsilon \neq \text{ } \\
\{r ; [*]\}[] \rightarrow f &\equiv \{r\}[] \rightarrow G f \\
\{r ; b[*]\}[] \rightarrow f &\stackrel{*}{\equiv} \{r\}[] \rightarrow (f \& X(f W ! b)) \text{ if } \varepsilon \neq \text{ } \\
\{[*] ; r\}[] \rightarrow f &\stackrel{*}{\equiv} G(\{r\}[] \rightarrow f) \\
\{b[*] ; r\}[] \rightarrow f &\stackrel{*}{\equiv} (! b) R (\{r\}[] \rightarrow f) \text{ if } \varepsilon \neq \text{ } \\
\{r_1 ; r_2\}[] \rightarrow f &\stackrel{*}{\equiv} \{r_1\}[] \rightarrow X(\{r_2\}[] \rightarrow f) \text{ if } \varepsilon \neq r_1 \text{ and } \varepsilon \neq r_2 \\
\{r_1 : r_2\}[] \rightarrow f &\stackrel{*}{\equiv} \{r_1\}[] \rightarrow (\{r_2\}[] \rightarrow f) \\
\{r_1 \mid r_2\}[] \rightarrow f &\stackrel{*}{\equiv} (\{r_1\}[] \rightarrow f) \& (\{r_2\}[] \rightarrow f) \\
\{[*]\}\langle \rangle \rightarrow f &\equiv F f \\
\{b[*]\}\langle \rangle \rightarrow f &\equiv f M b \\
\{b[+]\}\langle \rangle \rightarrow f &\equiv f M b \\
\{r[*0..j]\}\langle \rangle \rightarrow f &\equiv \{r[*1..j]\}\langle \rangle \rightarrow f \\
\{r[*i..j]\}\langle \rangle \rightarrow f &\stackrel{*}{\equiv} \{r\}\langle \rangle \rightarrow X(\{r\}\langle \rangle \rightarrow X(\dots \langle \rangle \rightarrow X(r[*1..j-i+1]))) \text{ if } i \geq 1 \text{ and } \varepsilon \neq \text{ } \\
\{r ; [*]\}\langle \rangle \rightarrow f &\equiv \{r\}\langle \rangle \rightarrow F f \\
\{r ; b[*]\}\langle \rangle \rightarrow f &\stackrel{*}{\equiv} \{r\}\langle \rangle \rightarrow (f \mid X(f M b)) \text{ if } \varepsilon \neq \text{ } \\
\{[*] ; r\}\langle \rangle \rightarrow f &\stackrel{*}{\equiv} F(\{r\}\langle \rangle \rightarrow f) \\
\{b[*] ; r\}\langle \rangle \rightarrow f &\stackrel{*}{\equiv} b U (\{r\}\langle \rangle \rightarrow f) \text{ if } \varepsilon \neq \text{ } \\
\{r_1 ; r_2\}\langle \rangle \rightarrow f &\stackrel{*}{\equiv} \{r_1\}\langle \rangle \rightarrow X(\{r_2\}\langle \rangle \rightarrow f) \text{ if } \varepsilon \neq r_1 \text{ and } \varepsilon \neq r_2 \\
\{r_1 : r_2\}\langle \rangle \rightarrow f &\stackrel{*}{\equiv} \{r_1\}\langle \rangle \rightarrow (\{r_2\}\langle \rangle \rightarrow f) \\
\{r_1 \mid r_2\}\langle \rangle \rightarrow f &\stackrel{*}{\equiv} (\{r_1\}\langle \rangle \rightarrow f) \mid (\{r_2\}\langle \rangle \rightarrow f)
\end{aligned}$$

Here are the basic rewritings for the weak closure and its negation:

$$\begin{array}{ll}
\{r[*]\} \equiv \{r\} & !\{r[*]\} \equiv !\{r\} \\
\{r;1\} \equiv \{r\} \quad \text{if } \varepsilon \neq & !\{r;1\} \equiv !\{r\} \quad \text{if } \varepsilon \neq \\
\{r;1\} \equiv 1 \quad \text{if } \varepsilon r & !\{r;1\} \equiv 0 \quad \text{if } \varepsilon r \\
\{r_1;r_2\} \equiv \{r_1\} \quad \text{if } \varepsilon \neq \wedge \varepsilon r_2 & !\{r_1;r_2\} \equiv !\{r_1\} \quad \text{if } \varepsilon \neq \wedge \varepsilon r_2 \\
\{r_1;r_2\} \equiv \{r_1\} \mid \{r_2\} \quad \text{if } \varepsilon r_1 \wedge \varepsilon r_2 & !\{r_1;r_2\} \equiv !\{r_1\} \& !\{r_2\} \quad \text{if } \varepsilon r_1 \wedge \varepsilon r_2 \\
\{b;r\} \stackrel{*}{\equiv} b \& X\{r\} & !\{b;r\} \stackrel{*}{\equiv} (!b) \mid X!\{r\} \\
\{b[*i..j];r\} \stackrel{*}{\equiv} \underbrace{b \& X(b \dots \& X\{b[*0..j-i];r\})}_{i \text{ occurrences of } b} & !\{b[*i..j];r\} \stackrel{*}{\equiv} \underbrace{(!b) \mid X((!b) \dots \mid X!\{b[*0..j-i];r\})}_{i \text{ occurrences of } !b} \\
\{b[*i..j]\} \stackrel{*}{\equiv} \underbrace{b \& X(b \& X(\dots b))}_{i \text{ occurrences of } b} & !\{b[*i..j]\} \stackrel{*}{\equiv} \underbrace{(!b) \mid X((!b) \mid X(\dots (!b)))}_{i \text{ occurrences of } !b} \\
\{r_1 \mid r_2\} \stackrel{*}{\equiv} \{r_1\} \mid \{r_2\} & !\{r_1 \mid r_2\} \stackrel{*}{\equiv} !\{r_1\} \& !\{r_2\}
\end{array}$$

5.4.2. Simplifications for Eventual and Universal Formulas

The class of *pure eventuality* and *purely universal* formulas are described in section 4.1.

In the following rewritings, we use the following notation to distinguish the class of subformulas:

f, f_i, g, g_i	any PSL formula
e, e_i	a pure eventuality
u, u_i	a purely universal formula
q, q_i	a pure eventuality that is also purely universal

$$\begin{array}{llll}
Fe \equiv e & f \cup e \equiv e & e \mathcal{M} g \equiv e \& g & u_1 \mathcal{M} u_2 \stackrel{*}{\equiv} (F u_1) \& u_2 \\
F(u) \mid q \equiv F(u \mid q) & f \cup (g \mid e) \equiv (f \cup g) \mid e & f \mathcal{M} (g \& u) \equiv (f \mathcal{M} g) \& u & q \cup X f \equiv X(q \cup f) \\
& f \cup (g \& q) \equiv (f \cup g) \& q & (f \& q) \mathcal{M} g \equiv (f \mathcal{M} g) \& q & \\
Gu \equiv u & u \mathcal{W} g \equiv u \mid g & f \mathcal{R} u \equiv u & e_1 \mathcal{W} e_2 \stackrel{*}{\equiv} (G e_1) \mid e_2 \\
G(e) \& q \equiv G(e \& q) & f \mathcal{W} (g \mid e) \equiv (f \mathcal{W} g) \mid e & f \mathcal{R} (g \& u) \equiv (f \mathcal{R} g) \& u & q \mathcal{R} X f \equiv X(q \mathcal{R} f) \\
Xq \equiv q & q \& X f \equiv X(q \& f) & q \mid X f \equiv X(q \mid f) & \\
& X(q \& f) \equiv q \& X f & X(q \mid f) \equiv q \mid X f &
\end{array}$$