1. Reasoning with Infinite Sequences 3
   1.1. Finite and Infinite Sequences 3
   1.2. Usage in Model Checking 3

2. Temporal Syntax & Semantics 4
   2.1. Boolean Constants 4
      2.1.1. Semantics 4
   2.2. Atomic Propositions 4
      2.2.1. Examples 5
      2.2.2. Semantics 5
   2.3. Boolean Operators (for Temporal Formulas) 5
      2.3.1. Semantics 6
      2.3.2. Trivial Identities (Occur Automatically) 6
   2.4. Temporal Operators 6
      2.4.1. Semantics 7
      2.4.2. Syntactic Sugar 7
      2.4.3. Trivial Identities (Occur Automatically) 7
   2.5. SERE Operators 8
      2.5.1. Semantics 8
      2.5.2. Syntactic Sugar 9
      2.5.3. Trivial Identities (Occur Automatically) 10
   2.6. SERE-LTL Binding Operators 11
      2.6.1. Semantics 11
      2.6.2. Syntactic Sugar 12
      2.6.3. Trivial Identities (Occur Automatically) 12

3. Grammar 13
   3.1. Operator precedence 13

4. Properties 15
   4.1. Pure Eventualities and Purely Universal Formulas 15
   4.2. Syntactic Hierarchy Classes 16

5. Rewritings 18
   5.1. Unabbreviations 18
   5.2. LTL simplifier 18
   5.3. Negative normal form 19
   5.4. Simplifications 19
      5.4.1. Basic Simplifications 20
      5.4.2. Simplifications for Eventual and Universal Formulas 24
      5.4.3. Simplifications Based on Implications 25
A. Defining LTL with only one of U, W, R, or M

B. Syntactic Implications

Bibliography
1. Reasoning with Infinite Sequences

1.1. Finite and Infinite Sequences

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

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

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

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 \textit{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 \( \text{AP} \) be a set of atomic propositions, for instance \( \text{AP} = \{r, y, g\} \). A configuration of the model is a function \( \rho : \text{AP} \rightarrow B \) (or \( \rho \in B^{\text{AP}} \)) that associates a truth value \( (B = \{0, 1\}) \) to each atomic proposition. A behavior of the model is an infinite sequence \( \sigma \) of such configurations. In other words: \( \sigma \in (B^{\text{AP}})^\omega \).

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

When a formula \( \varphi \) holds on a finite sequence \( \sigma \), we write \( \sigma \Vdash \varphi \).
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

\[ \sigma \neq 0 \]
\[ \sigma = 1 \]

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 propositions, and the double quotes are not necessary.

Here is the list of reserved keywords:

- ‘true’, ‘false’ (both are case insensitive)

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
- ‘FX100’ is not an atomic proposition, it is equivalent to the formula ‘F(X100)’, where ‘X100’ is an atomic proposition by rule

2.2.2. Semantics

For any atomic proposition a, we have

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

In other words a holds if and only if it is true in the first configuration of \(\sigma\).

2.3. Boolean Operators (for Temporal Formulas)

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

<table>
<thead>
<tr>
<th>operation</th>
<th>preferred syntax</th>
<th>other supported syntaxes</th>
<th>UTF8 characters supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td>!f</td>
<td>~f</td>
<td>¬ U+00AC</td>
</tr>
</tbody>
</table>
| disjunction | f \| g            | f \| g \/
                        | f \| g               | f \| g \| g \| g          | \/or U+2228 \/or U+222A |
| conjunction | f \& g           | f \& g \& f            | \& U+2227 \& U+2229       |
| implication | f \rightarrow g  | f \rightarrow g \rightarrow g | \rightarrow U+2192 \rightarrow U+27F6 \rightarrow U+21D2 U+27F9 |
| exclusion   | f \xor g         | f \xor g \xor g \xor g \xor g | \xor U+2295               |
| equivalence | f \leftrightarrow g | f \leftrightarrow g \leftrightarrow g | \leftrightarrow U+2194 \leftrightarrow U+21D4 |

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 ‘a’ 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

\[\sigma \models ! f \iff (\sigma \not\models f)\]
\[\sigma \models f \land g \iff (\sigma \models f) \land (\sigma \models g)\]
\[\sigma \models f \lor g \iff (\sigma \models f) \lor (\sigma \models g)\]
\[\sigma \models f \rightarrow g \iff (\sigma \not\models f) \lor (\sigma \models g)\]
\[\sigma \models f \oplus g \iff ((\sigma \models f) \land (\sigma \not\models g)) \lor ((\sigma \not\models f) \land (\sigma \models g))\]
\[\sigma \models f \leftrightarrow g \iff ((\sigma \models f) \land (\sigma \models g)) \lor ((\sigma \not\models f) \land (\sigma \not\models g))\]

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’.

\[!0 = 1\]
\[1 \rightarrow f = f\]
\[f \rightarrow 1 = 1\]
\[!1 = 0\]
\[0 \rightarrow f = 1\]
\[f \rightarrow 0 = !f\]
\[!!f = f\]
\[f \rightarrow f = 1\]

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

\[0 \& f = 0\]
\[0 \lor f = f\]
\[0 \oplus f = f\]
\[0 \leftrightarrow f = !f\]
\[1 \& f = f\]
\[1 \lor f = 1\]
\[1 \oplus f = !f\]
\[1 \leftrightarrow f = f\]
\[f \& f = f\]
\[f \lor f = f\]
\[f \oplus f = 0\]
\[f \leftrightarrow f = 1\]

The ‘&’ and ‘|’ 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.

<table>
<thead>
<tr>
<th>operator</th>
<th>preferred syntax</th>
<th>other supported syntaxes</th>
<th>UTF8 characters supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Weak) Next</td>
<td>(X f)</td>
<td>() (f)</td>
<td>☐ U+25CB ☐ U+25EF</td>
</tr>
<tr>
<td>Strong Next</td>
<td>(X[!] f)</td>
<td>(&lt;&gt; f)</td>
<td>☐ U+24CD</td>
</tr>
<tr>
<td>Eventually</td>
<td>(F f)</td>
<td>(\Box f)</td>
<td>☐ U+25C7 ☐ U+2204 ☐ U+2662</td>
</tr>
<tr>
<td>Always</td>
<td>(G f)</td>
<td>(\diamond f)</td>
<td>☐ U+25A1 ☐ U+2B1C ☐ U+25FB</td>
</tr>
<tr>
<td>(Strong) Until</td>
<td>(f \Uparrow g)</td>
<td>(f \vee g)</td>
<td>☐ U+25CB ☐ U+25EF</td>
</tr>
<tr>
<td>Weak Until</td>
<td>(f \Updownarrow g)</td>
<td>(f \land g)</td>
<td>☐ U+25CB ☐ U+25EF</td>
</tr>
<tr>
<td>(Weak) Release</td>
<td>(f \Leftrightarrow g)</td>
<td>(f \Leftrightarrow g)</td>
<td>☐ U+25CB ☐ U+25EF</td>
</tr>
<tr>
<td>Strong Release</td>
<td>(f \Leftrightarrow g)</td>
<td>(f \Leftrightarrow g)</td>
<td>☐ U+25CB ☐ U+25EF</td>
</tr>
</tbody>
</table>
2.4.1. Semantics

\[ \sigma \models X f \iff \sigma^{n} \models f \]
\[ \sigma \models X[!] f \iff \sigma^{n} \models f \]
\[ \sigma \models F f \iff \exists i \in \mathbb{N}, \sigma^{i} \models f \]
\[ \sigma \models G f \iff \forall i \in \mathbb{N}, \sigma^{i} \models f \]
\[ \sigma \models f \cup g \iff \exists j \in \mathbb{N}, \left\{ \begin{array}{l}
\forall i < j, \sigma^{j-i} \models g \\
\sigma^{j-i} \models f
\end{array} \right. \]
\[ \sigma \models f \rightarrow g \iff (\sigma \models f \cup g) \lor (\sigma \models G f) \]
\[ \sigma \models f \rightarrow \neg g \iff (\sigma \models f \cup \neg g) \lor (\sigma \models G f) \]

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 [14] 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.

\[ X[n] f = X X \ldots X f \]
\[ F[n; m] f = X X \ldots X (f \mid X(f \mid X(\ldots \mid X f)) \mid \ldots) \]
\[ G[n; m] f = X X \ldots X (f \& X(f \& X(\ldots \& X f)) \mid \ldots) \]
\[ X[!] n f = X[!] X[!] \ldots X[!] f \]
\[ F[n; m!] f = X[!] X[!] \ldots X[!] (f \mid X[!] (f \mid X[!] (\ldots \mid X[!] f)) \mid \ldots) \]
\[ G[n; m!] f = X[!] X[!] \ldots X[!] (f \& X[!] (f \& X[!] (\ldots \& X[!] f)) \mid \ldots) \]

2.4.3. Trivial Identities (Occur Automatically)

\[ X[!] 0 = 0 \]
\[ F 0 = 0 \]
\[ G 0 = 0 \]
\[ X 1 = 1 \]
\[ F 1 = 1 \]
\[ G 1 = 1 \]
\[ F f F f = F f \]
\[ G f G f = G f \]
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” [11], “Sugar Extended Regular Expression” [4], or “Semi-Extended Regular Expression” [12]. 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.

<table>
<thead>
<tr>
<th>operation</th>
<th>preferred syntax</th>
<th>other supported syntaxes</th>
<th>UTF8 characters supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty word</td>
<td>([\ast]0)</td>
<td>([\ast])</td>
<td>∨ U+2228 U+2222</td>
</tr>
<tr>
<td>union</td>
<td>(f \mid g)</td>
<td>(f \mid __ g)</td>
<td>(\land U+2229 \land U+2227)</td>
</tr>
<tr>
<td>intersection</td>
<td>(f \land g)</td>
<td>(f \land __ g)</td>
<td></td>
</tr>
<tr>
<td>NLM intersection</td>
<td>(f \land g)</td>
<td>(f \land __ g)</td>
<td></td>
</tr>
<tr>
<td>concatenation</td>
<td>(f ; g)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fusion</td>
<td>(f : g)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bounded :-iter.</td>
<td>([i..j])</td>
<td>([i..j])</td>
<td></td>
</tr>
<tr>
<td>unbounded :-iter.</td>
<td>([i..j])</td>
<td>([i..j])</td>
<td></td>
</tr>
<tr>
<td>bounded :-iter.</td>
<td>([i..j])</td>
<td>([i..j])</td>
<td></td>
</tr>
<tr>
<td>unbounded :-iter.</td>
<td>([i..j])</td>
<td>([i..j])</td>
<td></td>
</tr>
<tr>
<td>first match</td>
<td>(\text{first_match}(f))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The character ‘$’ or the string ‘inf’ can also be used as value for \( j \) in the above operators to denote an unbounded range. 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.

\[
\begin{align*}
\sigma \models 0 & \quad \iff \quad |\sigma| = 0 \\
\sigma \models 1 & \quad \iff \quad |\sigma| = 1 \\
\sigma \models [*0] & \quad \iff \quad |\sigma| = 0 \\
\sigma \models a & \quad \iff \quad \sigma(0)(a) = 1 \land |\sigma| = 1 \\
\sigma \models f \mid g & \quad \iff \quad (\sigma \models f) \lor (\sigma \models g) \\
\sigma \models f \land g & \quad \iff \quad (\sigma \models f) \land (\sigma \models g) \\
\sigma \models f \land g & \quad \iff \quad \exists k \in \mathbb{N}, \begin{cases} (\sigma \models f) \land (\sigma^{0..k-1} \models g) \\ (\sigma^{0..k-1} \models f) \land (\sigma^k \models g) \end{cases} \\
\end{align*}
\]

\(^2\text{Non-Length-Matching intersection.}\)

\(^3\text{SVA uses ‘$’ while PSL uses ‘inf’.}\)
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.

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.
The following identities also hold if $i > 0\). $j$

### 2.5.3. Trivial Identities (Occur Automatically)

The following identities also hold if $j$ or $l$ are missing (assuming they are then equal to $\infty$). $f$ can be any SERE, while $b$, $b_1$, $b_2$ are assumed to be Boolean formulas.
The following semantics assume that $r$ is a SERE, while $f$ is a PSL formula.

---

2.6. SERE-LTL Binding Operators

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

<table>
<thead>
<tr>
<th>operation</th>
<th>preferred syntax</th>
<th>other supported syntaxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(universal) suffix implication</td>
<td>$[r] \rightarrow f$</td>
<td>$f$</td>
</tr>
<tr>
<td>existential suffix implication</td>
<td>$[r] \leftarrow f$</td>
<td>$f$</td>
</tr>
<tr>
<td>weak closure</td>
<td>$[r]$</td>
<td>$f$</td>
</tr>
<tr>
<td>negated weak closure</td>
<td>$\neg [r]$</td>
<td>$f$</td>
</tr>
</tbody>
</table>

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 $\neg$ and $[r]$.

UTF-8 input may combine one box or diamond character from section 2.4 with one arrow character from section 2.5 to replace the operators $[\rightarrow], \leftarrow$, as well as the operators $[\Rightarrow]$ and $\Leftarrow$ that will be defined in 2.6.2. Additionally, $\Leftarrow$ may be replaced by $\backsimeq \backslash U+21A6$, and $\Rightarrow$ by $\Leftrightarrow U+2907$.

2.6.1. Semantics

The following semantics assume that $r$ is a SERE, while $f$ is a PSL formula.
\[ \sigma \models \{ r \}\triangleleft f \iff \exists k \geq 0, (\sigma^{0,k} \models r) \land (\sigma \models f) \]
\[ \sigma \models \{ r \} \downarrow \rightarrow f \iff \forall k \geq 0, (\sigma^{0,k} \models r) \rightarrow (\sigma \models f) \]
\[ \sigma \models \{ r \} \iff (\exists k \geq 0, \sigma^{0,k} \models r) \lor (\forall k \geq 0, \exists \pi \in (B^{AP})^*, (\sigma^{0,k} < \pi) \land (\pi \models r)) \]
\[ \sigma \models \{ r \} \iff (\forall k \geq 0, \sigma^{0,k} \not\models r) \land (\exists \pi \in (B^{AP})^*, (\sigma^{0,k} < \pi) \rightarrow (\pi \not\models r)) \]

The \(<\) symbol should be read as “is a prefix of”. So the semantic for ‘\(\sigma \models \{ r \}\)’ is that either there is a (non-empty) finite prefix of \(\sigma\) that is a model of \(r\), or any prefix of \(\sigma\) can be extended into a finite sequence \(\pi\) that is a model of \(r\). An infinite sequence \(a;a;a;a;\ldots\) 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 \(\uparrow\), the opposite rewritings are also preformed on output to ease reading.

\[ \{ r \} \triangleright \rightarrow f = \{ r ; 1 \} \triangleright \rightarrow f \]
\[ \{ r \} \downarrow \rightarrow f = \{ r ; 1 \} \downarrow \rightarrow f \]
\[ \{ r \} \uparrow = \{ r \} \triangleright \rightarrow 1 \]
\[ \{ r \} \downarrow \rightarrow f \uparrow = \{ r ; 1 \} \downarrow \rightarrow f \]

\(\triangleright \rightarrow\) and \(\downarrow \rightarrow\) are synonyms in the same way as \(\downarrow \rightarrow\) and \(\downarrow \rightarrow\) are.

The \(\{ r \} \uparrow\) 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).

\[ \{ 0 \} \downarrow \rightarrow f = 1 \quad \{ 0 \} \triangleright \rightarrow f = 0 \quad \{ 0 \} = 0 \quad \uparrow \{ 0 \} = 1 \]
\[ \{ 1 \} \downarrow \rightarrow f = f \quad \{ 1 \} \triangleright \rightarrow f = f \quad \{ 1 \} = 1 \quad \uparrow \{ 1 \} = 0 \]
\[ \{ [+]0 \} \downarrow \rightarrow f = 1 \quad \{ [+]0 \} \triangleright \rightarrow f = 0 \quad \{ [+]0 \} = 0 \quad \uparrow \{ [+]0 \} = 1 \]
\[ \{ b \} \downarrow \rightarrow f = (\uparrow b) \lor f \quad \{ b \} \triangleright \rightarrow f = b \& f \quad \{ b \} = b \quad \uparrow \{ b \} = \uparrow b \]
\[ \{ r \} \downarrow \rightarrow 1 = 1 \quad \{ r \} \triangleright \rightarrow 0 = 0 \]
3. Grammar

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

\[
\begin{align*}
\text{constant} & ::= 0 \mid 1 \\
\text{atomic\_prop} & ::= \text{see secn 2.2} \\
\text{bformula} & ::= \text{constant} \mid (\text{bformula}) \mid \text{bformula} \text{ xor } \text{bformula} \\
& \quad \mid \text{atomic\_prop} \mid \text{atomic\_prop}=0 \mid \text{atomic\_prop}=1 \\
& \quad \mid !\text{bformula} \mid \text{bformula} \text{ & } \text{bformula} \mid \text{bformula} \rightarrow \text{bformula} \\
\text{sere} & ::= \text{bformula} \mid [^*i..j] \mid \#\#\text{sere} \\
& \quad \mid \{\text{sere}\} \mid [+ \text{sere}] \mid \#\#[i..j] \text{sere} \\
& \quad \mid (\text{sere}) \mid \text{sere}[^*i..j] \mid \text{sere}[^*] \mid \text{first\_match}(\text{sere}) \\
& \quad \mid \text{sere} & \text{sere} \mid \text{sere} \text{ & } \text{sere} \\
& \quad \mid \text{sere} ; \text{sere} \mid \text{sere}[^i..j] \mid \text{sere}[^*] \\
& \quad \mid \text{sere} ;\text{sere} \mid \text{sere}[->i..j] \\
\text{tformula} & ::= \text{bformula} \mid X\text{tformula} \mid \{\text{sere}\} [] \rightarrow \text{tformula} \\
& \quad \mid (\text{tformula}) \mid X[i..j]\text{tformula} \mid \{\text{sere}\} []\rightarrow\text{tformula} \\
& \quad \mid !\text{tformula} \mid X[i..j]\text{tformula} \mid \{\text{sere}\} <>\rightarrow\text{tformula} \\
& \quad \mid \text{tformula} \& \text{tformula} \mid X[i..j]\text{tformula} \mid \{\text{sere}\} \leftrightarrow\rightarrow\text{tformula} \\
& \quad \mid \text{tformula} \mid \text{tformula} \mid F\text{tformula} \mid \{\text{sere}\} \\
& \quad \mid \text{tformula} \rightarrow \text{tformula} \mid F[i..j]\text{tformula} \mid \{\text{sere}\} ! \\
& \quad \mid \text{tformula} \text{ xor } \text{tformula} \mid F[i..j]\text{tformula} \\
& \quad \mid \text{tformula} \leftrightarrow \text{tformula} \mid G\text{tformula} \\
& \quad \mid \text{tformula} \text{ U } \text{tformula} \mid G[i..j]\text{tformula} \\
& \quad \mid \text{tformula} \text{ W } \text{tformula} \mid G[i..j]\text{tformula} \\
& \quad \mid \text{tformula} \text{ R } \text{tformula} \mid \text{tformula} \text{ M } \text{tformula}
\end{align*}
\]

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 \text{\rightarrow}, \leftrightarrow, \text{U}, \text{R}, \text{W}, and \text{M} to get closer to the PSL standard \cite{PSL}.
<table>
<thead>
<tr>
<th>assoc. operators</th>
<th>priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>right [[]][][], []=[], []&lt;[][][]</td>
<td>lowest</td>
</tr>
<tr>
<td>left ;</td>
<td></td>
</tr>
<tr>
<td>left :</td>
<td></td>
</tr>
<tr>
<td>left ##i, ##[i..j]</td>
<td></td>
</tr>
<tr>
<td>right [][], []&lt;[]</td>
<td></td>
</tr>
<tr>
<td>left xor</td>
<td></td>
</tr>
<tr>
<td>left &amp;</td>
<td></td>
</tr>
<tr>
<td>left &amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>right &amp;, W, M, R</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;, &amp;&amp;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 \&\, W as left-associative, while Goal (hence Büchi store), LTL2AUT, and LTL2Büchi (from JavaPathFinder) have \&\, W 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:

- `is_boolean()` Whether the formula uses only Boolean operators.
- `is_sugar_free_boolean()` Whether the formula uses only & , |, and ! operators. (Especially, no -> or <-> are allowed.)
- `is_in_nenoform()` Whether the formula is in negative normal form. See section 5.3.
- `is_X_free()` Whether the formula avoids the X operator.
- `is_ltl_formula()` Whether the formula uses only LTL operators. (Boolean operators are also allowed.)
- `is_psl_formula()` Whether the formula uses only PSL operators. (Boolean and LTL operators are also allowed.)
- `is_sere_formula()` Whether the formula uses only SERE operators. (Boolean operators are also allowed, provided no SERE operator is negated.)
- `is_finite()` Whether a SERE describes a finite language (no unbounded stars), or an LTL formula uses no temporal operator but X.
- `is_eventual()` Whether the formula is a pure eventuality.
- `is_universal()` Whether the formula is purely universal.
- `is_syntactic_safety()` Whether the formula is a syntactic safety property.
- `is_syntactic_guarantee()` Whether the formula is a syntactic guarantee property.
- `is_syntactic_obligation()` Whether the formula is a syntactic obligation property.
- `is_syntactic_recurrence()` Whether the formula is a syntactic recurrence property.
- `is_syntactic_persistence()` Whether the formula is a syntactic persistence property.
- `is_marked()` Whether the formula contains a special “marked” version of the <>-> or !{r} operators.\(^3\)
- `accepts_eword()` Whether the formula accepts [\*0]. (This can only be true for a SERE formula)
- `has_lbt_atomic_props()` Whether the atomic propositions of the formula are all of the form “\(pnn\)” where \(nn\) 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 Etessami and Holzmann\(^{13}\) 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 \(\phi\) is a left-append closed formula, then \(F \phi \equiv \phi\).

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 \(\phi\) is a suffix-closed formula, then \(G \phi \equiv \phi\).

\(^3\)These “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/
The hierarchy of linear temporal properties was introduced by Manna and Pnueli [15] 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 Chang et al. [7], but other presentations have been done including negation [6] and weak until [16].

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 \texttt{is\_syntactic\_safety()}, \texttt{is\_syntactic\_guarantee()}, \texttt{is\_syntactic\_obligation()}, \texttt{is\_syntactic\_recurrence()}, and \texttt{is\_syntactic\_persistence()} listed on page 15).

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 \( \varphi_B \) 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.

### 4.2. Syntactic Hierarchy Classes

Let \( \varphi \) denote any arbitrary formula and \( \varphi_E \) (resp. \( \varphi_U \)) denote any instance of a pure eventuality (resp. a purely universal) formula. We have the following grammar rules:

\[
\varphi_E ::= 0 \mid 1 \mid X \varphi_E \mid X[!] \varphi_E \mid F \varphi \mid G \varphi_E \mid \varphi_E \land \varphi_E \mid (\varphi_E \land \varphi_E) \mid ! \varphi_U \\
\mid \varphi U \varphi_E \mid 1 U \varphi \mid \varphi_E R \varphi_E \mid \varphi_E W \varphi_E \mid \varphi_E M \varphi_E \mid \varphi M 1
\]

\[
\varphi_U ::= 0 \mid 1 \mid X \varphi_U \mid X[!] \varphi_U \mid F \varphi_U \mid G \varphi \mid \varphi_U \land \varphi_U \mid (\varphi_U \land \varphi_U) \mid ! \varphi_E \\
\mid \varphi_U U \varphi_U \mid \varphi R \varphi_U \mid 0 R \varphi \mid \varphi_U W \varphi_U \mid \varphi W 0 \mid \varphi_U M \varphi_U
\]
\[\varphi_B ::= 0 | 1 | v | ! \varphi_B | \varphi_B \land \varphi_B | (\varphi_B \lor \varphi_B) | \varphi_B \leftrightarrow \varphi_B | \varphi_B \xor \varphi_B | \varphi_B \rightarrow \varphi_B | X \varphi_B \]

| \{r_F\} | ! \{r_F\} \\

\[\varphi_G ::= \varphi_B | ! \varphi_S | \varphi_C \land \varphi_C | (\varphi_G \land \varphi_G) | \varphi_G \rightarrow \varphi_G | X \varphi_G | F \varphi_G | \varphi_G \lor \varphi_G | \varphi_G \land \varphi_G \]

| ! \{r\} | \{r\} \leftrightarrow \varphi_G | \{r_F\} \rightarrow \varphi_G \\

\[\varphi_S ::= \varphi_B | ! \varphi_G | \varphi_S \land \varphi_S | (\varphi_S \land \varphi_S) | \varphi_S \rightarrow \varphi_S | X \varphi_S | G \varphi_S | \varphi_S \lor \varphi_S | \varphi_S \land \varphi_S \]

| \{r\} | \{r\} \leftrightarrow \varphi_S | \{r\} \rightarrow \varphi_S \\

\[\varphi_O ::= \varphi_G | \varphi_S | ! \varphi_O | \varphi_O \land \varphi_O | (\varphi_O \land \varphi_O) | \varphi_O \leftrightarrow \varphi_O | \varphi_O \xor \varphi_O | \varphi_O \rightarrow \varphi_O \]

| X \varphi_O | \varphi_O \lor \varphi_O | \varphi_O \land \varphi_O | \varphi_O \rightarrow \varphi_O | \{r_F\} \rightarrow \varphi_O | \{r\} \rightarrow \varphi_O \\

\[\varphi_P ::= \varphi_O | ! \varphi_R | \varphi_P \land \varphi_P | (\varphi_P \land \varphi_P) | \varphi_P \rightarrow \varphi_P | \varphi_P \lor \varphi_P | \varphi_P \land \varphi_P \]

| X \varphi_P | F \varphi_P | \varphi_P \lor \varphi_P | \varphi_P \land \varphi_P | \varphi_P \rightarrow \varphi_P | \{r_F\} \rightarrow \varphi_P | \{r\} \rightarrow \varphi_P \\

\[\varphi_R ::= \varphi_O | ! \varphi_P | \varphi_R \land \varphi_R | (\varphi_R \land \varphi_R) | \varphi_R \rightarrow \varphi_R | \varphi_R \lor \varphi_R | \varphi_R \land \varphi_R | \varphi_R \rightarrow \varphi_R | \varphi_R \rightarrow \varphi_R \]

| X \varphi_R | G \varphi_R | \varphi_R \lor \varphi_R | \varphi_R \land \varphi_R | \varphi_R \rightarrow \varphi_R | \varphi_R \land \varphi_R | \varphi_R \land \varphi_R | \varphi_R \land \varphi_R |

| \{r\} \rightarrow \varphi_R | \{r\} \rightarrow \varphi_R | \{r\} \rightarrow \varphi_R \\

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 \((G(q \land FG(p)) \land G(r \land FG(p))) \land G(q \land GR)\) is not syntactically safe, yet it is a safety formula equivalent to \(Gq \land GR\). 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 -> g = (! f) | g
"e"   f <-> g = (f & g) | (! g) & (! f))
"~e"  f xor g = (f & ! g) | (g & ! f)
"~e" without "e"  f xor g = !(f <-> g)
"F"   Fe = e  when e is a pure eventuality
"F"   F f = 1 U f
"G"   G u = u  when u is purely universal
"G" without "R"  G f = 0 R f
"GR" without "W"  G f = f W 0
"GRW"  G f = ! F ! f
"M"   f Me = F(f & e)  when e is a pure eventuality
"M"   f M g = g U (g & f)
"R"   f R u = u  when u is purely universal
"R" without "W"  f R g = g W (f & g)
"RM"  f R g = g U ((f & g) | G g)
"W"   f W u = G(f | u)
"W" without "R"  f W g = G R (g | f)
"WR"  f W g = f U (g | G f)
```

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 [10, 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.

\[
\begin{align*}
!Xf & = X!f & !(f \cup g) & = (!f) \mathcal{R} (!g) & !(f \& g) & = (!f) \mathcal{I} (!g) \\
!Ff & = G!f & !(f \mathcal{R} g) & = (!f) \mathcal{I} (!g) & !(f \mathcal{I} g) & = (!f) \mathcal{R} (!g) \\
!Gf & = F!f & !(f \mathcal{I} g) & = (!f) \mathcal{R} (!g) & !(f \& g) & = (!f) \& (!g) \\
\left(\left\{r\right\}\right) & = \left\{r\right\} & !(f \mathcal{R} g) & = (!f) \mathcal{I} (!g) & !(f \& g) & = (!f) \& (!g) \\
\end{align*}
\]

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

\[
\begin{align*}
fxorg & = ((!f) \& g) \mathcal{I} (f \& !g) & !(fxorg) & = ((!f) \& (!g)) \mathcal{I} (f \& g) & !(f \& g) & = (!f) \mathcal{I} (!g) \\
f\mathcal{R}g & = ((!f) \& (!g)) \mathcal{I} (f \& g) & !(f \mathcal{R}g) & = ((!f) \& (!g)) \mathcal{I} (f \& g) & !(f !g) & = (!f) \& (!g) \\
f\rightarrow g & = (f !g) \& g & !(f \rightarrow g) & = (f !g) \& g & !(f !g) & = (f !g) \& (!g) \\
\end{align*}
\]

Note that the above rules include the “unabbreviation” of operators “\(-\)”, “\(\rightarrow\)”, and “\(\&\)”, corresponding to the rules “eq” 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 ‘nenoform_stop_on_boolean’ is set, the above recursive rewritings are not applied to Boolean subformulas. For instance calling ‘tl_simplifier::negative_normal_form’ on \(!FQ(a \& b)\) will produce \(GF(!((!a) \& (!b)) \mathcal{I} (a \& b))\) if ‘nenoform_stop_on_boolean’ is unset, while it will produce \(GF(!((!a) \& b))\) if ‘nenoform_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: \(\bigvee\beta_i \& X\psi_i\) where \(\beta_i\)s are Boolean formulas and \(\psi_i\)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., \(F(f \mathcal{I} g)\) is better than the equivalent \(F(f) \mathcal{I} (Fg)\)), but not before X (\(XFf\) is better than \(FXf\)). Because F f incurs some indeterminism, it is best to factorize these terms to limit the sources of indeterminism.

Rewritings defined with \(\mathcal{R}\) 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 \(\mathcal{I}\) 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 \( \triangleq \) 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 `dnf(f)` and `cnf(f)` denote respectively the disjunctive and conjunctive normal forms if \( f \), handling non-Boolean sub-formulas as if they were atomic propositions.

\[
\begin{align*}
X F G f & = F G f \\
X G F f & = G F f \\
F X f & = X F f \\
G X f & = X G f \\
X 0 & = 0 \\
G F f & \triangleq G F (d n f (f)) \\
F G f & \triangleq F G (c n f (f))
\end{align*}
\]

\[
G (f_1 \mid \ldots \mid f_n \mid G F (g_1) \mid \ldots \mid G F (g_m)) = G (f_1 \mid \ldots \mid f_n) \mid G F (g_1 \mid \ldots \mid g_m)
\]

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

\[
\begin{align*}
1 U f & = F f \\
I f & = F f \\
(X f) U (X g) & = X (f U g) \\
(X f) M (X g) & = X (f M g) \\
(X f) U b & \triangleq b \mid X (b M f) \\
(X f) M b & \triangleq b \& X (b U f) \\
f U (G f) & = G f \\
f M (F f) & = F f \\
f U (g | G (f)) & = f W g \\
f M (g | F (f)) & = f M g \\
f U (g \& f) & = g M f \\
f M (g \& f) & = g U f \\
f U (g \mid f) & = g U f \\
f R (g \mid f) & = g W f
\end{align*}
\]
Here are the basic rewriting rules for \( n \)-ary operators (\& and \mid):

\[
\begin{align*}
(FGf) \& (FGg) &= FG(f \& g) \\
(Xf) \& (Xg) &= X(f \& g) \\
(Xf) \& (FGg) &= X(f \& FGg) \\
(Gf) \& (Gg) &= G(f \& g) \\
(f_1 U f_2) \& (f_3 U f_2) &= (f_1 \& f_3) U f_2 \\
(f_1 U f_2) \& (f_3 W f_2) &= (f_1 \& f_3) U f_2 \\
(f_1 W f_2) \& (f_3 W f_2) &= (f_1 \& f_3) W f_2 \\
(f_1 R f_2) \& (f_3 R f_2) &= (f_1 R f_3) R f_2 \\
(f_1 R f_2) \& (f_3 M f_3) &= (f_1 M f_2) R f_3 \\
(f_1 M f_2) \& (f_3 M f_3) &= (f_1 \& f_3) M f_3 \\
(Fg) \& (f U g) &= f U g \\
(Fg) \& (f W g) &= f U g \\
(Ff) \& (f R g) &= f M g \\
(Ff) \& (f M g) &= f M g \\
(f \& ((Xf) W g) &= g R f \\
(f \& ((Xf) U g) &= g M f \\
(f \mid g) \& (X(g \mid Ff)) &= g R f \\
f \mid (g \& X(g \mid Ff)) &= g M f
\end{align*}
\]

The above rules are applied even if more terms are present 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 \( f \& XGf = GF F \) and \( f \mid XFf = FF \):

\[
\begin{align*}
f \& X(G(f \& g \& \ldots) \& h \& \ldots) &= G(f) \& X(G(g \& \ldots) \& h \& \ldots) \\
f \mid X(F(f) \mid h \mid \ldots) &= F(f) \mid X(h \mid \ldots)
\end{align*}
\]

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

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

\[
F(f_1) \mid \ldots \mid F(f_n) \mid GF(g) \underset{\text{w}}{=} F(f_1 \mid \ldots \mid f_n \mid GF(g))
\]

Basic Simplifications for SERE Operators

The following rules, mostly taken from Cimatti et al. [8] are not complete yet. We only show those that are implemented.

The following simplification rules are used for the \( n \)-ary operators \&\&, \& and \mid. The patterns are of course commutative. \( b \) or \( b_i \) denote any Boolean formula while \( r \) or \( r_i \) denote any SERE.
\[ b \land r[i..j] = \begin{cases} b \land r & \text{if } i \leq 1 \leq j \\ 0 & \text{else} \end{cases} \]

\[ b \land \{r_1 : \ldots : r_n\} = b \land r_1 \land \ldots \land \land r_n \]

\[ b \land \{r_1 \ldots \lceil r \rceil n \} = \begin{cases} b \land r_i & \text{if } \exists i, r \not= r_i \\ b \land (r_1 \ldots \lceil r \rceil n \wedge 0) & \text{else} \end{cases} \]

\[ \{b_1 ; r_1\} \land \{b_2 ; r_2\} = \{b_1 \land b_2 ; \{r_1 \land r_2\} \}
\]

\[ \{b_1 : r_1\} \land \{b_2 : r_2\} = \{b_1 \land b_2 ; \{r_1 \land r_2\} \}
\]

\[ \{b_1 ; r_1\} \land \{b_2 ; r_2\} = \{b_1 \land b_2 ; \{r_1 \land r_2\} \}
\]

\[ \{b_1 : r_1\} \land \{b_2 : r_2\} = \{b_1 \land b_2 ; \{r_1 \land r_2\} \}
\]

\[ \{b_1 : r_1\} \land \{b_2 : r_2\} = \{b_1 \land b_2 ; \{r_1 \land r_2\} \}
\]

\[ \text{first_match}(b[i..j]) = b[i]
\]

\[ \text{first_match}(r[i..j]) = \text{first_match}(r[*])
\]

\[ \text{first_match}(r[i..j]) = \text{first_match}(r[*])
\]

\[ \text{first_match}(r_1 : r_2[i..j]) = \text{first_match}(r_1 ; r_2[*])
\]

\[ \text{first_match}(r_1 ; r_2[0..]) = \text{first_match}(r_1 ; r_2[0..])
\]

\[ \text{first_match}(r_1 : r_2[*]) = \text{first_match}(r_1 ; r_2[*])
\]

\[ \text{first_match}(b ; r) = b ; \text{first_match}(r)
\]

\[ \text{first_match}(b[i..j]) = b[i]; \text{first_match}(b[0..j-i..]) ; r)
\]

\[ \text{first_match}(b[i..j]) = b[i-1]; \text{first_match}(b[1..j-i+1..]) ; r) \text{ if } i > 1
\]

\[ \text{first_match}(r_1 ; r_2) = \text{first_match}(r_1 \wedge r_2) \text{ if } r_1 \not= r_2
\]

\[ \text{first_match}(\text{first_match}(r_1) ; r_2) = \text{first_match}(r_1 ; r_2)
\]

\[ \text{first_match}(\text{first_match}(r_1) ; r_2) = \text{first_match}(r_1 ; r_2)
\]

Starred subformulas are rewritten in Star Normal Form with:

\[ r[*] = r^\circ[*] \]

where \( r^\circ \) is recursively defined as follows:

\[ r^\circ = r \text{ if } r \not= r \]

\[ [0]^\circ = 0 \]

\[ r[i..j]^\circ = r^\circ \text{ if } i = 0 \text{ or } r \not= r \]

\[ (r_1 \mid r_2)^\circ = r_1^\circ \mid r_2^\circ \text{ if } r_1 \not= r_2 \]

Note: the original SNF definition does not include ‘\&’ and ‘\&\&’ operators, and it guarantees that \( \forall r, r \not= r^\circ \) because \( r^\circ \) 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:

\[ r[i..j] = r^\circ[0..j] \text{ if } r \not= r^\circ, \text{ and } j > 1 \]

\[ r[i..j] = r^\circ[1..j] \text{ if } r \not= r^\circ, \text{ and } j > 1 \]

\[ r[i..j] = r \text{ if } r \not= r \text{ and } j = 1 \]
where \( r^\circ \) is recursively defined as follows:

\[
\begin{align*}
    r_i^\circ & = r_i \text{ if } \varepsilon \not\models r_i \\
    \{\ast\} & = 0 \\
    r[i..j]^\circ & = r[i..j] \text{ if } i = 0 \text{ or } \varepsilon \not\models r \\
    (r_1 \cdot r_2)^\circ & = r_1 \cdot r_2 \\
    (r_1 \cdot r_2)^\circ & = r_1 \cdot r_2 \\
    (r_1 \cdot r_2)^\circ & = r_1 \cdot r_2 \\
    (r_1 \cdot r_2)^\circ & = r_1 \cdot r_2
\end{align*}
\]

The differences between \( ^\circ \) and \( \ast \) are in the handling of \( r[i..j] \) and in the handling of \( r_1 \cdot r_2 \).

### Basic Simplifications SERE-LTL Binding Operators

The following rewritings are applied to the operators \( \square \rightarrow \) and \( <>\rightarrow \). They assume that \( b \), denote a Boolean formula.

As noted at the beginning for section 5.4.1 rewritings denoted with \( \ast \) can be disabled by setting the \( \text{tl_simplifier_options::reduce_size_strictly} \) option to true.

\[
\begin{align*}
    \{\{\ast\}\} & \rightarrow f = G f \\
    \{b(\ast)\} & \rightarrow f = f \top \land b \\
    \{b(\ast)\} & \rightarrow f = f \top \land b \\
    \{r[\ast\ast..\ast]\} & \rightarrow f = \{r[\ast\ast..\ast]\} \rightarrow f \\
    \{r[i..j]\} & \rightarrow f = \{r[i..j]\} \rightarrow X(\{r[i..j]\} \rightarrow X(\{r[i..j]\})) \text{ if } i \geq 1 \text{ and } \varepsilon \not\models r \\
    \{r ; \{\ast\}\} & \rightarrow f = \{r ; \{\ast\}\} \rightarrow G f \\
    \{r ; b(\ast)\} & \rightarrow f = \{r ; b(\ast)\} \rightarrow (f \land X(f \top \land b)) \text{ if } \varepsilon \not\models r \\
    \{\{\ast\} ; r\} & \rightarrow f = G(\{r\} \rightarrow f) \\
    \{b(\ast) ; r\} & \rightarrow f = (b \cdot X(\{r\} \rightarrow f)) \text{ if } \varepsilon \not\models r \\
    \{r_1 ; r_2\} & \rightarrow f = \{r_1\} \rightarrow X(\{r_2\} \rightarrow f) \text{ if } \varepsilon \not\models r_1 \text{ and } \varepsilon \not\models r_2 \\
    \{r_1 ; r_2\} & \rightarrow f = \{r_1\} \rightarrow X(\{r_2\} \rightarrow f) \\
    \{r_1 \cdot r_2\} & \rightarrow f = \{r_1\} \rightarrow (\{r_2\} \rightarrow f) \\
    \{\{\ast\}\} & \rightarrow f = F f \\
    \{b(\ast)\} & \rightarrow f = f \top \land b \\
    \{b(\ast)\} & \rightarrow f = f \top \land b \\
    \{r[\ast\ast..\ast]\} & \rightarrow f = \{r[\ast\ast..\ast]\} \rightarrow f \\
    \{r[i..j]\} & \rightarrow f = \{r[i..j]\} \rightarrow X(\{r[i..j]\} \rightarrow X(\{r[i..j]\})) \text{ if } i \geq 1 \text{ and } \varepsilon \not\models r \\
    \{r ; \{\ast\}\} & \rightarrow f = \{r ; \{\ast\}\} \rightarrow F f \\
    \{r ; b(\ast)\} & \rightarrow f = \{r ; b(\ast)\} \rightarrow (f \land X(f \top \land b)) \text{ if } \varepsilon \not\models r \\
    \{\{\ast\} ; r\} & \rightarrow f = G(\{r\} \rightarrow f) \\
    \{b(\ast) ; r\} & \rightarrow f = b \cdot X(\{r\} \rightarrow f) \text{ if } \varepsilon \not\models r \\
    \{r_1 ; r_2\} & \rightarrow f = \{r_1\} \rightarrow X(\{r_2\} \rightarrow f) \text{ if } \varepsilon \not\models r_1 \text{ and } \varepsilon \not\models r_2 \\
    \{r_1 ; r_2\} & \rightarrow f = \{r_1\} \rightarrow X(\{r_2\} \rightarrow f) \\
    \{r_1 \cdot r_2\} & \rightarrow f = \{r_1\} \rightarrow (\{r_2\} \rightarrow f) \\
    \{r_1 \cdot r_2\} & \rightarrow f = \{r_1\} \rightarrow (\{r_2\} \rightarrow f) \\
    \{r_1 \cdot r_2\} & \rightarrow f = \{r_1\} \rightarrow (\{r_2\} \rightarrow f)
\end{align*}
\]
Here are the basic rewritings for the weak closure and its negation:

\[
\begin{align*}
\{r[*]\} & = \{r\} & \neg\{r[*]\} & = \neg\{r\} \\
\{r; 1\} & = \{r\} & \neg\{r; 1\} & = \neg\{r\} \\
\{r; 1\} & = 1 & \neg\{r; 1\} & = 0 \\
\{r_1; r_2\} & = \{r_1\} & \neg\{r_1; r_2\} & = \neg\{r_1\} \\
\{r; 1\} & = 1 & \neg\{r; 1\} & = 0 \\
\{r_1; r_2\} & = \{r_1\} & \neg\{r_1; r_2\} & = \neg\{r_1\} \\
\{b; r\} & = b \land X \{r\} & \neg\{b; r\} & = \neg (b \land X \{r\}) \\
\{b[*i..j]; r\} & = b \land X(b \ldots \land X \{b[*0..j-i]\}; r) & \neg\{b[*i..j]; r\} & = \neg (b \land X((!b) \ldots \land X \{b[*0..j-i]\}; r)) \\
\{b[*i..j]\} & = b \land X(b \land X(...b)) & \neg\{b[*i..j]\} & = \neg (b \land X((!b) \land X(...(!b)))) \\
\{r_1 \land r_2\} & = \{r_1\} \land \{r_2\} & \neg\{r_1 \land r_2\} & = \neg\{r_1\} \land \neg\{r_2\}
\end{align*}
\]

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:

\[
\begin{align*}
F e & = e \\
F(u) \mid q & = F(u) \mid q \\
F(u)(g \mid e) & = f \cup e = e \\
F(u)(g \mid e) & = f \cup e = e \\
G u & = u \\
G(c) \mid q & = G(c \mid q) \\
G(c) \mid q & = G(c \mid q) \\
X q & = q \\
X q & = q
\end{align*}
\]
Finally the following rule is applied only when no other terms are present in the OR arguments:

\[ F(f_1 | \ldots | f_n | q_1 | \ldots | q_p) \]

### 5.4.3. Simplifications Based on Implications

The following rewriting rules are performed only when we can prove that some subformula \( f \) implies another subformula \( g \). Shown such implication can be done in two ways:

**Syntactic Implication Checks** were initially proposed by Somjeni and Bloem [12]. This detection is enabled by the “tl_simplifier_options::synt_impl” option. This is a cheap way to detect implications, but it may miss some. The rules we implement are described in Appendix [B].

**Language Containment Checks** were initially proposed by Tauriainen [18]. This detection is enabled by the “tl_simplifier_options::containment_checks” option.

In the following rewritings rules, \( f \Rightarrow g \) means that \( g \) was proved to be implied by \( f \) using either of the above two methods. Additionally, implications denoted by \( f \Rightarrow g \) are only checked if the “tl_simplifier_options::containment_checks_stronger” option is set (otherwise the rewriting rule is not applied). We write \( f \Rightarrow g \) iff \( f \Rightarrow g \) and \( g \Rightarrow f \).

As in the previous section, formulas \( e \) and \( u \) represent respectively pure eventualities and purely universal formulas.

Finally \( |f|_b \) denote the length of \( f \) were all Boolean subformulas are counted as one.

```plaintext
if \( f \Rightarrow g \) then \( f \mid g = 1 \)
if \( f \Rightarrow g \) then \( f \& g = 0 \)
if \( (f \Rightarrow g) \land (|f|_b < |g|_b) \) then \( f \mid g = f \)
if \( f \Rightarrow g \) then \( f \mid g = g \)
if \( (f \Rightarrow g) \land (|g|_b < |f|_b) \) then \( f \& g = g \)
```
if \( f \Rightarrow g \) then \( f \& g = f \)
if \( f \Rightarrow g \) then \( f \leftarrow g = g \Rightarrow f \)
if \( f \Rightarrow g \) then \( f \Rightarrow g = 1 \)
if \( ! f \Rightarrow g \) then \( f \text{ xor } g = g \Rightarrow ! f \)
if \( f \Rightarrow \top \) then \( f \text{ xor } g = (\top) \Rightarrow f \)
if \( (f \Rightarrow g) \land (|f| < |g|) \) then \( f \cup g = f \)
if \( f \Rightarrow g \) then \( f \cup g = g \)
if \( (f \cup g) \Rightarrow \top \) then \( f \cup g = g \)
if \( ! f \Rightarrow g \) then \( f \cup g = F \)
if \( g \Rightarrow \top \) then \( e \cup g = F \)
if \( f \Rightarrow g \) then \( f \cup (g \cup h) = g \cup h \)
if \( f \Rightarrow g \) then \( f \cup (g \uparrow h) = g \uparrow h \)
if \( g \Rightarrow f \) then \( f \cup (g \cup h) = f \cup h \)
if \( f \Rightarrow h \) then \( f \cup (g \uparrow (h \cup k)) = g \uparrow (h \cup k) \)
if \( f \Rightarrow h \) then \( f \cup (g \uparrow (h \uparrow k)) = g \uparrow (h \uparrow k) \)
if \( f \Rightarrow h \) then \( f \cup (g \uparrow (h \uparrow k)) = g \uparrow (h \uparrow k) \)
if \( f \Rightarrow h \) then \( f \cup (g \uparrow (h \uparrow k)) = g \uparrow (h \uparrow k) \)
if \( f \Rightarrow h \) then \( (f \cup g) \uparrow h = g \uparrow h \)
if \( f \Rightarrow h \) then \( (f \uparrow g) \uparrow h = g \uparrow h \)
if \( g \Rightarrow h \) then \( (f \uparrow g) \uparrow h = (f \uparrow g) \uparrow h \)
if \( g \Rightarrow h \) then \( (f \uparrow g) \uparrow h = f \uparrow h \)
if \( ! f \Rightarrow g \) then \( f \uparrow g = 1 \)
if \( (f \Rightarrow g) \land (|f| < |g|) \) then \( f \uparrow g = f \)
if \( f \Rightarrow g \) then \( f \uparrow g = g \)
if \( (f \uparrow g) \Rightarrow \top \) then \( f \uparrow g = g \)
if \( f \Rightarrow g \) then \( f \uparrow g = f \)
if \( g \Rightarrow f \) then \( f \uparrow g = G \)
if \( ! u \Rightarrow g \) then \( u \uparrow g = G \)
if \( g \Rightarrow f \) then \( f \uparrow (g \uparrow h) = g \uparrow h \)
if \( g \Rightarrow f \) then \( f \uparrow (g \uparrow h) = g \uparrow h \)
if \( f \Rightarrow g \) then \( f \uparrow (g \uparrow h) = f \uparrow h \)
if \( h \Rightarrow f \) then \( (f \uparrow g) \uparrow h = g \uparrow h \)
if $h \Rightarrow f$ then $(f \& g) R h = g R h$
if $g \Rightarrow h$ then $(f R g) R h = (f \& g) R h$
if $g \Rightarrow h$ then $(f \& g) R h = (f \& g) R h$
if $h \Rightarrow g$ then $(f \& g) R h = f R h$
if $(f \Rightarrow g) \land (|f|_b < |g|_b)$ then $f \& g = f$
if $g \Rightarrow f$ then $f \& g = g$
if $g \Rightarrow ! f$ then $f \& g = 0$
if $g \Rightarrow f$ then $f \& (g \& h) = g \& h$
if $f \Rightarrow g$ then $f \& (g \& h) = f \& h$
if $f \Rightarrow g$ then $f \& (g \& h) = f \& h$
if $h \Rightarrow f$ then $(f \& g) \& h = g \& h$
if $h \Rightarrow f$ then $(f \& g) \& h = f \& h$
if $h \Rightarrow g$ then $(f \& g) \& h = f \& h$
if $h \Rightarrow g$ then $(f \& g) \& h = f \& h$

Many of the above rules were collected from the literature [17, 18, 19] and sometimes generalized to support operators such as $\&$ and $\|$. The following rules mix implication-based checks with formulas that are pure eventualities ($e$) or that are purely universal ($u$).

if $(! f) \Rightarrow g$ then $f \& (g \& e) = F(g \& e)$
if $f \Rightarrow ! g$ then $f \& (g \& e) = G(g \& e)$
if $(! f) \Rightarrow g$ then $G(f) \& (g \& e) = f \& (g \& e)$
if $f \Rightarrow ! g$ then $F(f) \& (g \& e) = f \& (g \& e)$
A. Defining LTL with only one of $U$, $W$, $R$, or $M$

The operators ‘F’, ‘G’, ‘U’, ‘R’, ‘M’, and ‘W’ can all be defined using only Boolean operators, ‘X’, and one of ‘U’, ‘W’, ‘R’, or ‘M’. This property is usually used to simplify proofs. These equivalences can also help to understand the semantics of section 2.4.1 if you are only familiar with some of the operators.

Equivalences using $U$:

$$
\begin{align*}
F f & = 1 U f \\
G f & = ! F ! f = ! (1 U f) \\
f W g & = (f U g) \land G f = (f U g) \land ! (1 U f) \\
& = f U (g \land G f) = f U (g \land ! (1 U f)) \\
f M g & = g U (f \land g) \\
f R g & = g W (f \land g) = (g U (f \land g)) \land ! (1 U g) \\
& = g U ((f \land g) \land ! (1 U g))
\end{align*}
$$

Equivalences using $W$:

$$
\begin{align*}
F f & = ! G ! f = ! ((f) W 0) \\
G f & = 0 R f = f W 0 \\
f U g & = (f W g) \land (F g) = (f W g) \land ! ((g) W 0) \\
f M g & = (g W (f \land g)) \land F (f \land g) = (g W (f \land g)) \land ! ((f \land g) W 0) \\
f R g & = g W (f \land g)
\end{align*}
$$

Equivalences using $R$:

$$
\begin{align*}
F f & = ! G ! f = ! (0 R ! f) \\
G f & = 0 R f \\
f U g & = (((X g) R f) \land F g) \land g = (((X g) R f) \land ! (0 R ! g)) \land g \\
f W g & = g R (f \land g) \\
& = ((X g) R f) \land g \\
f M g & = (f R g) \land F f = (f R g) \land ! (0 R ! f) \\
& = f R (g \land F g) = f R (g \land ! (0 R ! f))
\end{align*}
$$

Equivalences using $M$:

$$
\begin{align*}
F f & = f M 1 \\
G f & = ! F ! f = ! ((f) M 1) \\
f U g & = g M (f \land g) \\
& = ((X g) M f) \land g \\
f W g & = (f U g) \land G f = ((X g) M f) \land g \land ! ((f) M 1) \\
f R g & = (f M g) \land G g = (f M g) \land ! ((g) M 1)
\end{align*}
$$
These equivalences make it possible to build artificially complex formulas. For instance by applying
the above rules to successively rewrite $U \rightarrow M \rightarrow R \rightarrow U$ we get

$$f \cup g = ((X g) \land f) \lor g$$

$$= (((((X g) \land f) \land (0 \lor ! X g)) \lor g$$

$$= (((((f \cup (X g \land f)) \lor (1 \lor ! f)) \land ! (((!! X g) \cup (0 \land ! X g)) \lor (1 \land !! X g))) \lor g$$

$$= (((((f \cup (X g \land f)) \lor (1 \lor ! f)) \land (1 \land X g)) \lor g$$

Spot is able to simplify most of the above equivalences, but it starts to have trouble when the $X$
operator is involved. For instance $(f \land \land X g = f \cup g$ is one of the rewriting rules from 5.4.1. But the
formula $(f \land \land X g) \land X g$, which looks like it should be reduced similarly to $f \cup X g$, will be rewritten
instead to $(f \land \land X g) \land X g$, because $X g = F X g$ is another rule that gets applied first during the
bottom-up rewriting.
B. Syntactic Implications

Syntactic implications are used for the rewriting of Section 5.4.3. The rules presented below extend those first presented by Somenzi and Bloem [17].

A few words about implication first. For two PSL formulas \( f \) and \( g \), we say that \( f \implies g \) if \( \forall \sigma, (\sigma \models f) \implies (\sigma \models g) \). For two SERE \( f \) and \( g \), we say that \( f \implies g \) if \( \forall \pi, (\pi \models f) \implies (\pi \models g) \).

The recursive rules for syntactic implication are rules described in Table B.1, in which \( \implies \) denotes the syntactic implication, \( f, f_1, f_2, g, g_1 \) and \( g_2 \) denote any PSL formula in negative normal form, and \( f_U \) and \( g_E \) denote a purely universal formula and a pure eventuality.

The form on the left of the table syntactically implies the form on the top of the table if the condition in the corresponding cell holds.

Note that a given formula may match several forms, and require multiple recursive tests. To limit the number of recursive calls, some rules have been removed when they are already implied by other rules.

For instance it one would legitimately expect the rule “\( F f \implies F g \) if \( f \implies g \)” to appear in the table. However in that case \( F g \) is pure eventuality, so we can reach the same conclusion by chaining two rules: “\( F f \implies F g \) if \( f \implies g \)” and then “\( f \implies F g \) if \( f \implies g \)”.

The rules from Table B.1 should be completed by the following cases, where \( f_b \) and \( g_b \) denote Boolean formulas:

<table>
<thead>
<tr>
<th>( f )</th>
<th>( g )</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>always</td>
<td>( f \implies 1 )</td>
</tr>
<tr>
<td>0</td>
<td>always</td>
<td>( 0 \implies g )</td>
</tr>
<tr>
<td>( f_b )</td>
<td>( g_b )</td>
<td>( BDD(f_b) \land BDD(g_b) = BDD(f_b) )</td>
</tr>
</tbody>
</table>

Note that a given formula may match several forms, and require multiple recursive tests. To limit the number of recursive calls, some rules have been removed when they are already implied by other rules.

For instance it one would legitimately expect the rule “\( F f \implies F g \) if \( f \implies g \)” to appear in the table. However in that case \( F g \) is pure eventuality, so we can reach the same conclusion by chaining two rules: “\( F f \implies F g \) if \( f \implies g \)” and then “\( f \implies F g \) if \( f \implies g \)”.

The rules from Table B.1 should be completed by the following cases, where \( f_b \) and \( g_b \) denote Boolean formulas:

<table>
<thead>
<tr>
<th>( f )</th>
<th>( g )</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>always</td>
<td>( f \implies 1 )</td>
</tr>
<tr>
<td>0</td>
<td>always</td>
<td>( 0 \implies g )</td>
</tr>
<tr>
<td>( f_b )</td>
<td>( g_b )</td>
<td>( BDD(f_b) \land BDD(g_b) = BDD(f_b) )</td>
</tr>
<tr>
<td>$\Rightarrow$</td>
<td>$g$</td>
<td>$e_1$</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>$f$</td>
<td>$f = g$</td>
<td>$f \Rightarrow g_2$</td>
</tr>
<tr>
<td>$f \cup f_2$</td>
<td>$f \Rightarrow g_1$</td>
<td>$f \Rightarrow g_2$</td>
</tr>
<tr>
<td>$f \setminus f_2$</td>
<td>$f \Rightarrow g_1$</td>
<td>$f \Rightarrow g_2$</td>
</tr>
<tr>
<td>$f \setminus f_2$</td>
<td>$f \Rightarrow g_1$</td>
<td>$f \Rightarrow g_2$</td>
</tr>
<tr>
<td>$f \setminus f_2$</td>
<td>$f \Rightarrow g_1$</td>
<td>$f \Rightarrow g_2$</td>
</tr>
<tr>
<td>$F$</td>
<td>$f \Rightarrow g_1$</td>
<td>$f \Rightarrow g_2$</td>
</tr>
</tbody>
</table>

Table B.1: Recursive rules for syntactic implication.
Bibliography


