The Theory and Practice of SALT

Andreas Bauer[1] and Martin Leucker[2]

[1]NICTA Canberra Research Lab and The Australian National University
[2]Institut für Softwaretechnik und Programmiersprachen,
University of Lübeck, Germany

Abstract. SALT is a general purpose specification and assertion language developed for creating concise temporal specifications to be used in industrial verification environments. It incorporates ideas of existing approaches, such as PSL or Specification Patterns, in that it provides operators to express scopes and exceptions, as well as support for a subset of regular expressions. On the one hand side, SALT exceeds specific features of these approaches, for example, in that it allows the nesting of scopes and supports the specification of real-time properties. On the other hand, SALT is fully translatable to LTL, if no real-time operators are used, and to TLTL (also known as state-clock logic), if real-time operators appear in a specification. The latter is needed in particular for verification tasks to do with reactive systems imposing strict execution times and deadlines. SALT's semantics is defined in terms of a translation to temporal (real-time) logic, and a compiler is freely available from the project web site, including an interactive web interface to test drive the compiler. This tutorial paper details on the theoretical foundations of SALT as well as its practical use in applications such as model checking and runtime verification.

1 Introduction

When considering specification language formalisms, we have at least three different characteristics for their classification, which are (i) *expressiveness*, (ii) *conciseness*, and (iii) *readability*. In simple words, expressiveness means, which kind of languages can be defined at all within the considered formalism, while conciseness studies the question, how long the shortest spefications for a given family of languages is. Readability, on the other hand, deals with the question, how *easy* it is, to specify a certain language within the given formalism for a typical human beeing—and is thus a vague, not formal notion. SALT, which is an acronym for *structured assertion language for temporal logic*, aims to be a general purpose specification and assertion language, and was first introduced in [1]. It has been designed especially with *readability* in mind. Thus, one of the main goals of SALT is to offer users, who are not necessarily experts in formal specification and verification, a versatile tool that allows them to express system properties in a formal and concise, yet intelligible manner. In that respect, SALT has the look and feel of a general purpose programming language (e.g., it uses if-then-else constructs, supports (a subset of) regular expressions, and allows the definition

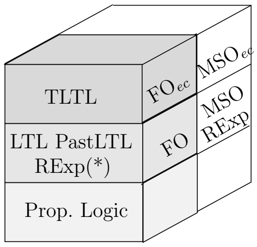

Fig. 1. Relationships between propositional, first-order, and temporal logics

of macros), yet it is fully translatable into standard temporal logics, such as *linear time temporal logic* (LTL [2]) or, if dedicated real-time operators appear inside a specification, to TLTL (also known as *state-clock logic* [3,4]). In other words, the untimed fragment of SALT is equally expressive as LTL, whereas the timed fragment is equally expressive as TLTL. D'Souza has shown in [4] that TLTL corresponds exactly to the first-order fragment of monadic second order logic interpreted over timed words. This resembles the correspondence of LTL and first-order logic over words as shown by Kamp [5]. However, LTL is strictly less expressive than second-order logic over words which, in turn, is expressively equivalent to ω-regular expressions. This also explains why full support of regular expressions is not possible when only LTL-expressible properties are in question (see Figure 1 for an overview).

As such it is possible to employ SALT as a higher level specification front-end to be used with standard model checking or runtime verification tools that otherwise would accept plain LTL formulae with a minimal set of operators as input. As a matter of fact, the freely available SALT compiler[1], which takes as input a SALT specification and returns a temporal logic formula, already supports the syntax of two powerful and commonly used model checking tools, namely SMV [6] and SPIN [7], such that deployment should be relatively straightforward, irrespective of the choice of verification tool.

The emphasis of this paper, however, is less on motivating the overall approach (which was already done in [1]), but rather to give an overview of the main language features, and to demonstrate how it can be used to specify complex temporal properties in a concise and accurate manner. Another important objective, which did not play an important role in [1], is to deepen our understanding of the similarities between SALT and some closely related approaches, in particular

- Dwyer et al.'s frequently cited *specification patterns* [8] which have become part of the Bandera system used to model checking Java programs [9],
- the *Property Specification Language* (PSL [10,11]), which is also a high-level temporal specification language, predominantly used for the specification of integrated circuits, and recently standardised by the IEEE (IEEE-Std 1850^{TM}–2005),

[1] For details, see `http://salt.in.tum.de/` and/or the authors' homepages.

As a matter of fact, parts of the design of SALT were directly influenced by the features existent in these approaches, but the goal was not to create yet another domain-specific tool, but a generic one which like LTL is more or less application-agnostic.

For instance, SALT offers operators to express complex temporal scopes (e.g., by means of `from`, `upto`, `between`, etc.), which is one of the main features underlying the specification patterns. On the other hand, SALT also offers operators to express so called exceptions (by means of `accepton` and `rejecton`), which similarly appear in PSL. While SALT's use of scopes exceeds the possibilities of the specification patterns, in that they allow the nesting of scopes, Sec. 4 will show that the exception operators in SALT are basically equivalent to those of PSL.

The rest of the paper is structured as follows. The next section is a guided tour of the language itself and highlights its main features; as such this section intersects most with work already presented in [1], except for the extended list of practical examples. Sec. 3 provides details on the translation of SALT expressions into LTL, i.e., semantics and implementation. In Sec. 4, we discuss in a more detailed manner the differences and similarities of SALT and other languages, in particular PSL and the Bandera input language, whereas in Sec. 5, we discuss complexity and experimental results of using SALT as a general purpose specification language. Finally, Sec. 6 concludes the paper.

2 Feature Overview

A SALT specification contains one or many assertions that together formulate the requirements associated with a system under scrutiny. Each assertion is translated into a separate LTL/TLTL formula, which can then be used in, say, a model checker or a runtime verification framework. SALT uses mainly textual operators, so that the frequently used LTL formula $\Box(p \rightarrow \Diamond q)$ would be written as

```
    assert always (p implies eventually q).
```

Note that the `assert` keyword precedes all SALT specifications, except metadefinitions such as macros.

The SALT language itself consists of the following three layers, each covering different aspects of a specification:

– *The propositional layer* provides the atomic, Boolean propositions as well as the well-known Boolean operators.
– *The temporal layer* encapsulates the main features of the SALT language for specifying temporal system properties. The layer is divided into a future fragment and a symmetrical past fragment.
– *The timed layer* adds real-time constraints to the language. It is equally divided into a future and a past fragment, similar to the temporal layer.

Within each layer, macros and parameterised expressions can be defined and instantiated by iteration operators, enlarging the expressiveness of each layer into the orthogonal dimension of functions.

As pointed out in the introduction, depending on which layers are used for specification, the Salt compiler generates either LTL or TLTL formulae (resp. with or without past operators). For instance, if only operators from the propositional layer are used, the resulting formulae are purely propositional formulae. If only operators from the temporal and the propositional layer are used, the resulting formulae are LTL formulae, whereas if the timed layer is used, the resulting formulae are TLTL formulae.

2.1 Propositional Layer

Atomic propositions. Boolean propositions are the atomic elements from which Salt expressions are built. They usually resemble variables, signals, or complete expressions of the system under scrutiny. Salt is parameterised with respect to the propositional layer: any term that evaluates to either *true* or *false* can be used as atomic proposition. This allows, for example, propositions to be Java expressions when used for runtime verification of Java programs, or, simple bit-vectors when Salt is used as front end to verification tools like SMV [6].

Usually, every identifier that is used in the specification and that was not defined as a macro or a formal parameter is treated as an atomic proposition, which means that it appears in the output as it has been written in the specification. Additionally, arbitrary strings can be used as atomic propositions. For example,

```
assert always "state!=ERROR"
```

is a valid Salt specification and results in the output (here, in SMV syntax)

```
LTLSPEC G state!=ERROR.
```

However, the Salt compiler can also be called with a customised parser provided as a command line parameter, which is then used to perform additional checks on the syntactic structure of the propositions thus, making the use of structured propositions more reliable.

Boolean operators. The well-known set of Boolean operators $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$ can be used in Salt both as symbols (`!`, `&`, `|`, `->`, `<->`), or as textual operators (`not`, `and`, `or`, `implies`, `equals`). Additionally, the conditional operators `if-then` and `if-then-else`, which have been already mentioned in the introduction, can be used. They tend to make specifications easier to read, because `if-then-else` constructs are familiar to programmers in almost any language. With the help of conditional operators, the introductory example could be reformulated as

```
assert always (if p then eventually q).
```

More so, any such formula can be arbitrarily combined using the Boolean connectives.

2.2 Temporal Layer

The temporal layer consists of a future and a past fragment. Although past operators do not add expressiveness [12], they can help to write formulae that are easier to understand and more efficient for processing [13].

In the following, we concentrate on the future fragment of Salt. The past fragment is, however, completely symmetrical. Salt's future operators are translated using only LTL future operators, and past operators are translated using only LTL past operators. This leaves users the complete freedom as to whether they do or do not want to have past operators in the result. This is useful as not all verification frameworks support both fragments. That said, it would be likewise possible to extend the current compilation process of Salt: The plain Salt compiler translates past operators into LTL with past operators. If the specification should be used say for a verification tool that does not support past operators, a further translation process may be started compiling past formulas to equivalent, possibly non-elementary longer future formulas. If, on the other hand, both future and past operators are supported, either output might be used, depending on how efficiently past operators are supported.

Standard LTL operators. Naturally, Salt provides the common LTL operators U, W, R, $\Box$, $\Diamond$ and $\circ$, written as `until`, `until weak`, `releases`, `always`, `eventually`, and `next`.

Extended operators. Salt provides a number of extended operators that help express frequently used requirements.

– `never`. The `never` operator is dual to `always` and requires that a formula never holds. While this could of course be easily expressed with the standard LTL operators, using `never` can, again, help to make specifications easier to understand.
– Extended `until`. Salt provides an extended version of the LTL U operator. The users can specify whether they want it to be *exclusive* (i. e., in φ U ψ, φ has to hold until the moment ψ occurs) or *inclusive* (i. e., φ has to hold until and during the moment ψ occurs)[2]

  They can also choose whether the end condition is *required* (i. e., must eventually occur), *weak* (i. e., may or may not occur), or *optional* (i. e., the expression is only considered if the end condition eventually occurs).
– Extended `next`. Instead of writing long chains of `next` operators, Salt users can specify directly that they want a formula to hold at a certain step in the future. It is also possible to use the extended `next` operator with an interval, e. g., specifying that a formula has to hold at some time between 3 and 6 steps in the future. Note that this operator refers only to states at certain positions in the sequence, not to real-time constraints.

[2] This has nothing to do with strict or non-strict U: 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.

Counting quantifiers. SALT provides two operators, `occurring` and `holding`, that allow to specify that an event has to occur a certain number of times. `occurring` deals with events that may last more than one step and are separated by one or more steps in which the condition does not hold. `holding` considers single steps in which a condition holds. Both operators can also be used with an interval, e. g., expressing the fact that an event has to occur *at most* 2 times in the future. To express this requirement manually in LTL, one would have to write

$$\neg p \text{ W } (p \text{ W } (\neg p \text{ W } (p \text{ W } \Box \neg p))).$$

The corresponding SALT specification is written concisely as

```
    assert occurring[<=2] p.
```

Exceptions. SALT also includes exception operators, named `rejecton` and `accepton`, which interrupt the evaluation of a formula upon occurrence of an abort condition. `rejecton` evaluates a formula to false if the abort condition occurs and the formula has not been accepted before. For example, monitoring a formula $\Diamond\varphi$ when there has been no occurrence of φ yet would evaluate to false. The dual operator, `accepton`, evaluates a formula to true if it has not been rejected before.

While exceptions do not add expressiveness to the language (i.e., untimed SALT using exceptions is fully translatable to standard LTL), they can be very useful, for example, when specifying a communication protocol that requires certain messages to be sent, but allows to abort the communication at any time by sending a reset message. This would be expressed in SALT as

```
  assert (con_open and next (data until con_close))
     accepton reset.
```

Exceptions also play an important role in the specification and verification of hardware systems. This is why languages such as PSL or ForSpec, which are used in this domain, both include this feature (see Sec. 4).

Scope operators. Many temporal specifications use requirements restricted to a certain *scope*, i. e., they state that the requirement has to hold only before, after, or between some events, and not on the whole sequence [8]. This can be expressed in SALT using the operators `upto` (or `before`), `from` (or `after`) and `between`.

Figure 2 illustrates scopes. It should be clear from the figure that it is mandatory in SALT to specify whether the delimiting events are part of the interval (i. e., *inclusive*) or not (i. e., *exclusive*).

Furthermore, for scope operators it has to be stated whether the occurrence of the delimiting events is strictly required. For example, the following specification

```
    assert p
      between inclusive optional call,
              inclusive optional answer
```

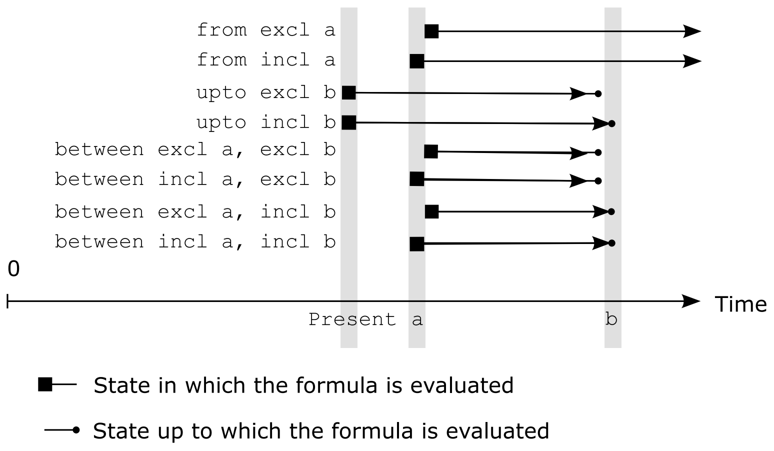

Fig. 2. Scopes of `upto`, `from` and `between`

means that p has to hold within the interval delimited by *call* and *answer*, provided such an interval exists. Without the keyword *optional*, such an interval would be required and within this interval, p must occur.

While it is possible to implement a translation of the `from` operator into LTL relatively straightforward (see Sec. 3), the `upto` operator proves to be more difficult, as can be seen in the following example.

A specification `always` φ `upto` b expresses that φ must always hold until the occurrence of the end condition b. A naïve translation into LTL would be φ W b. This is in order for a purely propositional φ, but might be wrong when temporal operators are used: Consider for example $\varphi :=$ `p -> (eventually s)` yielding the formula $(p \rightarrow \Diamond s) \mathrm{W} b$, intending to say "`p` should be followed by `s` before `b`". The sequence `pbs` is a model for the latter formula, although `s` occurs after the end condition `b`, which clearly violates our intuitions. To meet our intuition, the negated end condition b has to be inserted into the U and $\circ$ statements of φ in various places, e. g., like this: $(p \rightarrow (\neg b \text{ U } (\neg b \wedge s)))$ W b. Dwyer et al. describe this procedure in the notes of their specification pattern system [8]. It is however a tedious and highly error-prone task if undertaken manually.

Salt supports automatic translation by internally defining a stop operator. Using stop, the above example can be formulated as $((p \rightarrow \Diamond s) \text{ stop } b) \mathrm{W} b$ with stop b expressing that $(p \rightarrow \Diamond s)$ shall not take into account states after the occurrence of b. It is then transformed into an LTL expression in a similar way as the `rejecton` and `accepton` operators. Details can be found in Sec. 3.

Regular expressions. Regular expressions are well-known to many programmers. They provide a convenient way to express complex patterns of events, and appear also in many specification languages (see Sec. 4). However, arbitrary regular languages can be defined using regular expressions, while LTL only allows to define so-called *star-free* languages (cf. [14]). Thus, regular expressions have to be restricted in Salt in order to stay translatable to standard LTL. The main operators of Salt regular expressions (SREs) are concatenation (`;`), union (`|`)

and Kleene-star operators (*), but no complement. The argument of a Kleene-star operator is required to be a propositional formula. The advantage of this operator set—in contrast to the usual operator set for star-free regular expressions, which contains concatenation, union and complement—is that it can be translated more efficiently into LTL.

Salt provides further SRE operators that do not increase the expressiveness, but, arguably, make dealing with expressions more convenient for users. For example, the overlapping sequence operator `:` states that one expression follows another one, overlapping in one step. The `?` and `+` operators (optional expression and repetition at least once) are common extensions of regular expressions. Moreover, there are a number of variations of the Kleene-star operator such as `*[=n]` to express how many steps from now the argument has to consecutively hold, `*[>n]` (resp. `*[>n]`) to express a minimum (resp. maximum) bound on the consecutive occurrence of the argument, `*[n..m]` to express an exact bound, etc. All these operators, however, have to adhere to the same restriction as the standard Kleene-star operator; that is, their argument needs to be a propositional formula.

While traditional regular expressions match only finite sequences, a Salt regular expression holds on an (infinite) sequence if it matches a finite prefix of the sequence.

Finally, with the help of regular expressions, we can rewrite the example using exception operators as

```
assert /con_open; data*; con_close/ accepton reset.
```

2.3 Timed Layer

Salt contains a timed extension that allows the specification of real-time constraints. Timed operators are translated into TLTL [3,4], a timed variant of LTL.

Timing constraints in Salt are expressed using the modifier `timed[`$\sim$`]`, which can be used together with several untimed Salt operators in order to turn them into 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 $\sim c$. This corresponds to the operator $\rhd_{\sim c}\varphi$ in TLTL.
- 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 $\sim c$. The extended variants of `until` can be used as timed operators as well.
- `always timed[`$\sim c$`]` φ
  states that φ must always be true within the time bounds $\sim c$.
- `never timed[`$\sim c$`]` φ
  states that φ must never be 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 $\sim c$.

2.4 Macros and Parameterised Expressions

SALT allows user-defined sub-expressions as *macros* and to parameterise macros and sub-expressions. Macro definitions do not begin with the `assert` keyword. They can be called in the same way as built-in SALT operators. Within certain limits, this allows the user to extend the SALT language using their own operators. For example, the following macro is called in infix notation:

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

Iteration operators allow to instantiate a parameterised sub-expression or macro with a list of values provided by the user. For example, the following specification states that either `a` or `!b` or `c` must hold forever.

```
assert someof list [a, !b, c] as i in always i
```

Parameters defined in a macro or an iteration expression can also be used to parameterise Boolean variables, as in the following example, which states that exactly one of the four variables, `state_1`, `state_2`, `state_3` and `state_4`, must be true.

```
assert exactlyoneof enumerate[1..4] as i in state_$i$
```

Macros can help to make a specification easier to understand, because complicated sub-expressions can be transparently hidden from the user, and accessed via an intuitive name that explains what the expression actually stands for. Sub-expressions that are used several times have to be written down only once.

2.5 Further Examples

In this section, a concluding look at some more SALT specifications is taken, and their corresponding LTL versions examined. The examples are mostly borrowed from the survey presented in [8], except where indicated otherwise. Note that propositions appearing in the specifications are not necessarily marked as such and are denoted in plain text only, indicating their intuitive meaning wrt. the respective application.

1. The requirement that a system should operate until a queue of jobs is either empty, or an abort signal issued can be formulated in LTL as

$$\neg((\neg(\mathit{queuelength} == 0 \vee \mathit{abort}))\ \mathrm{U}\ (\neg \mathit{working} \wedge (\neg(\mathit{queuelength} == 0 \vee \mathit{abort})))).$$

   The accompanying SALT specification would be:

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

   where `abort` is a proposition, and not the abort operator.

2. To specify idle behaviour, the following LTL specification could be used:

$$\Box(\neg \mathit{return\_Execute} \vee (\mathit{return\_Execute} \wedge ((\Diamond \mathit{call\_Execute}) \Rightarrow$$
$$(\neg(\neg \mathit{call\_Execute} \ \mathrm{U}\ (\mathit{call\_doWork} \wedge \neg \mathit{call\_Execute})))))).$$

   It asserts that between the moment in which an execution completes, and before a new one begins, there is no work done. In SALT, this example would be written as:

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

3. Coming back to an example from the area of protocol specification, one might assert that an answer was immediately preceded by a request. In LTL this would be written as:

$$\Box(\mathit{answer} \Rightarrow (\ocircle \mathit{request})).$$

   Using a macro, in SALT, `precedes` can be expressed as follows:

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

4. A system with n input channels, may be using at most one at a time. Given that $n = 4$, this simple requirement would require

$$\Box(((\mathit{in\_0} \wedge (\neg(\mathit{in\_1} \vee (\mathit{in\_2} \vee \mathit{in\_3}))))\vee$$
$$((\mathit{in\_1} \wedge (\neg(\mathit{in\_0} \vee (\mathit{in\_2} \vee \mathit{in\_3}))))\vee$$
$$((\mathit{in\_2} \wedge (\neg(\mathit{in\_0} \vee (\mathit{in\_1} \vee \mathit{in\_3}))))\vee$$
$$(\mathit{in\_3} \wedge (\neg(\mathit{in\_0} \vee (\mathit{in\_1} \vee \mathit{in\_2})))))))\vee$$
$$(\neg(\mathit{in\_0} \vee (\mathit{in\_1} \vee (\mathit{in\_2} \vee \mathit{in\_3})))))$$

   if specified in LTL. The shorter SALT specification appears to be less error-prone and more readable:

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

5. To show that regular expressions can be very useful for specification purposes, in the following it is expressed that a connection signal is eventually answered by an acknowledgement, followed by at least four data packets and a close signal. Again, this is first examined in LTL:

$$\Box(\mathit{connection} \Rightarrow$$
$$(\Diamond(\mathit{answer} \wedge (\ocircle(\mathit{data}\ \mathrm{U}\ (\mathit{data} \wedge$$
$$(\ocircle(\mathit{data} \wedge (\ocircle(\mathit{data} \wedge (\ocircle(\mathit{data} \wedge (\ocircle\ \mathit{close})))))))))))).$$

Now, consider the SALT counterpart using a regular expression:

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

6. Consider an elevator: A possible requirement could be that 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. In SALT, this can be specified as:

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

7. This section is now concluded by extending this example further and thus, showing most of SALT's features in one use-case. The following specification describes the following behaviour: On all three floors in a building, calling the elevator at floor i implies that it may pass at most two times at that floor without opening its doors, and that it must finally open its doors at that floor within 60 seconds.

```
define max_twice_at_floor_before_open(i) :=
 always (occurring[<=2] atfloor_$i$
             between inclusive optional call_$i$,
                    exclusive optional open_$i$)

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_twice_at_floor_before_open(floor)
         and max_60s_before_open(floor)
```

The modifiers `optional` in the `between`-statement make sure that `atfloor_`i is only checked provided `call_`i occurs.

Note that the equality between the LTL specifications in the above examples and their SALT counterparts, was established using the model checker SMV. For this purpose the SALT specifications were first compiled into plain LTL using the SALT compiler and then compared with the manually written requirements.

3 Semantics

SALT comes with a precisely defined semantics. It can be translated into either LTL or TLTL; the latter only when timed operators are used in a specification. Therefore, we define the semantics of SALT's operators by means of their corresponding LTL or respectively TLTL formulae.

More precisely, we define a translation function $\mathcal{T}$ to translate a valid Salt specification ψ into a temporal logic formula $\mathcal{T}(\psi)$, and define that an infinite word w over a finite alphabet of actions satisfies ψ iff $w \models \mathcal{T}(\psi)$ (using the standard satisfaction relation $\models$ defined for LTL/TLTL [15]).

For brevity, we exemplify the translation on a few selected operators only and refer to the extensive language reference and manual available from Salt's homepage at `http://salt.in.tum.de/` for the remaining cases.

In what follows, let ψ, φ, and φ' denote Salt specifications. Many of Salt's operators can be considered as simple syntactic sugaring and are easily translated to LTL. For example, $\mathcal{T}(\varphi\ \texttt{or}\ \varphi'))$ is translated inductively to $\mathcal{T}(\varphi) \vee \mathcal{T}(\varphi')$. The aforementioned `accepton` operator, which adds an exception to a specification is inductively defined as follows:

$$\begin{aligned}
\mathcal{T}(b\ \texttt{accepton}\ a) &= b \vee a\\
\mathcal{T}((\neg\varphi)\ \texttt{accepton}\ a) &= \neg\mathcal{T}(\varphi\ \texttt{rejecton}\ a)\\
\mathcal{T}((\varphi \wedge \psi)\ \texttt{accepton}\ a) &= \mathcal{T}(\varphi\ \texttt{accepton}\ a) \wedge \mathcal{T}(\psi\ \texttt{accepton}\ a)\\
\mathcal{T}((\varphi \vee \psi)\ \texttt{accepton}\ a) &= \mathcal{T}(\varphi\ \texttt{accepton}\ a) \vee \mathcal{T}(\psi\ \texttt{accepton}\ a)\\
\mathcal{T}((\varphi\ \mathrm{U}\ \psi)\ \texttt{accepton}\ a) &= \mathcal{T}(\varphi\ \texttt{accepton}\ a)\ \mathrm{U}\ \mathcal{T}(\psi\ \texttt{accepton}\ a)\\
\mathcal{T}((\circ\varphi)\ \texttt{accepton}\ a) &= (\circ\mathcal{T}(\varphi\ \texttt{accepton}\ a)) \vee a\\
\mathcal{T}((\Box\varphi)\ \texttt{accepton}\ a) &= \neg(\neg a\ \mathrm{U}\ \neg\mathcal{T}(\varphi\ \texttt{accepton}\ a))\\
\mathcal{T}((\Diamond\varphi)\ \texttt{accepton}\ a) &= \Diamond\mathcal{T}(\varphi\ \texttt{accepton}\ a),
\end{aligned}$$

Whereas the `rejecton` operator, which is used in the above definition, is given in terms of:

$$\begin{aligned}
\mathcal{T}(b\ \texttt{rejecton}\ r) &= b \wedge \neg r\\
\mathcal{T}((\neg\varphi)\ \texttt{rejecton}\ r) &= \neg\mathcal{T}(\varphi\ \texttt{accepton}\ r)\\
\mathcal{T}((\varphi \wedge \psi)\ \texttt{rejecton}\ r) &= \mathcal{T}(\varphi\ \texttt{rejecton}\ r) \wedge \mathcal{T}(\psi\ \texttt{rejecton}\ r)\\
\mathcal{T}((\varphi \vee \psi)\ \texttt{rejecton}\ r) &= \mathcal{T}(\varphi\ \texttt{rejecton}\ r) \vee \mathcal{T}(\psi\ \texttt{rejecton}\ r)\\
\mathcal{T}((\varphi\ \mathrm{U}\ \psi)\ \texttt{rejecton}\ r) &= \mathcal{T}(\varphi\ \texttt{rejecton}\ r)\ \mathrm{U}\ \mathcal{T}(\psi\ \texttt{rejecton}\ r)\\
\mathcal{T}((\circ\varphi)\ \texttt{rejecton}\ r) &= (\circ\mathcal{T}(\varphi\ \texttt{rejecton}\ r)) \wedge \neg r\\
\mathcal{T}((\Box\varphi)\ \texttt{rejecton}\ r) &= \Box\mathcal{T}(\varphi\ \texttt{rejecton}\ r)\\
\mathcal{T}((\Diamond\varphi)\ \texttt{rejecton}\ a) &= \neg r\ \mathrm{U}\ \mathcal{T}(\varphi\ \texttt{rejecton}\ r).
\end{aligned}$$

However, not all Salt operators translate in such a straightforward inductive manner, since their translation depends on what is defined by the according subformulae occurring in a given expression. To guide the translation process for such operators, we have introduced an artificial or helper operator, `stop`, which is inductively defined as follows:

$$
\begin{aligned}
\mathcal{T}(b \text{ stop}_{\text{excl}}\ s) &= b\\
\mathcal{T}((\neg\varphi) \text{ stop}_{\text{excl}}\ s) &= \neg\mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s)\\
\mathcal{T}((\varphi \wedge \psi) \text{ stop}_{\text{excl}}\ s) &= \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s) \wedge \mathcal{T}(\psi \text{ stop}_{\text{excl}}\ s)\\
\mathcal{T}((\varphi \vee \psi) \text{ stop}_{\text{excl}}\ s) &= \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s) \vee \mathcal{T}(\psi \text{ stop}_{\text{excl}}\ s)\\
\mathcal{T}((\varphi \text{ U } \psi) \text{ stop}_{\text{excl}}\ s) &= (\neg s \wedge \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s)) \text{ U } (\neg s \wedge \mathcal{T}(\psi \text{ stop}_{\text{excl}}\ s))\\
\mathcal{T}((\varphi \text{ W } \psi) \text{ stop}_{\text{excl}}\ s) &= \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s) \text{ W } (s \vee \mathcal{T}(\psi \text{ stop}_{\text{excl}}\ s))\\
\mathcal{T}((\circ\varphi) \text{ stop}_{\text{excl}}\ s) &= \circ(\neg s \wedge \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s))\\
\mathcal{T}((\circ_W\varphi) \text{ stop}_{\text{excl}}\ s) &= \circ(s \vee \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s))\\
\mathcal{T}((\Box\varphi) \text{ stop}_{\text{excl}}\ s) &= \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s) \text{ W } s\\
\mathcal{T}((\Diamond\varphi) \text{ stop}_{\text{excl}}\ s) &= (\neg s) \text{ U } (\neg s \wedge \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s))
\end{aligned}
$$

where b denotes an atomic proposition from the action alphabet and s an arbitrary formula, possibly atomic also.

Thus, `stop` selects certain aspects of a formula, and in $\psi \equiv \varphi_1$ `stop` φ_2, intuitively asserts that the validity of ψ does not depend on events occurring after φ_2 has occurred. Again, for brevity, we consider only the exclusive variant of `stop` and only for the future fragment of SALT. The past fragment and inclusive semantics, however, are each symmetrical.

The more complicated scope operator `upto`, which was discussed earlier in Sec. 2.2, and whose translation depends on `stop`, is then defined as:

$$
\begin{array}{lll}
\mathcal{T}(\varphi \,\texttt{upto excl req}\, b) & = & \\
\quad\quad \text{if } \mathcal{T}(\varphi) = \Box\psi\text{:} & & (\psi \text{ stop}_{\text{excl}}\ b) \text{ U } b\\
\quad\quad \text{if } \mathcal{T}(\varphi) = \neg\Diamond\psi\text{:} & & (\neg\psi \text{ stop}_{\text{excl}}\ b) \text{ U } b\\
\quad\quad \text{else:} & & (\Diamond b) \wedge (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b)\\
\mathcal{T}(\varphi \,\texttt{upto excl opt}\, b) & = & \\
\quad\quad \text{if } \mathcal{T}(\varphi) = \Diamond\psi\text{:} & & \neg((\neg\psi \text{ stop}_{\text{excl}}\ b) \text{ U } b)\\
\quad\quad \text{else:} & & (\Diamond b) \rightarrow (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b)\\
\mathcal{T}(\varphi \,\texttt{upto excl weak}\, b) & = & (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b)\\
\mathcal{T}(\texttt{req}\,\varphi \,\texttt{upto excl req}\, b) & = & \\
\quad\quad \text{if } \mathcal{T}(\varphi) = \Box\psi\text{:} & & \neg b \wedge ((\psi \text{ stop}_{\text{excl}}\ b) \text{ U } b)\\
\quad\quad \text{if } \mathcal{T}(\varphi) = \neg\Diamond\psi\text{:} & & \neg b \wedge ((\neg\psi \text{ stop}_{\text{excl}}\ b) \text{ U } b)\\
\quad\quad \text{else:} & & (\Diamond b) \wedge \neg b \wedge (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b)\\
\mathcal{T}(\texttt{req}\,\varphi \,\texttt{upto excl opt}\, b) & = & \\
\quad\quad \text{if } \mathcal{T}(\varphi) = \Diamond\psi\text{:} & & \neg((\neg\psi \text{ stop}_{\text{excl}}\ b) \text{ U } b)\\
\quad\quad \text{else:} & & (\Diamond b) \rightarrow (\neg b \wedge (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b))
\end{array}
$$

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

where, of course, $\mathrm{stop}_{\mathrm{excl}}$ and $\mathrm{stop}_{\mathrm{incl}}$ are references to the exclusive and inclusive variants of `stop`, respectively.

Similar translation schemes are defined for the remaining operators' semantics, which are detailed in the SALT language reference and manual.

4 Comparison with Existing Approaches

As already mentioned in the introduction, the design of SALT is influenced by a number of existing (domain-specific) specification languages, in particular PSL and specification patterns. In order to see the differences between these approaches and SALT, besides their domain-specifitivity, we go again through the main list of SALT features and discuss similarities and differences between the approaches.

4.1 Overview

Like SALT, the Property Specification Language PSL [11] is a high level specification language, but predominantly used in the area of integrated circuit design. The initial version of PSL, which underwent standardisation by the IEEE, became available in March 2003, whereas the latest version, version 1.1, became available in April 2004. The PSL standard incorporates concepts and ideas of various other specification languages, such as ForSpec, which has been developed at Intel and been donated to Accellera in order to facilitate the then ongoing standardisation efforts (cf. [16]). PSL also comprises different layers, (i) a Boolean, (ii) temporal, (iii) verification, and (iv) modelling layer. The Boolean layer is much like SALT's propositional layer, and available in different flavours, depending on the concrete application domain; for instance, there exists a VHDL flavour which means that Boolean connectives follow roughly the same syntax used in

VHDL, which is a standard hardware specification language and simulation environment (cf. [17]). The temporal layer, in turn, is divided into two separate languages: the *foundation language* and the *optional branching extension*, with the main difference being that the former employs a linear model of time, and the latter a branching one. Therefore, in the following comparison with SALT, we focus mainly on the foundation language, although some features are the same for both languages. The foundation language, basically, consists of

- the usual Boolean connectives,
- future-time LTL operators,
- clocking operators,
- Sequential Extended Regular Expressions (SEREs), as well as
- an `abort` operator to model exceptions.

The verification layer consists of directives which describe how the temporal properties should be used by verification tools. Unlike SALT, the `assert` keyword is part of the verification layer, and not inherent to all specifications alike. Instead of `assert` it is also possible to `assume` that a specification holds, or to check if certain parts of a trace are `cover`ed by an SERE. As such, the verification layer instructs an employed verification tool how to treat the specification. Note that SALT specifications, basically, always use `assert`, except to define macros. Finally, the modelling layer of PSL is used to introduce domain-specific modelling constructs, e.g., in a VHDL-flavour or other hardware description language, to model directly the behaviour of hardware designs, and therefore augment what is possible using PSL alone. For example, it can be used to calculate an expected value of an output or use custom data structures, but then typically exceeding what is expressible using ω-regular languages alone. In fact, there exists currently no accepted formal semantics for PSL's modelling layer (cf. [18]). However, when augmented models are merely used for simulation, e.g., to compare the observed behaviour with a specified one (runtime verification), then such formal properties of the system are of no concern as a runtime verification toolkit would not care *how* a trace has been generated. SALT, on the other hand, aiming to be used as a general specification front-end, rather than a system modelling tool does not currently offer the integration of third-party languages as a means of extension.

Like PSL, the Bandera system [19,20] is also a domain-specific tool, in that it targets the Java language as platform to perform software model checking on. Dwyer et al.'s specification patterns [8] have been adopted by the Bandera Specification Language (BSL), which has a compiler to translate high-level specifications to LTL. Basically, Dwyer et al. analysed ca. 600 real-world specifications in order to identify common patterns among them [8]. These patterns were then formalised, and formed the foundation of their well-known *specification patterns*. Conceptually, specification patterns are similar to design patterns in software engineering [21]; that is, a pattern provides a solution to a recurring problem, often including notes about its advantages, drawbacks, and alternatives. As such it enables inexperienced users to reuse expert knowledge. The specification patterns themselves consist of *requirements*, such as "absence" (i. e., a condition is false) or "response" (i. e., an event triggers another one), that can be expressed under different *scopes*, like “globally”, “before an event r”, “after an event q”, or “between two events r and q”. Similarly to PSL and Salt, BSL consists of layers: the *assertion property specification layer* allows developers to define constraints on program contexts, whereas the *temporal property specification layer* provides support for temporal properties.

4.2 A Comparison of Features

In what follows, we go through a list of core features of the Salt language, as presented in Sec. 2, and discuss how the other specification languages mentioned above realise them, if at all. Since this discussion is guided by the features existing in Salt, it is not meant to distill a single best approach, but rather to show where the similarities and the differences are between all three languages. An objective comparison is also difficult because BSL and PSL each are optimised for a different purpose, namely hardware design/verification and software model checking.

Extended operators. Salt's extended operators aim at providing a richer set of LTL-like primitives, e.g., such as `never` as opposed to the frequently used LTL-operator `always`. PSL also has the `never` operator and its own equivalent operators to Salt's different versions of the `next` and `until` operators. BSL, on the other hand, discourages the use of low-level LTL primitives in favour of high-level patterns and scopes. Hence it does not provide these operators although it is easy to express them in terms of the standard LTL operators.

Scopes. Scopes have been identified by Dwyer et al. as an important issue in the specification pattern system. However, their pattern system is restricted to predefined requirements. That is, it does not allow nested scopes, and by default only certain combinations of inclusive/exclusive and required/optional delimiters. Some—but by far not all—scopes can also be expressed in PSL using the `next_event` and different variants of `before` operators. Salt's distinguishing feature here is that scope operators can be used with arbitrary formulae, even with nested scope operators as in the following example:

```
assert weak e between inclusive optional
  (eventually (required a before exclusive required b)
    from exclusive optional c),
  exclusive required d
```

Here, the outer-most scope is a `between`, which uses a `from` scope which, in turn, uses a `before` scope as one of its arguments. Admittedly, the example is hard to read and rather artificial, but it does highlight this particular feature of Salt in a very obvious manner.

Exceptions. Interestingly one of the main changes between versions 1.0 and 1.1 of PSL, besides precedence ordering, is the treatment of PSL's `abort` operator, and whose Salt counterpart are the operators `accepton` and `rejecton`.

The reason for this change is described in [22]. In this paper, Armoni et al. describe their discovery that the original definition of the `abort` operator would cause, in the worst-case, a non-elementary blow-up when translating a specification into an alternating Büchi automaton. In essence, this meant that PSL, as it was defined in version 1.0, could render subsequent formal verification an unnecessarily difficult if not impossible task, as the performance of most such tools, which use temporal specifications, directly depends on the size of the resulting automata representations. This problem has been addressed by basically adopting the semantics of a similar language wrt. this operator, called ForSpec [23], and which has been mainly developed at Intel and donated to Accellera in 2003. ForSpec also offers exception operators, called `accept` and `reject`, and specifications are translatable into a logic termed Reset-LTL in [22]. Although Reset-LTL contains two additional operators when compared to Pnueli's LTL, the two languages are actually equally expressive [22]. At this stage we only give an intuitive semantics of Reset-LTL, and refer the reader to the Appendix for a formal account.

An infinite word w at position i over some alphabet is said to satisfy a Reset-LTL formula, φ, if $\langle w^i, \text{false}, \text{false}\rangle \models \varphi$ holds. But unlike in standard LTL, this satisfaction relation is not only defined between an infinite word and a formula, but also between two additional Boolean formulae, which capture the exception conditions for the `accept` and `reject` operators, respectively. Let us refer to the former by a and the latter by r. Initially, when evaluating a formula, false and false are used for a and r, and the definition of the semantics (see Appendix) ensures that during evaluation, it is not possible for a and r to be true at the same time. That is, in some relation $\langle w^i, a, r\rangle \models \varphi$, if a is satisfied in state w_i then the entire word w^i is a model, irrespective of whether or not φ is satisfied by w^i using the standard LTL semantics. On the other hand, if r is satisfied, then w^i is not a model, irrespective of whether or not φ is satisfied using standard LTL semantics. As such a and r are, indeed, exception conditions, and set to a value other than false by the definition of the `accept` and `reject` operators above. What is interesting to note is that SALT's exception operators `accepton` and `rejecton` are, in fact, compatible with Reset-LTL's exception operators in the following sense.

Theorem 1. *The following relationship holds between the Reset-LTL operators* `abort` *and* `reject` *and the* SALT *operators* `accepton` *and* `rejecton`*:*

$$\langle w^i, \text{false}, \text{false}\rangle \models \texttt{accept}\ e\ \texttt{in}\ \phi \textit{ if and only if } w^i \models \phi\ \texttt{accepton}\ e,$$

and

$$\langle w^i, \text{false}, \text{false}\rangle \models \texttt{reject}\ e\ \texttt{in}\ \phi \textit{ if and only if } w^i \models \phi\ \texttt{rejecton}\ e.$$

Again, for a formal proof of this statement, see the Appendix.

Note also that although PSL adheres to Reset-LTL semantics with respect to the two exception operators, it has adopted its own keyword (`abort`) and, unlike SALT's exception operators which are defined in a mutually recursive manner in

Sec. 3, uses a direct definition, expressed in terms of two "helper" symbols, $\top$ and $\perp$, instead of the two Boolean context formulae as in Reset-LTL. These helper symbols are not part of the underlying alphabet. Basically, the symbol $\top$ is such that everything holds on it, including false, and $\perp$ is such that nothing holds on it, including true. As the two semantic definitions for the exception operators are expressively equivalent, we abstain from giving further details at this point, but the interested reader may refer to [22,24] and [10, §B2.1.1.2].

Regular expressions. As pointed out in Sec. 2, Salt supports a subset of regular expressions, which is translatable to LTL. Note that as is the case with PSL, Salt regular expressions (SREs) do not offer complementation as an operator. The reason being not to restrict expressiveness, but the fact that arbitrary use of complementation in a specification can lead to exponentially larger LTL formulae in the translation. It is, however, possible to negate the language an expression defines by using `not` as can be seen in the example already employed in Sec. 2:

```
assert not /con_open; data*; con_close/
  accepton reset
```

As SEREs form a superset of SREs (modulo a different semantics, e.g., SEREs are typically enclosed by brackets instead of slashes), the above is also a valid PSL expression. Salt basically supports the same repetition operators as SEREs do, but with further restrictions on their arguments to not allow specifications that would otherwise exceed the expressiveness of star-free languages, and thus LTL:

- The argument of `*`, `*[>`n`]`, `*[>=`n`]` and `+` has to be a propositional formula.
- All expressions except for the last in an SRE must be either Boolean propositions, or they must be other SRE combined by `|`. No other Boolean connectives are allowed for the combination of SRE (although they can be used to form propositional 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.

Other operators, like the overlapping sequence operator ("`:`") are also inspired from SEREs, and its semantics defined accordingly.

To the best of our knowledge, BSL does not currently offer any kind of support for regular expressions.

Real-time support. As also pointed out in Sec. 2, Salt has dedicated support for real-time specifications, in that temporal operators can be enriched with discrete timeouts as is shown in the last example in Sec. 2.5. Recall that all specifications employing real-time directives are translated into TLTL, and although the timing constraints that appear in a Salt specification can only be discrete, TLTL's underlying model of time is continuous [4]. TLTL basically enriches standard LTL with two operators, each accepting a discrete value as argument: one operator is used to express when a proposition was true in the past, and the other one to express when it will be true in the future. Based on these operators, it is easy to derive time-bounded variants of the typical LTL modalities.

Neither PSL nor BSL currently offer real-time support in the above sense. However, PSL supports clocked expressions using the "`@`" operator, which can be appended to unclocked expressions, similarly as time-bounds in SALT can be appended to untimed expressions. Clocks, however, are not used to model real-time, but to match (parts of) expressions with different parts of the clock cycles of the hardware system under scrutiny. For example, `@rose` defines that something has to hold on a rising edge, `@negedge` on a negative edge, and so on. As such, PSL adopts a hardware designer's point of view. SALT on the other hand adopts, more or less, a purely behavioural point of view, in that the intention is not to let users model the actual implementation of an event-driven real-time system, but its abstract behaviour. Arguably, a continuous model of time, as is offered by TLTL and discrete time-outs, are an adequate language to achieve this goal.

Macros and parameterised expressions. In comparison to SALT, PSL's macro definition capabilities are more akin to C or C++'s preprocessor. PSL defines the well-known directives for `#define`, `#ifdef`, `#undef`, etc. which behave in the expected way. In addition it offers two less common directives, `%for` and `%if`, whose semantics can be explained as follows. The `%for` directive replicates something a number of times. The syntax is as follows:

```
// using a range
%for var in expr1 .. expr2 do
...
%end
// using a list
%for var in { item1 , item2, ... , itemN } do
...
%end
```

where `var` is an identifier, `expr1` and `expr2` are statically computable expressions, and `item1`, `item2` etc. are either a number or a simple identifier. In the first case the text inside the `%for...%end` pairs will be replicated $expr2 - expr1 + 1$ times (assuming that $expr2 \geq expr1$). In the second case the text will be replicated according to the number of items in the list (cf. [11, §8.5]). The following PSL macro definition using `%for`

```
%for ii in 0..3 do
assign aa[ii] = ii > 2;
%end
```

is therefore equivalent to this slightly longer piece of PSL code:

```
assign aa[0] =  0 > 2;
assign aa[1] =  1 > 2;
assign aa[2] =  2 > 2;
assign aa[3] =  3 > 2;
```

As such, the `%for` directive is PSL's counterpart to SALT's enumeration operator, whereas `%if` is similar to the `#ifdef` construct known from C/C++. However

Table 1. Comparison of Salt language features with those of other specification languages

	Ext. ops	Scopes	Exceptions	Reg. exp.	Real-time	Macros	Iterators
Salt	●	●	●	◐	●	●	●
PSL	●	◐	●	●	○	●	●
BSL	○	●	○	○	○	◐	●

`%if` must be preferred over `#ifdef` when the condition refers to variables defined in an encapsulating `%for`. For further details, refer to [11,10].

While BSL doesn't directly support macros in the above sense, it has a rich and powerful *assertion language* as well as *predicate definition sublanguage*. While neither offers an if-then-else construct, the assertion language lets users define assertions of the form of C's conditional operator ":", which is also part of C++, Java, and other languages, e.g., as in `a? b : c`, which is equivalent to `if a then b else c`. Also, assertions in BSL define static properties, in that they are Boolean conditions which can be checked at certain control-flow points throughout the execution of a Java program, such as method entry and return. However, Salt's parameterised expressions (and as such PSL's `%for` operator) have a match in BSL. Consider, for example, the following excerpt from a specification given in [19]:

```
FullToNonFull: forall[b:BoundedBuffer].
 {Full(b)} leads to {!Full(b)} globally
```

which is translated into a parameterised specification, which during verification is instantiated accordingly by all objects of type `BoundedBuffer`:

$$\Box(\texttt{Full(b)} \rightarrow \circ(\neg\texttt{Full(b)})).$$

Obviously, this form of parameterisation using type information is geared towards the verification of Java programs.

Summary. As an overview, we present a brief summary of our findings in the form of a table in Table 1.

4.3 Further Related Work

EAGLE [25], is a temporal logic with a small but flexible set of primitives. The logic is based on recursive parameterised equations with fix-point semantics and merely three temporal operators: next-time, previous-time, and concatenation. Using these primitives, one can construct the operators known from various other formalisms, such as LTL or regular expressions. While EAGLE allows the specification of real-time constraints, it lacks most high level constructs such as nested scopes, exceptions, counting quantifiers currently present in Salt.

Duration calculus [26] and similar interval temporal logics overcome some of the limitations of LTL that we mentioned. These logics can naturally encode past operators, scoping, regular expressions, and counting. However, it is unclear how to translate specifications in these frameworks to LTL such that standard model checking and runtime verification tools based on LTL can be employed.

Notably, [27] describes a symmetric approach by providing a more low-level and formal framework in which the various different aspects of different temporal logics can be expressed. The observational mu-calculus is introduced as an "assembly language" for various extensions of temporal logic. In a follow-up paper [28], first results from an integration of the observational mu-calculus into the *Object Constraint Language* (OCL), which also forms part of the UML are described. However, the goal of this work was not to provide a more rich and natural syntax, but rather a sufficient set of temporal operators.

5 Realisation and Results

Specification languages like Salt, PSL, BSL, etc. aim at offering as many convenience operators to users as possible, in order to make the specifications more concise, thus readable, and the task of specification ultimately less error-prone. However, increased conciseness often comes at a price, namely that the complexity of these formalisms increases. Although, to the best of our knowledge, there does not exist a complexity result for PSL's satisfiability problem, there exist results for LTL and specific SERE features: While LTL is known to be PSpace-complete, it turns out that adding even just a single operator of the ones offered by SEREs makes the satisfiability problem at least ExpSpace hard [29]. On the other hand in [22] it is noted, that Reset-LTL, which we have used to express PSL's exception operators in Sec. 4 is only PSpace-complete. As, due to Theorem 1 we can easily create a Reset-LTL formula for every untimed Salt specification that uses only the LTL operators, extended operators, and exceptions, it follows that this fragment is also in PSpace. In fact, due to the PSpace-completeness of LTL, it follows that this fragment is PSpace-complete.

The situation is different when we consider the complete untimed fragment of Salt. In particular, it contains a variant of the $\circ$-operator, as in $\texttt{nextn}[n]\varphi$, which states that φ is required to hold n steps from now in the future. It was pointed out in [30], that the succinctness gains of this operator alone push the complexity of a logic up by one exponent as the formula $\texttt{nextn}[2^n]$ is only of length $O(n)$. This is the same argument used in [23] to explain ExpSpace-hardness of FTL, the logic underlying ForSpec. We thus get:

Theorem 2. *The untimed* Salt *fragment consisting of LTL-, extended-, and abort-operators is PSpace-complete. By adding the* `nextn` *operator, one obtains an ExpSpace-complete fragment.*

Note that we currently have no similar result for full Salt as it would require analysing many more features than the ones above. In fact, to the best of our knowledge, there does not exist a similar result for PSL, despite the fragments considered in [29], presumably for the same reason.

5.1 Experimental Results

We have implemented our concepts in terms of a compiler for the Salt language. The compiler front end is currently implemented in Java, while its back end, which also optimises specifications for size, is realised via the functional programming language Haskell. Basically, the compiler's input is a Salt specification and its output a temporal logic formula. Like with programming languages, compilation of Salt is done in several stages. First, user-defined macros, counting quantifiers and iteration operators are expanded to expressions using only a core set of Salt operators. Then, the Salt operators are replaced by expressions in the subset Salt--, which contains the full expressiveness of LTL/TLTL as well as exception handling and stop operators. The translation from Salt-- into LTL/TLTL is treated as a separate step since it requires weaving the abort conditions into the whole subexpression. The result is an LTL/TLTL formula in form of an abstract syntax tree that is transformed easily into concrete syntax via a so-called *printing function*. Currently, we provide printing functions for SMV [6] and SPIN [7] syntax, but the users can easily provide additional printing functions to support their tool of choice. The use of optimised, context-dependent translation patterns as well as a final optimisation step performing local changes also help reducing the size of the generated formulae.

As the time required for model checking depends exponentially on the size of the formula to check, efficiency was an important issue for the development of Salt and its compiler. Because of the arguments presented in the discussion above, one might suspect that generated formulae are necessarily bigger and less efficient to check than handwritten ones. But our experiments show that the compiler is doing a good job of avoiding this worst-case scenario in practice.

In order to quantify the efficiency of the Salt compiler, existing LTL formulae were compared to the formulae generated by the compiler from a corresponding Salt specification. This was done for two data sets: the specification pattern system [8] (50 specifications) and a collection of real-world example specifications, mostly from the Dwyer's et al.'s survey data [8] (26 specifications). The increase or decrease of the formula was measured using the following parameters:

BA [Fri]: Number of states of the Büchi automaton (BA) generated using the algorithm proposed by Fritz [31], which is one of the best currently known. This is probably the most significant parameter, as a BA is usually used for model checking, and the duration of the verification process depends highly on the size of this automaton.

BA [Odd]: Number of states of the BA generated using the algorithm proposed by Oddoux [32].

U: Number of U, R, $\Box$ and $\Diamond$ in the formula.

X: Number of $\circ$ in the formula.

Boolean: Number of Boolean leafs, i. e., variable references and constants. This is a good parameter for estimating the length of the formula.

The results can be seen in Figure 3. The formulae generated by the Salt compiler contain a greater number of Boolean leafs, but use *less temporal operators* and,

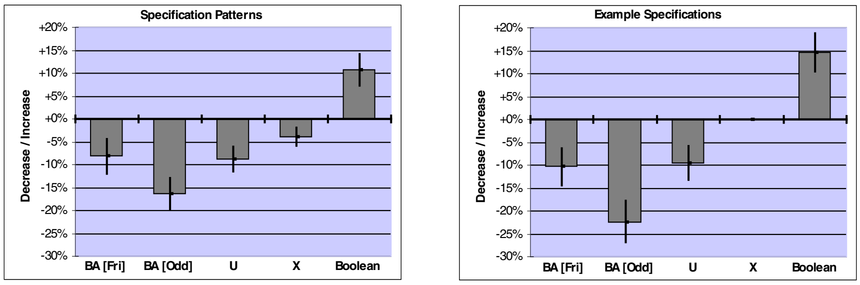

Fig. 3. Size of generated formulae

therefore, also yield a smaller BA. The error markers in the figure indicate the simple standard error of the mean.

Discussion. As it turned out, using SALT for writing specifications does not deprave model checking efficiency in practice. On the contrary, one can observe that it often leads to more succinct formulae. The reason for this result is that SALT performs a number of optimisations. For instance, when translating a formula of the form $\varphi \mathrm{W} \psi$, the compiler can choose between the two equivalent expressions

$$\neg(\neg\psi \;\mathrm{U}\; (\neg\varphi \wedge \neg\psi)) \quad \text{and} \quad (\varphi \;\mathrm{U}\; \psi) \vee \Box\varphi.$$

While the first expression duplicates ψ in the resulting formula, the second expression duplicates φ, and introduces a new temporal operator. In most cases, the first expression, which is less intuitive for humans, yields better technical results.

Another equivalence utilised by the compiler is: $\Box(\varphi \;\mathrm{W}\; \psi) \iff \Box(\varphi \vee \psi)$. With $\varphi \;\mathrm{W}\; \psi$ being equivalent to $(\varphi \;\mathrm{U}\; \psi) \vee \Box\varphi$, the left hand side reads as $\Box((\varphi \;\mathrm{U}\; \psi) \vee \Box\varphi)$. When φ and ψ are propositions, this expression results in a BA with four states (using the algorithm proposed by Fritz [31]). $\Box(\varphi \vee \psi)$, however, is translated into a BA with only a single state.

Of course, the benefit obtained from using the SALT approach is of no principle nature: The rewriting of LTL formulae could be done without having SALT as a high-level language. What is more, given an LTL-to-BA translator that produces a minimal BA for the language defined by a given formula, no optimisations on the formula level would be required, and such a translation function exists—at least theoretically[3]. Nevertheless, the high abstraction level realised by SALT makes the mentioned optimisations *easily* possible, and produces BAs that are smaller than without such optimisations—despite the fact that today's LTL-to-BA translators already perform many optimisations.

[3] As the class of BAs is enumerable and language equivalence of two BAs decidable, it is possible to enumerate the class of BAs ordered by size and take the first one that is equivalent to the one to be minimised. Clearly, such an approach is not feasible in practice—and feasible minimisation procedures are hard to achieve.

6 Conclusions

In this tutorial paper we gave an overview and a practical introduction to Salt, a high-level extensible specification and assertion language for temporal logic. We not only gave an overview over its core features, but also a detailed comparison with related approaches, in particular PSL and the Bandera input language BSL, as well as provided practical examples and results concerning the complexity of Salt. Our experimental results show that the higher level of abstraction, offered by Salt when compared to normal LTL, does not practically result in an efficiency penalty, as compiled specifications are often considerably smaller than manually written ones. This is somewhat in contrast with our more theoretical considerations, in that the satisfiability problem of Salt specifications, depending on which features are used, can be exponentially harder than that of LTL. However, the experiments show that this exponential gap does not show up in many practical examples, and that our compiler, on the contrary, is able to optimise formulae to result in smaller automata.

Our feature comparison between Salt, PSL, and the Bandera input language BSL shows that Salt incorporates many of the features present in these domain-specific languages, while still being fully translatable to standard temporal logic. However, one could argue that this is also a shortcoming of Salt, in that it is not possible to express the full fragment of ω-regular languages as can be done in other approaches, but then of course not being able to map all specifications to LTL formulae any longer. This fact could be compensated for by adding a direct translation of Salt into automata as is suggested, for example, in [33], which introduces a regular form of LTL, i.e., expressively complete wrt. ω-regular languages. Moreover, the feature comparison does not show a clear "winner" among specification languages, since they have been designed for different purposes. In fact, Salt could be used in combination with other approaches, such as BSL where it would be possible to use the output of the Salt compiler, i.e., standard LTL formulae, as input to BSL's temporal property specification layer, which offers support for LTL specifications.

Salt as presented in this tutorial paper is ready to use and we invite the reader to explore it via an interactive web interface at `http://salt.in.tum.de/`, or to download the compiler from the same location.

References

1. Bauer, A., Leucker, M., Streit, J.: SALT—Structured Assertion Language for Temporal Logic. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 757–775. Springer, Heidelberg (2006)
2. Pnueli, A.: The temporal logic of programs. In: Proc. 18th IEEE Symposium on the Foundations of Computer Science (FOCS), Providence, Rhode Island, pp. 46–57. IEEE, Los Alamitos (1977)
3. Raskin, J.-F., Schobbens, P.-Y.: State clock logic: A decidable real-time logic. In: Maler, O. (ed.) HART 1997. LNCS, vol. 1201, pp. 33–47. Springer, Heidelberg (1997)

4. D'Souza, D.: A logical characterisation of event clock automata. International Journal of Foundations of Computer Science 14(4), 625–639 (2003)
5. Kamp, J.A.W.: Tense Logic and the Theory of Linear Order. PhD thesis, University of California, Los Angeles (1968)
6. McMillan, K.L.: The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University (1992)
7. Holzmann, G.J.: The model checker Spin. IEEE Trans. on Software Engineering 23, 279–295 (1997)
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proc. 21st Int. Conf. on Software Engineering (ICSE), pp. 411–420. IEEE, Los Alamitos (1999)
9. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera: Extracting finite-state models from Java source code. In: Proc. 22nd Int. Conf. on Software Engineering (ICSE), IEEE, Los Alamitos (2000)
10. Accellera Property Specification Language. Reference Manual 1.1 (April 2004)
11. Eisner, C., Fisman, D.: A Practical Introduction to PSL (Series on Integrated Circuits and Systems). Springer, Heidelberg (2006)
12. Gabbay, D., Pnueli, A., Shelah, S., Stavi, J.: On the temporal analysis of fairness. In: Proc. 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 163–173. ACM, New York (1980)
13. Markey, N.: Temporal logic with past is exponentially more succinct, concurrency column. Bulletin of the EATCS 79, 122–128 (2003)
14. Lichtenstein, O., Pnueli, A., Zuck, L.D.: The glory of the past. In: Proc. Conference on Logic of Programs, pp. 196–218. Springer, Heidelberg (1985)
15. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems. Springer, Heidelberg (1995)
16. Fix, L.: Fifteen years of formal property verification in intel. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking, pp. 139–144. Springer, Heidelberg (2008)
17. Ashenden, P.J.: The Designer's Guide to VHDL, 2nd edn. Morgan Kaufmann Publishers Inc., San Francisco (2001)
18. Ferro, L., Pierre, L.: Formal semantics for PSL modeling layer and application to the verification of transactional models. In: Proc. Conference on Design, Automation and Test in Europe (DATE), pp. 1207–1212. European Design and Automation Association (2010)
19. Corbett, J.C., Dwyer, M., Hatcliff, J., Robby: A language framework for expressing checkable properties of dynamic software. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885. Springer, Heidelberg (2000)
20. Corbett, J., Dwyer, M., Hatcliff, J., Robby: Expressing checkable properties of dynamic systems: The Bandera specification language. Technical Report 04, Kansas State University, Department of Computing and Information Sciences (2001)
21. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1994)
22. Armoni, R., Bustan, D., Kupferman, O., Vardi, M.Y.: Resets vs. Aborts in linear temporal logic. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 65–80. Springer, Heidelberg (2003)
23. Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M.Y., Zbar, Y.: The forSpec temporal logic: A new temporal property-specification language. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 211–296. Springer, Heidelberg (2002)

24. Eisner, C.: PSL for Runtime Verification: Theory and Practice. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 1–8. Springer, Heidelberg (2007)
25. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Fifth International Conference on Verification, Model Checking and Abstract Interpretation (2004)
26. ChaoChen, Z., Hoare, T., Ravn, A.P.: A calculus of durations. Information Processing Letters 40(5), 269–276 (1991)
27. Bradfield, J., Stevens, P.: Observational mu calculus. In: Proc. Workshop on Fixed Points in Computer Science (FICS), pp. 25–27 (1998); An extended version is available as BRICS-RS-99-5
28. Bradfield, J.C., Filipe, J.K., Stevens, P.: Enriching OCL using observational mu-calculus. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 203–217. Springer, Heidelberg (2002)
29. Lange, M.: Linear time logics around psl: Complexity, expressiveness, and a little bit of succinctness. In: Caires, L., Li, L. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 90–104. Springer, Heidelberg (2007)
30. Alur, R., Henzinger, T.A.: Real-time logics: complexity and expressiveness. Technical report, Stanford, CA, USA (1990)
31. Fritz, C.: Constructing Büchi Automata from Linear Temporal Logic Using Simulation Relations for Alternating Büchi Automata. In: Ibarra, O.H., Dang, Z. (eds.) CIAA 2003. LNCS, vol. 2759, pp. 35–48. Springer, Heidelberg (2003)
32. Gastin, P., Oddoux, D.: Fast LTL to büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
33. Leucker, M., Sánchez, C.: Regular Linear Temporal Logic. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 291–305. Springer, Heidelberg (2007)

A Proofs

In this appendix, we summarise the formal semantics of Reset-LTL [22] and give a detailed proof of Theorem 1.

Definition 1 (Reset-LTL). *Let $\Sigma := 2^{AP}$ be a finite alphabet made up of propositions in the set AP, $w \in \Sigma^\omega$ an infinite word, and a, r be two Boolean formulae over AP, then*

- $\langle w^i, a, r\rangle \models p$ *if* $w^i \models a \vee (p \wedge \neg r)$,
- $\langle w^i, a, r\rangle \models \neg\varphi$ *if* $\langle w^i, r, a\rangle \not\models \neg\varphi$
- $\langle w^i, a, r\rangle \models \varphi \vee \psi$ *if* $\langle w^i, a, r\rangle \models \varphi$ *or* $\langle w^i, a, r\rangle \models \psi$
- $\langle w^i, a, r\rangle \models \circ\varphi$ *if* $w^i \models a$ *or* $\langle w^{i+1}, a, r\rangle \models \varphi$ *and* $w^i \not\models r$,
- $\langle w^i, a, r\rangle \models \varphi U \psi$ *if* $\exists k \geq i.\ \langle w^k, a, r\rangle \models \psi \wedge \forall i \leq l < k.\ \langle w^l, a, r\rangle \models \varphi$,
- $\langle w^i, a, r\rangle \models$ `accept` e `in` φ *if* $\langle w^i, a \vee (e \wedge \neg r), r\rangle \models \varphi$,
- $\langle w^i, a, r\rangle \models$ `reject` e `in` φ *if* $\langle w^i, a, r \vee (e \wedge \neg a)\rangle \models \varphi$,

where w^i denotes w's infinite suffix after the i-th position, i.e., $w^i = w_i w_{i+1} \ldots$

Theorem 1. The following relationship holds between the Reset-LTL operators `abort` and `reject` and the SALT operators `accepton` and `rejecton`:

$$\langle w^i, \text{false}, \text{false}\rangle \models \texttt{accept}\ e\ \texttt{in}\ \phi \text{ if and only if } w^i \models \phi\ \texttt{accepton}\ e,$$

and

$$\langle w^i, \text{false}, \text{false}\rangle \models \texttt{reject}\ e\ \texttt{in}\ \phi \text{ if and only if } w^i \models \phi\ \texttt{rejecton}\ e.$$

Proof. By structural induction. Note that the relevant semantic definitions for SALT's exception operators are given in Sec. 3. Let $p \in AP$ and $\phi := p$.

$$\begin{aligned} &\langle w^i, \text{false}, \text{false}\rangle \models \texttt{accept}\ e\ \texttt{in}\ p \\ \Leftrightarrow\ &\langle w^i, e, \text{false}\rangle \models p \\ \Leftrightarrow\ &w^i \models e \vee p \\ \Leftrightarrow\ &w^i \models p\ \texttt{accepton}\ e. \end{aligned} \qquad \begin{aligned} &\langle w^i, \text{false}, \text{false}\rangle \models \texttt{reject}\ e\ \texttt{in}\ p \\ \Leftrightarrow\ &\langle w^i, \text{false}, e\rangle \models p \\ \Leftrightarrow\ &w^i \models p \wedge \neg e \\ \Leftrightarrow\ &w^i \models p\ \texttt{rejecton}\ e. \end{aligned}$$

$\phi := \varphi \vee \psi$:

$$\begin{aligned} &\langle w^i, \text{false}, \text{false}\rangle \models \texttt{accept}\ e\ \texttt{in}\ \varphi \vee \psi \\ \Leftrightarrow\ &\langle w^i, e, \text{false}\rangle \models \varphi \vee \psi \\ \Leftrightarrow\ &\langle w^i, e, \text{false}\rangle \models \varphi \vee \langle w^i, e, \text{false}\rangle \models \psi \\ \Leftrightarrow\ &w^i \models \varphi\ \texttt{accepton}\ e \vee w^i \models \psi\ \texttt{accepton}\ e \\ \Leftrightarrow\ &w^i \models \varphi \vee \psi\ \texttt{accepton}\ e \end{aligned}$$

The case for `reject` e `in` $\varphi \vee \psi$ is analogous.
$\phi := \circ\varphi$:

$$\begin{aligned} &\langle w^i, \text{false}, \text{false}\rangle \models \texttt{accept}\ e\ \texttt{in}\ \circ\varphi \\ \Leftrightarrow\ &w^i \models e \vee \langle w^{i+1}, e, \text{false}\rangle \models \varphi \\ \Leftrightarrow\ &w^i \models e \vee w^{i+1} \models \varphi\ \texttt{accepton}\ e \\ \Leftrightarrow\ &w^i \models e \vee w^i \models \circ(\varphi\ \texttt{accepton}\ e) \\ \Leftrightarrow\ &w^i \models (\circ\varphi)\ \texttt{accepton}\ e. \end{aligned}$$

The case for `reject` e `in` $\circ\varphi$ is analogous.
$\phi := \varphi\ \mathrm{U}\ \psi$:

$$\begin{aligned} &\langle w^i, \text{false}, \text{false}\rangle \models \texttt{accept}\ e\ \texttt{in}\ \varphi\ \mathrm{U}\ \psi \\ \Leftrightarrow\ &w^i \models e \vee \langle w^i, e, \text{false}\rangle \models \varphi\ \mathrm{U}\ \psi \\ \Leftrightarrow\ &\exists k \geq i.\ \langle w^k, e, \text{false}\rangle \models \psi \wedge \forall i \leq l < k.\ \langle w^l, e, \text{false}\rangle \models \varphi \\ \Leftrightarrow\ &\exists k \geq i.\ w^k \models \psi\ \texttt{accepton}\ e \wedge \forall i \leq l < k.\ w^l \models \varphi\ \texttt{accepton}\ e \\ \Leftrightarrow\ &w^i \models \varphi\ \texttt{accepton}\ e\ \mathrm{U}\ \psi\ \texttt{accepton}\ e \\ \Leftrightarrow\ &w^i \models \varphi\ \mathrm{U}\ \psi\ \texttt{accepton}\ e. \end{aligned}$$

The case for `reject` e `in` $\varphi\ \mathrm{U}\ \psi$ is analogous.
$\phi := \neg\varphi$: Negation is somewhat a special case due to the mutual recursive definition of the semantics. Here, we treat the Reset-LTL side first by itself, and use the duality between the `accept` and `reject` operators as follows.

$$\begin{aligned} &\langle w^i, \text{false}, \text{false}\rangle \models \texttt{accept}\ e\ \texttt{in}\ \neg\varphi \\ \Leftrightarrow\ &\langle w^i, \text{false}, \text{false}\rangle \models \neg(\texttt{reject}\ e\ \texttt{in}\ \varphi) \\ \Leftrightarrow\ &\langle w^i, \text{false}, \text{false}\rangle \not\models (\texttt{reject}\ e\ \texttt{in}\ \varphi). \end{aligned}$$

Next, we observe that the following holds in SALT:

$$w^i \models \neg\varphi \;\texttt{accepton}\; e \Leftrightarrow w^i \not\models \varphi \;\texttt{rejecton}\; e.$$

Now, equivalence follows from case two of the induction hypothesis, i.e.,

$$w^i \models \varphi \;\texttt{rejecton}\; e \Leftrightarrow \langle w^i, \text{false}, \text{false}\rangle \models (\texttt{reject}\; e \;\texttt{in}\; \varphi).$$

The case for `reject` e `in` $\neg\varphi$ is dual. □