SALT—Structured Assertion Language for Temporal Logic

Andreas Bauer, Martin Leucker*, and Jonathan Streit

Institut für Informatik, Technische Universität München
{baueran,leucker,streit}@informatik.tu-muenchen.de

Abstract. This paper presents SALT. 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 specification patterns, but also provides nested scopes, exceptions, support for regular expressions and real-time. The latter is needed in particular for verification tasks to do with reactive systems imposing strict execution times and deadlines. However, unlike other formalisms used for temporal specification of properties, SALT does not target a specific domain. The paper details on the design rationale, syntax and semantics of SALT in terms of a translation to temporal (real-time) logic, as well as on the realisation in form of a compiler. Our results will show that the higher level of abstraction introduced with SALT does not deprave the efficiency of the subsequent verification tools—rather, on the contrary.

1 Introduction

Temporal logics, such as linear time temporal logic (LTL) [Pnu77], are specification formalisms suited to express desired properties of a set of traces and come with a rigorous semantics. Yet more importantly, automatic verification techniques, such as model checking [CGP99], are successfully used to verify such specifications over finite-state system models.

However, despite obvious advantages over semi-formal or even informal notations, temporal logic is often completely disregarded in (industrial) practice. Instead, a considerable amount of verification related questions are answered only partially by means of testing and simulation—with well-known drawbacks, namely that testing alone can never show the absence of bugs, but merely their presence if at all (cf. [DDH72]). Temporal logic, on the other hand, is still widely considered to be a vehicle for specially skilled verification engineers, if not even an academic toy.

We argue that this point-of-view is misleading. We do, however, admit that, for example, LTL's syntax—together with the typical reduction to a minimal set of operators which is done in most research papers—makes it additionally hard for formulating concise and correct specifications, even for specialists.

* Part of this work was done during the author's stay in Stanford, USA, and supported by ARO DAAD 19-03-1-0197.

For example, consider the simple requirement "s precedes p after q", which is formulated in LTL by Dwyer et al. [DAC99] as $(\Box\neg q) \vee \Diamond(q \wedge (\neg p \text{ W } s))$. At first sight, this looks correct: "either q never holds or, when q becomes true, there is no p before an s". Nevertheless, the formula contains a very subtle error: it states that *eventually* $q \wedge (\neg p \text{ W } s)$ holds, but does not require it to be the first occurrence of q. The sequence `qpqs` satisfies the formula, although it is clear that it should not. Consequently, the correct formula would be $(\Box\neg q) \vee \neg q \text{ U } (q \wedge (\neg p \text{ W } s))$. Avoiding this kind of mistake in specifications altogether is practically impossible. LTL's minimalistic set of operators, however, forces its users to build complex, error-prone formulas for even very simple requirements as can be seen above.

Despite this, it is very unlikely that a completely different specification formalism—of whatever kind—would stand a chance to compete with LTL for at least two different reasons: 1. LTL has a well-accepted precise semantics, 2. powerful model checking tools and runtime verification approaches based on LTL exist already.

Contribution. In this paper, we remedy LTL's and timed LTL's (short: TLTL [RS97,D'S03]) weaknesses for industrial specifications by introducing the specification and assertion language Salt, which is an acronym for *Structured Assertion Language for Temporal Logic* (see also `http://salt.in.tum.de/`).

To programmers, Salt looks a lot like a programming language, while still being translatable to LTL, or—in case real-time operators are used—to TLTL. As such, Salt is also suitable as a front end to already existing model checking and runtime verification tools. Furthermore, a precise semantics of Salt is given in terms of translation rules, which are realised in an accompanying Salt compiler prototype.

More importantly, being closer to a general purpose (programming) language, Salt is—as the examples throughout the paper will show—more intuitive to use and understand than standard LTL. For example, besides LTL's temporal operators, Salt provides sophisticated scoping rules, support for (limited) regular expressions, exceptions, iterators, counting quantifiers, and user-defined macros.

In other words, using Salt, users are able to specify properties on a higher level of abstraction than with many other formalisms, such as standard LTL.

While compiling a high level programming language to a low level representation often has a negative impact in terms of efficiency, we will show that LTL (resp. TLTL) formulas resulting from Salt specifications tend to be rather compact when compared to their manually-written counterparts in LTL; one reason lies in that humans tend to choose the most readable formula among equivalent ones, while our compiler can optimise purely for the size of a formula.

However, plain LTL's limited flexibility in real-world scenarios has also been noted by other users (see Section 2). For instance, for the hardware domain, Sugar/PSL [BBDE$^+$01] has been designed as a high level specification language aimed as a "syntactic sugaring" for temporal logic.

Dwyer et al. [DAC99] have analysed real-world specifications to identify typical *patterns* for property specifications, similar to *design patterns* encountered in

software engineering [GHJV94]. Using patterns allows even inexperienced users to reuse expert knowledge.

SALT takes over some of the ideas present in PSL and is heavily inspired by the pattern approach. However, SALT is a language and patterns are turned into operators of the language. Furthermore, the additional concepts listed above, like macro definitions, counting quantifiers etc., round off the specification language and push SALT ahead of existing approaches.

SALT's language reference, a compiler, an interactive web interface, as well as further example specifications written in SALT are available from the web site (`http://salt.in.tum.de/`).

Outline. Section 2 describes the context of SALT by means of already existing and mostly domain-specific approaches, as well as a classification of SALT with respect to its underlying semantics and expressiveness. Then, in Section 3, we take a detailed look at the language itself and highlight its main features. We discuss SALT's formal semantics in Section 4. In Section 5, we will show that SALT specifications can be efficiently compiled to standard temporal (real-time) logics, often resulting in even more compact representations. Section 6 concludes the paper.

2 Classification

In the following, we detail on the context of SALT in terms of related work as well as in terms of its underlying semantics and expressiveness.

2.1 Existing approaches

Sugar/PSL. Sugar/PSL (Property Specification Language) [BBDE+01] is a high level specification language tailored for hardware design, originally aimed as a "syntactic sugaring" of the branching time logic CTL. Sugar 2.0 is based on a linear view of time while keeping branching time as an optional extension and is currently undergoing standardisation by the IEEE under the name PSL [FMW05].

The PSL specification language is structured into boolean, temporal, verification, and modelling *layers*. The boolean layer provides operators for propositional logic, while the operators of the temporal layer are used to combine propositional formulas to temporal ones. The verification layer allows to define what the verification tool is expected to do with the specified properties (e. g., check that a property holds, assume that a property holds, etc.). The modelling layer, in turn, is used to model the input to the design or external hardware.

PSL provides a rich set of operators for reasoning over boolean conditions (e. g., bit-vector operations) and for regular expressions. A so-called clocking operator allows to state that an expression is evaluated only in cycles where its clocking condition holds. PSL comes with an abort operator that can be used to model resets: it evaluates a pending expression to false on the occurrence

of an exceptional (abort) condition. Furthermore, PSL allows the use of macro directives similar to those of the C preprocessor. Parameterised properties can be instantiated for a set of concrete values. However, PSL does not contain temporal past operators which can be rather intuitive to use as well as make specifications more succinct (cf. [Mar03]), and no real-time constraints used frequently for modelling and verifying properties of reactive systems imposing strict execution times and deadlines, such as embedded systems.

PSL is often directly used as input to a verification tool, both for formal verification and for generating checks that are executed by a simulation tool. The latter corresponds to a runtime analysis of a simulated hardware design. However, PSL is specific to the hardware domain and a translation into LTL is possible only for a subset of PSL [TS05]. Therefore it cannot be easily used with existing LTL-based verification tools.

PSL's goals are orthogonal to the Salt approach. With Salt, we wanted to go further in terms of abstracting from LTL's syntax and thus providing a more convenient-to-use language. On the other hand, Salt is not dedicated to either model checking, runtime verification, or strictly to the hardware domain. As such, Salt does not impose its own verification and modelling layer.

SpecPatterns. The Salt approach was also influenced by work of Dwyer et al., in which various real-world specifications have been analysed [DAC99]. Frequently used patterns have been identified and a *pattern system* for property specifications, similar to the design patterns in software engineering [GHJV94] has been elaborated. A pattern provides a solution to a reoccurring problem, often including notes about its advantages, drawbacks, and alternatives. As such it enables inexperienced users to reuse expert knowledge.

Basically, the patterns of Dwyer et al. 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". The specification pattern approach has been adopted by the Bandera Specification Language [CDHR01] and a compiler that translates such specifications into LTL is part of the Bandera system.

Dwyer et al. convincingly argue that scopes are needed in many real-world specifications. However, specification patterns as defined by Dwyer et al. suffer from the fact that they cannot be nested: only propositional formulas may be used as their parameters. In other words, adding a new requirement to the pattern system means having to manually write an LTL formula for each scope.

Others. The previous two approaches are not the only specification languages tailored for domain specific tasks. For instance, the ForSpec Temporal Logic (FTL) [AFF$^+$02] is a specification language developed at Intel, and is based on a linear view of time, aimed for the formal verification of hardware circuits. Much like Sugar/PSL, ForSpec provides regular and clocked expressions as well as accept and reject operators for modelling resets. However, ForSpec does not

contain real-time operators, only limited support for references to the past, and cannot be completely translated to LTL.

EAGLE [BGHS04] 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 [CHR91] 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.

2.2 Expressiveness

Clearly, existing approaches have shaped various practical considerations in the design rationale of the language SALT. However, from a purely theoretical point-of-view, SALT's features are more oriented towards the varying expressiveness of the supported logics.

SALT currently supports translation into propositional logics, LTL, as well as TLTL, a natural extension of LTL for the formulation of real-time constraints [RS97]. D'Souza has shown [D'S03] 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, shown by Kamp [Kam68]. However, LTL is strictly less expressive than second-order logic over words, which is expressively equivalent to ω-regular expressions. This implies that full support of regular expressions is not possible when LTL properties are in question (see Figure 1).

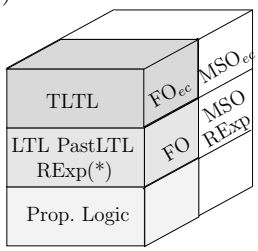

Fig. 1. Relationships between propositional, first-order, and temporal logics.

For practitioners, regular expressions are an established formalism, often used to specify custom search-patterns. Therefore, SALT provides support for simpli-

fied regular expressions that do not go beyond star-free languages (where "star" refers to the Kleene operator) and that can be efficiently translated into LTL.

The design of the language SALT also follows a strictly layered approach (see Section 3), in that the language supports specifications that can be translated into either formalism depicted in Figure 1. More so, by reflecting and differentiating between the different levels of expressiveness in the language, SALT is extensible to support other logics in the future as well.

3 Features of the SALT language

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).
```

Basically, the SALT language consists of the following three layers, each covering different aspects of the 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.

Depending on which layers are used for specification, the SALT compiler generates either LTL or TLTL formulas (resp. with or without past operators). For instance, if only operators from the propositional layer are used, the resulting formulas are purely propositional formulas. If only operators from the temporal and the propositional layer are used, the resulting formulas are LTL formulas, whereas if the timed layer is used, the resulting formulas are TLTL formulas.

3.1 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 [McM92].

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 customized 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` can be used, which appear similarly also in the ForSpec language. Conditional operators tend to make specifications easier to read, because `if-then-else` constructs are familiar to programmers of almost every language. Using this operator, 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.

3.2 Temporal layer

The temporal layer consists of a future and a past fragment. Although past operators do not add expressiveness [GPSS80], they can help to write formulas that are easier to understand and more efficient for processing [Mar03].

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.

Standard LTL operators. SALT provides the common LTL operators U, W, R, $\Box$, $\Diamond$ and $\bigcirc$, written as `until`, `until weak`, `releases`, `always`, `eventually`, and `next`. Thus, untimed SALT has the same expressiveness as LTL (see Section 2.2).

Extended operators. Similar to Sugar/PSL, SALT also 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 help to make specifications easier to understand.
- Extended `until`. SALT provides an extended version of the LTL U operator. The user 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) [1]
  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).
  The `until` operator family of Sugar/PSL provides a similar choice between inclusive/exclusive and weak/strong end conditions.
- 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.

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 as

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

Exceptions. SALT includes the exception operators `rejecton` and `accepton` that 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

[1] 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.

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.

Exceptions can be 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.
```

Similar `rejecton` and `accepton` operators can be found in ForSpec and in PSL. The formal semantics of LTL enriched with those two operators (called Reset-LTL) is explored in detail elsewhere [ABKV03].

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 [DAC99]. 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*).

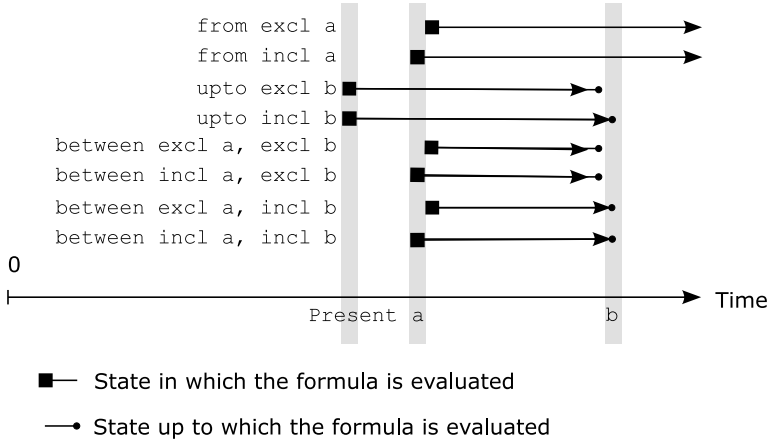

Fig. 2. Scopes of `upto`, `from` and `between`.

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
```

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.

Scopes have been identified by Dwyer et al. as an important issue in the specification pattern system, and the Bandera language. However, their pattern system is restricted to predefined requirements. 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 Sugar/PSL using the `next_event` and `before` operators. SALT's distinguishing feature here is that scope operators can be used with arbitrary formulas, even with nested scope operators.

While it is possible to implement a translation of the `from` operator into LTL relatively straightforward (see Section 4), 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: $(p \rightarrow (\neg b \ \mathrm{U} \ (\neg b \wedge s))) \ \mathrm{W} \ b$. Dwyer et al. describe this procedure in the notes of their specification pattern system [DAC99]. 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 Section 4.

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, e. g., such as Sugar/PSL. However, arbitrary regular languages can be defined using regular expressions, while LTL only allows to define so-called *star-free* languages. Thus, regular expressions have to be restricted in SALT.

SALT regular expressions provide concatenation (`;`), union (`|`) and Kleene-star operators (`*`), but no complement. The argument of the Kleene-star 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 efficiently

into LTL. We agree with Sugar/PSL, which also provides regular expressions without a complement operator, that many relevant properties can be expressed conveniently without it.

Additionally, SALT provides operators that do not increase the expressiveness of its regular expressions, but makes dealing with them more convenient for users. The overlapping sequence operator `:` is inspired by Sugar/PSL and 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. The `*` operator extended with a range of natural numbers allows to specify that an expression has to hold at least, at most, exactly, or in between n and m times.

Traditional regular expressions match finite sequences. A SALT regular expression holds on an (infinite) sequence if it matches a finite prefix of the sequence.

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

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

3.3 Timed layer

SALT contains a timed extension that allows the specification of real-time constraints. Timed operators are translated into TLTL [RS97,D'S03], 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 $\triangleright_{\sim c}\varphi$ 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 $\sim c$.

3.4 Macros and parameterised expressions

SALT allows user-defined sub-expressions as *macros* and to parameterise macros and sub-expressions. Macros 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.

Up to an extent, support for user-defined macros and iteration over parameterised expressions is a part of many high-level specification languages, e. g., such as Sugar/PSL.

3.5 Example

Let us conclude this section by looking at a slightly longer example showing most of SALT's features. The following specification describes an elevator and is partially based on an example presented by Dwyer et al. [DAC99]: 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)
```

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

4 Semantics

SALT comes with a precisely defined semantics. As outlined in Section 2.2, SALT 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 formulas.

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 [MP95]).

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 operator `never` is then translated as $\mathcal{T}(\texttt{never } \varphi) = \neg\Diamond T(\varphi)$, whereas a weak inclusive until as in $\varphi_1 \texttt{ until incl weak } \varphi_2$ is then defined, for instance, as

$$\mathcal{T}(\varphi_1 \texttt{ until incl weak } \varphi_2) = \mathcal{T}(\varphi_1) \text{ W } (\mathcal{T}(\varphi_1) \wedge \mathcal{T}(\varphi_2)).$$

However, not all SALT operators translate in such a straightforward inductive manner, since their translation depends on what is defined by the according subformulas 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:

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

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

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 Section 3.2, and whose translation depends on `stop`, is then defined as:

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

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

Similar translation schemes are defined for SALT's exception operators, i. e., `accepton` and `rejecton`. Those and the remaining operators' semantics are detailed in the SALT language reference and manual.

5 Realisation and 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.

5.1 The SALT compiler

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 [McM92] and SPIN [Hol97] 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 formulas.

5.2 Experimental results

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. One might suspect that generated formulas are bigger and less efficient to check than handwritten ones. But our experiments show that this is not the case.

In order to quantify the efficiency of the SALT compiler, existing LTL formulas were compared to the formulas generated by the compiler from a corresponding SALT specification. This was done for two data sets: the specification pattern system [DAC99] (50 specifications) and a collection of real-world example specifications, mostly from the Dwyer's et al.'s survey data [DAC99] (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 [Fri03], 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 [GO01].

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

X: Number of $\bigcirc$ 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 formulas generated by the SALT compiler contain a greater number of boolean leafs, but use *less temporal operators* and, therefore, also yield a smaller BA. The error markers in the figure indicate the simple standard error of the mean.

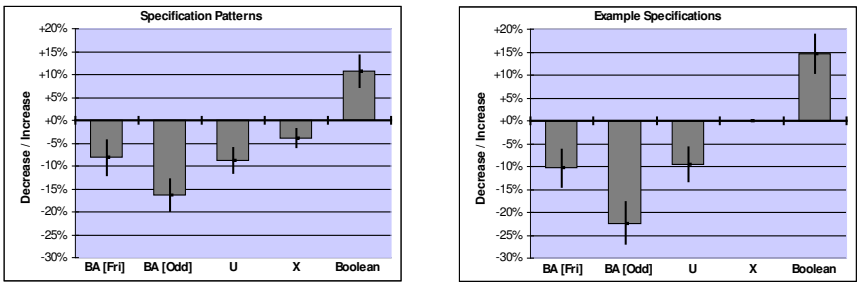

Fig. 3. Size of generated formulas.

Discussion. Using SALT for writing specifications does not deprave model checking efficiency. On the contrary, one can observe that it often leads to more succinct formulas.

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: $\Box(\varphi \text{ W } \psi) \iff \Box(\varphi \vee \psi)$. With $\varphi \text{ W } \psi$ being equivalent to $(\varphi \text{ U } \psi) \vee \Box\varphi$, the left hand side reads as $\Box((\varphi \text{ U } \psi) \vee \Box\varphi)$. When φ and ψ are propositions, this expression results in a BA with four states (using the algorithm proposed by Fritz [Fri03]). $\Box(\varphi \vee \psi)$, 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 formulas 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.[2] 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.

6 Conclusions

In this paper we presented SALT, a high-level extensible specification and assertion language for temporal logic. It is designed for intuitive usage for both verification experts as well as more practically oriented system engineers.

The development of SALT originates mainly from difficulties we faced in our industrial cooperations, when trying to apply and transfer certain state-of-the-art verification methods into industrial practice. But also within our academic cooperations (see, e. g., [BKKS05]), we have learned that LTL is often difficult to use for a typical software engineer.

SALT aims to ease some of these problems by introducing on the one hand side a higher level of abstraction for the specification of temporal assertions. This makes specifications easier to understand and more convenient to express for its users. At the same time, SALT is designed to look and feel like a programming language to be easily accessible to software engineers.

Our experimental results have shown that the higher level of abstraction does not result in an efficiency penalty, as compiled specifications are often considerably smaller than manually-written ones.

We have integrated SALT into AUTOFOCUS [HSSS96], a modelling and verification tool used within several industrial cooperations, and first reactions of AutoFocus users are very promising.

SALT as presented in this paper is ready to use and we invite the reader to explore it in-depth via its interactive web interface at `http://salt.in.tum.de/`.

[2] 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.

References

[ABKV03] R. Armoni, D. Bustan, O. Kupferman, and M. Y. Vardi. Resets vs. aborts in linear temporal logic. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 65–80. Springer, 2003.

[AFF+02] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. The ForSpec temporal logic: A new temporal property-specification language. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 296–211, 2002.

[BBDE+01] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. The temporal logic Sugar. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV)*, pages 363–367, London, UK, 2001. Springer.

[BGHS04] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, 2004.

[BKKS05] J. Botaschanjan, L. Kof, C. Kühnel, and M. Spichkova. Towards Verified Automotive Software. In ACM Press, editor, *Proceedings of the 2nd International ICSE Workshop on Automotive Software*. ACM, New York, May 2005.

[CDHR01] James Corbett, Matthew Dwyer, John Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The Bandera specification language. Technical Report 04, Kansas State University, Department of Computing and Information Sciences, 2001.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CHR91] Zhou ChaoChen, Tony Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[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.

[DDH72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, London, 1972.

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

[FMW05] Harry Foster, Erisch Marschner, and Yaron Wolfsthal. IEEE 1850 PSL: The next generation. In *DVCon*, 2005.

[Fri03] Carsten Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In Oscar H. Ibarra and Zhe Dang, editors, *Implementation and Application of Automata. Eighth International Conference (CIAA)*, volume 2759 of *Lecture Notes in Computer Science*, pages 35–48, Santa Barbara, CA, USA, 2003.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV)*, pages 53–65, London, UK, 2001. Springer.

[GPSS80] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 163–173, New York, NY, USA, 1980. ACM Press.

[Hol97] Gerard J. Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23:279–295, May 1997.

[HSSS96] Franz Huber, Bernhard Schatz, Alexander Schmidt, and Katharina Spies. AutoFocus: A tool for distributed systems specification. In *Proceedings of Formal Techniques in Real-Time and Fault Tolerant Systems*, pages 467–470. Springer, 1996.

[Kam68] Johan Anthony Willem Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, 1968.

[Mar03] Nicolas Markey. Temporal logic with past is exponentially more succinct, concurrency column. *Bulletin of the EATCS*, 79:122–128, 2003.

[McM92] K. L. McMillan. The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems*. Springer, New York, 1995.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, October 31–November 2 1977. IEEE Computer Society Press.

[RS97] Jean-François Raskin and Pierre-Yves Schobbens. State clock logic: A decidable real-time logic. In Oded Maler, editor, *HART*, volume 1201 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 1997.

[TS05] T. Tuerk and K. Schneider. From PSL to LTL: A formal validation in HOL. In *Theorem Proving in Higher Order Logic (TPHOL)*, Lecture Notes in Computer Science, Oxford, UK, 2005. Springer.