Structured **A**ssertion **L**anguage for **T**emporal Logic

Language Reference and Compiler Manual

Version 1.0.1 — April 2006

Jonathan Streit
Institut für Informatik
Technische Universität München
http://salt.in.tum.de

Contents

1 General Information **1**
- 1.1 The Salt language . . . 1
- 1.2 Licensing and Contact . . . 1
- 1.3 Typographical conventions . . . 2
- 1.4 Salt version history . . . 2

2 Installation **3**

3 Usage **5**
- 3.1 Command line parameters . . . 5
- 3.2 Extending the Salt compiler . . . 7

4 Getting started with Salt **8**
- 4.1 First steps . . . 8
- 4.2 Salt for Ltl users . . . 9
- 4.3 Scope operators . . . 10
- 4.4 Regular expressions . . . 10
- 4.5 Exception operators . . . 11
- 4.6 Macros . . . 11
- 4.7 Iteration operators . . . 11
- 4.8 Further steps . . . 12

5 Salt Language Reference **13**
- 5.1 General structure . . . 13
- 5.2 Propositional layer . . . 15
- 5.3 Temporal layer . . . 17
  - 5.3.1 Simple temporal operators . . . 17
  - 5.3.2 Scope operators . . . 20
  - 5.3.3 Exception operators . . . 24
  - 5.3.4 Regular expressions . . . 25
  - 5.3.5 Counting quantifiers . . . 27
  - 5.3.6 Past operators . . . 28
- 5.4 Timed layer . . . 30
- 5.5 Macros and parameterised expressions . . . 33
  - 5.5.1 Parameterised expressions . . . 33
  - 5.5.2 Macros . . . 33
  - 5.5.3 Iteration . . . 35

6 Translation schema **37**
6.1 Replacement of non-core Salt operators . . . 38
6.1.1 **never** . . . 38
6.1.2 **releases** . . . 38
6.1.3 **nextn** . . . 38
6.1.4 **occurring** . . . 39
6.1.5 **holding** . . . 39
6.1.6 Regular expressions, part I . . . 40
6.1.7 Iteration operators . . . 40
6.2 Translation of core Salt into Salt-- . . . 41
6.2.1 **until** . . . 41
6.2.2 **upto** . . . 42
6.2.3 **from** . . . 43
6.2.4 **between** . . . 43
6.2.5 Exception operators . . . 43
6.2.6 Regular expressions, part II . . . 43
6.3 Translation of Salt-- into Ltl . . . 44
6.3.1 acc . . . 44
6.3.2 rej . . . 45
6.3.3 $\text{stop}_{\text{incl}}$ . . . 45
6.3.4 $\text{stop}_{\text{excl}}$ . . . 46
6.4 Optimisation . . . 46
6.5 Operator replacement . . . 46
6.6 Translation of timed operators . . . 47
6.6.1 Timed Salt into timed Salt-- . . . 47
6.6.2 Timed Salt-- into extended Tltl . . . 48
6.6.3 Extended Tltl into pure Tltl . . . 48

7 Examples **49**

Bibliography **52**

Index **53**

Chapter 1

General Information

1.1 The Salt language

Salt (Structured Assertion Language for Temporal Logic) is a high-level temporal specification language designed for the comfortable creation of concise specifications to be used in model checking and runtime verification. Unlike other specification languages, Salt does not target a specific domain.

Besides the common temporal operators, Salt provides exception operators, counting quantifiers and support for simplified regular expressions, as well as scope operators, allowing to express that a property has to hold before, after or in between some events. Frequently occurring patterns can be defined as parameterisable macros and can be used in a similar way as operators of the language. A timed extension allows to express real-time constraints.

In contrast to many proprietary specification languages, Salt can be translated into Ltl (Linear Temporal Logic)—or in the case of real-time properties into Tltl—and thus be used as a front end to existing verification tools. The Salt compiler generates optimised formulae, that are usually at least as efficient as hand-written ones, often even better.

1.2 Licensing and Contact

The Salt language and compiler are Open Source Software released under the terms of the GNU GPL. The full license text can be found in the file `LICENSE`.

Developers: Jonathan Streit

Contributions from: Andreas Bauer, Martin Leucker

Example specifications from: Matthew B. Dwyer, George S. Avrunin, James C. Corbett, Laura Dillon, Leonid Kof

Third party software used: ANTLR parser generator by Terence Parr

Contact. The authors can be contacted at
`salt AT mailbroy.informatik.tu-muenchen.de`.
Feedback—positive as well as critical—is greatly appreciated.

The SALT compiler can be downloaded from `http://salt.in.tum.de` as binary or source distribution. There is also a web interface available that allows to translate a SALT specification without having to install the compiler.

Additional information on the development and theoretical background of SALT can be found in [Str06, BLS06].

1.3 Typographical conventions

In this manual, SALT specifications are written in typewriter style with bold keywords (e. g., `variable`, **`until`**), while mathematical style and symbols (e. g., $\vee$, U) are used for LTL expressions. Placeholders for boolean propositions are denoted with italic lower case letters (e. g., a, b). Temporal formulae are denoted with Greek letters (e. g., φ, ψ).

Product names, registered names and trademarks may appear in this manual without being marked as such. This does not imply that they can be freely used.

1.4 SALT version history

0.7 Preliminary version

1.0 Revised manual, AUTOFOCUS support

1.0.1 Bugfix: additional parentheses in output needed for Cadence SMV

Chapter 2

Installation

This chapter describes how to install the SALT command line compiler. Most of its features can also be accessed using the web interface on the SALT homepage, without having to install the compiler.

The SALT compiler can be used on Windows, Linux and Unix systems. The following software is needed in order to install and run the compiler:

- The SALT compiler binaries. Binaries as well as the corresponding source code can be downloaded from `http://salt.in.tum.de`. The same binaries can be used for all platforms. There is no need for the average user to build the SALT compiler themselves, although this is possible with the source distribution.
- A Java Runtime Environment. The SALT compiler works with JRE version 1.3 or higher (preferably 1.5). The latest JRE can be downloaded from `http://www.java.sun.com`.
- A Haskell interpreter. The SALT compiler works with the Hugs 98 Haskell interpreter as well as with the Glasgow Haskell Compiler GHC, although Hugs is preferable as it is smaller and faster. The SALT compiler has been tested with Hugs November 2003 and GHC 6.4.1 on Windows 2000 and SuSE Linux. Hugs 98 can be downloaded from `www.haskell.org/hugs`. GHC can be downloaded from `www.haskell.org/ghc`.

Installation procedure

1. Install the Java Runtime Environment unless you already have it installed. Set the environment variable `JAVA_HOME` so that it points to the directory where Java is installed (which normally contains a subdirectory called `bin` with the executable `java`). See your operation system manual for information on how to set environment variables.
2. Install the Haskell interpreter of your choice unless you already have it installed.
3. The SALT compiler comes as a zip archive. Unzip the archive to the directory of your choice. This directory will be referred to as the SALT home directory. On a Linux or Unix system, you may have to make the file `salt.sh` executable by running the command `chmod +x salt.sh`.

4. Rename the file `SALT_HOME/config/hs.properties.template` to `SALT_HOME/config/hs.properties` and open it with a text editor to edit the following values:

   - `hs.interpreter` must be set to either `hugs` or `ghc`, depending on the Haskell interpreter to be used.
   - `hs.tempfile` may optionally be set. The file named here will be used for the intermediate Haskell code. This option may help if your system-wide temp directory's name contains spaces and Hugs is unable to find the intermediate code.
   - `hugs.path` must be set if you want to use Hugs. It must point to the directory where Hugs is installed. Hugs on Windows seems to have problems with spaces in the directory name, so you may have to use the 8-letter DOS name.
   - `hugs.command` must be set if you want to use Hugs. It is the name of the executable to be used (`runhugs.exe` for Windows and `runhugs` for Linux). `hugs.command` is interpreted relative to `hugs.path`.
   - `hugs.librarypath` must be set if you want to use Hugs. It is the name of the directory (relative to `hugs.path`) where the Hugs library files are located, usually called `libraries`.
   - `hugs.options` may optionally be set if you want to use Hugs. These are additional options that can be passed to Hugs.
   - `ghc.path` must be set if you want to use GHC. It must point to the directory where the GHC binaries are located (normally a directory `bin` relative to where GHC is installed).
   - `ghc.command` must be set if you want to use GHC. It is the name of the executable to be used (`runghc.exe` for Windows and `runghc` for Linux). `ghc.command` is interpreted relative to `ghc.path`.
   - `ghc.options` may optionally be set if you want to use GHC. These are additional options that can be passed to GHC.

5. If you want to call the SALT compiler from a directory different than the SALT home directory, you have to set the environment variable `SALT_HOME` accordingly.

6. Typing `salt.bat -f "assert always a"` in a DOS box or `./salt.sh -f "assert always a"` in a Linux shell should output the following:
   ```
   LTLSPEC G a
   ```

Chapter 3

Usage

This chapter describes the usage of the SALT compiler command line tool. It is called via the shell scripts `salt.bat` (for Windows) and `salt.sh` (for Linux). The compiler assumes that you either call it from its home directory or that you have set the environment variable `SALT_HOME`.

3.1 Command line parameters

The following parameters can be provided to the compiler:

- *file*
  Sets the input file to be read. The SALT specification will be read from the given file. If no input file is provided, the specification is read from standard in (normally the console). This allows the usage of Unix pipes.
- `-f "`*spec*`"`
  Processes a specification from the command line. This option allows to translate small specifications provided on the command line.
- `-o` *file*
  Sets the output file to be used. The result is written to the given file. If this option is not present, the result will be written to standard out (normally the console). This allows the usage of Unix pipes.
- `-e`
  Switches to embedded SALT mode. This is useful when a SALT specification forms a part of another file, for example an SMV model. The compiler searches the input for SALT specifications delimited by `BEGINSALT` and `ENDSALT`. The rest of the file is copied to the output, with the SALT specifications replaced by the resulting LTL formulae. See chapter 4 for an example.
- `-parser` *module*
  Enables a custom proposition parser plugin. The proposition parser is called to check and/or transform atomic propositions. For information on how to provide a custom implementation see section 3.2.

- `-smv`
  Chooses SMV syntax for the output (default). SMV syntax uses `! & | -> <->` for boolean and `G F U V X H O S T Y Z` for temporal operators. It is understood by SMV model checkers.

- `-spin`
  Chooses SPIN output syntax. SPIN output syntax uses `! && || -> <->` for boolean and `[] <> U V X` for temporal operators. It is understood by the SPIN model checker. SPIN does not allow past operators in LTL formulae.

- `-latex`
  Chooses LaTeX output syntax. LaTeX syntax allows to include LTL formulae easily into LaTeX documents. The packages `amsmath` and `amssymb` have to be included.

- `-printer` *module*
  Enables a custom printing function plugin. The printing function is used to print the final LTL formula in the desired output syntax. For information on how to provide a custom implementation see section 3.2.

- `-ltl`
  Chooses LTL generation (default). The result will be an LTL formula.

- `-rltl`
  Chooses intermediate SALT-- generation. The result will be a formula that contains rej , acc and stop operators in addition to the standard LTL operators. This option exists mainly for troubleshooting.

- `-hs`
  Chooses intermediate Haskell code generation. The result will be a Haskell program. The Haskell interpreter is not invoked. This option exists mainly for troubleshooting.

- `-tltl`
  Chooses TLTL for timed operators (default). Timed SALT operators will be translated using an event predicting (`|>`) and event recording (`<|`) operator, as defined in State Clock Logic [RS99].

- `-xtltl`
  Chooses extended TLTL for timed operators. In addition to the event predicting and event recording operator, timed U, W, □ and ◊ as well as the corresponding past operators will be used in the result. Extended TLTL is much easier to read than pure TLTL.

- `-notimed`
  Don't allow timed operators. The use of timed operators will produce an error message.

- `-nopast`
  Don't allow past operators. The use of past operators will produce an error message. Note that past operators are not allowed anyway in SPIN output syntax.

- `-nonext`
  Don't allow next operators. The use of the **`next`** operator as well as of other SALT operators that are translated using the **`next`** operator (e. g., regular expressions) produces an error message. This ensures that the formula is stutter-invariant.

- `-v`
  Choose verbose mode. The compiler will output some more status messages.

- `-h` or `-?`
  Show help screen.

3.2 Extending the SALT compiler

Plugins. The SALT compiler can be extended via a plugin mechanism. Plugins are Haskell modules stored in the directory `hs`. They have to be enabled with a command line parameter.

Proposition parsing plugins allow to perform checks or transformations on the atomic propositions that are used in a SALT specification. The default proposition parser checks that if a **`declare`**-statement is present, all propositions used in the specification are listed. For custom implementations, copy and modify the file `hs/PropositionCheck.hs`. Custom implementations are enabled using the command line parameter `-parser`. All implementations have to define a function `parseProposition`. Custom proposition parsing is particularly interesting for quoted propositions that may contain arbitrary text. For example, a custom proposition parser could check whether the atomic propositions are valid Java boolean expressions.

Printing function plugins allow to define a custom output syntax for LTL formulae. The default implementations `hs/LTL2`*xxx*`.hs` provide SMV, SPIN and LaTeX syntax. For custom implementations, copy and modify the file `hs/LTL2SMV.hs`. Custom implementations are enabled using the command line parameter `-printer`. All implementations have to define a function `printLTL`.

Further modifications. In order to extend the compiler beyond the plugin mechanism, you have to download the SALT source distribution and use ANT with the file `build.xml` to build it. JUnit is required for the compiler self-tests.

Chapter 4

Getting started with SALT

This chapter provides a short tutorial helping you to learn SALT. You should however have some basic knowledge of temporal logic. In chapter 5, you will find a detailed language reference. Chapter 7 contains some example specifications. See the index if you want to know about a specific operator.

This chapter assumes that you have the SALT compiler installed, as described in chapter 2, or that you have access to the SALT web interface.

4.1 First steps

We start with a very simple specification. Imagine some kind of client-server constellation, where we want to specify that every request is eventually answered. The SALT specification for this is

```
assert always (request implies eventually answer)
```

The keyword **assert** starts an assertion. There can be more than one assertion in a SALT specification, and each of them is translated into a formula of its own. **always**, **implies** and **eventually** are keywords. Their names should make clear what their meaning is. `request` and `answer` represent two boolean variables in the model to be checked. Any identifier that is not a keyword is automatically interpreted as a boolean variable by the compiler.

When we run the compiler on the specification, we obtain

```
LTLSPEC G (request -> (F answer))
```

which corresponds to the LTL formula

$$\Box(request \rightarrow \Diamond answer)$$

The compiler is by default set to SMV output syntax, and therefore uses `G` and `F` for $\Box$ and $\Diamond$ and begins each formula with the keyword `LTLSPEC`.

If we prefer the SPIN model checker instead, we have to call the compiler with the option `-spin` and obtain

```
[] (request -> (<> answer))
```

We can also write SALT specifications as embedded part of another file, like the following SMV file:

```
MODULE main
VAR
erroroccured : boolean;
ASSIGN
init(erroroccured) := ...

BEGINSALT
  assert never erroroccured
ENDSALT
```

The model checker can then be invoked using a piping command like `salt -e test.salt | nuSMV`.

4.2 SALT for LTL users

We have seen in the previous example that a SALT expression has a similar structure as an LTL formula. Experienced LTL users probably want to know how to denote the common LTL operators in SALT. Here they are:

LTL	SALT	
$\neg$	`!`	or **`not`**
$\wedge$	`&`	or **`and`**
$\vee$	`|`	or **`or`**
$\rightarrow$	`->`	or **`implies`**
$\leftrightarrow$	`<->`	or **`equals`**
U	**`until`**	
W	**`until weak`**	
R	**`releases`**	
$\Box$	**`always`**	
$\Diamond$	**`eventually`**	
$\circ$	**`next`**	
S	**`since`**	or **`untilinpast`**
W	**`since weak`**	or **`untilinpast weak`**
T	**`triggered`**	or **`releasesinpast`**
$\blacksquare$	**`historically`**	or **`alwaysinpast`**
$\blacklozenge$	**`once`**	or **`eventuallyinpast`**
$\bullet$	**`previous`**	or **`nextinpast`**
$\bullet_W$	**`previous weak`**	or **`nextinpast weak`**

The difference between symbolic and textual boolean operators (e. g., `|` and **`or`**) is that the symbolic operators have a higher precedence. Furthermore, unary operators have a higher precedence than binary operators. If you do not want to care about operator precedences, just set enough parentheses to avoid any ambiguity. It is a good idea to use symbolic operators to create purely propositional formulae and textual operators to combine temporal expressions.

The following three expressions all have the same meaning:

```
assert always a | b or eventually c | d
assert always (a or b) or eventually (c or d)
assert (always a | b) | (eventually c | d)
```

4.3 Scope operators

Until now, all we have done is to define a different syntax for LTL. Let's benefit a bit more from using SALT. Assume we want to specify that a program returns a result before terminating. We can do this by writing

```
assert (eventually result) before term
```

However, we will get an error message from the compiler saying

```
ERROR:
Operator must be used with inclusive/exclusive and
   required, optional or weak at line 1:30
```

Our specification is ambiguous. For example, it is not clear what happens if `result` and `term` become true at the same time. Also, the specifications does not clarify what is expected if the program never sends a `term`. When writing temporal specifications, one often forgets these special cases. In order to avoid erroneous specifications, the compiler requires us to specify exactly what we mean. We add the keyword **`exclusive`** to emphasise that the step when `term` becomes true does not any more belong to the denoted interval and that therefore `result` has to become true before that step. We also add the keyword **`required`** that states that `term` has to occur at some point. If this seems annoying to you, think of the time you might have needed to find out that your specification was expressing the wrong requirement (and not that your model was wrong).

The correct specification looks like this:

```
assert (eventually result) before exclusive required term
```

The keywords **`inclusive`**, **`optional`** or **`weak`** would have lead to a different meaning. Note that there are also other scope operators, like **`from`** and **`between`**.

4.4 Regular expressions

Specifying sequences of consecutive events in LTL requires a lot of nested $\wedge\bigcirc(\dots)$ or $\wedge\Diamond(\dots)$. SALT allows to describe such sequences in a concise way: by regular expressions. In the following, we again specify the data flow between a client and a server, this time a bit more in detail. The request consists of a begin signal, followed (in the next step) by an optional header and one or more data signals, and finally an end signal. The answer consists of a begin signal, any number of data signals and an end signal.

```
assert always ( /request_begin;
                 request_header?;
                 request_data+;
                 request_end /
               implies eventually
                /answer_begin;
                 answer_data*;
                 answer_end / )
```

Consecutive expressions are separated by the concatenation operator `;`. The `+` operator states that an element is repeated one or more times, while the `*` operator allows any number of repetitions. Note that the specification given above does *not* prevent `request_begin` or any other of the signals from occurring again at a step where they are not named explicitly. For example, `request_begin` might remain true throughout the whole communication.

More operators and details about the usage of regular expressions can be found in the language reference. In particular, some restrictions have to be taken into account when using the `*` operator.

4.5 Exception operators

Let's take the example a little further. Imagine that the communication can be aborted by the client at any moment by sending a reset signal. The implication for our specification is that it must be satisfied by any communication that begins correctly and is then interrupted by the reset signal. In SALT, we can use exception operators to express this:

```
assert always (( /request_begin;
                  ...
                  answer_end /
                ) accepton request_reset)
```

4.6 Macros

Have a look again at the first specification in this tutorial. It states that each request to a server is eventually answered. However, the **`implies eventually`** does not really tell us in an intuitive way what behaviour it expects. We therefore extract it into a macro definition that encapsulates the expression "is answered by":

```
define answeredby(x, y) := x implies eventually y
assert always (request answeredby answer)
```

Note how we can use our macro like the predefined operators in infix notation.

4.7 Iteration operators

Specifications often contain similar requirements for a set of signals or variables. In order to deal with this, SALT allows to instantiate a parameterised expression with a list of concrete values and to combine the resulting expressions in a certain way. For example, the following specification states that eventually all four processes must send a termination signal `p_`*i*`_finished`.

```
assert allof enumerate[1..4] as i in
         eventually p_$i$_finished
```

4.8 Further steps

You have reached the end of this short tutorial. You might start to write some specifications of your own now, or have a look at the example specifications in chapter 7. There are also various operators in SALT that are not covered by this tutorial. You can read more about them in chapter 5.

Chapter 5

SALT Language Reference

This chapter describes the SALT language in detail. Formal semantics are defined in chapter 6 via a translation schema. See chapter 4 for a tutorial if you want to learn SALT and chapter 7 for some example specifications.

5.1 General structure

Assertions

A SALT specification contains one or many assertions. An assertion formulates a requirement that is expected to be satisfied by the system under scrutiny. Each assertion is translated into a separate formula, which can then be used in a model checker or another verification tool.

Syntax

The syntax of SALT is described in this manual as an EBNF grammar, meant to provide an overview of the syntactical structure. An actual parsing grammar might be more complicated, as for example operator precedences have to be respected.

For better readability, the grammar has been separated into several fragments related to different concepts of the language. Each of the sections in this chapter contains one fragment of the grammar.

```
<specification> ::= ( <variable_declaration> )*
                    ( <macro_definition> )*
                    ( <assertion> )+

<assertion> ::= 'assert' <expression>

<expression> ::= '(' <expression> ')'
               | <propositional_expression>
               | <temporal_expression>
               | <timed_expression>
```

Figure 5.1: SALT syntax: general structure

Comments in SALT start with `--` and end at the end of the line. Keywords and identifiers are case-sensitive. The following operator precedences apply (from high to low):

1. `( )` parentheses
2. `!` symbolic unary boolean operators
3. `& | -> <->` symbolic binary boolean operators
4. `* + ?` repetition operators
5. `; :` sequence operators
6. prefix macro calls, unary built-in operators like **`always`** or **`next`** and modifiers like **`optional`**
7. infix macro calls and binary built-in operators like **`until`** or **`and`**
8. **`if`**-**`then`**, **`if`**-**`then`**-**`else`** and iteration operators

Layers

The SALT language consists of three layers:

- The *propositional layer* deals with atomic boolean propositions and boolean operators.
- The *temporal layer* encapsulates the main features of the SALT language that reason about temporal behaviour. It is divided into a future fragment and a symmetrical past fragment.
- The *timed layer* adds real-time constraints to the language. Similar to the temporal layer, it is divided into a future and a past fragment.

Within each layer, parameterised macros can be defined and instantiated. Iteration operators allow the instantiation of parameterised expressions for a set of concrete values.

The kind of formula that is generated from a SALT specification depends on the layers that are used in it. If only operators from the propositional layer are employed, the resulting formulae are propositional formulae. If only operators from the temporal and the propositional layer are employed, the resulting formulae are LTL formulae. If the timed layer is used, the resulting formulae are TLTL formulae. The resulting formulae are pure future LTL/TLTL formulae if only operators from the future fragments are employed, and LTL/TLTL+past formulae if past operators are used.

5.2 Propositional layer

The propositional layer deals with atomic boolean propositions and boolean operators. All boolean operators can however also be used to combine temporal expressions.

```
<variable_declaration> ::= 'declare' <identifier> ( ',' <identifier> )*

<identifier> ::= ('a'..'z' | 'A'..'Z' | '_')
                 ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*

<propositional_expression> ::=
                  <expression> <binary_bool_operator> <expression>
                | <unary_bool_operator> <expression>
                | 'if' <expression> 'then' <expression>
                  [ 'else' <expression> ]
                | <atomic_proposition>
                | <constant>

<binary_bool_operator> ::= '&' | '|' | '->' | '<->'
                         | 'and' | 'or' | 'implies' | 'equals'

<unary_bool_operator> ::= '!' | 'not'

<atomic_proposition> ::= <identifier>
                       | ('a'..'z' | 'A'..'Z' | '_' |
                          <parameter_reference>)
                         ('a'..'z' | 'A'..'Z' | '_' | '0'..'9' |
                          <parameter_reference>)*
                       | '"' (* | <parameter_reference>)* '"'

<parameter_reference> ::= '$' <identifier> '$'

<constant> ::= 'true' | 'false'
```

Figure 5.2: SALT syntax: propositional layer

Simple boolean variables. Boolean variables are the simplest atomic propositions from which SALT expressions can be built. Every identifier that was not defined as a macro or a formal parameter is treated as a boolean variable. This means that it appears in the output as it has been written in the specification.

The boolean literals are denoted as **`true`** and **`false`**.

Quoted boolean propositions. Additionally, arbitrary strings between `""` can be employed as atomic propositions. This allows the use of predicates like `state==START` or even Java expressions that can be interpreted by another tool (e. g., the model checker or the runtime monitor) in the processing chain. The text between `""` appears unchanged in the output (a `"` or `$` inside the quoted proposition must be escaped with \).

Quoted propositions make the SALT language independent from the domain it is used in and the models and tools that influence the rest of the verification process. In order to make the use of complex propositions more reliable, custom proposition parser plugins can be provided to check and/or transform atomic propositions that appear in a SALT specification (see section 3.2).

Example:

```
assert condition & "state==START"
```

yields the output

```
LTLSPEC condition & state==START
```

Parameterised propositions. Inside a boolean proposition, function parameters or iteration variables may appear between `$$`. During translation they are replaced by the value of the parameter. See section 5.5 for details on parameterised expressions.

Example:

```
define isok(process) :=
      $process$_started & !$process$_error
assert isok("main")
```

yields the output

```
LTLSPEC main_started & !main_error
```

Explicit declaration of atomic propositions. By default, declaring boolean variables is not mandatory in SALT. It is however possible to do so using the **`declare`** keyword. If at least one declaration appears in the specification, all atomic propositions have to be declared explicitly and the compiler issues an error message for atomic propositions that are used in the specification but not listed in the declaration. This allows to detect typos rapidly.

Custom proposition parser plugins can define their own behaviour for proposition checking (see section 3.2).

Boolean operators. The usual semantics apply for the boolean operators `| & ! -> <->` (logical or, and, not, implication and equivalence).
The alternative notations **`and`**, **`or`**, **`not`**, **`equals`** and **`implies`** have the same meaning, but the operator precedence of a macro call.

`if` φ **`then`** ψ **`else`** ρ expresses that ψ must hold if φ holds, and that ρ (if present) must hold if φ does not hold. The advantage of **`if`**-**`then`**-**`else`** over `->` is that it helps to write specifications in a more natural way, because it makes clear to the reader that the first expression is a condition. Nested **`if`**-**`then`**-**`else`** have to be enclosed in parentheses in order to clarify to which **`if`** an **`else`** belongs.

5.3 Temporal layer

The temporal layer is the heart of the SALT language. It allows to express temporal properties by combining propositional expressions with temporal operators. The temporal layer consists of a future fragment and a completely symmetrical past fragment. For the sake of brevity, this section presents only future operators explicitly. The corresponding past operators are introduced in 5.3.6.

5.3.1 Simple temporal operators

The common LTL temporal operators can be used in SALT specifications:

- **`always`** φ
  states that φ must hold forever from now on, including in the current step.
- **`never`** φ
  states that φ must never hold from now on, including in the current step.
- **`eventually`** φ
  states that φ must hold at some time in the future or at the current step.
- **`next`** φ
  states that φ must hold in the next step. When used inside an **`upto`** statement, **`next`** acts as a strong operator, i. e., even **`next true`** does not hold if there is no next step. **`next weak`** is the corresponding weak operator.
- φ **`until`** ψ
  states that ψ must eventually hold and that φ must hold from now on until this step.
- φ **`until weak`** ψ
  states that either φ must hold forever from now on, or that ψ must eventually hold and that φ must hold from now on until this step.
- φ **`releases`** ψ
  states that either ψ must hold forever from now on, or that φ must eventually hold and that ψ must hold from now on until and during this step.

Extended `until`

Besides the two well-known versions of **`until`**, SALT provides some more:

- φ **`until exclusive required`** ψ
  is the same as φ **`until`** ψ.
- φ **`until exclusive optional`** ψ
  states that if ψ eventually holds, φ must hold from now on until this step. Nothing is required if ψ never holds.
- φ **`until exclusive weak`** ψ
  is the same as φ **`until weak`** ψ.

```
<temporal_expression> ::=
         <modifier>* <expression> <ternary_temp_operator>
         <modifier>* <expression> ',' <modifier>* <expression>
       | <modifier>* <expression> <binary_temp_operator>
         <modifier>* <expression>
       | <unary_temp_operator> <modifier>* <expression>
       | <quantified_temp_operator> <range> <expression>
       | <regular_expression>

<modifier> ::= 'required' | 'req'
             | 'optional' | 'opt'
             | 'weak'
             | 'exclusive' | 'excl'
             | 'inclusive' | 'incl'

<ternary_temp_operator> ::= 'between' | 'betweeninpast'

<binary_temp_operator> ::= 'until' | 'untilinpast' | 'since'
                         | 'releases' | 'releasesinpast' | 'triggered'
                         | 'upto' | 'before' | 'uptoinpast'
                         | 'from' | 'after' | 'frominpast'
                         | 'rejecton' | 'accepton'

<unary_temp_operator> ::= 'always' | 'alwaysinpast' | 'historically'
                        | 'never' | 'neverinpast'
                        | 'eventually' | 'eventuallyinpast' | 'once'
                        | 'next' | 'nextinpast' | 'previous'

<quantified_temp_operator> ::= 'nextn' | 'nextninpast' | 'previousn'
                             | 'occurring' | 'occurringinpast'
                             | 'holding' | 'holdinginpast'

<regular_expression> ::=
         '/ ' [<expression>] [<repetition_operator>]
              ( <sequence_operator>
                [<expression>] [<repetition_operator>] )* '/ '
       | '\ ' [<expression>] [<repetition_operator>]
              ( <sequence_operator>
                [<expression>] [<repetition_operator>] )* '\ '

<repetition_operator> ::= '?'
                        | '*' [ <range> ]
                        | '+'

<sequence_operator> ::= ';'
                      | ':'

<range> ::= '[' ('=' | '>' | '<' | '>=' | '<=') <number> ']'
          | '[' <number> '..' <number> ']'
          | '[' <number> ']'

<number> ::= ('1'..'9') ('0'..'9')*
           | '0'
```

Figure 5.3: SALT syntax: temporal layer

- φ **until inclusive required** ψ
  states that ψ must eventually hold and that φ must hold from now on until and during this step.
- φ **until inclusive optional** ψ
  states that if ψ eventually holds, φ must hold from now on until and during this step. Nothing is required if ψ never holds.
- φ **until inclusive weak** ψ
  is the same as ψ **releases** φ.

The abbreviations **req**, **opt**, **incl** and **excl** may be used instead of the long keywords.

Note that **inclusive**/**exclusive** has nothing to do with the strict or non-strict **until** operators that can be defined in LTL: strictness refers to whether the present state (i. e., the left end of the interval where φ is required to hold) is included or not in the evaluation, while **inclusive**/**exclusive** defines whether φ has to hold in the state where ψ occurs (i. e., the right end of the interval). Strict SALT operators can be created by adding a preceding **next** operator.

Extended `next`

There is also an abbreviation for consecutive **next** operators:

- **nextn[**=n**]**φ or **nextn[**n**]**φ states that φ is required to hold n steps from now in the future.
- **nextn[**n`..`m**]**φ states that φ is required to hold at some time in the future, at least n steps from now and at most m steps from now (both inclusive).
- **nextn[**`>=`n**]**φ states that φ is required to hold eventually in the future, but at least n steps from now (inclusive).
- **nextn[**`<=`n**]**φ states that φ is required to hold eventually in the future, but at most n steps from now (inclusive).
- **nextn[**`>`n**]**φ, **nextn[**`<`n**]**φ similarly.

5.3.2 Scope operators

Scope operators allow to specify that an expression has to hold before, after or in between some events (represented by boolean propositions).

The `upto` operator

```
φ upto exclusive required b
φ upto exclusive optional b
φ upto exclusive weak b
required φ upto exclusive required b
required φ upto exclusive optional b
required φ upto exclusive weak b
weak φ upto exclusive required b
weak φ upto exclusive optional b
weak φ upto exclusive weak b
φ upto inclusive required b
φ upto inclusive optional b
φ upto inclusive weak b
```

The **`upto`** operator states that an expression φ must hold in the time before the first occurrence of a boolean end condition b. φ is evaluated in the current step, but considering only the time up to b. This means that for example **`always`** `x` is true if `x` holds at least until the occurrence `b`. It does not matter if `x` becomes false afterwards. See below for a more detailed explanation.

The alternative name **`before`** can be used instead of **`upto`**.

The `from` operator

```
φ from exclusive required a
φ from exclusive optional a
φ from inclusive required a
φ from inclusive optional a
```

The **`from`** operator states that an expression φ must hold in the time after the first occurrence of a start condition a.

The alternative name **`after`** can be used instead of **`from`**.

The `between` operator

The **`between`** operator is a combination of both **`from`** and **`upto`**: it states that an expression φ must hold in the time after a start condition a, but before an end condition b.

Start and end conditions

`upto`, **`from`** and **`between`** require the user to specify what behaviour is expected in a case where the condition does not occur at all. There are three possibilities, expressed by prefixing the condition with a modifier keyword.

- **required** b states that b is expected to hold at some time in the future. The whole expression evaluates to false if there is no occurrence of b.
- **optional** b states that φ shall be considered only if b eventually holds. The whole expression evaluates to true if there is no occurrence of b.
- **weak** b states that b is an end condition that may or may not hold in the future. φ is evaluated for the time until b, or for the whole sequence if b never holds. **weak** may only be used with **upto** or for the end condition of **between**.

Example:

Trace	always a upto excl req b	always a upto excl opt b	always a upto excl weak b
aaab...	true	true	true
aaaa...	false	true	true
----...	false	true	false
---b...	false	false	false

Occurrences of end condition b of a **between** statement before the first occurrence of start condition a are not taken into account. When **between** is used with the combination **optional**-**optional**, both conditions have to eventually hold in order for φ to be evaluated.

The abbreviations **req** and **opt** may be used instead of the long keywords.

Inclusive and exclusive semantics

upto can either be *exclusive* (the step when b occurs is not taken into account any more) or *inclusive* (the step when b occurs is the last step of the denoted interval). To specify which behaviour is meant, the condition b must be prefixed with the modifier keyword **exclusive** or **inclusive**. The same applies to the **from** operator: inclusive means that the step when a occurs is the step φ is evaluated. Exclusive means that φ is evaluated in the next step after a. The **between** operator requires its start and end condition to be prefixed by **inclusive** or **exclusive**. A **between** operator with an exclusive start condition looks for occurrences of the end condition only from the next step on, i. e., it ignores b if it occurs together with the start condition a. The abbreviations **incl** and **excl** may be used instead of the long keywords. Figure 5.4 provides a visualisation of inclusive/exclusive semantics.

Behaviour for empty time intervals

If the end condition of an exclusive **upto** or **between** holds immediately at the current step, it is not clear whether the whole expression should evaluate to true or false, as expressions can only be evaluated over non-empty intervals. For example, p **upto excl req** b could be true or false if p and b are both true. Depending on the immediate argument of the **upto** or **between**, the following rules apply in this case:

- φ **until** ψ evaluates to **false**, because **until** requires its end condition to eventually occur, and of course this cannot happen if the **upto**

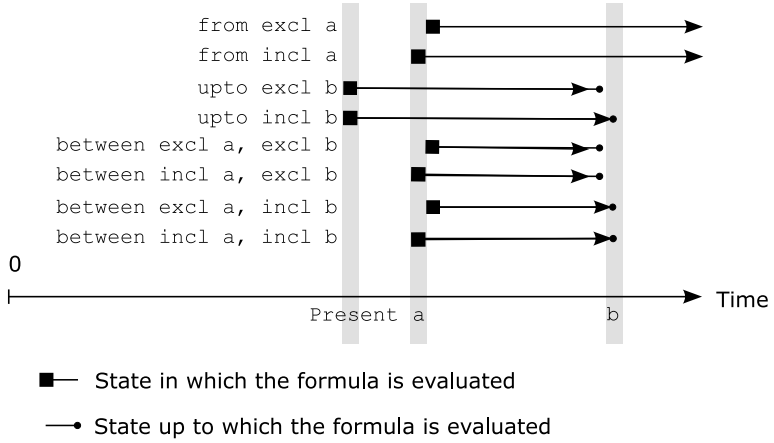


Figure 5.4: Inclusive and exclusive semantics of scope operators

ends immediately. The same applies to φ **`until excl req`** ψ and φ **`until incl req`** ψ.

- φ **`until weak`** ψ evaluates to **true**, because **`until weak`** allows the end condition to never occur, as long as the loop condition always holds. As there is no time for the loop condition to hold *not*, we can say that it holds forever. The same applies to φ **`releases`** ψ, φ **`until excl opt`** ψ, φ **`until incl opt`** ψ, φ **`until excl weak`** ψ and φ **`until incl weak`** ψ.
- **`always`** φ and **`never`** φ evaluate to **true**, for the same reasons that apply to **`until weak`**.
- **`eventually`** φ evaluates to **false**, as there is no time for φ to eventually occur.
- `! & | -> <->` have the usual boolean semantics. The arguments of the operator are required to recursively match one of the above patterns.
- **`weak`** φ evaluates to **true**. This is a possibility to specify explicitly what should happen in the case of an immediately occurring end condition.
- **`required`** φ evaluates to **false**. This is a possibility to specify explicitly what should happen in the case of an immediately occurring end condition. The abbreviation **`req`** may be used.
- all other φ are **illegal** as an argument to **`upto`** or **`between`** and produce an error during compilation.

These rules imply that some φ, like propositions, must be prefixed with either **`weak`** or **`required`** when used within an exclusive **`upto`**, while others, like **`always`**, may also be used without. None of this has to be considered when using an inclusive end condition, as this ensures that the time interval denoted is at least of length 1.

Details and special cases

All three operators may be freely nested.

Note that although **upto** and **from** seem to be very similar, there is a crucial difference: The **from** operator evaluates its argument from a certain step (where the condition a holds for the first time) on toward the future. The **upto** operator evaluates its argument at the current step, but on a future limited by the occurrence of the end condition b. See figure 5.4 for a visualisation.

Note also that neither of the operators contains an implicit **always**. In order to express that φ has to hold at *every* step before b, an explicit **always** has to be added to φ.

next ψ is always false when evaluated in the last step of the time interval delimited by the end condition b of an **upto**. Similarly, **next weak** ψ is always true in this situation.

Examples

```
assert always x upto inclusive required b
```

states that x must hold from now on until and including the step when b holds for the first time. b is required to eventually hold.

```
assert eventually y from exclusive optional b
```

states that if b becomes eventually true, y must hold at the following or any later step.

```
assert /x;y/ between incl opt a, excl opt b
```

states that x followed by y is expected to hold the step when a holds for the first time. If b holds during this or the next step, the expression evaluates to false, as the sequence /x;y/ could not be finished in time. If no a occurs, the whole expression is true. If no b occurs together with or after the first occurrence of a, the whole expression is also true (keyword **optional**), even if x and y do not show the desired behaviour.

```
assert always (weak (x->(eventually y)) upto excl weak b)
```

states that every x has to be followed by some y. This y must occur before b holds the next time. The first **weak** states that the **upto** expression shall be considered true if b occurs together with x. The **always** placed outside the **upto** makes that the expression is tested at any time in the future, even after the first occurrence of b. If it had been placed inside, any x after the first occurrence of b would not have been taken into account.

5.3.3 Exception operators

```
φ rejecton b
φ accepton b
```

The operators **rejecton** and **accepton** define an exception condition (also called abort condition) for a formula φ. The evaluation of φ stops when the condition occurs. If the exception condition never occurs, the operator is ignored. φ **rejecton** b rejects a formula φ (evaluates it to false) on occurrence of a condition b if φ has not been satisfied before. φ **accepton** b accepts a formula φ (evaluates it to true) on occurrence of a condition b if φ has not been violated before.

Examples:

```
assert (a until b) rejecton c
```

is true, if b occurs before c, and a is present until the occurrence of b. It is false if a becomes false before the occurrence of b, if b never occurs or if b occurs only together with or after c.

```
assert (a until b) accepton c
```

is true, if b occurs before c, and a is present until the occurrence of b. It is also true if a is present until the occurrence of c. It is false if a becomes false before the occurrence of b or c.

The semantics of **rejecton** and **accepton** have similarities to those of the **upto** operator (see 5.3.2). However, **rejecton** and **accepton** evaluate any pending formula to false resp. true on occurrence of the abort condition, while the result of an **upto** depends on the nature of φ.

Example:

Trace	always eventually a rejecton b	always eventually a accepton b	always eventually a upto excl weak b
ab...	false	true	true
-b...	false	true	false
aa...	true	true	true

rejecton and **accepton** do not have corresponding past operators (see 5.3.6). They influence both future and past operators in φ.

5.3.4 Regular expressions

SALT regular expressions (SRE) allow testing on complex patterns of conditions in a very concise way. They begin and end with a slash `/`. However, some restrictions have to be applied, as not every regular expression can be translated into LTL. The following table compares traditional regular expressions to SRE:

Traditional RE	**SRE**
terminal symbols	propositional formulae
. (concatenation)	`; :` (sequence operators)
$\cup$ (union)	`|` (or operator)
$*$ (Kleene star)	`* +` (repetition operators, with constraints)
? (optional part)	`?` (optional part)
$\neg$ (complement)	not implemented for efficiency reasons

Two sequence operators are available:

- p`;`q states that q must hold in the next step after p.
- p`:`q states that q must hold after p and overlap with p in one step. For two boolean propositions, this is equivalent to p`&`q.

Each element of a SRE can be suffixed with a *repetition operator*. The following repetition operators are available:

- p`*` states that p may hold an arbitrary number of consecutive steps from now on.
- p`*[=`n`]` or p`*[`n`]` states that p must hold during exactly n consecutive steps from now on.
- p`*[`n`..`m`]` states that p must hold during between n and m consecutive steps from now on.
- p`*[>`n`]` states that p must hold during more than n consecutive steps from now on.
- p`*[<`n`]` states that p must hold during less than n consecutive steps from now on (including the possibility that p does not hold at all).
- p`*[>=`n`]` states that p must hold during at least n consecutive steps from now on.
- p`*[<=`n`]` states that p must hold during at most n consecutive steps from now on (including the possibility that p does not hold at all).
- p`?` states that p may or may not hold (the same as p`*[<=1]`).
- p`+` states that p must hold during at least one step from now on (the same as p`*[>=1]`).

The SRE `/`a $*$ `[0]:`b`/` (empty sequence in combination with the `:` sequence operator) is equal to `/`b`/`. This implies that for instance the SRE `/`p`;`q`*:`r`/` is not satisfied by an occurrence of p and r at the same time.

Elements in an SRE may be left out, which is interpreted as **`true`**. For example, `/*;a/` is equivalent to `/`**`true`**`*;a/`, which is equivalent to **`eventually`** `a`.

As regular expressions can not be generally translated into LTL, a few additional rules have to be followed when composing SRE:

- The argument of `*`, `*[>`n`]`, `*[>=`n`]` and `+` may only be a purely boolean proposition. It may contain boolean operators like `&`, `|` or `!`.
- All expressions except for the last in an SRE must be either purely boolean propositions, or they must be other SRE combined by `|`. No other boolean operators are allowed for the combination of SRE (although they can be used to form boolean expressions).
- The last element in an SRE may be any SALT expression, however because of operator precedences it may be necessary to surround it with parentheses.

Examples of allowed sequences:

```
assert / a*: b; c /
assert / !a*; (always b) /
assert / /a/ | /b*/; c /
```

Examples of forbidden sequences:

```
assert / /a; b/*; c /
-- /a; b/ is not purely propositional
```

```
assert / !/a*/; b /
-- complement of reg. exp. /a*/ is not allowed
```

```
assert / /a*/ -> /b/; c /
-- -> can not be used to combine reg. exp.
```

Remember that an SRE does not state anything about conditions not named explicitly: `/a;b/` requires `a` to hold in the current and `b` to hold in the next step. It does not require `b` to be false in the current or `a` to be false in the next step, and therefore also matches for example a sequence where `a` and `b` hold all the time.

SRE match by default finite prefixes of a sequence. This implies that a trailing unbounded `*` operator is equivalent to true, because it includes the empty sequence, which is a prefix to any sequence. However, the last element of an SRE is allowed to be an arbitrary SALT expression, and can therefore also be for example `(always a)`, which does ensure that `a` is true until infinity.

5.3.5 Counting quantifiers

The counting quantifiers **occurring** and **holding** allow concise statements about conditions that have to hold a certain number of times. The difference between the two operators is that **holding** counts each step during which the condition holds separately, while **occurring** treats consecutive steps where the condition holds as one occurrence.

`occurring` operator

- **occurring**[=n]φ or **occurring**[n]φ states that φ occurs exactly n times in the future. An occurrence may last more than one step, and has to be separated from the next occurrence by a step where φ does not hold. The first occurrence may or may not begin immediately. After the last occurrence, φ must not hold again.
- **occurring**[n..m]φ states that φ occurs between n and m times in the future. After the last occurrence, φ must not hold again.
- **occurring**[>=n]φ states that φ occurs at least n times and is allowed to occur again afterwards.
- **occurring**[<=n]φ states that φ occurs at most n times and is not allowed to occur again afterwards.
- **occurring**[>n]φ, **occurring**[<n]φ similarly.

`holding` operator

- **holding**[=n]φ or **holding**[n]φ states that φ is required to hold during exactly n steps in the future. Those occurrences may or may not be separated from the next occurrence by a step where φ does not hold. The first occurrence may or may not begin immediately. After the last occurrence, φ must not hold again.
- **holding**[n..m]φ states that φ is required to hold during between n and m steps in the future. After the last occurrence, φ must not hold again.
- **holding**[>=n]φ states that φ is required to hold at least during n steps and is allowed to hold again afterwards.
- **holding**[<=n]φ states that φ is required to hold at most during n steps and is not allowed to hold again afterwards.
- **holding**[>n]φ, **holding**[<n]φ similarly.

5.3.6 Past operators

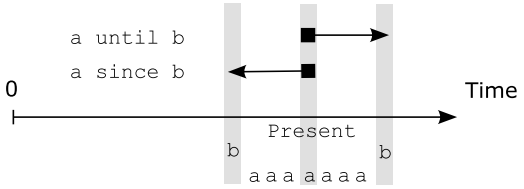
SALT supports past operators, that reason about past states instead of future ones. For every future operator there is a corresponding past operator, as shown in figure 5.5. Future operators are translated using only LTL future operators, and past operators are translated using only LTL past operators. This leaves the users the choice whether they do or do not want to use past operators.

Some operators have two names: there is always a generic name where the past operator has the name of the future operator with the suffix `inpast`. Additionally, there might be another more intuitive name like **`since`** or **`previous`**. The future operators **`from`** and **`upto`** have the alternative names **`after`** and **`before`**. Past regular expressions are written between `\ \` instead of `/ /`.

Future		Past	
`always`		**`historically`**	or **`alwaysinpast`**
`between`		**`betweeninpast`**	
`eventually`		**`once`**	or **`eventuallyinpast`**
`from`	or **`after`**	**`frominpast`**	
`holding`		**`holdinginpast`**	
`never`		**`neverinpast`**	
`next`		**`previous`**	or **`nextinpast`**
`next weak`		**`previous weak`**	or **`nextinpast weak`**
`nextn`		**`previousn`**	or **`nextninpast`**
`occurring`		**`occurringinpast`**	
`releases`		**`triggered`**	or **`releasesinpast`**
`until`		**`since`**	or **`untilinpast`**
`until weak`		**`since weak`**	or **`untilinpast weak`**
`upto`	or **`before`**	**`uptoinpast`**	

Figure 5.5: Overview of the future operators and their corresponding past operators

Reading direction of past operators. Past operators as defined in LTL bring with them an inherent pitfall connected to our understanding of time. Time always progresses from present to future, and we attribute intuitively the Western reading direction left-to-right to the progress of time from present to future. Past operators however are mirrored future operators: their direction is from present to past. While `a` **`until`** `b` steps forward on a sequence of `a` until hitting a `b`, the corresponding `a` **`since`** `b` steps backward on a sequence of `a` until hitting a `b`. This `b` occurs actually *before* the sequence of `a`, but when reading `a` **`since`** `b` from left to right the `b` appears *behind* the `a`. This can be a little confusing. When using past operators, imagine therefore always standing at the present point in time and facing backward, while reading the expression from left to right.



All SALT past operators follow this mirrored semantics, which is consistent with the definition of LTL past operators. This has the advantage of similar semantics for all past operators, as well as a similar parameter order for future and past operators. The drawback is, however, that some past operators have a meaning not expected at first sight. The most surprising case are probably regular expressions which, when read from left to right, have to be interpreted from present to past. The expression `\a;b;c\` matches a sequence where `a` is true in the present, `b` was true one step ago and `c` was true two steps ago. In other words, it matches the sequence `cba` when reaching the `a`.

Past operators and scope operators used together. Future scope operators do not limit past operators in their argument, i. e., a **`from`** does not contain an implicit **`uptoinpast`**. In the expression

```
assert (always x -> (once y)) from incl req a
```

the corresponding `y` is allowed to occur *before* the occurrence of `a`. In order to limit **`once`** `y` to the time after `a`, you have to write

```
assert (always x -> (once y uptoinpast incl req a))
    from incl req a
```

This expression however will look back for `y` only up to the first occurrence of `a` it can find (which might be different from the one that triggered **`from`**).

Past operators and exception operators used together. In contrast to **`from`** and **`upto`**, the operators **`rejecton`** and **`accepton`** influence both future and past operators. There are no separated versions for future and past.

5.4 Timed layer

SALT contains a timed extension that allows the specification of real-time constraints.

```
<expression> ::= <unary_timed_operator> <timing_constraint>
                 <expression>
               | <expression> <until_operator>
                 <timing_constraint> <expression>
               | <timing_constraint> <expression> <releases_operator>
                 <expression>

<unary_timed_operator> ::= 'next' | 'nextinpast' | 'previous' |
                           'always' | 'alwaysinpast' | 'historically' |
                           'never' | 'neverinpast' |
                           'eventually' | 'eventuallyinpast' | 'once'

<until_operator> ::= 'until' | 'untilinpast' | 'since'

<releases_operator> ::= 'releases' | 'releasesinpast' | 'triggered'

<timing_constraint> ::= 'timed' '['
                        ( '=' | '>' | '<' | '>=' | '<=' ) <float> ']'

<float> ::= <number> [ '.' ('0'..'9')+ ]
```

Figure 5.6: SALT syntax: timed layer

Timed SALT includes all features of untimed SALT as well as some timed operators. Timed operators are translated into a timed variant of LTL, here referred to as TLTL [D'S03]. TLTL adds the event predicting and event recording operators defined in [RS99]. The SALT compiler can produce either pure TLTL with $\triangleright_{\sim c}$ and $\triangleleft_{\sim c}$ as the only timed operators, or an extended TLTL with the additional timed operators $\mathrm{U}_{\sim c}$, $\mathrm{W}_{\sim c}$, $\Box_{\sim c}$ and $\Diamond_{\sim c}$ as well as the corresponding past operators. Extended TLTL is much easier to read than pure TLTL.

Timing constraints in SALT are expressed using the modifier **`timed`**`[`$\sim c$`]` that can be used together with several untimed SALT operators in order to make them timed operators. $\sim$ is one of `<`, `<=`, `=`, `>=`, `>` for **`next timed`** and either `<` or `<=` for all other timed operators.

- **`next timed`**`[`$\sim c$`]`φ
  states that the next occurrence of φ is within the time bounds $\sim c$. It is not taken into account if φ is true at the current step. **`next timed`**`[`$\sim c$`]`φ corresponds to the event predicting operator $\triangleright_{\sim c}\varphi$.

- φ **`until timed`**`[`$\sim c$`]` ψ
  states that φ is true until the next occurrence of ψ, and that this occurrence of ψ is within the time bounds $\sim c$. Occurrences of φ at the current step are accepted too. The extended variants of **`until`** (using **`required`**, **`optional`**, **`weak`**, **`inclusive`** and **`exclusive`**) can be used as timed operators as well.

- **`timed`**`[`$\sim c$`]` φ **`releases`** ψ
  states that ψ is true until and during the next occurrence of φ, if such

occurrence of φ is within the time bounds $\sim c$. Occurrences of φ at the current step are accepted too. If φ does not occur within the time bounds, ψ is required to hold during the whole specified time interval.

- **always timed**[$\sim c$] φ
  states that φ must be always true within the time bounds $\sim c$.
- **never timed**[$\sim c$] φ
  states that φ must be never true within the time bounds $\sim c$.
- **eventually timed**[$\sim c$] φ
  states that φ must be true at some point within the time bounds $\sim c$.

Past operators can be enriched in a similar way. Other SALT operators can not be combined with **timed[ ]**.

Examples:

```
assert always timed[<=3] p
```

states that p is true during the next 3 time units.

```
assert always (p -> (eventually timed[<3] q)
```

states that p is followed by q within less than 3 time units

```
assert p & (always (p -> (next timed[=1] p)))
```

states that p occurs periodically with a distance of 1 time unit.

Timed operators within upto and between. Timed operators within an **upto** statement have to be handled with care. Both the timing constraint and the **upto** specify an end condition, and it is not a priori clear what semantics they express when combined.

For example,

```
assert (always timed[<3] p) upto req excl b
```

could have three different meanings:

1. p has to hold during the next 3 time units or until the occurrence of b.
2. p has to hold during the next 3 time units and b is not allowed to occur during this time.
3. p has to hold during the next 3 time units regardless of whether b occurs.

Because of this ambiguity, the choice was made that **upto** and **between** do not influence timed operators and their arguments at all, i. e., the end condition is not woven into a timed sub-expression. This leaves the user full choice between the possible meanings, because they can (or rather must) manually add the desired constraints.

The SALT formulae yielding the correct semantics for the example from above are:

```
assert (always timed[<3] p or eventually timed[<3] b)
    upto req excl b
```

```
assert (always timed[<3] p and never timed[<3] b)
    upto req excl b
```

```
assert (always timed[<3] p)
    upto req excl b
```

5.5 Macros and parameterised expressions

SALT allows the definition of macros and parameterised expressions. This can help to make a specification easier to understand, because complex sub-formulae can be defined separately and accessed by a name. It also helps writing more concise specifications, because expressions that appear several times in a specification have to be written down only once. Iteration operators can be used to instantiate parameterised expressions for a list of concrete values.

5.5.1 Parameterised expressions

A parameterised expression is an expression that contains placeholders. Parameterised expressions allow to reuse a specification pattern for different concrete values.

Parameters can be used in two ways within an expression: First, they can be employed directly as part of an expression. Secondly, they can be used *within* an atomic proposition, e. g., as part of a variable name. This is expressed by referencing the parameter between `$$` in the atomic proposition (see section 5.2).

```
define mymacro1(x, y) := x implies eventually y
define mymacro2(x, y) := always input$x$ & input$y$
```

SALT parameters are expression-based and not—like for example C preprocessor macro parameters—character-based. Therefore, parameters always have to stand for complete expressions. It is not allowed (and hardly necessary) to use incomplete expressions like `a |` as a parameter. Parameters used within an atomic proposition should usually not be complex expressions.

5.5.2 Macros

Macro definitions. A macro definition starts with the keyword **`define`**, followed by the macro name and an optional parameter list in parentheses. The macro body appears after `:=`. All macros have to be defined before being used. Macro definitions have to appear before any assertion in the specification.

Simple macro calls. Macros can be accessed in four different ways:

- Without arguments. A macro that does not expect parameters can be accessed simply via its name.

  ```
  define any_a := a1 | a2 | a3 | a4
  assert always any_a
  ```

- As a prefix operator. A macro that expects exactly one parameter can be used as a prefix operator without the need to enclose the argument in parentheses, similar to the unary built-in operators like **`always`** or **`next`**.

  ```
  define stricteventually(x) := next eventually x
  assert stricteventually term
  ```

```
<macro_definition> ::= 'define' <identifier> [<formal_parameter_list>]
                       ':=' <expression>

<identifier> ::= ('a'..'z' | 'A'..'Z' | '_')
                 ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*

<formal_parameter_list> ::= '(' <formal_parameter>
                            (',' <formal_parameter>)* ')'

<formal_parameter> ::= <identifier>

<expression> ::= <macro_call>
               | <formal_parameter>
               | <iteration_expression>

<macro_call> ::= <nullary_call>
               | <prefix_call>
               | <infix_call>
               | <explicit_call>

<nullary_call> ::= <identifier>
                 | <formal_parameter>

<prefix_call> ::= <identifier> <actual_parameter>

<infix_call> ::= <actual_parameter> <identifier>
                 <actual_parameter_list>

<explicit_call> ::= <identifier>
                    '(' [ <actual_parameter_list> ] ')'
                  | <formal_parameter>
                    '(' [ <actual_parameter_list> ] ')'

<actual_parameter_list> ::= <actual_parameter> (','
    <actual_parameter>)*

<actual_parameter> ::= <expression>
                     | '@' <identifier>

<iteration_expression> ::= <iteration_operator> <list_creation>
                           'as' <identifier> 'in' <expression>

<iteration_operator> ::= 'allof' | 'someof' | 'noneof' | 'exactlyoneof'

<list_creation> ::= 'list' '[' [<expression> (',' <expression>)* ] ']'
                        ( ( 'with' | 'without' ) <expression> )*
                  | 'enumerate' <range>
                        ( ( 'with' | 'without' ) <expression> )*
```

Figure 5.7: SALT syntax: macros and parameterised expressions

- As an infix operator. A macro that expects two or more parameters can be used as an infix operator. The macro name appears after the first argument. If the macro expects more than two parameters, the remaining arguments are separated by commas. This is similar to the built-in binary and ternary operators, like **`until`**, **`upto`** and **`between`**.

```
define respondsto(x, y) := y implies eventually x
assert reply respondsto request
```

- With explicit arguments. Any macro can be accessed via its name followed by the comma-separated arguments enclosed in parentheses.

```
define my_macro(a, b, c) := a | b | !c
assert my_macro(x, u & v, z)
-- evaluates to a | (u & v) | !z
```

Indirect macro calls. A macro name may be passed as a parameter to another macro, which can then use the macro. This feature allows the definition of generic parameterised properties. The macro name has to be prefixed with `@` when passed as a parameter. For accessing it, the explicit call syntax `f(a, b,...)` has to be used; the prefix and infix variants are not allowed.

Example:

```
define myproperty(a, b) := a implies b
define circle(f,a,b,c) := f(a,b) and f(b,c) and f(c,a)
assert always circle(@myproperty, u, !v, w)
-- evaluates to G (u -> !v) & (!v -> w) & (w -> u)
```

5.5.3 Iteration

Many specifications have to define a certain assertion for a whole set of boolean variables like `input1 input2 input3`, or repeat an expression several times with a few parameters exchanged. Iteration operators allow easy handling of such sets of similar boolean variables and expressions.

The general syntax for an iteration expression is

`allof` *list* **`as`** *param* **`in`** φ
`noneof` *list* **`as`** *param* **`in`** φ
`someof` *list* **`as`** *param* **`in`** φ
`exactlyoneof` *list* **`as`** *param* **`in`** φ

Each of the elements in *list* is inserted as iteration parameter *param* in the expression φ. The resulting instantiated expressions are then combined using one of the four iteration operators.

List creation. The following operators can be used to create lists:

- **`list`** `[`φ`,`ψ`,...]`
  creates a list of expressions or identifiers.

- **enumerate** [n..m]
  creates a list containing the numbers between n and m. This list can be used to create for example a parameterised list of boolean variables with a common base name.

- *list* **without** φ
  removes the element φ (which must be an element of *list*) from the list.

- *list* **with** φ
  adds φ to the list.

Iteration operators.

- **allof**
  combines the instantiated expressions with a logical and, i. e., *all of* them have to be true in order to make the whole expression true.

- **noneof**
  combines the negated instantiated expressions with a logical and, i. e., *none of* them is allowed to be true in order to make the whole expression true.

- **someof**
  combines the instantiated expressions with a logical or, i. e., *some of* them have to be true in order to make the whole expression true.

- **exactlyoneof**
  requires *exactly one of* the instantiated expressions to be true and all the others to be false in order to make the whole expression true.

Examples:

```
assert allof list [a, b, c] as i in always i
-- is equal to (always a) & (always b) & (always c)
```

```
assert exactlyoneof list [a, b, c] as i in i
-- is equal to (a & !b & !c)|(!a & b & !c)|(!a & !b & c)
```

```
assert someof enumerate [1..3] as i in
         someof enumerate [1..3] without i as j in
            in$i$$j$
-- is equal to in12 | in13 | in21 | in23 | in31 | in32
```

Chapter 6

Translation schema

This chapter describes how the Salt language is translated into Ltl and Tltl and thereby defines the formal semantics of Salt.

The translation of past operators is left out for brevity, unless stated otherwise. It follows the same schema as the translation of the future operators. The translation of timed operators is described in section 6.6. The other sections of this chapter refer to untimed Salt.

Translation is done in several steps:

- Expansion of user-defined macros.
- Replacement of non-core Salt operators. Several Salt operators are replaced by expressions made out of a small set of core operators.
- Translation of core Salt into Salt--. The Salt operators are replaced by Salt-- expressions. Salt-- includes all Ltl operators as well as the acc and rej operators (corresponding to the Salt exception operators **`accepton`** and **`rejecton`**) and the exclusive and inclusive stop operators for future and past (introduced during the translation of **`upto`** and **`between`**).
- Translation of Salt-- into Ltl/Tltl. The translation of the Salt-- operators requires weaving their end conditions into the whole sub-expression.
- Optimisation. The Ltl/Tltl expression is optimised using a number of optimisation patterns.
- Ltl/Tltl output. The Ltl/Tltl expression is printed in the desired output syntax. This might require expressing certain operators through others (like W through U). Also, extended Tltl operators may be replaced by pure Tltl.

Each translation step is described in form of a translation function $\mathrm{T}(\varphi)$ that is applied by choosing the first translation that matches the current expression. Trivial translations that just descend recursively into the arguments of an operator, such as $\mathrm{T}(\varphi \wedge \psi) = \mathrm{T}(\varphi) \wedge \mathrm{T}(\psi)$, are left out in the following.

The LTL operators used during translation are:

true	$\top$	until	U	since	S
false	$\bot$	weak until	W	back to	B
logical negation	$\neg$	globally	$\Box$	historically	$\blacksquare$
logical and	$\wedge$	eventually	$\Diamond$	once	$\blacklozenge$
logical or	$\vee$	next	$\bigcirc$	previous	$\bullet$
logical implication	$\rightarrow$	weak next	$\bigcirc_W$	weak previous	$\bullet_W$
logical equivalence	$\leftrightarrow$				

And for timed expressions additionally:

timed until	$\mathrm{U}_{\sim c}$	timed since	$\mathrm{S}_{\sim c}$
timed weak until	$\mathrm{W}_{\sim c}$	timed weak since	$\mathrm{B}_{\sim c}$
timed globally	$\Box_{\sim c}$	timed historically	$\blacksquare_{\sim c}$
timed eventually	$\Diamond_{\sim c}$	timed once	$\blacklozenge_{\sim c}$
event predicting	$\rhd_{\sim c}$	event recording	$\lhd_{\sim c}$

The SALT-- operators used are:

accept	acc
reject	rej
exclusive stop	$\mathrm{stop}_{\mathrm{excl}}$
inclusive stop	$\mathrm{stop}_{\mathrm{incl}}$

6.1 Replacement of non-core SALT operators

6.1.1 `never`

$$\mathrm{T}(\textbf{never}\ \varphi) \quad = \quad \neg\Diamond\mathrm{T}(\varphi)$$

6.1.2 `releases`

$$\mathrm{T}(\varphi\ \textbf{releases}\ \psi) \quad = \quad \mathrm{T}(\psi\ \textbf{until incl weak}\ \varphi)$$

6.1.3 `nextn`

$$\begin{aligned}
&\mathrm{T}(\textbf{nextn[=}n\textbf{]}\varphi) && = \\
&\qquad \text{if } n = 0: && \mathrm{T}(\varphi) \\
&\qquad \text{else:} && \bigcirc\mathrm{T}(\textbf{nextn[=}n-1\textbf{]}\varphi) \\
&\mathrm{T}(\textbf{nextn[}n..m\textbf{]}\varphi) && = \mathrm{T}(\textbf{nextn[=}n\textbf{]}(\textbf{nextn[<=}m-n\textbf{]}\varphi)) \\
&\mathrm{T}(\textbf{nextn[<=}n\textbf{]}\varphi) && = \\
&\qquad \text{if } n = 0: && \mathrm{T}(\varphi) \\
&\qquad \text{else:} && \varphi \vee \bigcirc\mathrm{T}(\textbf{nextn[<=}n-1\textbf{]}\varphi) \\
&\mathrm{T}(\textbf{nextn[<}n\textbf{]}\varphi) && = \mathrm{T}(\textbf{nextn[<=}n-1\textbf{]}\varphi) \\
&\mathrm{T}(\textbf{nextn[>=}n\textbf{]}\varphi) && = \mathrm{T}(\textbf{nextn[=}n\textbf{]}\Diamond\varphi) \\
&\mathrm{T}(\textbf{nextn[>}n\textbf{]}\varphi) && = \mathrm{T}(\textbf{nextn[>=}n+1\textbf{]}\varphi)
\end{aligned}$$

6.1.4 occurring

$$
\begin{array}{lll}
\mathrm{T}(\texttt{occurring[=}n\texttt{]}\varphi) & = & \\
\quad \text{if } n = 0: & & \neg\Diamond\mathrm{T}(\varphi) \\
\quad \text{if } n = 1: & & \neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge ((\mathrm{T}(\varphi)\ \mathrm{W}\ \neg\Diamond\mathrm{T}(\varphi)))^{1} \\
\quad \text{else:} & & \neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge (\mathrm{T}(\varphi)\ \mathrm{U}\ (\neg\mathrm{T}(\varphi) \wedge \\
 & & \mathrm{T}(\texttt{occurring[=}n-1\texttt{]}\varphi))))
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{T}(\texttt{occurring[}n\texttt{..}m\texttt{]}\varphi) & = & \\
\quad \text{if } n = 0: & & \mathrm{T}(\texttt{occurring[<=}m\texttt{]}\varphi) \\
\quad \text{if } n = 1: & & \neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge (\mathrm{T}(\varphi)\ \mathrm{W}\ (\neg\mathrm{T}(\varphi) \wedge \\
 & & \mathrm{T}(\texttt{occurring[<=}m-1\texttt{]}\varphi))))^{1} \\
\quad \text{else:} & & \neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge (\mathrm{T}(\varphi)\ \mathrm{U}\ (\neg\mathrm{T}(\varphi) \wedge \\
 & & \mathrm{T}(\texttt{occurring[}n-1\texttt{..}m-1\texttt{]}\varphi))))
\end{array}
$$

$$\mathrm{T}(\texttt{occurring[<=}n\texttt{]}\varphi) = \neg\mathrm{T}(\texttt{occurring[>=}n+1\texttt{]}\varphi)$$

$$\mathrm{T}(\texttt{occurring[<}n\texttt{]}\varphi) = \neg\mathrm{T}(\texttt{occurring[>=}n\texttt{]}\varphi)$$

$$
\begin{array}{lll}
\mathrm{T}(\texttt{occurring[>=}n\texttt{]}\varphi) & = & \\
\quad \text{if } n = 0: & & \top \\
\quad \text{if } n = 1: & & \Diamond\mathrm{T}(\varphi) \\
\quad \text{else:} & & \Diamond(\mathrm{T}(\varphi) \wedge (\Diamond(\neg\mathrm{T}(\varphi) \wedge \\
 & & \mathrm{T}(\texttt{occurring[>=}n-1\texttt{]}\varphi))))
\end{array}
$$

$$\mathrm{T}(\texttt{occurring[>}n\texttt{]}\varphi) = \mathrm{T}(\texttt{occurring[>=}n+1\texttt{]}\varphi)$$

6.1.5 holding

$$
\begin{array}{lll}
\mathrm{T}(\texttt{holding[=}n\texttt{]}\varphi) & = & \\
\quad \text{if } n = 0: & & \neg\Diamond\mathrm{T}(\varphi) \\
\quad \text{if } n = 1: & & \neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge \circ_{W}\neg\Diamond\mathrm{T}(\varphi)))^{2} \\
\quad \text{else:} & & \neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge \circ\mathrm{T}(\texttt{holding[=}n-1\texttt{]}\varphi))
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{T}(\texttt{holding[}n\texttt{..}m\texttt{]}\varphi) & = & \\
\quad \text{if } n = 0: & & \mathrm{T}(\texttt{holding[<=}m\texttt{]}\varphi) \\
\quad \text{if } n = 1: & & \neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge \\
 & & \circ_{W}\mathrm{T}(\texttt{holding[<=}m-1\texttt{]}\varphi))^{2} \\
\quad \text{else:} & & \neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge \\
 & & \circ\mathrm{T}(\texttt{holding[}n-1\texttt{..}m-1\texttt{]}\varphi))
\end{array}
$$

$$\mathrm{T}(\texttt{holding[<=}n\texttt{]}\varphi) = \neg\mathrm{T}(\texttt{holding[>=}n+1\texttt{]}\varphi)$$

$$\mathrm{T}(\texttt{holding[<}n\texttt{]}\varphi) = \neg\mathrm{T}(\texttt{holding[>=}n\texttt{]}\varphi)$$

$$
\begin{array}{lll}
\mathrm{T}(\texttt{holding[>=}n\texttt{]}\varphi) & = & \\
\quad \text{if } n = 0: & & \top \\
\quad \text{if } n = 1: & & \Diamond\mathrm{T}(\varphi)^{2} \\
\quad \text{else:} & & \Diamond(\mathrm{T}(\varphi) \wedge \circ\mathrm{T}(\texttt{holding[>=}n-1\texttt{]}\varphi))
\end{array}
$$

$$\mathrm{T}(\texttt{holding[>}n\texttt{]}\varphi) = \mathrm{T}(\texttt{holding[>=}n+1\texttt{]}\varphi)$$

[1] Notice that the last occurrence of φ may last forever.

[2] A special case for $n = 1$ is required for situations where there is no next state, because either a surrounding **upto** ended or because we reached time zero in the past. In these

6.1.6 Regular expressions, part I

The `?` and `+` repetition operators can be expressed by the more general `*` operator as follows:

$$\mathrm{T}(\varphi\texttt{?}) = \mathrm{T}(\varphi\texttt{*[<=1]})$$

$$\mathrm{T}(p\texttt{+}) = \mathrm{T}(p\texttt{*[>=1]})$$

The different variants of the `*` repetition operator are translated as follows into core SALT, where only sequences and the `*[>=`n`]` repetition operator exist. The empty sequence is denoted by ε.

$$\mathrm{T}(\varphi\texttt{*[=}n\texttt{]}) =$$
$$\text{if } n = 0: \quad \varepsilon$$
$$\text{if } n = 1: \quad \mathrm{T}(\varphi)$$
$$\text{else:} \quad \mathrm{T}(\varphi\texttt{;}\varphi\texttt{*[=}n-1\texttt{]})$$

$$\mathrm{T}(\varphi\texttt{[}n\texttt{..}m\texttt{]}) =$$
$$\text{if } n = 0: \quad \mathrm{T}(\varphi\texttt{*[<=}m\texttt{]})$$
$$\text{else:} \quad \mathrm{T}(\varphi\texttt{*[=}n-1\texttt{];}\varphi\texttt{*[<=}m-n\texttt{];}\varphi)^3$$

$$\mathrm{T}(\varphi\texttt{*[<=}n\texttt{]}) =$$
$$\text{if } n = 0: \quad \varepsilon$$
$$\text{else:} \quad \varepsilon \vee \mathrm{T}(\overbrace{\varphi \vee \varphi\texttt{;}(\varphi \vee \varphi\texttt{;}(\ldots))}^{n})^4$$

$$\mathrm{T}(\varphi\texttt{*[<}n\texttt{]}) = \mathrm{T}(\varphi\texttt{*[<=}n-1\texttt{]})$$

$$\mathrm{T}(p\texttt{*[>}n\texttt{]}) = \mathrm{T}(p\texttt{*[>=}n+1\texttt{]})$$

6.1.7 Iteration operators

The iteration operators are translated as follows:

$$\mathrm{T}(\textbf{\texttt{allof}}\ \mathit{list}) = \bigwedge_{\varphi\in\mathit{list}} \mathrm{T}(\varphi)$$

$$\mathrm{T}(\textbf{\texttt{noneof}}\ \mathit{list}) = \neg \bigvee_{\varphi\in\mathit{list}} \mathrm{T}(\varphi)$$

$$\mathrm{T}(\textbf{\texttt{someof}}\ \mathit{list}) = \bigvee_{\varphi\in\mathit{list}} \mathrm{T}(\varphi)$$

$$\mathrm{T}(\textbf{\texttt{exactlyoneof}}\ \mathit{list}) = \bigvee_{\varphi\in\mathit{list}} (\mathrm{T}(\varphi) \wedge \neg \bigvee_{\psi\in\mathit{list},\psi\neq\varphi} \mathrm{T}(\psi))$$

---

situations, even $\bigcirc\top$ would be false, although the conditions for the **`holding`** operator have been fulfilled.

[3] The trailing φ is necessary for correct translation when followed by a `:` sequence operator.

[4] The schema used here repeats φ less times than the straightforward translation φ`*[=...]`$\vee\varphi$`*[=...]`$\vee\ldots$.

6.2 Translation of core SALT into SALT--

6.2.1 `until`

$$
\begin{aligned}
\mathrm{T}(\varphi \,\texttt{until excl req}\, \psi) &= \mathrm{T}(\varphi)\ \mathrm{U}\ \mathrm{T}(\psi) \\
\mathrm{T}(\varphi \,\texttt{until excl opt}\, \psi) &= (\Diamond \mathrm{T}(\psi)) \rightarrow (\mathrm{T}(\varphi)\ \mathrm{U}\ \mathrm{T}(\psi)) \\
\mathrm{T}(\varphi \,\texttt{until excl weak}\, \psi) &= \mathrm{T}(\varphi)\ \mathrm{W}\ \mathrm{T}(\psi) \\
\mathrm{T}(\varphi \,\texttt{until incl req}\, \psi) &= \mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge \mathrm{T}(\psi)) \\
\mathrm{T}(\varphi \,\texttt{until incl opt}\, \psi) &= (\Diamond \mathrm{T}(\psi)) \rightarrow (\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge \mathrm{T}(\psi))) \\
\mathrm{T}(\varphi \,\texttt{until incl weak}\, \psi) &= \mathrm{T}(\varphi)\ \mathrm{W}\ (\mathrm{T}(\varphi) \wedge \mathrm{T}(\psi))
\end{aligned}
$$

6.2.2 upto

$$
\begin{array}{lll}
\mathrm{T}(\varphi \ \textbf{upto excl req}\ b) & = & \\
\quad \text{if } \mathrm{T}(\varphi) = \Box\psi\text{:} & & (\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b \\
\quad \text{if } \mathrm{T}(\varphi) = \neg\Diamond\psi\text{:} & & (\neg\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b \\
\quad \text{else:} & & (\Diamond b) \wedge (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b)^{5} \\
\mathrm{T}(\varphi \ \textbf{upto excl opt}\ b) & = & \\
\quad \text{if } \mathrm{T}(\varphi) = \Diamond\psi\text{:} & & \neg((\neg\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b) \\
\quad \text{else:} & & (\Diamond b) \rightarrow (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b)^{5} \\
\mathrm{T}(\varphi \ \textbf{upto excl weak}\ b) & = & (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b) \\
\mathrm{T}(\textbf{req}\ \varphi \ \textbf{upto excl req}\ b) & = & \\
\quad \text{if } \mathrm{T}(\varphi) = \Box\psi\text{:} & & \neg b \wedge ((\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b) \\
\quad \text{if } \mathrm{T}(\varphi) = \neg\Diamond\psi\text{:} & & \neg b \wedge ((\neg\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b) \\
\quad \text{else:} & & (\Diamond b) \wedge \neg b \wedge (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b)^{5} \\
\mathrm{T}(\textbf{req}\ \varphi \ \textbf{upto excl opt}\ b) & = & \\
\quad \text{if } \mathrm{T}(\varphi) = \Diamond\psi\text{:} & & \neg((\neg\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b) \\
\quad \text{else:} & & (\Diamond b) \rightarrow (\neg b \wedge (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b))^{5} \\
\mathrm{T}(\textbf{req}\ \varphi \ \textbf{upto excl weak}\ b) & = & \neg b \wedge (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b) \\
\mathrm{T}(\textbf{weak}\ \varphi \ \textbf{upto excl req}\ b) & = & \\
\quad \text{if } \mathrm{T}(\varphi) = \Box\psi\text{:} & & (\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b \\
\quad \text{if } \mathrm{T}(\varphi) = \neg\Diamond\psi\text{:} & & (\neg\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b \\
\quad \text{else:} & & (\Diamond b) \wedge (b \vee (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b))^{5} \\
\mathrm{T}(\textbf{weak}\ \varphi \ \textbf{upto excl opt}\ b) & = & \\
\quad \text{if } \mathrm{T}(\varphi) = \Diamond\psi\text{:} & & b \vee \neg((\neg\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b) \\
\quad \text{else:} & & (\Diamond b) \rightarrow (b \vee (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b))^{5} \\
\mathrm{T}(\textbf{weak}\ \varphi \ \textbf{upto excl weak}\ b) & = & b \vee (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b) \\
\mathrm{T}(\varphi \ \textbf{upto incl req}\ b) & = & (\Diamond b) \wedge (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{incl}}\ b) \\
\mathrm{T}(\varphi \ \textbf{upto incl opt}\ b) & = & (\Diamond b) \rightarrow (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{incl}}\ b) \\
\mathrm{T}(\varphi \ \textbf{upto incl weak}\ b) & = & \\
\quad \text{if } \mathrm{T}(\varphi) = \Box\psi\text{:} & & \neg(\neg b\ \mathrm{U}\ \neg(\psi\ \mathrm{stop}_{\mathrm{incl}}\ b)) \\
\quad \text{if } \mathrm{T}(\varphi) = \neg\Diamond\psi\text{:} & & \neg(\neg b\ \mathrm{U}\ (\psi\ \mathrm{stop}_{\mathrm{incl}}\ b)) \\
\quad \text{else:} & & (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{incl}}\ b)^{5}
\end{array}
$$

[5] The specialised translations exist only for optimisation reasons.

6.2.3 `from`

$$
\begin{array}{lcl}
\mathrm{T}(\varphi \,\texttt{from incl req}\, a) & = & (\neg a)\ \mathrm{U}\ (a \wedge \mathrm{T}(\varphi)) \\
\mathrm{T}(\varphi \,\texttt{from incl opt}\, a) & = & \\
\quad \text{if } \mathrm{T}(\varphi) = \Box\psi\text{:} & & \Box(a \rightarrow \Box\psi) \\
\quad \text{if } \mathrm{T}(\varphi) = \neg\Diamond\psi\text{:} & & \Box(a \rightarrow \neg\Diamond\psi) \\
\quad \text{else:} & & (\neg a)\ \mathrm{W}\ (a \wedge \mathrm{T}(\varphi))^{6} \\
\mathrm{T}(\varphi \,\texttt{from excl req}\, a) & = & (\neg a)\ \mathrm{U}\ (a \wedge \bigcirc\mathrm{T}(\varphi)) \\
\mathrm{T}(\varphi \,\texttt{from excl opt}\, a) & = & (\neg a)\ \mathrm{W}\ (a \wedge \bigcirc\mathrm{T}(\varphi))
\end{array}
$$

6.2.4 `between`

$$\mathrm{T}(\varphi \,\texttt{between}\, a, b) \quad = \quad \mathrm{T}((\varphi \,\texttt{upto}\, b)\texttt{from}\, a)$$

6.2.5 Exception operators

$$
\begin{array}{lcl}
\mathrm{T}(\varphi \,\texttt{accepton}\, b) & = & \mathrm{T}(\varphi)\ \mathrm{acc}\ b \\
\mathrm{T}(\varphi \,\texttt{rejecton}\, b) & = & \mathrm{T}(\varphi)\ \mathrm{rej}\ b
\end{array}
$$

6.2.6 Regular expressions, part II

The `*[>=`n`]` repetition operator is translated as follows (its translation depends on the next element ψ in the sequence as well as on the sequence operator):

$$
\begin{array}{lcl}
\mathrm{T}(p\texttt{*[>=0];}\psi) & = & p\ \mathrm{U}\ \mathrm{T}(\psi) \\
\mathrm{T}(p\texttt{*[>=}n\texttt{];}\psi) & = & p\ \mathrm{U}\ \mathrm{T}(\overbrace{p\texttt{;}p\texttt{;}\ldots\texttt{;}p\texttt{;}}^{n}\ \psi) \\
\mathrm{T}(p\texttt{*[>=0]:}\psi) & = & \mathrm{T}(\psi) \vee \mathrm{T}(p\texttt{*[>=1]:}\psi) \\
\mathrm{T}(p\texttt{*[>=}n\texttt{]:}\psi) & = & p\ \mathrm{U}\ \mathrm{T}(\overbrace{p\texttt{;}p\texttt{;}\ldots\texttt{;}p\texttt{:}}^{n}\ \psi)
\end{array}
$$

For the translation of the sequence operators, we have to define the length of a regular expression:

$$
|\varphi| := \begin{cases}
|\varepsilon| & = 0 \\
|p| & = 1 \\
|p\texttt{*[>=}n\texttt{]}| & = \bot \\
|\varphi_1\texttt{;}\varphi_2| & = |\varphi_1| + |\varphi_2| \\
|\varphi_1\texttt{:}\varphi_2| & = |\varphi_1| + |\varphi_2| - 1
\end{cases}
$$

The sequence operators are then translated as follows:

[6] The specialised translations exist only for optimisation reasons.

$$
\begin{array}{lcl}
\mathrm{T}((\varphi_1 \vee \varphi_2)\texttt{;}\psi) & = & \begin{cases} \text{if } |\varphi_1| \neq |\varphi_2|: & \mathrm{T}(\varphi_1\texttt{;}\psi) \vee \mathrm{T}(\varphi_2\texttt{;}\psi) \\ \text{else:} & \mathrm{T}((\varphi_1 \vee \varphi_2)\texttt{;}\psi) \end{cases} \\
\mathrm{T}(\varphi\texttt{;}\psi) & = & \mathrm{T}(\varphi) \wedge \bigcirc^{|\varphi|}\mathrm{T}(\psi) \\
\mathrm{T}((\varphi_1 \vee \varphi_2)\texttt{:}\psi) & = & \begin{cases} \text{if } |\varphi_1| \neq |\varphi_2|: & \mathrm{T}(\varphi_1\texttt{:}\psi) \vee \mathrm{T}(\varphi_2\texttt{:}\psi) \\ \text{else:} & \mathrm{T}((\varphi_1 \vee \varphi_2)\texttt{:}\psi) \end{cases} \\
\mathrm{T}(\varphi\texttt{:}\psi) & = & \begin{cases} \text{if } \varphi = \varepsilon: & \mathrm{T}(\psi) \\ \text{else:} & \mathrm{T}(\varphi) \wedge \bigcirc^{|\varphi|-1}\mathrm{T}(\psi) \end{cases}
\end{array}
$$

6.3 Translation of SALT-- into LTL

During this step, the rej and acc operators (SALT-- equivalents of the SALT exception operators) as well as the stop operators (introduced during the translation of **`upto`** and **`between`**) are replaced by pure LTL expressions. This requires weaving the end conditions into all sub-expressions of the argument. The innermost operators are replaced first, so that the translation process does not have to deal explicitly with nested operators.

6.3.1 acc

$$
\begin{array}{lcl}
\mathrm{T}(b \text{ acc } a) & = & b \vee a \\
\mathrm{T}((\neg\varphi) \text{ acc } a) & = & \neg\mathrm{T}(\varphi \text{ rej } a) \\
\mathrm{T}((\varphi \wedge \psi) \text{ acc } a) & = & \mathrm{T}(\varphi \text{ acc } a) \wedge \mathrm{T}(\psi \text{ acc } a) \\
\mathrm{T}((\varphi \vee \psi) \text{ acc } a) & = & \mathrm{T}(\varphi \text{ acc } a) \vee \mathrm{T}(\psi \text{ acc } a) \\
\mathrm{T}((\varphi \text{ U } \psi) \text{ acc } a) & = & \mathrm{T}(\varphi \text{ acc } a) \text{ U } \mathrm{T}(\psi \text{ acc } a) \\
\mathrm{T}((\bigcirc\varphi) \text{ acc } a) & = & (\bigcirc\mathrm{T}(\varphi \text{ acc } a)) \vee a \\
\mathrm{T}((\Box\varphi) \text{ acc } a) & = & \neg(\neg a \text{ U } \neg\mathrm{T}(\varphi \text{ acc } a)) \\
\mathrm{T}((\Diamond\varphi) \text{ acc } a) & = & \Diamond\mathrm{T}(\varphi \text{ acc } a)
\end{array}
$$

The translation of $\rightarrow$, $\leftrightarrow$, W and $\bigcirc_W$ is done using the corresponding LTL equivalents in 6.5.

6.3.2 rej

$$
\begin{aligned}
\mathrm{T}(b \text{ rej } r) &= b \wedge \neg r \\
\mathrm{T}((\neg\varphi) \text{ rej } r) &= \neg\mathrm{T}(\varphi \text{ acc } r) \\
\mathrm{T}((\varphi \wedge \psi) \text{ rej } r) &= \mathrm{T}(\varphi \text{ rej } r) \wedge \mathrm{T}(\psi \text{ rej } r) \\
\mathrm{T}((\varphi \vee \psi) \text{ rej } r) &= \mathrm{T}(\varphi \text{ rej } r) \vee \mathrm{T}(\psi \text{ rej } r) \\
\mathrm{T}((\varphi \text{ U } \psi) \text{ rej } r) &= \mathrm{T}(\varphi \text{ rej } r) \text{ U } \mathrm{T}(\psi \text{ rej } r) \\
\mathrm{T}((\bigcirc\varphi) \text{ rej } r) &= (\bigcirc\mathrm{T}(\varphi \text{ rej } r)) \wedge \neg r \\
\mathrm{T}((\Box\varphi) \text{ rej } r) &= \Box\mathrm{T}(\varphi \text{ rej } r) \\
\mathrm{T}((\Diamond\varphi) \text{ rej } a) &= \neg r \text{ U } \mathrm{T}(\varphi \text{ rej } r)
\end{aligned}
$$

The translation of $\rightarrow$, $\leftrightarrow$, W and $\bigcirc_W$ is done using the corresponding LTL equivalents in 6.5.

6.3.3 $\text{stop}_{\text{incl}}$

$$
\begin{aligned}
\mathrm{T}(b \text{ stop}_{\text{incl}} s) &= b \\
\mathrm{T}((\neg\varphi) \text{ stop}_{\text{incl}} s) &= \neg\mathrm{T}(\varphi \text{ stop}_{\text{incl}} s) \\
\mathrm{T}((\varphi \wedge \psi) \text{ stop}_{\text{incl}} s) &= \mathrm{T}(\varphi \text{ stop}_{\text{incl}} s) \wedge \mathrm{T}(\psi \text{ stop}_{\text{incl}} s) \\
\mathrm{T}((\varphi \vee \psi) \text{ stop}_{\text{incl}} s) &= \mathrm{T}(\varphi \text{ stop}_{\text{incl}} s) \vee \mathrm{T}(\psi \text{ stop}_{\text{incl}} s) \\
\mathrm{T}((\varphi \text{ U } \psi) \text{ stop}_{\text{incl}} s) &= (\neg s \wedge \mathrm{T}(\varphi \text{ stop}_{\text{incl}} s)) \text{ U } \mathrm{T}(\psi \text{ stop}_{\text{incl}} s) \\
\mathrm{T}((\bigcirc\varphi) \text{ stop}_{\text{incl}} s) &= \neg s \wedge \bigcirc\mathrm{T}(\varphi \text{ stop}_{\text{incl}} s) \\
\mathrm{T}((\bigcirc_W\varphi) \text{ stop}_{\text{incl}} s) &= s \vee \bigcirc\mathrm{T}(\varphi \text{ stop}_{\text{incl}} s) \\
\mathrm{T}((\Box\varphi) \text{ stop}_{\text{incl}} s) &= \neg(\neg s \text{ U } \neg\mathrm{T}(\varphi \text{ stop}_{\text{incl}} s)) \\
\mathrm{T}((\Diamond\varphi) \text{ stop}_{\text{incl}} s) &= (\neg s) \text{ U } \mathrm{T}(\varphi \text{ stop}_{\text{incl}} s) \\
\mathrm{T}((\varphi \text{ S } \psi) \text{ stop}_{\text{incl}} s) &= \mathrm{T}(\varphi \text{ stop}_{\text{incl}} s) \text{ S } \mathrm{T}(\psi \text{ stop}_{\text{incl}} s)^{7} \\
\mathrm{T}((\bullet\varphi) \text{ stop}_{\text{incl}} s) &= \bullet\mathrm{T}(\varphi \text{ stop}_{\text{incl}} s)^{7}
\end{aligned}
$$

The past stop operators are translated in a similar way as the future stop operators, but affecting only past operators. The translation of W, $\rightarrow$ and $\leftrightarrow$ is done using the corresponding LTL equivalents in 6.5.

[7] Notice how the future stop operator affects only future operators and leaves the past operators unchanged. The past operators not listed here are translated similarly.

6.3.4 $\mathbf{stop_{excl}}$

$$
\begin{aligned}
\mathrm{T}(b \text{ stop}_{\text{excl}}\ s) &= b \\
\mathrm{T}((\neg\varphi) \text{ stop}_{\text{excl}}\ s) &= \neg\mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s) \\
\mathrm{T}((\varphi \wedge \psi) \text{ stop}_{\text{excl}}\ s) &= \mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s) \wedge \mathrm{T}(\psi \text{ stop}_{\text{excl}}\ s) \\
\mathrm{T}((\varphi \vee \psi) \text{ stop}_{\text{excl}}\ s) &= \mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s) \vee \mathrm{T}(\psi \text{ stop}_{\text{excl}}\ s) \\
\mathrm{T}((\varphi\ \mathrm{U}\ \psi) \text{ stop}_{\text{excl}}\ s) &= (\neg s \wedge \mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s))\ \mathrm{U}\ (\neg s \wedge \mathrm{T}(\psi \text{ stop}_{\text{excl}}\ s)) \\
\mathrm{T}((\varphi\ \mathrm{W}\ \psi) \text{ stop}_{\text{excl}}\ s) &= \mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s)\ \mathrm{W}\ (s \vee \mathrm{T}(\psi \text{ stop}_{\text{excl}}\ s)) \\
\mathrm{T}((\bigcirc\varphi) \text{ stop}_{\text{excl}}\ s) &= \bigcirc(\neg s \wedge \mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s)) \\
\mathrm{T}((\bigcirc_W\varphi) \text{ stop}_{\text{excl}}\ s) &= \bigcirc(s \vee \mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s)) \\
\mathrm{T}((\Box\varphi) \text{ stop}_{\text{excl}}\ s) &= \mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s)\ \mathrm{W}\ s \\
\mathrm{T}((\Diamond\varphi) \text{ stop}_{\text{excl}}\ s) &= (\neg s)\ \mathrm{U}\ (\neg s \wedge \mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s)) \\
\mathrm{T}((\varphi\ \mathrm{S}\ \psi) \text{ stop}_{\text{excl}}\ s) &= \mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s)\ \mathrm{S}\ \mathrm{T}(\psi \text{ stop}_{\text{excl}}\ s)^{7} \\
\mathrm{T}((\bullet\varphi) \text{ stop}_{\text{excl}}\ s) &= \bullet\mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s)^{7}
\end{aligned}
$$

The past stop operators are translated in a similar way as the future stop operators, but affecting only past operators. The translation of $\rightarrow$ and $\leftrightarrow$ is done using the corresponding LTL equivalents in 6.5.

6.4 Optimisation

The following equivalences are used for optimisation:

$$
\begin{aligned}
\top\ \mathrm{U}\ \varphi &\iff \Diamond\varphi \\
\neg\Diamond\neg\varphi &\iff \Box\varphi \\
\Box\Box\varphi &\iff \Box\varphi \\
\Diamond\Diamond\varphi &\iff \Diamond\varphi \\
\neg\varphi\ \mathrm{U}\ \varphi &\iff \Diamond\varphi \\
\Box(\varphi\ \mathrm{W}\ \psi) &\iff \Box(\varphi \vee \psi) \\
\varphi\ \mathrm{W}\ (\varphi \wedge \psi) &\iff \neg(\neg\psi\ \mathrm{U}\ \neg\varphi) \\
(\varphi \vee \psi)\ \mathrm{U}\ \psi &\iff \varphi\ \mathrm{U}\ \psi
\end{aligned}
$$

Furthermore, boolean operators with constant arguments (e. g., $\top \wedge a$) are eliminated.

6.5 Operator replacement

The following equivalences are used to express certain operators through others if necessary for the current output syntax.

$$
\begin{array}{lcl}
\Box\varphi & \Longleftrightarrow & \neg(\top \text{ U } \neg\varphi) \\
\Diamond\varphi & \Longleftrightarrow & \top \text{ U } \varphi \\
\mathsf{O}_W\psi & \Longleftrightarrow & \neg\mathsf{O}(\neg\varphi) \\
\varphi \text{ W } \psi & \Longleftrightarrow & \begin{cases} \text{if } |\psi| \leq |\varphi|^{8}\text{:} & \neg(\neg\psi \text{ U } (\neg\varphi \wedge \neg\psi)) \\ \text{else:} & (\varphi \text{ U } \psi) \vee \Box\varphi \end{cases} \\
\neg(\neg\varphi \text{ U } \neg\psi) & \Longleftrightarrow & \varphi \text{ R } \psi
\end{array}
$$

6.6 Translation of timed operators

6.6.1 Timed SALT into timed SALT--

$$
\begin{array}{lcl}
\mathrm{T}(\texttt{next timed[}\sim c\texttt{]}\varphi) & = & \rhd_{\sim c}\mathrm{T}(\varphi) \\
\mathrm{T}(\varphi \texttt{ until timed[}\sim c\texttt{] } \psi) & = & \mathrm{T}(\varphi) \text{ U}_{\sim c} \mathrm{T}(\psi) \\
\mathrm{T}(\varphi \texttt{ until timed[}\sim c\texttt{] weak } \psi) & = & \mathrm{T}(\varphi) \text{ W}_{\sim c} \mathrm{T}(\psi) \\
\mathrm{T}(\varphi \texttt{ until timed[}\sim c\texttt{] excl req } \psi) & = & \mathrm{T}(\varphi) \text{ U}_{\sim c} \mathrm{T}(\psi) \\
\mathrm{T}(\varphi \texttt{ until timed[}\sim c\texttt{] excl opt } \psi) & = & (\Diamond_{\sim c}\mathrm{T}(\psi)) \rightarrow \\
 & & (\mathrm{T}(\varphi) \text{ U}_{\sim c} \mathrm{T}(\psi)) \\
\mathrm{T}(\varphi \texttt{ until timed[}\sim c\texttt{] excl weak } \psi) & = & \mathrm{T}(\varphi) \text{ W}_{\sim c} \mathrm{T}(\psi) \\
\mathrm{T}(\varphi \texttt{ until timed[}\sim c\texttt{] incl req } \psi) & = & \mathrm{T}(\varphi) \text{ U}_{\sim c} (\mathrm{T}(\varphi) \wedge \mathrm{T}(\psi)) \\
\mathrm{T}(\varphi \texttt{ until timed[}\sim c\texttt{] incl opt } \psi) & = & (\Diamond_{\sim c}\mathrm{T}(\psi)) \rightarrow (\mathrm{T}(\varphi) \text{ U}_{\sim c} \\
 & & (\mathrm{T}(\varphi) \wedge \mathrm{T}(\psi))) \\
\mathrm{T}(\varphi \texttt{ until timed[}\sim c\texttt{] incl weak } \psi) & = & \mathrm{T}(\varphi) \text{ W}_{\sim c} (\mathrm{T}(\varphi) \wedge \mathrm{T}(\psi)) \\
\mathrm{T}(\texttt{timed[}\sim c\texttt{] } \varphi \texttt{ releases } \psi) & = & \mathrm{T}(\psi) \text{ W}_{\sim c} (\mathrm{T}(\psi) \wedge \mathrm{T}(\varphi)) \\
\mathrm{T}(\texttt{always timed[}\sim c\texttt{] } \varphi) & = & \Box_{\sim c}\mathrm{T}(\varphi) \\
\mathrm{T}(\texttt{eventually timed[}\sim c\texttt{] } \varphi) & = & \Diamond_{\sim c}\mathrm{T}(\varphi)
\end{array}
$$

[8] As a heuristic estimation for the size of a formula, the number of temporal operators in the formula is used.

6.6.2 Timed SALT-- into extended TLTL

acc (`accepton`):

$$
\begin{aligned}
\mathrm{T}((\rhd_{\sim c}\varphi)\ \mathrm{acc}\ a) &= a \vee \rhd_{\sim c}\mathrm{T}(\varphi\ \mathrm{acc}\ a)\\
\mathrm{T}((\varphi\ \mathrm{U}_{\sim c}\ \psi)\ \mathrm{acc}\ a) &= \mathrm{T}(\varphi\ \mathrm{acc}\ a)\ \mathrm{U}_{\sim c}\ \mathrm{T}(\psi\ \mathrm{acc}\ a)\\
\mathrm{T}((\varphi\ \mathrm{W}_{\sim c}\ \psi)\ \mathrm{acc}\ a) &= \begin{cases} \text{if } |\psi| \le |\varphi|^{9}\text{:} & \neg(\neg\mathrm{T}(\psi\ \mathrm{acc}\ a)\ \mathrm{U}_{\sim c}\\ & (\neg\mathrm{T}(\varphi\ \mathrm{acc}\ a) \wedge \neg\mathrm{T}(\psi\ \mathrm{acc}\ a)))\\ \text{else:} & (\mathrm{T}(\varphi\ \mathrm{acc}\ a)\ \mathrm{U}_{\sim c}\ \mathrm{T}(\psi\ \mathrm{acc}\ a))\vee\\ & \neg(\neg a\ \mathrm{U}_{\sim c}\ \neg\mathrm{T}(\varphi\ \mathrm{acc}\ a)) \end{cases}\\
\mathrm{T}((\Box_{\sim c}\varphi)\ \mathrm{acc}\ a) &= \neg(\neg a\ \mathrm{U}_{\sim c}\ \neg\mathrm{T}(\varphi\ \mathrm{acc}\ a))\\
\mathrm{T}((\Diamond_{\sim c}\varphi)\ \mathrm{acc}\ a) &= \Diamond_{\sim c}\mathrm{T}(\varphi\ \mathrm{acc}\ a)
\end{aligned}
$$

rej (`rejecton`):

$$
\begin{aligned}
\mathrm{T}((\rhd_{\sim c}\varphi)\ \mathrm{rej}\ r) &= \neg r \wedge \bigcirc(\neg r\ \mathrm{U}\ \mathrm{T}(\varphi\ \mathrm{rej}\ r)) \wedge \rhd_{\sim c}\mathrm{T}(\varphi\ \mathrm{rej}\ r)^{10}\\
\mathrm{T}((\varphi\ \mathrm{U}_{\sim c}\ \psi)\ \mathrm{rej}\ r) &= \mathrm{T}(\varphi\ \mathrm{rej}\ r)\ \mathrm{U}_{\sim c}\ \mathrm{T}(\psi\ \mathrm{rej}\ r)\\
\mathrm{T}((\varphi\ \mathrm{W}_{\sim c}\ \psi)\ \mathrm{rej}\ r) &= \begin{cases} \text{if } |\psi| \le |\varphi|^{11}\text{:} & \neg(\neg\mathrm{T}(\psi\ \mathrm{rej}\ r)\ \mathrm{U}_{\sim c}\\ & (\neg\mathrm{T}(\varphi\ \mathrm{rej}\ r) \wedge \neg\mathrm{T}(\psi\ \mathrm{rej}\ r)))\\ \text{else:} & (\mathrm{T}(\varphi\ \mathrm{rej}\ r)\ \mathrm{U}_{\sim c}\ \mathrm{T}(\psi\ \mathrm{rej}\ r))\vee\\ & \Box_{\sim c}\mathrm{T}(\varphi\ \mathrm{rej}\ r) \end{cases}\\
\mathrm{T}((\Box_{\sim c}\varphi)\ \mathrm{rej}\ r) &= \Box_{\sim c}\mathrm{T}(\varphi\ \mathrm{rej}\ r)\\
\mathrm{T}((\Diamond_{\sim c}\varphi)\ \mathrm{rej}\ r) &= \neg r\ \mathrm{U}_{\sim c}\ \mathrm{T}(\varphi\ \mathrm{rej}\ r)
\end{aligned}
$$

stop operators: The stop operators do not influence timed operators, i. e., any timed operator and its arguments are left unchanged.

6.6.3 Extended TLTL into pure TLTL

$$
\begin{aligned}
\mathrm{T}(\varphi\ \mathrm{U}_{\sim c}\ \psi) &= (\mathrm{T}(\varphi)\ \mathrm{U}\ \mathrm{T}(\psi)) \wedge (\mathrm{T}(\psi) \vee \rhd_{\sim c}\mathrm{T}(\psi))\\
\mathrm{T}(\varphi\ \mathrm{W}_{\sim c}\ \psi) &= (\mathrm{T}(\varphi)\ \mathrm{U}\ \mathrm{T}(\psi)) \vee (\mathrm{T}(\varphi) \wedge \neg \rhd_{\sim c}\neg\mathrm{T}(\varphi))\\
\mathrm{T}(\Box_{\sim c}\varphi) &= \mathrm{T}(\varphi) \wedge \neg(\rhd_{\sim c}\neg\mathrm{T}(\varphi))\\
\mathrm{T}(\Diamond_{\sim c}\varphi) &= \mathrm{T}(\varphi) \vee \rhd_{\sim c}\mathrm{T}(\varphi)
\end{aligned}
$$

[9] As a heuristic estimation for the size of a formula, the number of temporal operators in the formula is used.

[10] The $\bigcirc$ is required because $\rhd_{\sim c}\varphi$ is not supposed to match occurrences of φ at the current state, but U would.

[11] As a heuristic estimation for the size of a formula, the number of temporal operators in the formula is used.

Chapter 7

Examples

The following examples describe requirements in natural language and provide a corresponding SALT specification.

Simple specification

```
-- Requirement: A query is eventually answered

assert always (query -> (eventually answer))
```

Specification using `until`

This example makes use of a quoted atomic proposition to encapsulate a comparison predicate.

```
-- Requirement: The software is working until the
--   queue is empty or an abort signal comes.
--   Working may continue forever.

assert working until weak ("queuelength == 0" | abort)
```

Scheduler specification

The original specification for this example can be found on [DAC99].

```
-- Requirement: Between the moment in which an
--   execution completes and before a new execution
--   begins there is no work done.
-- Handwritten LTL: [](((return_Execute && <>call_Execute)
--   -> ((!call_doWork) U call_Execute))

assert always
  (never call_doWork
   between inclusive optional return_Execute,
           exclusive optional call_Execute)
```

Precedence specification

This example makes use of macros and past operators.

```
-- Requirement: An answer is preceded by a request

define precedes(x, y) := if y then once x
assert always (request precedes answer)
```

Elevator specification

The original specification for this example can be found in [DAC99].

```
-- Requirement: Between the time an elevator is called at
--   a floor and the time it opens its doors at that
--   floor, the elevator can arrive at that floor at most
--   twice.
-- Handwritten LTL: [](call & <>open) ->
--   ((!atfloor & !open) U
--     (open | ((atfloor & !open) U
--      (open | ((!atfloor & !open) U
--       (open | ((atfloor & !open) U
--        (open | (!atfloor U open)))))))))

assert always
 (occurring[<=2] atfloor
  between incl optional call, excl optional open)
```

Input channel iteration specification

This example makes use of iteration operators.

```
-- Requirement: Only one of the four input channels may
--    be active at a time

assert always
        (exactlyoneof enumerate [0..3] as i in in_$i$) |
        (noneof enumerate [0..3] as i in in_$i$)
```

Response pattern specification

This example makes use of regular expressions.

```
-- Requirement: A connection signal is eventually
--    answered by an ack signal, followed by at least
--    4 data states and a close signal.

assert always (if connection then eventually
               /answer; data*[>=4]; close/)
```

Real-time example

This example uses the timed extension of SALT.

```
-- Requirement: On all floors of a building,
--    the elevator must arrive at most 60s after
--    having been called.

define max_60s_before_open(i) :=
 always (call_$i$ implies
         eventually timed[<=60.0] open_$i$)

assert allof enumerate[1..3] as floor in
             max_60s_before_open(floor)
```

Bibliography

[BLS06] Andreas Bauer, Martin Leucker, and Jonathan Streit. SALT—structured assertion language for temporal logic. Technical Report TUM-I0604, Technische Universität München, Institut für Informatik, March 2006.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[D'S03] D. D'Souza. A logical characterisation of event clock automata. *International Journal of Foundations of Computer Science*, 14(4):625–639, August 2003.

[RS99] Jean-François Raskin and Pierre-Yves Schobbens. The logic of event clocks: decidability, complexity and expressiveness. *Automatica*, 4:247–282, 1999.

[Str06] Jonathan Streit. Development of a programming-language-like temporal logic specification language. Master's thesis, Technische Universität München, 2006.

Index

`*`, 25
`+`, 25
`--`, 13
`->`, 16
`:`, 25
`:=`, 33
`;`, 25
`<->`, 16
`?`, 25
`[ ]`
  with `*`, 25
  with **`timed`**, 30
  with counting quantifiers, 27
  with lists, 35
`$`, 16
`&`, 16
`\\`, 25, 28
`|`, 16
`!`, 16
`@`, 35
`-?`, 7
`-e`, 5
`-f`, 5
`-hs`, 6
`-h`, 7
`-latex`, 6
`-ltl`, 6
`-nonext`, 7
`-nopast`, 6
`-notimed`, 6
`-o`, 5
`-parser`, 5
`-printer`, 6
`-rltl`, 6
`-smv`, 6
`-spin`, 6
`-tltl`, 6
`-v`, 7
`-xtltl`, 6

`accepton`, 24
`after`, 20
`allof`, 36
`always`, 17
`alwaysinpast`, 28
`and`, 16
`as`, 35
`assert`, 13

`before`, 20
`BEGINSALT`, 5
`between`, 20
`betweeninpast`, 28

Comments, 13
Configuration, 4

`declare`, 16
`define`, 33

`else`, 16
Embedded Salt, 5
`ENDSALT`, 5
`enumerate`, 36
`equals`, 16
`eventually`, 17
`eventuallyinpast`, 28
`exactlyoneof`, 36
Examples, 49
`excl`
  with **`until`**, 17
  with **`upto`**, 21
`exclusive`
  with **`until`**, 17
  with **`upto`**, 21

`false`, 15
`from`, 20
`frominpast`, 28

Haskell Interpreter
  GHC, 3
  Hugs, 3

historically, 28
holding, 27
holdinginpast, 28
hs.properties, 4

if, 16
implies, 16
in, 35
incl
    with **until**, 17
    with **upto**, 21
inclusive
    with **until**, 17
    with **upto**, 21
Installation, 3
Iteration, 35

Java Runtime Environment, 3
JAVA_HOME, 3

Layer
    propositional, 15
    temporal, 17
Layers, 14
License, 1
list, 35
Literal, 15

Macros, 33
    binary, 35
    calling, 33
    definition, 33
    explicit calling, 35
    indirect calling, 35
    infix, 35
    nullary, 33
    prefix, 33
    unary, 33

never, 17
neverinpast, 28
next, 17
next weak, 17
nextinpast, 28
nextinpast weak, 28
nextn, 19
nextninpast, 28
noneof, 36
not, 16

occurring, 27
occurringinpast, 28
once, 28
Operators
    counting quantifiers, 27
    exception, 24
    infix, 35
    iteration, 35
    past, 28
    precedences, 14
    prefix, 33
    repetition, 25
    scope, 20
    sequence, 25
    timed, 30
opt
    with **until**, 17
    with **upto**, 21
optional
    with **until**, 17
    with **upto**, 21
or, 16

Parameters, 5
Past, 28
previous, 28
previous weak, 28
previousn, 28
Proposition
    atomic, 15
    declaration, 16
    parameterised, 16
    quoted, 15
    simple, 15

Real time, 30
Regular expressions, 25
rejecton, 24
releases, 17
releasesinpast, 28
req
    with **until**, 17
    with **upto**, 21
    with the argument of **upto**, 22
required
    with **until**, 17
    with **upto**, 21
    with the argument of **upto**, 22

SALT_HOME, 4
Sequences, 25

someof, 36
stutter-invariant, 7

then, 16
timed, 30
TLTL, 30
Translation, 37
  acc, 44
  rej, 45
  $\text{stop}_{\text{excl}}$, 46
  $\text{stop}_{\text{incl}}$, 45
  accepton, 43
  between, 43
  from, 43
  holding, 39
  never, 38
  nextn, 38
  occurring, 39
  rejecton, 43
  releases, 38
  timed, 48
  until, 41
  upto, 42
  iteration, 40
  regular expressions, 43
  repetition operators, 40
triggered, 28
true, 15
Tutorial, 8
Typographical conventions, 2

until, 17
until weak, 17
untilinpast, 28
upto, 20
uptoinpast, 28
Usage, 5

weak
  with **next**, 17
  with **until**, 17
  with **upto**, 21
  with the argument of **upto**, 22
with, 36
without, 36