Formal Validation of Core SALT Translation to LTL in Isabelle/HOL

David Trachtenherz

TUM–I1105
März 11
Formal Validation of Core SALT Translation to LTL in Isabelle/HOL

Formal semantics definition, translation to LTL, and formal translation validation for core SALT in the Isabelle/HOL theorem prover

David Trachtenherz
February 2011

Abstract

Temporal notations are widely accepted for formal specification of functional properties amenable to automated formal verification. The SALT temporal specification language was developed as an extension of the popular LTL notation to simplify creating temporal specifications: it provides, among others, concise operators and restricted regular expressions. SALT formulas can be translated to LTL by a freely available compiler and thereby directly used for model checking. Clearly defined semantics of the specification notation is indispensable for creating precise unambiguous descriptions of the desired behavioural properties and for making subsequent formal verification meaningful. SALT semantics has been given through translation to LTL so far, which is in parts rather sophisticated and not easily comprehensible. This report presents a clear and explicit semantics formalisation for a substantial language subset of SALT through translation to an expressive interval temporal logic with explicit time variables. The formal definition and validation is performed in the Isabelle/HOL theorem prover. In the course of the formal validation we particularly prove that the semantics resulting from translation to LTL is equivalent to the explicit semantics definition.
# Contents

1 Introduction and motivation \hfill 3  

2 ILET \hfill 4  
   2.1 Shallow embedding \hfill 4  
   2.2 Regular expressions (deep embedding) \hfill 7  
      2.2.1 Syntax \hfill 7  
      2.2.2 Semantics \hfill 7  
      2.2.3 Sequence operators and expressions matching empty words \hfill 8  

3 LTL \hfill 11  
   3.1 Syntax \hfill 11  
   3.2 Semantics \hfill 11  

4 Core SALT \hfill 12  
   4.1 Syntax \hfill 12  
   4.2 Translation to LTL \hfill 13  
      4.2.1 Translation of regular expressions to LTL \hfill 13  
      4.2.2 Translation of $until$ and $from$ operators to LTL \hfill 15  
      4.2.3 Translation of core SALT formulas to LTL \hfill 16  
   4.3 Semantics \hfill 18  
      4.3.1 Translation of regular expressions to ILET \hfill 18  
      4.3.2 Translation of $until$ and $from$ operators to ILET \hfill 19  
      4.3.3 Translation of core SALT formulas to ILET \hfill 19  
   4.4 Sequence operators and expressions matching empty words \hfill 20  
   4.5 Formal validation of core SALT translation to LTL \hfill 22  
      4.5.1 Selected auxiliary translation validation lemmas \hfill 22  
      4.5.2 Main translation validation theorem \hfill 23  

5 Additional results for core SALT \hfill 23  
   5.1 LTL operators $until$, weak $until$, $release$ in core SALT \hfill 23  
   5.2 Expressive equivalence of core SALT and LTL \hfill 24
1 Introduction and motivation

SALT [BLS06] is a temporal specification language based on the linear temporal logic LTL [Pnu77] and incorporating aspects of further specification formalisms and frameworks [BBDE+01, DAC99], e.g., restricted regular expressions, specification patterns and further operators. SALT is meant particularly as a bridge to formal but not always user-friendly LTL specification – allowing macro definitions and using textual operator names it much more resembles a programming language than LTL does, and furthermore the operators provided by SALT make it possible to conveniently specify requirements, which can hardly be formulated in LTL without errors due to the complexity of corresponding LTL formulas – compare, for instance, a simple SALT regular expression and the corresponding LTL formula:

\[ p \,* \, [\geq 3] \, ; \, r \, \iff \, p \land \circ (q \land \circ (q \land \circ (q \land \circ r))) \]

This example also shows an important and critical point about SALT translation to LTL – the concise and quite intuitive SALT operators have to be expressed using the well-defined but minimalist set of LTL operators so that the translation is in parts complex and therefore not easy to comprehend and especially being checked for correctness.

The meaning of SALT operators is informally explained in [Str06]. The translation of SALT to LTL, described in [Str06] and implemented by the SALT compiler [SAL], implicitly gives a SALT semantics. However, there existed no explicit formal semantics definition so far. The advantages of creating such an explicit semantics definition are manifold. [Gor03, Section 2] discusses several aspects that motivated the semantics validation for PSL [Acc04]. This discussion largely applies also to our work, particularly the issues of obtaining a machine-processible semantics as well as the prospect of combining model checking and theorem proving for formal verification of temporal properties of programs.

The main motivation concerns the actual purpose of SALT as language for formal specification of program properties. Similarly to LTL, SALT and many other formal notations are intended to be used for clear and unambiguous specification of functional properties, and in many instances for a subsequent verification. It is thus of crucial importance that their own semantics is clearly and precisely defined. Formalising SALT in a mechanized theorem prover and proving the correctness of its translation to LTL provides both a clear, machine-processible semantics definition of SALT and a formal evidence for the fact that formal verification (e.g., by model checking) of an LTL specification generated from a SALT specification is equivalent to formally verifying the original SALT specification. This would give the firm confidence that we can, instead of manually creating LTL specifications, safely use SALT for creating formal functional specifications and then automatically translate them by means of the SALT compiler into LTL for further applications, especially model checking.

Our first goal is an explicit definition of semantics for a selected SALT subset, comprising most of the core SALT operators (cf. Section 4), performed by translation to the expressive temporal logic ILET [Tra09, Chapter 4.2] [Tra11]. We have chosen ILET for several reasons. Firstly, it makes use of simple syntax and semantics with few basic constructs and allows explicit access to time variables, thus simplifying definitions of both further temporal operators and complete temporal logic notations. Secondly, there already exists a developed Isabelle/HOL theory for its temporal operators including verified results for time intervals and temporal operators, which are directly transferable to temporal logic notations defined through translation to ILET. Finally, it includes operators and verification results for working with bounded time intervals, which is significant with regard to future work comprising translation validation for SALT operators that simulate bounded time intervals (e.g., the upto operator).

The explicit semantics definition through translation to ILET prepares the ground for the second goal of formally validating the translation of the selected SALT subset to LTL by verifying that the semantics resulting from translation to LTL is equivalent to the explicit semantics definition.

We perform the semantics definition and the formal translation validation in the Isabelle/HOL interactive theorem prover. Familiarity with higher-order logic and Isabelle/HOL notation or similar ones is not required to understand the proof documentation in the presented work, though it would be helpful when reading it. A detailed tutorial on Isabelle/HOL can be found in [NPW02].
2 ILET

ILET (Interval Logic with Explicit Time, [Tra11], BPDL in [Tra09, Chapter 4.2]) is a propositional interval temporal logic providing explicit access to time variables and intervals and using natural numbers as time domain.

The propositional part of ILET provides atomic propositions on system computation states and the common Boolean operators. Due to explicit time variables, propositions can be evaluated on states for any point of time given by an arithmetic expression on time variables.

The temporal part of ILET has a simple syntax and semantics with three basic constructs:

- Temporal operators □ and ◇ corresponding to universal and existential quantification on time domain.
- Interval step operator inext calculating the next element of an interval $I \subseteq \mathbb{N}$ with respect to a given element $n \in I$.
- Interval cut operators $\downarrow<$ and $\downarrow\leq$ restricting an interval $I \subseteq \mathbb{N}$ to its elements less/less or equal a given cutting point $n \in \mathbb{N}$.

These constructs are sufficient to define further operators, common to various linear temporal logics, e.g., next or until.

2.1 Shallow embedding

Selected definitions and results for ILET.

Interval cut operators

Cutting intervals/sets at given point. The resulting interval contains all elements of original intervals less/less or equal the cutting point.

```plaintext
consts
  cut-le :: 'a::linorder set ⇒ 'a ⇒ 'a set     (infixl $\downarrow\leq$ 100 )
  cut-less :: 'a::linorder set ⇒ 'a ⇒ 'a set     (infixl $\downarrow<$ 100 )
defs
  cut-le-def: $I \downarrow\leq t = \{ x \in I. x \leq t \}$
  cut-less-def: $I \downarrow< t = \{ x \in I. x < t \}$
```

Relations between cut operators:

```plaintext
lemma cut-less-le-conv: $I \downarrow< t = (I \downarrow\leq t) - \{t\}$
lemma cut-less-le-conv-if: $I \downarrow< t = (if t \in I then (I \downarrow\leq t) - \{t\} else (I \downarrow\leq t))$
lemma nat-cut-le-less-conv: $I \downarrow\leq t = I \downarrow< \text{Suc } t$
lemma nat-cut-less-le-conv: $\theta < t \implies I \downarrow< t = I \downarrow\leq (t \text{ - Suc } \theta)$
```

Operator inext for stepping forwards through intervals

Minimal element of a well-ordered set.

```plaintext
constdefs
  iMin :: 'a::wellorder set ⇒ 'a
  iMin I ≡ LEAST x. x ∈ I
```

Function returning the next element of a natural interval/set $I$ with respect to a given number $n$. If $I$ contains no greater elements ($n$ is maximal element) or $n$ is not in $I$, then $n$ is returned.

```plaintext
constdefs
  inext :: nat ⇒ nat set ⇒ nat
  inext n I ≡ ( if (n \in I \land (I \downarrow> n \neq \{}))
```

---

1Here ILET constructs required below are introduced. A complete ILET definition (including further constructs, e.g., operator iprev, which is dual to inext and calculates the previous element of $I \subseteq \mathbb{N}$ w.r.t. some $n \in I$) is given in [Tra09] Chapter 4.
then $\text{imin}(I \upharpoonright> n)$
else $n$)

Operator $\text{inext}$ on continuous natural intervals.

**Lemma** $\text{inext-atLeast}$: $n \leq t \Rightarrow \text{inext} t \{n..\} = \text{Suc} \ t$

**Lemma** $\text{inext-atMost}$: $t < n \Rightarrow \text{inext} t \{..n\} = \text{Suc} \ t$

**Lemma** $\text{inext-lessThan}$: $\text{Suc} \ t < n \Rightarrow \text{inext} t \{..<n\} = \text{Suc} \ t$

**Lemma** $\text{inext-atLeastAtMost}$: $[m \leq t; t < n] \Rightarrow \text{inext} t \{m..n\} = \text{Suc} \ t$

### Temporal operators

ILET uses natural numbers as time domain.

**Types**
- $\text{Time} = \text{nat}$
- $\text{itt} = \text{Time set}$

Basic operators $\text{always}$ and $\text{eventually}$ corresponding to universal/existential quantification for time variables over time intervals.

**Constants**
- $\text{iAll} :: \text{itt} \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$ — Always
- $\text{iEx} :: \text{itt} \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$ — Eventually

**Definitions**
- $\text{iAll-def} : \text{iAll} \ i \ p \equiv \forall t \in I. \ p \ t$
- $\text{iEx-def} : \text{iEx} \ i \ p \equiv \exists t \in I. \ p \ t$

**Syntax (xsymbols)**
- $\lnot \text{iAll} :: \text{Time} \Rightarrow i\text{t} \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$ $(\{(\exists \ Diamond ~ -/- ~ ) [0, 0, 10] 10\})$
- $\lnot \text{iEx} :: \text{Time} \Rightarrow i\text{t} \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$ $(\{(\exists \ Diamond ~ -/- ~ ) [0, 0, 10] 10\})$

**Translations**
- $\Box t \ . \ p = \text{iAll} \ i \ (\lambda t . \ p)$
- $\Diamond t \ . \ p = \text{iEx} \ i \ (\lambda t . \ p)$

Weak and strong $\text{next}$ operator. The bound formula is evaluated at the next time point in $I$ relatively to $t_0$. If $\text{inext} t_0 I = t_0$ (i.e., $t_0$ is maximal element or $t_0 \notin I$) then weak $\text{next}$ evaluates to $\text{true}$ and strong $\text{next}$ to $\text{false}$.

**Constants**
- $\text{iNextWeak} :: \text{Time} \Rightarrow i\text{t} \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$
- $\text{iNextStrong} :: \text{Time} \Rightarrow i\text{t} \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$

**Definitions**
- $\text{iNextWeak-def} : \text{iNextWeak} \ t_0 \ i \ p \equiv (\Box t \ {\text{inext} \ t_0 \ i} \ |> t_0. \ p \ t)$
- $\text{iNextStrong-def} : \text{iNextStrong} \ t_0 \ i \ p \equiv (\Diamond t \ {\text{inext} \ t_0 \ i} \ |> t_0. \ p \ t)$

**Syntax (xsymbols)**
- $\lnot \text{iNextWeak} :: \text{Time} \Rightarrow \text{Time} \Rightarrow i\text{t} \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$ $(\{(\exists \ DiamondW ~ -/- ~ ) [0, 0, 10] 10\})$
- $\lnot \text{iNextStrong} :: \text{Time} \Rightarrow \text{Time} \Rightarrow i\text{t} \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$ $(\{(\exists \ DiamondS ~ -/- ~ ) [0, 0, 10] 10\})$

**Translations**
- $\Diamond t \ t_0 \ i . \ p = \text{iNextWeak} \ t_0 \ i \ (\lambda t . \ p)$
- $\Box t \ t_0 \ i . \ p = \text{iNextStrong} \ t_0 \ i \ (\lambda t . \ p)$

Operator $\text{until}$: the second formula $Q$ must hold at some time $t \in I$ and the first formula $P$ must hold until this time point.

**Constants**
- $\text{iuUntil} :: \text{itt} \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$

**Definitions**
- $\text{iuUntil-def} : \text{iuUntil} \ i \ p \ q \equiv \Diamond t . \ q \ t \land (\Box t' \ (I \downarrow< t). \ p \ t')$

**Syntax (xsymbols)**
- $\text{iUntil} :: \text{Time} \Rightarrow \text{Time} \Rightarrow i\text{t} \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$
  $(\{(\exists \ Diamond ~ -/- ~ ) [10, 0, 0, 10] 10\})$

**Translations**
- $P . \ t \ \downarrow \ t' . \ q . i \ = \text{iuUntil} \ i \ (\lambda t . \ p) \ (\lambda t' . \ q)$
Operator \textit{weak until} (also \textit{waiting for, unless}): either the previously defined \textit{until} operator must hold, or the first formula \(P\) must always hold in \(I\).

\begin{verbatim}
consts
  iWeakUntil :: iT ⇒ (Time ⇒ bool) ⇒ (Time ⇒ bool) ⇒ bool
defs
  iWeakUntil-def : iWeakUntil I P Q ≡ 
  \((\square t I. P t) ∨ (\diamond t I. Q t ∧ (\square t′ (I \lessdot t). P t'))\)
syntax (xsymbols)
  -iWeakUntil :: Time ⇒ Time ⇒ iT ⇒ (Time ⇒ bool) ⇒ (Time ⇒ bool) ⇒ bool
                            \((-/- (3W - -)./-) [\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset] 10)\)
translations
  P. t \∈ W t' I. Q = iWeakUntil I (\(\lambda t. P\)) (\(\lambda t'. Q\))
\end{verbatim}

Selected results for temporal operators

The interval conversions given below hold for arbitrary intervals/sets of natural numbers \(I \subseteq \mathbb{N}\).

Conversion between basic operators \textit{always} and \textit{eventually}.

\textbf{lemma} \(i\text{all}-i\text{Ex}-\text{conv}\): \((\square t I. P t) = (\neg (\diamond t I. \neg P t))\)

\textbf{lemma} \(i\text{Ex}-i\text{All}-\text{conv}\): \((\diamond t I. P t) = (\neg (\square t I. \neg P t))\)

Expressing \textit{eventually} operator through \textit{until} operator analogously to the LTL rule \(\diamond \varphi = true \U \varphi\).

\textbf{lemma} \(i\text{Until}-i\text{Ex}-\text{conv}\): \((true. t′ \mathcal{U} t I. P t) = (\diamond t I. P t)\)

Conversions between \textit{until} and \textit{weak until}.

\textbf{lemma} \(i\text{WeakUntil}-i\text{Until}-\text{conv}\):
\((P. t′ \mathcal{W} t \in I. Q t) = ((P. t′ \mathcal{U} t I. Q t) \mathcal{V} (\square t I. P t))\)

\textbf{lemma} \(i\text{Until}-i\text{WeakUntil}-\text{conv}\):
\((P. t′ \mathcal{U} t I. Q t) = ((P. t′ \mathcal{W} t I. Q t) \mathcal{V} (\diamond t I. Q t))\)

\textbf{lemma} \(i\text{WeakUntil}-i\text{Until}-\text{conv}\):
\((P. t1 t \mathcal{W} t2 t I. (P t2 \mathcal{V} Q t2)) = (\neg (Q t1. t1 \mathcal{U} t2 I. \neg P t2))\)

Conversion between \textit{release} and \textit{weak until}.

\textbf{lemma} \(i\text{Release}-i\text{WeakUntil}-\text{conv}\):
\((P. t′. t′ \mathcal{R} t I. Q t) = (Q t′. t′ \mathcal{W} t I. (Q t \mathcal{A} P t))\)

Weak and strong \textit{next} operators are dual.

\textbf{lemma} \(\neg i\text{NextWeak}\): \((\neg (\O W t t0 I. P t)) = (\emptyset s t0 I. \neg P t)\)

\textbf{lemma} \(\neg i\text{NextStrong}\): \((\neg (\O S t t0 I. P t)) = (\emptyset W t t0 I. \neg P t)\)

Weak and strong \textit{next} operators are equivalent for infinite intervals (provided the evaluation time point is in the interval, which is always true for the interval \(\{0..\}\) used in LTL and core SALT semantics definition, cf. functions \textit{ltl-valid} in Sec. 3.2 and \textit{core-salt-valid} in Sec. 3.3).

\textbf{lemma} \(\text{infin-imp-iNextStrong-eq-iNextWeak}\):
\[ \text{infinite } I. t0 ∈ I \] \(⇒\) \((\emptyset S t0 I. P t) = (\emptyset W t t0 I. P t)\)

On the interval \(\{0..\}\) weak and strong \textit{next} operators are equivalent to adding 1 to the time point of evaluation.

\textbf{lemma}
iNextWeak-atLeast-\( \emptyset \): \( (\bigcap_W t \cdot \{ \emptyset \} . P t) = P (\text{Suc} \ t \emptyset) \) and
iNextStrong-atLeast-\( \emptyset \): \( (\bigcap_S t \cdot \{ \emptyset \} . P t) = P (\text{Suc} \ t \emptyset) \)

2.2 Regular expressions (deep embedding)

Though ILET is formalised in shallow embedding manner, the regular expressions are first deeply embedded using standalone data types, which allow imposing syntactical restrictions (needed for the repetition operator \( \ast \)), and dedicated evaluation functions, which process a regular expression recursively over its structure. The ILET regular expressions can then be translated to the conventional shallow embedding formulas. We use the deep embedding of regular expressions as an intermediate step that especially facilitates working with ILET regular expressions because the shallow embedded ILET formulas giving the semantics of regular expressions do not resemble familiar regular expression notations.

2.2.1 Syntax

Data types for ILET regular expressions.

\[
\begin{align*}
\text{datatype} & \quad \text{ilet-reg-exp-bool} = \\
& \quad \text{BREAtom} \quad \text{Time} \Rightarrow \text{bool} \quad (\text{BREAtom} - [115,115]) \\
& \quad \text{BRENot} \quad \text{ilet-reg-exp-bool} \quad (\neg\text{bre} - [40,40]) \\
& \quad \text{BREAnd} \quad \text{ilet-reg-exp-bool ilet-reg-exp-bool} \quad ((\land\land\text{bre}) - [36,35,35]) \\
& \quad \text{BREOr} \quad \text{ilet-reg-exp-bool ilet-reg-exp-bool} \quad ((\lor\lor\text{bre}) - [31,30,30]) \\
& \quad \text{BREImp} \quad \text{ilet-reg-exp-bool ilet-reg-exp-bool} \quad ((\neg\rightarrow\text{bre}) - [26,25,25]) \\
& \quad \text{BREequiv} \quad \text{ilet-reg-exp-bool ilet-reg-exp-bool} \quad ((\neg\leftrightarrow\text{bre}) - [26,25,25])
\end{align*}
\]

\[
\begin{align*}
\text{datatype} & \quad \text{ilet-reg-exp} = \\
& \quad \text{BREBool} \quad \text{ilet-reg-exp-bool} \quad (\text{BREBool} - [115,115]) \\
& \quad \text{BREEmpty} \quad (\varepsilon) \\
& \quad \text{BRERegExpOr} \quad \text{ilet-reg-exp ilet-reg-exp} \quad ((\lor\lor\text{bre}) - [30,31,30]) \\
& \quad \text{BRESeqSubsequent} \quad \text{ilet-reg-exp ilet-reg-exp} \quad ((\rightarrow\text{bre}) - [111,110,110]) \\
& \quad \text{BRESeqOverlap} \quad \text{ilet-reg-exp ilet-reg-exp} \quad ((\rightarrow\leftrightarrow\text{bre}) - [111,110,110]) \\
& \quad \text{BRERegOp-StarGe} \quad \text{ilet-reg-exp-bool nat} \quad ((\rightarrow\ast\text{bre}) - [110,110,111])
\end{align*}
\]

2.2.2 Semantics

Validity function for Boolean terms in ILET regular expressions.

\[
\begin{align*}
\text{consts} & \quad \text{ilet-reg-exp-bool-valid} :: \text{Time} \Rightarrow \text{ilet-reg-exp-bool} \Rightarrow \text{bool} \\
& \quad (\text{ilet-reg-exp-bool-}\text{-}\text{valid} - [80,80,80])
\end{align*}
\]

\[
\begin{align*}
\text{primrec} & \quad \models\text{bre}\text{bool} \quad \text{BREAtom} \quad a \quad t \\
& \quad \models\text{bre}\text{bool} \quad (\neg\text{bre} \quad f) \quad (\models\text{bre}\text{bool} \quad t \quad f) \\
& \quad \models\text{bre}\text{bool} \quad (f \quad \land\land\text{bre} \quad f) \quad (\models\text{bre}\text{bool} \quad t \quad f \quad 1 \quad \land\land\text{bre}\text{bool} \quad t \quad f) \\
& \quad \models\text{bre}\text{bool} \quad (f \quad \lor\lor\text{bre} \quad f) \quad (\models\text{bre}\text{bool} \quad t \quad f \quad 1 \quad \lor\lor\text{bre}\text{bool} \quad t \quad f) \\
& \quad \models\text{bre}\text{bool} \quad (f \quad \rightarrow\rightarrow\text{bre} \quad f) \quad (\models\text{bre}\text{bool} \quad t \quad f \quad 1 \quad \rightarrow\rightarrow\text{bre}\text{bool} \quad t \quad f) \\
& \quad \models\text{bre}\text{bool} \quad (f \quad \leftarrow\leftarrow\text{bre} \quad f) \quad (\models\text{bre}\text{bool} \quad t \quad f \quad 1 \quad \leftarrow\leftarrow\text{bre}\text{bool} \quad t \quad f)
\end{align*}
\]

We now define the evaluation function for ILET regular expressions. Though evaluation of ILET regular expressions is principally possible for all intervals (e.g. for modulo-intervals of the form \( \{ n \mid n \geq n_0 \land n \text{ mod } m = r \} \), we consider for reasons of simplicity only continuous intervals of the form \( [n_1 \ldots n_2] = \{n_1, n_1+1, \ldots, n_2-1\} \). Thus, passing lower and upper bounds of a time interval suffices for matching a regular expression to this interval. \( t_2 - t_1 \) indicates the length of the regular expression: the expression begins at time point \( t_1 \) and ends exactly before time point \( t_2 \).

\[
\begin{align*}
\text{consts} & \quad \text{ilet-reg-exp-match} :: \text{Time} \Rightarrow \text{Time} \Rightarrow \text{ilet-reg-exp} \Rightarrow \text{bool} \\
& \quad (\text{ilet-reg-exp-}\text{-}\text{match} - [80,80,80,80])
\end{align*}
\]

\(^2\text{The abbreviation BRE stands for Boolean Regular Expression, the identifier BREAtom is required merely for technical syntactical reasons in Isabelle/HOL.} \)
transfer, once as regular expression

operators and hence represent convenience constructs. Here, for example, a simple protocol pattern for data

For example, \texttt{/a*;b*/} matches "aaabbb.." with \(t_1 = 0\), \(t_2 = 6\) as follows:

\begin{align*}
\text{ilet_reg_exp_match } 0 6 /a*;b*/ \text{ returns true with } t = 3, \text{ because} \\
\text{ilet_reg_exp_match } 0 3 /a*/ \text{ matches "aaa", as } s[0]=s[1]=s[2]=a \text{ and} \\
\end{align*}

The regular expressions are, similar to derived operators like \texttt{until}, directly translatable to basic ILET operators and hence represent convenience constructs. Here, for example, a simple protocol pattern for data transfer, once as regular expression / \texttt{start; data \* [\geq 3]; finish} / and once using basic ILET operators.

\textbf{lem}mma \texttt{ilet-RegExp1-start-data-finish}:

\begin{align*}
\text{let } t t' = \left( \text{BREBool (BREATom start)) } \text{'} } \text{b} \\
\text{let } ((\text{BREATom data }) } \text{'} n \geq 3 ] \text{b} \\
\text{let } (\text{BREBool (BREATom finish})) = \\
\text{t t t'}. \\
\text{start t + t = t + t} \\
\text{t' t' t1 t2 t1 t2 t1 + t2 + t' = t' + 1)}
\end{align*}

2.2.3 Sequence operators and expressions matching empty words

The sequence overlap operator : requires additional considerations for expressions able to match the empty

word \(\varepsilon\) with regard to well-formed ILET and SALT formulas, as explained later in this section and in

Section 4.

Results for sequence operators with the empty word \(\varepsilon\) as left operand.

\textbf{lem}ma \texttt{bre-reg-exp-overlap-epsilon}:

\(\neg (\text{let } t t' (\varepsilon ) (\varepsilon ) b))\)

\textbf{lem}ma \texttt{bre-reg-exp-subsequent-epsilon}:

\(\neg (\text{let } t t' (\varepsilon ) (\varepsilon ) b) = (\text{let } t t' b)\)

Empty word \(\varepsilon\) matches any interval of length 0.

\textbf{def}inition \texttt{ilet-reg-exp-matches-epsilon} \text{}\texttt{::} \texttt{ilet-reg-exp \Rightarrow bool where} \\
\texttt{ilet-reg-exp-matches-epsilon} \text{\texttt{r = let} 0 \Theta r} \\
\textbf{lem}ma \texttt{ilet-reg-exp-matches-epsilon-any-time} \text{::} \texttt{let} t t r = \text{ilet-reg-exp-matches-epsilon} \text{r} \\
All expressions matching \(\varepsilon\).

\textbf{lem}ma \texttt{ilet-reg-exp-matches-epsilon-conv}:

\((r = \varepsilon) \lor \\
(\exists b. r = (b ' s') \geq 0) \lor \\
(\exists a b. (r = (a \lor b) \land \\
 \texttt{(ilet-reg-exp-matches-epsilon} a \lor \texttt{ilet-reg-exp-matches-epsilon}} b)) \lor \\
(\exists a b. (r = (a ' s') \lor \\
 \texttt{ilet-reg-exp-matches-epsilon} a \land \texttt{ilet-reg-exp-matches-epsilon}} b)) = \\
\texttt{(ilet-reg-exp-matches-epsilon}} r)\)

Function determining regular expressions where there is at least one sequence whose last element

matches \(\varepsilon\).

\textbf{fun} \texttt{ilet-reg-exp-seq-last-matches-epsilon} \text{::} \texttt{ilet-reg-exp \Rightarrow bool where}
 Analogue function determining regular expressions where there is at least one sequence whose first element matches $\varepsilon$.

**fun**

| let-reg-exp-seq-first-matches-epsilon :: let-reg-exp $\Rightarrow$ bool 
| where 
| let-reg-exp-seq-first-matches-epsilon $\triangleq$ let-reg-exp-seq-first-matches-epsilon $\vee$ let-reg-exp-seq-overlap-with-epsilon $\omega$ 
| $\omega$ $= \omega \vee \omega$ 

Function determining regular expressions, in which there is at least one sequence overlap operator $\omega$ for which at least one operand matches $\varepsilon$.

**fun**

| let-reg-exp-overlap-with-epsilon :: let-reg-exp $\Rightarrow$ bool 
| where 
| let-reg-exp-overlap-with-epsilon $\triangleq$ let-reg-exp-overlap-with-epsilon $\omega$ $\vee$ let-reg-exp-overlap-with-epsilon $\omega$ $\vee$ let-reg-exp-overlap-with-epsilon $\omega$ 
| $\omega$ $= \omega \vee \omega$ 

Some examples of ILET regular expressions with and without overlaps with empty words:

**lemma**

| $a_1 = \text{BREBool} \ a_1$; $a_2 = \text{BREBool} \ a_2$; $a_3 = \text{BREBool} \ a_3$; $a_4 = \text{BREBool} \ a_4$; $a_5 = \text{BREBool} \ a_5$; $a_6 = \text{BREBool} \ a_6$; $a_7 = \text{BREBool} \ a_7$ 
| in 
| (let-reg-exp-overlap-with-epsilon $\omega$) 
| (let-reg-exp-overlap-with-epsilon $\omega$) 
| (let-reg-exp-overlap-with-epsilon $\omega$) 
| (let-reg-exp-overlap-with-epsilon $\omega$) 


expressions with proper overlap operators. It follows as corollaries that sequence and overlap operator are associative with each other on regular expressions with proper overlap operators: an ILET regular expression is considered well-formed w.r.t. proper overlaps: an ILET regular expression is considered well-formed w.r.t. proper overlaps if for every overlap operator both operands cannot match the empty word/interval.

**Definition of well-formedness condition w.r.t. proper overlaps:** an ILET regular expression is considered well-formed w.r.t. overlap operator if for every overlap operator both operands cannot match the empty word/interval.

**Definition**

\[ \text{ilet-reg-exp-proper-overlap} :: \text{ilet-reg-exp} \to \text{bool} \text{ where} \]

\[ \text{ilet-reg-exp-proper-overlap} \equiv \neg (\text{ilet-reg-exp-overlap-with-epsilon} \ r) \]

The sequence and overlap operators are associative.

**Lemma**

**ILETRegExp-subsequent-assoc:**

\[ (\text{\text{ILETRegExp-overlap-subsequent-assoc}}) \]

\[ \text{ILETRegExp-overlap-subsequent-assoc} \equiv \neg (\text{ILETRegExp-overlap-subsequent-overlap-assoc}) \]

The sequence and overlap operators are associative with each other only if the middle operand cannot match the empty word.

**Lemma**

**ILETRegExp-subsequent-overlap-assoc:**

\[ (\text{\text{ILETRegExp-subsequent-overlap-assoc}}) \]

\[ \text{ILETRegExp-subsequent-overlap-assoc} \equiv \neg (\text{ILETRegExp-subsequent-overlap-assoc}) \]

It follows as corollaries that sequence and overlap operator are associative with each other on regular expressions with proper overlap operators.

**Corollary**

**ILETRegExp-subsequent-overlap-assoc-proper-overlap:**

\[ (\text{\text{ILETRegExp-subsequent-overlap-assoc-proper-overlap}}) \]

\[ \text{ILETRegExp-subsequent-overlap-assoc-proper-overlap} \equiv \neg (\text{ILETRegExp-subsequent-overlap-assoc}) \]

If a regular expression matching the empty word neighbours an overlap operator (improper overlap) then different parenthesis of the sequence can result in different formula meaning:

**Lemma**

**ILETRegExp-subsequent-overlap-epsilon-left:**

\[ (\text{\text{ILETRegExp-subsequent-overlap-epsilon-left}}) \]

\[ \text{ILETRegExp-subsequent-overlap-epsilon-left} \equiv \neg (\text{ILETRegExp-subsequent-overlap-epsilon-right}) \]

Consequently sequence and overlap operator can in general be non-associative with each other:

**Lemma**

**NOT-ILETRegExp-subsequent-overlap-assoc:**

\[ (\text{\text{NOT-ILETRegExp-subsequent-overlap-assoc}}) \]

\[ \text{NOT-ILETRegExp-subsequent-overlap-assoc} \equiv \neg (\text{NOT-ILETRegExp-subsequent-overlap-assoc}) \]
3 LTL

3.1 Syntax

Syntax of deep embedding of LTL.

Data type for LTL formulas:

datatype 'a ltl-formula =
  LTLAtom 'a ⇒ bool
  | LTLNot 'a ltl-formula
  | LTLAnd 'a ltl-formula 'a ltl-formula
  | LTLor 'a ltl-formula 'a ltl-formula
  | LTLImp 'a ltl-formula 'a ltl-formula
  | LTLEquiv 'a ltl-formula 'a ltl-formula
  | LTLEventually 'a ltl-formula
  | LTLUntil 'a ltl-formula 'a ltl-formula
  | LTLUntilWeak 'a ltl-formula 'a ltl-formula
  | LTLrelease 'a ltl-formula 'a ltl-formula

3.2 Semantics

Validity function for LTL formulas – definition through translation to (shallow embedding of) ILET:

consts
  ltl-valid :: (Time ⇒ 'a) ⇒ Time ⇒ 'a ltl-formula ⇒ bool
  ( ( | | ) ) ( )

primrec
  s | =tl t (LTLAtom a) = a (s t)
  s | =tl t (¬tl f) = (¬(s | =tl t f))
  s | =tl t (f1 ∧tl t f2) = ((s | =tl t f1) ∧ (s | =tl t f2))
  s | =tl t (f1 ∨tl t f2) = ((s | =tl t f1) ∨ (s | =tl t f2))
  s | =tl t (¬t1 f) = ((s | =tl t f1) → (s | =tl t f2))
  s | =tl t (tl f) = (tl t (0 s). s | =tl t f)
  s | =tl t (tl f) = (tl t (t s). s | =tl t f)
  s | =tl t (tl f) = (tl t (t s). s | =tl t f)
  s | =tl t (tl f) = (0 t1 {t}. s | =tl t f)
  s | =tl t (tl f) = (tl t (0 t). s | =tl t f)
  s | =tl t (tl f) = (0 t1 {t}. s | =tl t f)
  s | =tl t (tl f) = (t1 t2 {t}. s | =tl t f)
  s | =tl t (tl f) = (t1 t2 {t}. s | =tl t f)
  s | =tl t (tl f) = (t1 t2 {t}. s | =tl t f)

Convenience shortcuts for Boolean constants in LTL formulas:

consts
  LTLTrue :: 'a ltl-formula
  LTLFalse :: 'a ltl-formula

defs
  LTLTrue-def[simp] : LTLTrue ≡ LTLAtom (λx. True)
  LTLFalse-def[simp] : LTLFalse ≡ LTLAtom (λx. False)

lemma
  LTLTrue-conv: (s | =tl t LTLTrue) = True and
  LTLFalse-conv: (s | =tl t LTLFalse) = False

LTL is often defined on basis of Boolean operators not, and and temporal operators until, next. Further

Boolean operators or, implies, equiv and temporal operators eventually, always, weak until, release can
be then defined as abbreviations. The commonly used abbreviations and the explicit semantics definition
through translation to ILET are equivalent:

lemma
  ltl-disj-equiv: (s | =tl t (f1 ∨tl t f2)) = (s | =tl t ¬tl (¬tl t (f1 ∧tl t f2) ∧tl t f2)) and
\[
\text{ltl-imp-equiv: } (s \models_{ltl} t (f1 \rightarrow_{ltl} f2)) = (s \models_{ltl} t ((\neg_{ltl} f1) \lor_{ltl} f2)) \quad \text{and}
\]
\[
\text{ltl-imp-equiv: } (s \models_{ltl} t (f1 \rightarrow_{ltl} f2)) = (s \models_{ltl} t ((f1 \rightarrow_{ltl} f2) \land_{ltl} (f2 \rightarrow_{ltl} f1)))
\]

**Lemma**

\[
\text{ltl-eventually-equiv: } (s \models_{ltl} t (\diamond_{ltl} f)) = (s \models_{ltl} t (\text{LTLTrue} U_{ltl} f)) \quad \text{and}
\]
\[
\text{ltl-eventually-equiv: } (s \models_{ltl} t (\diamond_{ltl} f)) = (s \models_{ltl} t (\neg_{ltl} \diamond_{ltl} (\neg_{ltl} f)))
\]

**Lemma**

\[
\text{ltl-untilweak-equiv: } (s \models_{ltl} t (f1 W_{ltl} f2)) = (s \models_{ltl} t ((f1 U_{ltl} f2) \lor_{ltl} f2 \land_{ltl} f1))
\]

**Lemma**

\[
\text{ltl-release-equiv: } (s \models_{ltl} t (f1 R_{ltl} f2)) = (s \models_{ltl} t (f2 W_{ltl} (f2 \land_{ltl} f1)))
\]

## 4 Core SALT

We consider following core SALT language constructs:

- **Boolean operators** `not`, `and`, `or`, `implies`, `equals`.
- **Common temporal operators** `next`, `always`, `eventually`.
- **Extended `until` operator capable of encoding LTL operators until, until weak, release.**
- **from** operator.
- **Restricted regular expressions**
  - Boolean operators on propositions.
  - Disjunction on regular expressions.
  - Repetition operator `*$n$*` with $n \in \mathbb{N}$ for propositional expressions.
  - Operators `;` and `:` expressing successive and overlapping sequences, respectively.

Few core SALT constructs are not treated here and are considered part of future work:

- **Scope operators using the SALT-- stop operators (e.g., `upto`).**
- **Exception operators** `accepton`, `rejecton`.

As the SALT-- translation step [Str06 Section 6.2] is only needed for translation of the omitted operators, we do not have to consider it and can translate core SALT directly to LTL.

### 4.1 Syntax

Syntax of deep embedding of core SALT.

**Data types for parameters of some core SALT operators.**

```plaintext
datatype SALT-req-opt-weak =
  req ( req )
| opt ( opt )
| weak ( weak )
```

```plaintext
datatype SALT-req-opt =
  req2 ( req )
| opt2 ( opt )
```

```plaintext
datatype SALT-excl-incl =
  excl ( excl )
| incl ( incl )
```

**Data types for core SALT regular expressions:**

```plaintext
datatype 'a core-salt-reg-exp-bool =
  CoreSREAtom 'a => bool  ( CoreSREAtom - [115] 115)
| CoreSRENot 'a core-salt-reg-exp-bool  ( not - [49] 49)
| CoreSREAnd 'a core-salt-reg-exp-bool 'a core-salt-reg-exp-bool
  ( (- and -) [36, 35] 35)
| CoreSREOr 'a core-salt-reg-exp-bool 'a core-salt-reg-exp-bool
  ( (- or -) [31, 30] 30)
| CoreSREImp 'a core-salt-reg-exp-bool 'a core-salt-reg-exp-bool
  ( (- implies -) [26, 25] 25)
```

12
4.2 Translation to LTL

Translation of core SALT to LTL according to [Str06, Section 6].

4.2.1 Translation of regular expressions to LTL

Translating Boolean regular expressions to LTL.

```ml
| CoreSREEqv 'a core-salt-reg-exp-bool 'a core-salt-reg-exp-bool ( (- equals -) [26, 25] 25)
| CoreSREImp 'a core-salt-reg-exp-bool 'a core-salt-reg-exp-bool ( (- implies -) [26, 25] 25)
| CoreSRENot 'a core-salt-reg-exp-bool ( not- [40] 40)
| CoreSREAnd 'a core-salt-reg-exp-bool 'a core-salt-reg-exp-bool ( (- and -) [36, 35] 35)
| CoreSREOr 'a core-salt-reg-exp-bool 'a core-salt-reg-exp-bool ( (- or -) [31, 30] 30)
| CoreSREEqv 'a core-salt-formula 'a core-salt-formula ( (- equals -) [26, 25] 25)
| CoreSRENext 'a core-salt-formula (next -) [50] 50)
| CoreSREAlways 'a core-salt-formula (always -) [50] 50)
| CoreSREEventually 'a core-salt-formula (eventually -) [50] 50)
| CoreSREUntilExt 'a core-salt-formula SALT-excl-incl SALT-req-opt-weak 'a core-salt-formula ( (- until - -) [50, 50, 50, 51] 50)
| CoreSREFrom 'a core-salt-formula SALT-excl-incl SALT-req-opt ( (- from - -) [50, 50, 50, 51] 50)
| CoreSRERegExp 'a core-salt-reg-exp ( ("\\" - "\\"[coresre]) [50] 50)
| CoreSRERegExpSeqSaltFinish 'a core-salt-reg-exp 'a core-salt-formula ( ("\\" - "\\"end - "\\"[coresre]) [51.50] 50)
```

Data type for core SALT formulas:

```ml
datatype 'a core-salt-formula =
  CoreSALTAtom 'a ⇒ bool ( CoreSALTAtom - [115] 115)
  CoreSALTNot 'a core-salt-formula ( not- [40] 40)
  CoreSALTAnd 'a core-salt-formula 'a core-salt-formula ( (- and -) [36, 35] 35)
  CoreSREOr 'a core-salt-formula 'a core-salt-formula ( (- or -) [31, 30] 30)
  CoreSREImp 'a core-salt-formula 'a core-salt-formula ( (- implies -) [26, 25] 25)
  CoreSREEqv 'a core-salt-formula 'a core-salt-formula ( (- equals -) [26, 25] 25)
  CoreSRENext 'a core-salt-formula (next -) [50] 50)
  CoreSREAlways 'a core-salt-formula (always -) [50] 50)
  CoreSREEventually 'a core-salt-formula (eventually -) [50] 50)
  CoreSREUntilExt 'a core-salt-formula SALT-excl-incl SALT-req-opt-weak 'a core-salt-formula ( (- until - -) [50, 50, 50, 51] 50)
  CoreSREFrom 'a core-salt-formula SALT-excl-incl SALT-req-opt ( (- from - -) [50, 50, 50, 51] 50)
  CoreSRERegExp 'a core-salt-reg-exp ( ("\\" - "\\"[coresre]) [50] 50)
  CoreSRERegExpSeqSaltFinish 'a core-salt-reg-exp 'a core-salt-formula ( ("\\" - "\\"end - "\\"[coresre]) [51.50] 50)
```

4.2 Translation to LTL
Function for constructing an LTL formula consisting of $n$ subsequent next operators applied to a parameter LTL formula $f$. The resulting formula states that $f$ holds for $n$ steps after current time.

consts
nextn-ltl :: nat ⇒ 'a ltl-formula ⇒ 'a ltl-formula
(_[Suc n]) ([50,50] 50)
primrec
(0) $f$ = $f$
(Suc $f$) = (_ $f$)

Function expressing a bounded always operator in LTL. It constructs an LTL formula stating that $f$ holds for $n$ steps (in an interval $[t \ldots t + n]$ when evaluated at time $t$).

consts
alwaysn-ltl :: nat ⇒ 'a ltl-formula ⇒ 'a ltl-formula
(_[Suc n]) ([50,50] 50)
primrec
(0) $f$ = LTLTrue
(Suc $f$) = ($f$ _ $f$)

Properties of nextn-ltl and alwaysn-ltl.

lemma nextn-ltl-conv: \[ t. (s \models (\bigcirc_{ltl}[n] f)) = (s \models (\bigcirc_{ltl}[t + n] f)) \]
lemma alwaysn-ltl-conv: \[ t. (s \models (\bigcirc_{ltl}[n] f)) = (\Box (t..t + n). (s \models t' f)) \]

Translating sequence operators to LTL (mutually recursive function definitions):

fun
sre-subsequent-to-ltl :: 'a core-salt-reg-exp ⇒ 'a ltl-formula and
sre-overlap-to-ltl :: 'a core-salt-reg-exp ⇒ 'a ltl-formula

where
(sre-subsequent-to-ltl (CoreSREBool b) f) =
((core-salt-reg-exp-bool-to-ltl b) _ $f$)
| (sre-subsequent-to-ltl 'b f) = $f$
| (sre-subsequent-to-ltl a or b f) =
((sre-subsequent-to-ltl a f) _ (sre-subsequent-to-ltl b f))
| sre-subsequent-to-ltl-simp-subeq:
(sre-subsequent-to-ltl a 'b f) =
(sre-subsequent-to-ltl a (sre-subsequent-to-ltl b f))
| sre-overlap-subsequent-to-ltl-simp-subeq:
(sre-subsequent-to-ltl a 'b f) =
(sre-overlap-subsequent-to-ltl a (sre-subsequent-to-ltl b f))
| (sre-subsequent-to-ltl (b 'b) $f$) =
((core-salt-reg-exp-bool-to-ltl b) _ $f$
| (sre-overlap-to-ltl (CoreSREBool b) f) = ((core-salt-reg-exp-bool-to-ltl b) _ $f$
| (sre-overlap-to-ltl 'b f) = $f$
| (sre-overlap-to-ltl a or b f) =
((sre-overlap-to-ltl a f) _ (sre-overlap-to-ltl b f))
| sre-overlap-subsequent-to-ltl-simp-subeq:
(sre-overlap-to-ltl a 'b f) =
(sre-subsequent-to-ltl a (sre-overlap-to-ltl b f))
| sre-overlap-to-ltl-simp-subeq:
(sre-overlap-to-ltl a 'b f) =
(sre-overlap-to-ltl a (sre-overlap-to-ltl b f))
| (sre-overlap-to-ltl (b 'b) $f$) =
;if $n = 0$ then

14
\[(f \lor_{ltl} ((\text{core-salt-reg-exp-bool-to-ltl } b) \ U_{ltl} ((\text{core-salt-reg-exp-bool-to-ltl } b) \wedge_{ltl} f)))\]

else

\[((\text{core-salt-reg-exp-bool-to-ltl } b) \ U_{ltl} (((\text{core-salt-reg-exp-bool-to-ltl } b) \wedge_{ltl} (O_{ltl}[n-l] f))))\]

Translating all core SALT regular operators:

fun

\text{sre-core-to-ltl} :: 'a \text{ core-salt-reg-exp} \Rightarrow 'a \text{ ltl-formula}

where

\text{sre-core-to-ltl} (\text{CoreSREbool } b)) = (\text{core-salt-reg-exp-bool-to-ltl } b)

| (\text{sre-core-to-ltl} (\text{a or } b)) = ((\text{sre-core-to-ltl } a) \lor_{ltl} (\text{sre-core-to-ltl } b))

| sre-core-to-ltl-simp-subseq:

(sre-core-to-ltl (a \ 'b)) = (sre-subsequent-to-ltl a (sre-core-to-ltl b))

| sre-core-to-ltl-simp-overlap:

(sre-core-to-ltl (a \ 'b)) = (sre-overlap-to-ltl a (sre-core-to-ltl b))

| (sre-core-to-ltl (b \ 'c [\geq n]\text{corere})) = (O_{ltl}[n] (\text{core-salt-reg-exp-bool-to-ltl } b))

Core SALT sequence operators are associative, not only w.r.t. to the semantical equivalence of the LTL formulas resulting from the translation but even syntactically, i.e., the resulting LTL formulas are syntactically equal.

lemma

\text{sre-core-to-ltl-subsequent-assoc}:
\hspace{1em} sre-core-to-ltl ((a \ 'b) \ 'c) = sre-core-to-ltl (a \ 'b \ 'c) and

\text{sre-core-to-ltl-overlap-assoc}:
\hspace{1em} sre-core-to-ltl ((a \ 'b) \ 'c) = sre-core-to-ltl (a \ 'b \ 'c) and

\text{sre-core-to-ltl-subsequent-overlap-assoc}:
\hspace{1em} sre-core-to-ltl ((a \ 'b) \ 'c) = sre-core-to-ltl (a \ 'b \ 'c) and

\text{sre-core-to-ltl-overlap-subsequent-assoc}:
\hspace{1em} sre-core-to-ltl ((a \ 'b) \ 'c) = sre-core-to-ltl (a \ 'b \ 'c)

Contrary to ILET, core SALT sequence operators are associative without well-formedness preconditions. The reason is that due to translation definition all sequences are considered right-associative independently of the actual parenthesis. Consider the translation of the sequences \((a; \varepsilon) : c : \text{and } a; (\varepsilon : c), which are both translated according to the right-associative interpretation \(a; \varepsilon : c = a; c \) where the empty word \(\varepsilon\) is "consumed" by \(c\) (which corresponds to the LTL formula \(a \wedge \triangledown c\) if \(a\) and \(c\) are Boolean expressions).

lemma

\text{sre-core-subsequent-overlap-epsilon-left}:
\hspace{1em} sre-core-to-ltl ((a \ 'c) \ 'c) = sre-core-to-ltl (a \ 'c) and

\text{sre-core-subsequent-overlap-epsilon-right}:
\hspace{1em} sre-core-to-ltl (a \ 'c \ 'c) = sre-core-to-ltl (a \ 'c)

Obviously we cannot provide a sound semantics for all formulas if the translation syntactically forces the sequence operators to be right associative and at the same time the semantics of the expressions \((a; \varepsilon): c = a : c\) and \(a; (\varepsilon : c) = a; c\) are different (the interpretation \(\varepsilon : c\) corresponds to the description in [Str06 p. 42]; in ILET the semantics of \(\varepsilon : c\) is False, cf. lemma ILETRegExp-subsequent-overlap-epsilon-right in Section 2.2.3).

Hence, for proving the correctness of the translation of core SALT to LTL we will have to restrict the set of well-formed core SALT regular expressions by the condition that an expression matching the empty word \(\varepsilon\) may not neighbour the overlap operator : (cf. Section 4.4).

4.2.2 Translation of until and from operators to LTL

Translating the extended until operator to LTL.

consts

\text{ltl-untilext} :: SALT-excl-incl \Rightarrow SALT-opt-opt-weak \Rightarrow 'a \text{ ltl-formula} \Rightarrow
\hspace{1em} 'a \text{ ltl-formula} \Rightarrow 'a \text{ ltl-formula}
\begin{verbatim}
ltl-untilexcl-incl :: SALT-excl-incl ⇒ 'a ltl-formula ⇒ 'a ltl-formula
primes
ltl-untilexcl-excl f1 f2 = f2
ltl-untilexcl-incl f1 f2 = (f1 \land f2)

primrec
ltl-untilexclincl req f1 f2 = (f1 \until (ltl-untilexclincl exclincl f1 f2))
ltl-untilexclincl opt f1 f2 = ((\o f2) \rightarrow (f1 \until (ltl-untilexclincl exclincl f1 f2)))
ltl-untilexclincl weak f1 f2 = (f1 \wedge (ltl f1 \until (ltl-untilexclincl exclincl f1 f2)))

The translation function for the extended until operator returns exactly the LTL formulas given in the SALT language reference [Str06, p. 40].

lemma
ltl-untilexcl-excl req: ltl-untilexcl excl req f1 f2 = (f1 \until (ltl-untilexclincl exclincl f1 f2)) and
ltl-untilexcl-excl opt: ltl-untilexcl excl opt f1 f2 = ((\o f2) \rightarrow (f1 \until (ltl-untilexclincl exclincl f1 f2))) and
ltl-untilexcl-excl weak: ltl-untilexcl excl weak f1 f2 = (f1 \wedge (ltl f1 \until (ltl-untilexclincl exclincl f1 f2))) and
ltl-untilexclincl req: ltl-untilexcl incl req f1 f2 = (f1 \until (f1 \land f2)) and
ltl-untilexclincl opt: ltl-untilexcl incl opt f1 f2 = ((\o f2) \rightarrow (f1 \until (f1 \land f2))) and
ltl-untilexclincl weak: ltl-untilexcl incl weak f1 f2 = (f1 \wedge (f1 \until (ltl f1 \land f2)))

Translating the \textit{from} operator to LTL.

consts
ltl-from-exclincl :: SALT-excl-incl ⇒ 'a ltl-formula ⇒ 'a ltl-formula
primes
ltl-from-exclincl incl f = f
ltl-from-exclincl excl f = (\o f)

constdefs
ltl-from :: SALT-excl-incl ⇒ SALT-req-opt ⇒ 'a ltl-formula ⇒ 'a ltl-formula
ltl-from exclincl requopt f a ≡ (case requopt of req ⇒ LTLUntil | opt ⇒ LTLUntilWeak)
ð\o LTLAtom a \land (ltl-from-exclincl exclincl f))

The translation function for the \textit{from} operator returns exactly the LTL formulas given in the SALT language reference [Str06, p. 42].

lemma
ltl-from-excl req: ltl-from excl req f a = (ltl f \until (ltl LTLAtom a \land (\o f))) and
ltl-from-excl opt: ltl-from excl opt f a = (ltl f \until (ltl LTLAtom a \land (\o f))) and
ltl-from-incl req: ltl-from incl req f a = (ltl f \until (ltl LTLAtom a \land (\o f))) and
ltl-from-incl opt: ltl-from incl opt f a = (ltl f \until (ltl LTLAtom a \land (\o f)))

4.2.3 Translation of core SALT formulas to LTL

Main function for translation of core SALT to LTL.

consts
core-salt-to-ltl :: 'a core-salt-formula ⇒ 'a ltl-formula
primes
core-salt-to-ltl (CoreSALTAtom a) = LTLAtom a
core-salt-to-ltl (not f) = (\o (ltl core-salt-to-ltl f))
core-salt-to-ltl (f1 \land f2) = (core-salt-to-ltl f1 \land (core-salt-to-ltl f2))
\end{verbatim}
core-salt-to-ltl (f₁ or f₂) = (core-salt-to-ltl f₁ ∨_{ltl} core-salt-to-ltl f₂)
core-salt-to-ltl (f₁ implies f₂) = (core-salt-to-ltl f₁ →_{ltl} core-salt-to-ltl f₂)
core-salt-to-ltl (f₁ equals f₂) = (core-salt-to-ltl f₁ ↔_{ltl} core-salt-to-ltl f₂)
core-salt-to-ltl (next f) = (◊_{ltl} core-salt-to-ltl f)
core-salt-to-ltl (always f) = (□_{ltl} core-salt-to-ltl f)
core-salt-to-ltl (eventually f) = (◊_{ltl} core-salt-to-ltl f)
core-salt-to-ltl (f₁ until exclincl reqoptweak f₂) =
(ltl-untilext exclincl reqoptweak (core-salt-to-ltl f₁) (core-salt-to-ltl f₂))
core-salt-to-ltl (f from exclincl reqopt (core-salt-to-ltl f) a) =
(ltl-from exclincl reqopt (core-salt-to-ltl f)) a)
core-salt-to-ltl ('|' r 'r'coresre ) = (sre-core-to-ltl r)
core-salt-to-ltl ('|' r 'r'end f 'f'coresre ) =
(sre-subsequent-to-ltl r (core-salt-to-ltl f))
core-salt-to-ltl ('|' r 'r'end f 'f'coresre ) =
(sre-overlap-to-ltl r (core-salt-to-ltl f))

Below we define auxiliary functions for showing the equivalence of the translation definition used here
and the translation definition in the SALT language reference [Str06, p. 42] for the regular operators star *,
sequence ;, and overlap |.

Function for constructing a sequence of n + 1 repetitions of a regular expression.

consts
subsequentn-coresre :: nat ⇒ 'a core-salt-reg-exp ⇒ 'a core-salt-reg-exp

primrec
subsequentn-coresre θ r = r
subsequentn-coresre (Suc n) r = (r 'r' subsequentn-coresre n r)

Function for constructing a SALT formula, which is a regular expression containing n repetitions of a
Boolean expression (empty word for n = 0).

consts
defs
subsequentn-core-salt :: nat ⇒ 'a core-salt-reg-exp-bool ⇒ 'a core-salt-formula
subsequentn-core-salt n b ≡ 'r' (case n of
θ ⇒ ε | Suc n' ⇒ subsequentn-coresre n' (CoreSREBool b) 'r'coresre

Function for constructing a SALT formula, which is a regular expression containing n repetitions of a
Boolean expression, followed by a further core SALT formula.

consts
defs
subsequentn-tail-core-salt :: nat ⇒ 'a core-salt-reg-exp-bool ⇒
'a core-salt-formula ⇒ 'a core-salt-formula
subsequentn-tail-core-salt n b f ≡ (case n of
θ ⇒ f | Suc n' ⇒ 'r' subsequentn-coresre n' (CoreSREBool b) 'r'end f 'f'coresre

Function for constructing a SALT formula, which is a regular expression containing n repetitions of a
Boolean expression, followed by an overlapping core SALT formula.

consts
defs
subsequentn-tail-overlap-core-salt :: nat ⇒ 'a core-salt-reg-exp-bool ⇒
'a core-salt-formula ⇒ 'a core-salt-formula
subsequentn-tail-overlap-core-salt n b f ≡ (case n of
θ ⇒ f | Suc n' ⇒ 'r' subsequentn-coresre n' (CoreSREBool b) 'r'end f 'f'coresre

Some examples of generating regular expressions representing n subsequent repetitions of a given
regular expression r or a Boolean regular expression b, possibly followed by a core SALT formula f.

lemma subsequentn-coresre-3:
subsequentn-coresre 3 r = r 'r' 'r' r 'r' r
lemma subsequentn-core-salt-θ:
subsequentn-core-salt θ b = ('r' ε 'f'coresre)
lemma subsequentn-core-salt-3: let r = CoreSREBool b in
subsequentn-core-salt 3 b = ('r' 'f' 'r' 'r' 'r' r 'f'coresre)
lemma subsequentn-tail-core-salt-θ:

5The operator : is written in apostrophes solely to distinguish it from punctuation marks.
subsequentn-tail-core-salt θ b f = f

**Lemma** subsequentn-tail-core-salt-3: let r = CoreSREBool b in subsequentn-tail-core-salt 3 b f = (′r f′ r f′ r f′ end f |′ coresre)

For translating the star operator $s[≥ n]$ to LTL we use the function alwaysn-ltl (syntax $\square_{ltl}[^n]$) (cf. sre-core-to-ltl). Here the equivalence of this definition and the definition in the SALT language reference [Str06, p. 42] is shown.

**Lemma** core-salt-StarGe-equiv-alwaysn-ltl: $\forall t$. s |=ltl t (core-salt-to-ltl (′b b s |′ n coresre ;′ end f |′ coresre)) = s |=ltl t (core-salt-reg-exp-bool-to-ltl b) $U_{ltl}$ (core-salt-to-ltl (subsequentn-tail-core-salt n b f))

For translating the star operator $s[≥ n]$ with the sequence operator ; to LTL we use the functions alwaysn-ltl and nextn-ltl (syntax $\bigcirc_{ltl}[^1]$) (cf. sre-subsequent-to-ltl). Here the equivalence of this definition and the definition in the SALT language reference [Str06, p. 42] is shown.

**Lemma** core-salt-StarGe-Subsequent-equiv-alwaysn-ltl: $\forall t$. s |=ltl t (core-salt-to-ltl (′b b s |′ n coresre ;′ end f |′ coresre)) = s |=ltl t (core-salt-reg-exp-bool-to-ltl b) $U_{ltl}$ (core-salt-to-ltl (subsequentn-tail-overlap-core-salt n b f))

Finally, the analogue equivalence of the translation definition of the star operator $s[≥ n]$ with the overlap operator : to LTL and the definition in the SALT language reference [Str06, p. 42] is shown.

**Lemma** core-salt-StarGe-Overlap-equiv-alwaysn-ltl: $\forall t$. θ < n $\implies$ s |=ltl t (core-salt-to-ltl (′b b s |′ n coresre ;′ end f |′ coresre)) = s |=ltl t (core-salt-reg-exp-bool-to-ltl b) $U_{ltl}$ (core-salt-to-ltl (subsequentn-tail-overlap-core-salt n b f))

### 4.3 Semantics

Definition of core SALT semantics by translation of core SALT formulas to ILET. A formula is first translated to an ILET formula, which can contain ILET regular expressions if the core SALT formula contains regular expressions. In the final step the ILET regular expressions are translated to ILET – the resulting formula gives the formal semantics of the core SALT formula.

#### 4.3.1 Translation of regular expressions to ILET

Translating Boolean regular expressions to ILET.

**consts**

core-salt-reg-exp-bool-to-ilet :: (Time $\Rightarrow$ 'a) $\Rightarrow$ 'a core-salt-reg-exp-bool $\Rightarrow$ illet-reg-exp-bool

**primrec**

core-salt-reg-exp-bool-to-ilet s (CoreSREAtom a) = (BREAtom ($\lambda$x. a (s x)))
core-salt-reg-exp-bool-to-ilet s (not f) = (¬bre core-salt-reg-exp-bool-to-ilet s f)
core-salt-reg-exp-bool-to-ilet s (f1 and f2) = (core-salt-reg-exp-bool-to-ilet s f1 $\land_{bre}$ core-salt-reg-exp-bool-to-ilet s f2)
core-salt-reg-exp-bool-to-ilet s (f1 or f2) = (core-salt-reg-exp-bool-to-ilet s f1 $\lor_{bre}$ core-salt-reg-exp-bool-to-ilet s f2)
core-salt-reg-exp-bool-to-ilet s (f1 implies f2) = (core-salt-reg-exp-bool-to-ilet s f1 $\implies_{bre}$ core-salt-reg-exp-bool-to-ilet s f2)
core-salt-reg-exp-bool-to-ilet s (f1 equals f2) = (core-salt-reg-exp-bool-to-ilet s f1 $\equiv_{bre}$ core-salt-reg-exp-bool-to-ilet s f2)

Translating regular expressions to ILET.

**consts**

sre-core-to-ilet :: (Time $\Rightarrow$ 'a) $\Rightarrow$ 'a core-salt-reg-exp $\Rightarrow$ illet-reg-exp

**primrec**

(sre-core-to-ilet s (CoreSREBool b)) = BREBool (core-salt-reg-exp-bool-to-ilet s b)
(sre-core-to-ilet s $\varepsilon$) = $\varepsilon$
(sre-core-to-ilet s (a or b)) = (sre-core-to-ilet s a ∨ sre-core-to-ilet s b)
(sre-core-to-ilet s (a 'b)) = (sre-core-to-ilet s a 'bre sre-core-to-ilet s b)
(sre-core-to-ilet s (a 'b)) = (sre-core-to-ilet s a 'bre sre-core-to-ilet s b)
(sre-core-to-ilet s (b 'w' ≥ n[core-sre])) =
((core-salt-reg-exp-bool-to-ilet s b) 'w' [≥ n[bre]])

4.3.2 Translation of until and from operators to ILET

Translating the extended until operator to ILET.

consts
salt-exclincl-to-cut :: SALT-excl-incl ⇒ (iT ⇒ Time ⇒ iT)
salt-reqoptweak-to-ilet ::
SALT-req-opt-weak ⇒ (Time ⇒ bool ⇒ (Time ⇒ bool) ⇒ (Time ⇒ bool)
primrec
salt-exclincl-to-cut excl = (op |<)
salt-exclincl-to-cut incl = (op ≤)
primrec
salt-reqoptweak-to-ilet req f1 f2 = (λt. False)
salt-reqoptweak-to-ilet opt f1 f2 = (λt. ⊢ t2 {t..}. f1 t2)
salt-reqoptweak-to-ilet weak f1 f2 = (λt. ⊢ t1 {t..}. f1 t1)
constdefs
salt-untilext-to-ilet ::
Time ⇒ SALT-excl-incl ⇒ SALT-req-opt-weak ⇒
(Time ⇒ bool) ⇒ (Time ⇒ bool) ⇒
bool
salt-untilext-to-ilet t exclincl reqoptweak f1 f2 ≡
(◊ t2 {t..}. (f2 t2 ∧ (□ t1 ((salt-exclincl-to-cut exclincl) {t..} t2). f1 t1))) ∨
(salt-reqoptweak-to-ilet reqoptweak f1 f2) t

The excl/incl parameter for the extended until operator specifies, whether the time point, at which f2 becomes true, is excluded from the interval, in which f1 must hold. This is done by selecting the corresponding interval cut operator, excluding (|≤) or including (|<) the time point, where the interval is cut.

lemma
salt-exclincl-to-cut--excl: (salt-exclincl-to-cut excl I t) = (I |< t) and
salt-exclincl-to-cut--incl: (salt-exclincl-to-cut incl I t) = (I |≤ t)

Translating the from operator to ILET. Here the excl/incl parameter specifies, whether f must become true at the time point, where a is valid, or at the next time point. The req/opt parameter specifies, whether the formula is also fulfilled, if a never becomes true (parameter value opt).

constdefs
salt-from-to-ilet ::
Time ⇒ SALT-excl-incl ⇒ SALT-req-opt ⇒
(Time ⇒ bool) ⇒ (Time ⇒ bool) ⇒
bool
salt-from-to-ilet t exclincl reqopt f a ≡
(◊ a t2 {t..}. (a t2 ∧ (□ t1 {(t..) |< t2}. f a t1) ∧
(f (case exclincl of excl ⇒ Suc t2 | incl ⇒ t2))) ∨
(case reqopt of req ⇒ False | opt ⇒ (□ t1 {t..}. f a t1)))

4.3.3 Translation of core SALT formulas to ILET

The semantics of a core SALT formula is given by its translation to ILET.

consts
core-salt-valid :: (Time ⇒ 'a) ⇒ Time ⇒ 'a core-salt-formula ⇒ bool
( (¬ |=_{core-salt} - ) [80,80] 80)
primrec
4.4 Sequence operators and expressions matching empty words

Definition of well-formedness condition w.r.t. proper overlaps: a core SALT regular expression is considered well-formed w.r.t. the overlap operator if for every overlap operator both operands cannot match the empty word/interval.

Core SALT expressions matching the empty word, i.e., an interval of length 0.

**primrec**

```plaintext
core-salt-reg-exp-matches-epsilon :: 'a core-salt-reg-exp ⇒ bool
where

| core-salt-reg-exp-matches-epsilon (CoreSREBool b) = False
| core-salt-reg-exp-matches-epsilon ε = True
| core-salt-reg-exp-matches-epsilon (a or b) = (core-salt-reg-exp-matches-epsilon a ∨ core-salt-reg-exp-matches-epsilon b)
| core-salt-reg-exp-matches-epsilon (a ∨ b) = (core-salt-reg-exp-matches-epsilon a ∧ core-salt-reg-exp-matches-epsilon b)
| core-salt-reg-exp-matches-epsilon (b 'a' ≥ n [coreSRE]) = (n = 0)
```

Function indicating whether the first expression in a sequence matches the empty word.

**fun**

```plaintext
core-salt-reg-exp-seq-first-matches-epsilon :: 'a core-salt-reg-exp ⇒ bool
where

| core-salt-reg-exp-seq-first-matches-epsilon (a 'b' b) = core-salt-reg-exp-seq-first-matches-epsilon a
| core-salt-reg-exp-seq-first-matches-epsilon (a 'b' b) = core-salt-reg-exp-seq-first-matches-epsilon a ∨ core-salt-reg-exp-seq-first-matches-epsilon b
| core-salt-reg-exp-seq-first-matches-epsilon (a or b) = core-salt-reg-exp-seq-first-matches-epsilon a ∨ core-salt-reg-exp-seq-first-matches-epsilon b
| core-salt-reg-exp-seq-first-matches-epsilon r = core-salt-reg-exp-matches-epsilon r
```

Analogue function indicating whether the last expression in a sequence matches the empty word.

**fun**

```plaintext
core-salt-reg-exp-seq-last-matches-epsilon :: 'a core-salt-reg-exp ⇒ bool
where

| core-salt-reg-exp-seq-last-matches-epsilon (a 'b' b) = core-salt-reg-exp-seq-last-matches-epsilon a
| core-salt-reg-exp-seq-last-matches-epsilon (a 'b' b) = core-salt-reg-exp-seq-last-matches-epsilon b
```
core-salt-reg-exp-seq-last-matches-epsilon (a or b) =
  (core-salt-reg-exp-seq-last-matches-epsilon a ∨
   core-salt-reg-exp-seq-last-matches-epsilon b)

Function determining whether the core SALT regular expression contains a sequence where an expression matching the empty word neighbours the overlap operator.

fun core-salt-reg-exp-overlap-with-epsilon :: 'a core-salt-reg-exp ⇒ bool
where
  core-salt-reg-exp-overlap-with-epsilon (a ·' b) =
    (core-salt-reg-exp-seq-first-matches-epsilon a ∨
     core-salt-reg-exp-seq-last-matches-epsilon b)
  core-salt-reg-exp-overlap-with-epsilon a = True

Some examples of core SALT regular expressions with and without overlaps with empty words.

lemma let
  a1 = CoreSREBool a1; a2 = CoreSREBool a2; a3 = CoreSREBool a3; a4 = CoreSREBool a4;
  a5 = CoreSREBool a5; a6 = CoreSREBool a6; a7 = CoreSREBool a7
in
  (core-salt-reg-exp-overlap-with-epsilon ((a1 ·' a2) ·' (a3 ·' (a4 ·' a5) ·'))
     (a6 ·' a7))) = False ∧
  (core-salt-reg-exp-overlap-with-epsilon ((a1 ·' a2) or (a3 ·' (a4 ·' a5) ·'))
     (a6 ·' a7))) = False ∧
  (core-salt-reg-exp-overlap-with-epsilon ((a1 ·' a2) or (ε ·' (a4 ·' a5) ·'))
     (a6 ·' a7))) = True ∧
  (core-salt-reg-exp-overlap-with-epsilon ((a1 ·' ε) or (a3 ·' (a4 ·' a5) ·'))
     (a6 ·' a7))) = True ∧
  (core-salt-reg-exp-overlap-with-epsilon ((a1 ·' a2) or (a3 ·' ((b ·' [≥ 1]coresre) ·' a5) ·'))
     (a6 ·' a7))) = False ∧
  (core-salt-reg-exp-overlap-with-epsilon ((a1 ·' a2) or (a3 ·' ((b ·' [≥ 0]coresre) ·' a5) ·'))
     (a6 ·' a7))) = True)

A core SALT regular expression is well-formed if no regular expression matching the empty word neighbours the overlap operator.

definition core-salt-reg-exp-proper-overlap :: 'a core-salt-reg-exp ⇒ bool where
  core-salt-reg-exp-proper-overlap r ≡ ¬ (core-salt-reg-exp-overlap-with-epsilon r)

Remarkably, a core SALT regular expression is well-formed iff its translation to ILET is well-formed.

lemma core-salt-reg-exp-to-ilet--proper-overlap-eq:
  (ilet-reg-exp-proper-overlap (sre-core-to-ilet s r)) =
  (core-salt-reg-exp-proper-overlap r)

A core SALT formula is well-formed if all regular expressions in it are well-formed w.r.t. overlaps with expressions matching empty words.

consts core-salt-proper-overlap :: 'a core-salt-formula ⇒ bool
primrec
  core-salt-proper-overlap (CoreSALTAtom a) = True
  core-salt-proper-overlap (not f) = (core-salt-proper-overlap f)
  core-salt-proper-overlap (f1 and f2) =
    (core-salt-proper-overlap f1 ∧ core-salt-proper-overlap f2)
  core-salt-proper-overlap (f1 or f2) =
    (core-salt-proper-overlap f1 ∧ core-salt-proper-overlap f2)
  core-salt-proper-overlap (f1 implies f2) =
4.5.1 Selected auxiliary translation validation lemmas

By its ILET translation. The translation of core S

4.5 Formal validation of core SALT compiler but not suitable for formal semantics definition. Ignoring parentheses in sequences, which would be a purely syntactic solution, reasonable for a pragmatic

cannot distinguish such formulas, e.g., by forcing all regular expressions to be right associative and hence

therefore mapping two different core S

ALT

ALT

validation theorem \textit{core-salt-to-ltl-equiv-core-salt-valid} in Sec. 4.5.2 because this theorem will consider core SALT formulas with proper overlaps in regular expressions and hence well-defined semantics. It will not state anything about core SALT formulas with improper overlaps, e.g., / a; b * [0] c / because for them no well-defined semantics exist. Consider the example of the two formulas / (a; e) . c / and / a; e : c / . They are mapped to two different ILET regular expressions and thus assigned two different meanings: / (a; e) . c / = / a : c / . Itlet, because the empty word e is "consumed" by a, while / a; e : c / = / a ; False / = False, because the empty word in the sub-expression e : c cannot match any interval of length > 0, as required by the sub-formula |=let t (Suc t) ε in the definition of \textit{ilet-reg-exp-match}. At the same time the translation to LTL yields the right associative interpretation / a ; e : c / = a \lor c for both formulas therefore mapping two different core SALT formulas with different meanings to the same LTL formula. Thus, a proper semantics definition for such cases is not possible, unless we use a semantics definition that cannot distinguish such formulas, e.g., by forcing all regular expressions to be right associative and hence ignoring parentheses in sequences, which would be a purely syntactic solution, reasonable for a pragmatic compiler but not suitable for formal semantics definition.

4.5 Formal validation of core SALT translation to LTL

The translation of core SALT to LTL is validated by proving that the semantics of an LTL formula obtained by translating a core SALT formula is equivalent to the semantics of the core SALT formula directly given by its ILET translation.

4.5.1 Selected auxiliary translation validation lemmas

Translation validation for the regular repetition operator *:

\textbf{lemma} \textit{core-salt-to-ltl-equiv-core-salt-valid--RegExp-StarGe}:

\((s \models_{i1} t (sre-core-to-ilt (b \; \text{\textquoteleft} [ \geq n]_{\text{corestr}}))) =\)

\((\text{\textquoteleft} (\text{\textquoteleft} (t \ldots)) \models_{\text{\textquoteleft} bre} t \; t2 (sre-core-to-ilet s (b \; \text{\textquoteleft} [ \geq n]_{\text{corestr}})))\)

Translation validation for the sequence operator ; and the sequence overlap operator \(\cdot\):

\textbf{lemma} \textit{core-salt-to-ltl-equiv-core-salt-valid--RegExp-Subsequent}: \(\land t.\)

\textbf{lemma} \textit{core-salt-to-ltl-equiv-core-salt-valid--RegExp-SeqLastMatches-Epsilon}:

\((s \models_{i1} t (sre-overlap-to-ilt t f)) =\)

\((\text{\textquoteleft} (\text{\textquoteleft} (t \ldots)) \models_{\text{\textquoteleft} bre} t \; t2 (sre-core-to-ilet s r) \land (s \models_{i1} t2 f))\) and

\textbf{lemma} \textit{core-salt-to-ltl-equiv-core-salt-valid--RegExp-Overlap}:

\((s \models_{i1} t (sre-overlap-to-ilt t f)) =\)

\((\text{\textquoteleft} (\text{\textquoteleft} (t \ldots)) \models_{\text{\textquoteleft} bre} t \; Suc t2 (sre-core-to-ilet s r) \land (s \models_{i1} t2 f))\)

Translation validation for regular expressions:

\textbf{lemma} \textit{core-salt-to-ltl-equiv-core-salt-valid--RegExp}: \(\land t.\)
lemmas about expressing LTL operators

5.1 LTL operators

5.1.1 Until

ALT

4.5.2 Main translation validation theorem

Core SALT translation to LTL yields the same semantics as the core SALT semantics given by direct translation to ILET:

themorem core-salt-to-ltl-equiv-core-salt-valid: \( \forall t. \)

core-salt-proper-overlap \( f \) \( \Rightarrow \)
\( \forall t. \) \( s \models_{\text{LT}} t \) (core-salt-to-ltl \( f \) ) \( \Rightarrow \)
\( s \models_{\text{core-salt}} t \) \( f \) \( \)

The precondition core-salt-proper-overlap \( f \) indicates that we consider core SALT formulas with proper overlaps in regular expressions and hence well-defined semantics.

5 Additional results for core SALT

5.1 LTL operators

lemmas about expressing LTL operators \( U, V, R \) using the extended until operator in core SALT.

5.1.1 Until

ALT

5.1.2 Weak Until

ALT

5.1.3 Release

ALT

\footnote{The theorem does not state anything about core SALT formulas with improper overlaps, e.g., \( /a; b] \geq 0; c/ \), because for them no well-defined semantics exist.}
Translation functions from LTL to core SALT:

**consts**

\[
\text{lsl-to-core-salt} :: 'a \text{ ltl-formula} \Rightarrow 'a \text{ core-salt-formula}
\]

**primrec**

\[
\begin{align*}
\text{lsl-to-core-salt} \text{ (LTLAtom a)} &= \text{CoreSALTAtom a} \\
\text{lsl-to-core-salt} (\lnot f) &= (\text{not} (\text{lsl-to-core-salt} f)) \\
\text{lsl-to-core-salt} (f_1 \land f_2) &= ((\text{lsl-to-core-salt} f_1) \text{ and} (\text{lsl-to-core-salt} f_2)) \\
\text{lsl-to-core-salt} (f_1 \lor f_2) &= ((\text{lsl-to-core-salt} f_1) \text{ or} (\text{lsl-to-core-salt} f_2)) \\
\text{lsl-to-core-salt} (f_1 \rightarrow f_2) &= ((\text{lsl-to-core-salt} f_1) \text{ implies} (\text{lsl-to-core-salt} f_2)) \\
\text{lsl-to-core-salt} (f_1 \leftrightarrow f_2) &= ((\text{lsl-to-core-salt} f_1) \text{ equals} (\text{lsl-to-core-salt} f_2)) \\
\text{lsl-to-core-salt} (\exists f) &= (\text{next} \text{lsl-to-core-salt} f) \\
\text{lsl-to-core-salt} (\forall f) &= (\text{always} \text{lsl-to-core-salt} f) \\
\text{lsl-to-core-salt} (\exists f) &= (\text{eventually} \text{lsl-to-core-salt} f) \\
\text{lsl-to-core-salt} (f_1 U f_2) &= (\text{lsl-to-core-salt} f_1 \text{ until excl req} \text{lsl-to-core-salt} f_2) \\
\text{lsl-to-core-salt} (f_1 W f_2) &= (\text{lsl-to-core-salt} f_1 \text{ until incl weak} \text{lsl-to-core-salt} f_2) \\
\text{lsl-to-core-salt} (\text{req} f) &= (\text{lsl-to-core-salt} f_1 \text{ req} f_2) \\
\end{align*}
\]

Translation functions from core SALT to LTL and vice versa are inverse:

**lemma** \text{lsl-to-core-salt-to-ltl-equiv:} \text{:} \land.

\[
(s \models_{\text{lsl}} t \text{ core-salt-to-ltl} (\text{lsl-to-core-salt} f)) = (s \models_{\text{lsl}} t f)
\]

**lemma** \text{core-salt-to-ltl-to-core-salt-equiv:}

\[
(s \models_{\text{core-salt}} t \text{ core-salt-to-ltl} (\text{core-salt-to-ltl} f)) = (s \models_{\text{core-salt}} t f)
\]

Each core SALT property can be expressed in LTL:

**lemma** \text{core-salt-subset-ltl:}

\[
\forall (f': 'a \text{ core-salt-formula}). \text{ core-salt-proper-overlap} f \rightarrow \exists (f': 'a \text{ ltl-formula}). (s \models_{\text{core-salt}} t f) = (s \models_{\text{lsl}} t f')
\]

Each LTL property can be expressed in core SALT:

**lemma** \text{lsl-subset-core-salt:}

\[
\forall (f': 'a \text{ ltl-formula}). \exists (f': 'a \text{ core-salt-formula}). (s \models_{\text{lsl}} t f) = (s \models_{\text{core-salt}} t f')
\]

Core SALT and LTL have equivalent expressiveness, i.e., the sets of properties on system runs \(s\) for a given time point \(t\) expressible in core SALT (considering well-formed formulas) and in LTL are equal:

**theorem** \text{core-salt-ltl-equiv:}

\[
\{ p. \exists (f': 'a \text{ core-salt-formula}). \text{ core-salt-proper-overlap} f \land p s t = (s \models_{\text{core-salt}} t f) \} = \{ p. \exists (f': 'a \text{ ltl-formula}). p s t = (s \models_{\text{lsl}} t f) \}
\]

References


[BLS06] Andreas Bauer, Martin Leucker, and Jonathan Streit. SALT – Structured Assertion Language for Temporal Logic. In Zhiming Liu and Jifeng He, editors, Formal Methods and Software


[SAL] SALT Compiler. [http://salt.in.tum.de/](http://salt.in.tum.de/)

