Munich University of Technology
Department of Informatics

Diploma Thesis

in Informatics

Development of a programming-language-like
temporal logic specification language

Jonathan Streit

Date: April 12th, 2006

Supervisor:
Prof. Dr. Dr. h.c. Manfred Broy

Instructors:
Andreas Bauer
Dr. Martin Leucker

Acknowledgements

I would like to express my gratitude to my project instructors, Andreas Bauer and Martin Leucker, for fruitful discussions, fast and detailed answers to my questions, their honesty and time.

Furthermore I would like to thank Elmar Jürgens and Friedemann Gerster-Streit for reading and commenting on this thesis, as well as Gerhard Dirlich for both comments on this thesis and advice on statistics.

Abstract

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

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

In contrast to many proprietary specification languages, SALT can be translated into LTL—or in the case of real-time properties into TLTL—and thus be used as a front end to existing verification tools. A compiler performing this task has been implemented and is presented in this thesis. Experimental results show that the higher level of abstraction SALT provides does not come at the expense of efficiency—rather, on the contrary.

Contents

Chapter 1

Introduction

Developing a hardware or software system inevitably means producing errors, with consequences reaching from annoyance for the users of an erroneous software to the death of humans caused by wrongly controlled devices (e. g., [LT93]). Among the numerous techniques for finding defects, formal methods like model checking [VW86, CM05] can prove automatically and with absolute certainty that a system satisfies a given set of properties. This is an obvious advantage over traditional testing, which can merely show the presence of defects, but never their absence [DDH72].

Still, formal methods are often disregarded in industrial practice. One obstacle, the feasibility of model checking for real-world systems, has been tackled in the past years with increased computing power, significantly improved algorithms and alternative approaches like runtime verification [CM05]. Another barrier, still hindering the practical application of formal methods, is the limited usability of the specification formalism. Temporal logics, especially linear temporal logic (Ltl) [Pnu77, MP92], are traditionally used to express the desired temporal properties of a system in a formal way. However, complex Ltl formulae are quite difficult to write—not only for novices but also for experienced users—and they are often not very intuitive. Consider for instance the requirement

"s precedes p after q"

found in [DAC99] and its representation in Ltl

$$(\Box\neg q) \vee \Diamond(q \wedge (\neg p \text{ W } s)).$$

We can note three things about this formula:

1. It does not resemble much the initial requirement.
2. q appears twice in the formula although it is mentioned in the requirement only once.
3. The formula is wrong (as we will see in chapter 3.1).

The latter is probably the worst—of what use is the formal verification of a system against a specification, if the specification itself is erroneous?

Still, LTL has two significant advantages that justify its use: first, it has a well-accepted precise semantics, a precondition for the application of formal verification methods. Second, powerful model checking and runtime verification tools that process specifications in LTL exist already.

This thesis relies on the conviction that LTL is in general a suitable formal basis for the creation of formal specifications, but located on a level of abstraction too low for comfortable use in daily practice. A model for resolving this problem can be found in the field of programming languages: for decades, new languages have been built atop of lower level languages, raising the level of abstraction while keeping the lower level as a well-accepted basis and as an interface to existing tools. Compilers hide the complexity of the translation process (e. g., UML model to C++, C++ to C, C to assembler code, assembler code to binary code) from the users. In this work, we will apply the same approach to specification languages and design a higher level specification language that can be translated into LTL with the help of a compiler.

Contribution

This thesis proposes a new general purpose temporal specification language called SALT (**S**tructured **A**ssertion **L**anguage for **T**emporal Logic). In order to be comfortable for engineers, this language was designed to resemble natural language and common programming languages. It aims at providing the abstractions and operators necessary for the creation of concise specifications. A timed extension permits the expression of real-time constraints.

Untimed SALT is completely translatable into LTL. TLTL [D'S03], an efficiently decidable real-time logic, is used as target language for the timed extension. LTL is preferred over CTL (i. e., linear time over branching time) for mainly two reasons: CTL is inherently unsuitable for runtime verification, as observed traces are linear instead of tree-like, and it is less intuitive than LTL [Var01].

A compiler performing the translation has been implemented and is presented in this thesis. Experimental results show that the higher level of abstraction SALT provides does not come at the expense of efficiency—rather, on the contrary.

Related approaches

Various high-level specification languages (e. g., Sugar/PSL [Acc04] and ForSpec [AFF$^+$02]) have been developed in the hardware domain, where formal verification techniques can be applied more easily (because of a smaller state space), and where they are successfully used by the industry. Those languages are however focused on hardware design and aimed as front ends to proprietary verification tools. They are therefore not apt for translation into LTL.

Dwyer et al. [DAC99] have analysed real-world specifications and developed a pattern system for property specification, similar to design patterns encountered in software engineering [GHJV94]. Patterns allow inexperienced users to profit from expert knowledge. The specification patterns focus especially on requirements that are limited to a scope, i. e., that have to hold before, after or between some events. Unfortunately, patterns cannot be combined freely, and provide only little help for new requirements not contained in the pattern catalogue.

Both Sugar/PSL and the pattern approach have deeply inspired this thesis, as they represent sound knowledge about the needs of the users.

Outline

This thesis is organised as follows:

- Chapter 2 introduces the necessary preliminaries and formalisms, in particular LTL and TLTL.
- Chapter 3 describes existing specification formalisms and discusses their advantages and drawbacks. From the discussion we derive goals for the development of a new specification language.
- Chapter 4 presents the main features of the new specification language SALT. Furthermore it describes in detail how a mapping to LTL can be obtained for two important non-trivial aspects, namely the scope operators and simplified regular expressions.
- Chapter 5 deals with the implementation of the SALT compiler.
- Chapter 6 presents and discusses experimental results on the efficiency of the SALT compiler. In particular, it provides evidence that the additional processing step does not increase model checking cost, but in most cases even improves efficiency.
- Chapter 7 summarises the results and contributions of this thesis.
- The appendix contains the full language reference and compiler manual for the SALT language, as distributed together with the compiler. Besides the detailed description of the language, the manual includes installation and configuration instructions as well as a tutorial and several example specifications. Furthermore, it provides a complete mapping of SALT into LTL and TLTL and thereby defines the formal semantics of the language.

Typographical conventions.

SALT specifications are written in typewriter style with bold keywords (e. g., `variable`, **`until`**), while mathematical style and symbols (e. g., $\vee$, U) are used for LTL expressions. A full list of LTL operators can be found in section 2.3. Placeholders for propositional formulae are denoted with italic lower case letters (e. g., a, b). Temporal formulae are denoted with Greek letters (e. g., φ, ψ).

Chapter 2

Preliminaries

This chapter presents the context and preliminaries necessary for the understanding of this thesis, namely verification methods and temporal logic.

2.1 Model checking

Model checking [VW86, BR05] is a formal verification technique, i. e., it can prove that a system satisfies a specification. This is an obvious advantage over traditional testing, which can merely show the presence of defects, but never their absence [DDH72]. A model checking tool enumerates—explicitly or implicitly—all reachable states of the system under scrutiny in an automated way. The system is described by a model, usually a finite state automaton, which can be derived from the system design (e. g., the program code) or written by hand (e. g., in the verification language ProMeLa [Hol90]). The properties to be verified are usually expressed by temporal logic formulae, from which Büchi automata are constructed for the verification process [VW86, Wol02, GO01, Fri03].

Model checking suffers from the so-called state space explosion problem—an exponential blow up of the state space—that occurs when combining the automata representing the system and the specified properties. Therefore model checking is, in industrial practice, currently mainly employed for hardware design, embedded systems and communication protocols, where the state space is usually not as big as for software systems. Substantial progress has however been made in the optimisation of the underlying algorithms, and there is more and more research in the application of model checking to software, even for programs written in Java [CDHR01, VHBP00].

The model checkers relevant for this thesis are NuSMV [CCGR99], a reimplementation of SMV [McM], and SPIN [Hol97]. Both tools can be downloaded for free[1].

2.2 Runtime verification

The application of model checking to large systems is often not feasible because of the state space explosion problem. In these cases, *runtime verification* tech-

[1] `http://nusmv.irst.itc.it` and `http://www.spinroot.com`

niques [CM05] offer an alternative approach. As in model checking, the desired behaviour is specified in temporal logic and translated into an automaton, the so-called *monitor*. Then, the system under test and the monitor are executed in parallel, and the monitor is fed with output or events from the system (this might require instrumenting the system under scrutiny with event-emitting code). If the monitor detects a violation of the specification, it triggers an alarm signal. In a productive environment, it might even take further action, like resetting the system to a safe state. As a variation of the described *on-line* monitoring, output from the system can be saved and analysed *off-line* later.

Of course, runtime verification cannot provide a proof of correctness, because it examines only a few out of all possible execution traces. It is nevertheless a useful technique where traditional testing does not yield the desired reliability and model checking is not feasible. An advantage of runtime verification over model checking is that it can test the implementation of a system in a real environment, while model checking is limited to checking a model of the system and thus cannot assure that the model is implemented correctly.

Various tools for runtime verification are described in [Dru00, BGHS04b, HR04, ABLS05]. An experimental comparison between different verification techniques can be found in [BDG+04].

2.3 Temporal Logic

Linear temporal logic (LTL) [Pnu77, MP92] is a formalism for reasoning about sequences of states. It extends propositional logic with temporal operators that are based on a linear view of time. By way of contrast, temporal operators in the *branching time logic* CTL [CE82] carry a universal or existential quantifier and therefore consider all possible continuations of a sequence in the future. In this thesis, LTL is preferred over CTL for two reasons: CTL is inherently unsuitable for runtime verification, as observed traces are linear instead of tree-like, and it is less intuitive than LTL [Var01].

Syntax. We define the set of LTL formulae over a set of atomic propositions AP inductively as

$$\varphi := \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \mathrel{\mathrm{U}} \varphi \mid \circ\varphi \mid \varphi \mathrel{\mathrm{S}} \varphi \mid \bullet\varphi$$

with $p \in AP$. As in propositional logic, $\top$, $\neg$ and $\wedge$ are read as *true*, *logical* or *boolean not*, and *logical* or *boolean and*. The operators U, S, $\circ$ and $\bullet$ are read as *until*, *since*, *next* and *previous*.

Formulae that do not contain S or $\bullet$ are called *future formulae*. Formulae that do not contain U or $\circ$ are called *past formulae*. Formulae that contain none of U, $\circ$, S or $\bullet$ are called *state formulae* or *propositional formulae*. Formulae that do not contain $\circ$ or $\bullet$ are called *stutter invariant*.

Let $\Sigma = 2^{AP}$ be the alphabet of states over AP, $w = a_0a_1a_2\ldots \in \Sigma^\omega$ an infinite sequence of states and $i \in \mathbb{N}_0$ a position. Then w_i describes the $i+1$th (counting starts at zero) state of w, and w^i describes the suffix of w starting at i (i. e., $a_ia_{i+1}\ldots$). $w, i \models \varphi$ means that a sequence w *satisfies* the LTL formula φ at position i. One can also say that φ *holds* or *evaluates to true* at position i of w. When considering a propositional formula p as indicator of an event, a position where p is satisfied is an *occurrence* of p.

Semantics. The semantics of LTL is defined inductively as follows:

$$\begin{array}{lcl}
w, i \models \top & & \\
w, i \models p & \iff & p \in w_i \\
w, i \models \neg\varphi & \iff & w, i \not\models \varphi \\
w, i \models \varphi \wedge \psi & \iff & w, i \models \varphi \text{ and } w, i \models \psi \\
w, i \models \varphi \mathbin{\mathrm{U}} \psi & \iff & \exists j | i \leq j : \begin{array}{l} w, j \models \psi \text{ and} \\ \forall k | i \leq k < j : w, k \models \varphi \end{array} \\
w, i \models \circ\varphi & \iff & w, i+1 \models \varphi \\
w, i \models \varphi \mathbin{\mathrm{S}} \psi & \iff & \exists j | i \geq j \geq 0 : \begin{array}{l} w, j \models \psi \text{ and} \\ \forall k | i \geq k > j : w, k \models \varphi \end{array} \\
w, i \models \bullet\varphi & \iff & i \neq 0 \text{ and } w, i-1 \models \varphi
\end{array}$$

Although past operators do not add expressiveness to LTL [GPSS80], their use allows more concise and efficient formulae [Mar03].

Additional operators. Furthermore, we define a number of abbreviations:

$\bot$	$:= \neg\top$	(read *false*)
$\varphi \vee \psi$	$:= \neg(\neg\varphi \wedge \neg\psi)$	(read *logical* or *boolean or*)
$\varphi \rightarrow \psi$	$:= \neg\varphi \vee \psi$	(read *implies*)
$\varphi \leftrightarrow \psi$	$:= (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$	(read *equals*)
$\Diamond\varphi$	$:= \top \mathbin{\mathrm{U}} \varphi$	(read *eventually*)
$\Box\varphi$	$:= \neg\Diamond\neg\varphi$	(read *globally* or *always*)
$\varphi \mathbin{\mathrm{W}} \psi$	$:= (\varphi \mathbin{\mathrm{U}} \psi) \vee (\Box\varphi)$	(read *weak until* or *waiting for*)
$\varphi \mathbin{\mathrm{R}} \psi$	$:= \neg(\neg\varphi \mathbin{\mathrm{U}} \neg\psi)$	(read *releases*)
$\blacklozenge\varphi$	$:= \top \mathbin{\mathrm{U}} \varphi$	(read *once* or *eventually in past*)
$\blacksquare\varphi$	$:= \neg\blacklozenge\neg\varphi$	(read *historically* or *globally in past*)
$\varphi \mathbin{\mathrm{B}} \psi$	$:= (\varphi \mathbin{\mathrm{S}} \psi) \vee (\blacksquare\varphi)$	(read *weak since* or *back to*)
$\varphi \mathbin{\mathrm{T}} \psi$	$:= \neg(\neg\varphi \mathbin{\mathrm{S}} \neg\psi)$	(read *triggered*)
$\bullet_W \varphi$	$:= \neg\bullet\neg\varphi$	(read *weak previous*)

W, B and $\bullet_W$ are called *weak* operators in contrast to their *strong* equivalents U, S and $\bullet$, because they do not require an occurrence of ψ resp. the existence of a previous step in the sequence. When reasoning over finite sequences, a weak next operator $\circ_W$ can be defined similarly to weak previous.

The operators U, W, $\Box$, $\Diamond$, S, B, $\blacksquare$ and $\blacklozenge$ defined here are all *reflexive* or *non-strict*. This means they consider the present as part of both future and past. One can also define *strict* or *non-reflexive* variants of these operators, like for example $\varphi \mathbin{\hat{\mathrm{U}}} \psi := \circ(\varphi \mathbin{\mathrm{U}} \psi)$. We will, however, not use them in this work.

2.4 Ltl on finite sequences

Normally, Ltl formulae are defined over infinite sequences. However, there are two reasons for considering finite sequences too. First, when performing incomplete verification methods (like runtime verification), only a finite, incomplete sequence can be observed and must be sufficient for a decision. Second, specifications may contain the requirement that the evaluation of a formula can be *aborted* on occurrence of a certain event, or that a formula has to be fulfilled *before* a certain event. These situations can be imagined as evaluating the formula on a sequence that has been truncated at the moment when the event occurs.

Incomplete sequences in runtime verification. Finite sequences in runtime verification can be dealt with in two ways. On the one hand, the formula can be modified by replacing all strong operators with weak operators [EFH$^+$03]. This makes that for example an until expression does not fail as long as it could be satisfied by an extension of the sequence. Having to change the formula is however a clumsy, unsatisfying solution. On the other hand, 3-valued semantics can be defined for Ltl on finite sequences that allow the verification tool and its underlying automaton not only to output *true* or *false*, but also *inconclusive* for sequences that do not yet allow a final judgement [ABLS05].

Truncated sequences. The meaning of Ltl on truncated sequences is explored in [EFH$^+$03] and presented shortly in the following. On the one hand, truncation can be performed under connected *strong* and *weak* views. On the other hand, truncation can also be seen under a *neutral* view.

Strong and weak truncation. On a sequence truncated under the weak view, a formula evaluates to true if there is no evidence in the truncated sequence against it. This implies that $\Diamond p$ is always true on such a truncated sequence (because it could still be satisfied by a later occurrence of p)[2]. $\Box p$ is true only if $p = \top$ throughout the whole sequence. Under the strong view a formula evaluates to false unless there is evidence in the truncated sequence that satisfies the formula. This implies that $\Box p$ is always false in such a truncated sequence, and that $\Diamond p$ is only true if there is a state in the sequence that satisfies p.

In other words: a pending formula φ (i. e., a formula that has U or $\circ$ operators referring to states beyond the truncation) yields true resp. false when evaluated in a sequence truncated under the weak resp. strong view. Weak and strong view are dual to each other: negation switches from weak to strong and vice versa.

Truncation of a sequence under the weak and the strong view is connected to the accepton and rejecton operators described in [ABKV03]. They truncate a sequence on occurrence of an abort condition. It is possible to translate a formula containing accepton and rejecton operators into pure Ltl using the definition in [ABKV03], hence they do not add expressiveness to Ltl. The definition of accepton and rejecton will be used in section 4.3.5 of this thesis.

[2]Even $\Diamond\bot$ is true on a sequence truncated under the weak view, although this formula can of course never be satisfied. The Sugar 2.0 Language [BBDE$^+$01], on which the Accellera Property Specification Language PSL is based, contained an `abort` operator with different semantics, which was considered too difficult to be implemented [ABKV03] and therefore changed for version 1.1 of PSL [Acc04].

Neutral truncation. While the weak and the strong view evaluate any pending formula to true resp. false on occurrence of the abort condition, the result of neutral truncation is based on what can be reasonably expected from a truncated sequence: $\Diamond p$ evaluates to false if there is no state satisfying p, as there is no evidence that p would eventually hold. $\Box p$ evaluates to true if p is satisfied throughout the sequence, as there is no evidence that p would eventually not hold. Truncation under the neutral view will be used in section 4.3.5 for defining a stop operator that is needed to translate properties of the form "φ before b".

2.5 Timed LTL

Various extensions of LTL have been proposed in order to express real-time constraints. The most prominent include TPTL [AH94], MTL [Koy90] and MITL [AFH96]. A choice has to be made between *discrete* modelling of time (e. g., time represented by natural numbers) and *dense* modelling of time (e. g., time represented by real numbers), and between *pointwise* semantics (each state is attributed a time value and formulae are evaluated at a position $i \in \mathbb{N}_0$) and *interval* semantics (each state is attributed an interval of time and formulae may be evaluated at any time $t \in \mathbb{R}$) [BCM05]. Alur and Henzinger provide a good overview in [AH92].

However many of the real-time logics mentioned above are hard or impossible to decide. We will therefore use another real-time logic called TLTL (also *Timed* LTL) [D'S03] in this thesis.

Syntax and semantics of TLTL. TLTL extends LTL by two operators $\rhd_{\sim c}\varphi$ and $\lhd_{\sim c}\varphi$ (read *event predicting operator* and *event recording operator*) with $\sim \in \{<, \leq, =, \geq, >\}$ and $c \in \mathbb{R}_0^+$.

Let w be a *timed sequence*, i. e., a sequence where each state w_i is attributed a time stamp τ_i. Then the semantics of the event predicting and event recording operator are defined as follows:

$$w, i \models \rhd_{\sim c}\varphi \iff \begin{array}{ll} \exists j | i < j : & w, j \models \varphi \text{ and} \\ & \forall k | i < k < j : w, k \not\models \varphi \text{ and} \\ & \tau_j - \tau_i \sim c \end{array}$$

$$w, i \models \lhd_{\sim c}\varphi \iff \begin{array}{ll} \exists j | i > j \geq 0 : & w, j \models \varphi \text{ and} \\ & \forall k | i > k > j : w, k \not\models \varphi \text{ and} \\ & \tau_i - \tau_j \sim c \end{array}$$

The semantics of the other operators remain the same.

Expressiveness of TLTL. TLTL is based on *State Clock Logic* proposed by Raskin and Schoebbens in [RS97, RS99] [3]. TLTL is less expressive than other extensions like MITL [RS97]. For example, the MITL formula $\Diamond_{>1.0}p$ ("There is an occurrence of p after more than 1.0 time units, although there might also

[3]The semantics of $\rhd_{\sim c}p$ and $\lhd_{\sim c}p$ defined in [RS97] are slightly different. The semantics used here appear in [RS99], with both operators being strict instead of non-strict. This allows the specification of properties like $\Box(p \rightarrow \rhd_{=1.0}p)$ that could not have been expressed before, as the $\rhd$ operator would have always matched the current p.

be occurrences of p before that") cannot be expressed in Tltl, because $\rhd p$ can only reason about the *next* occurrence of p. Many real-world requirements can be formulated in Tltl, though [RS99].

D'Souza has shown in [D'S03] that Tltl corresponds directly to the first-order fragment of monadic second order logic interpreted over timed words. Because formulae expressed in Ltl correspond to properties defined in first-order logic over words [Kam68], Tltl is a natural real-time extension of Ltl.

Tltl is decidable and can be translated into Event Clock Automata [AFH94, RS99]. Therefore it is suitable for model checking and runtime verification. A runtime verification framework based on Tltl is being developed [ABLS05].

Additional operators. We define a number of abbreviations:

$$
\begin{array}{ll}
\varphi \ \mathrm{U}_{\sim c}\ \psi & := (\varphi \ \mathrm{U}\ \psi) \wedge (\psi \vee \rhd_{\sim c}\psi) \\
\Diamond_{\sim c}\varphi & := \varphi \vee \rhd_{\sim c}\varphi\ ^{4} \\
\Box_{\sim c}\varphi & := \neg\Diamond_{\sim c}\neg\varphi \\
\varphi \ \mathrm{W}_{\sim c}\ \psi & := (\varphi \ \mathrm{U}\ \psi) \vee \Box_{\sim c}\varphi \\
\varphi \ \mathrm{S}_{\sim c}\ \psi & := (\varphi \ \mathrm{S}\ \psi) \wedge (\psi \vee \lhd_{\sim c}\psi) \\
\blacklozenge_{\sim c}\varphi & := \varphi \vee \lhd_{\sim c}\varphi\ ^{4} \\
\blacksquare_{\sim c}\varphi & := \neg\blacklozenge_{\sim c}\neg\varphi \\
\varphi \ \mathrm{B}_{\sim c}\ \psi & := (\varphi \ \mathrm{S}\ \psi) \vee \blacksquare_{\sim c}\varphi
\end{array}
$$

$\sim$ is restricted to be either $<$ or $\leq$. This ensures that the semantics of $\varphi \ \mathrm{U}_{\sim c}\ \psi$ in Tltl are similar to those in Mitl, Mtl and Tptl.

[4] In [RS99], $\Diamond_{\sim c}\varphi$ is defined as $\top \ \mathrm{U}_{\sim c}\ \varphi$, which is an abbreviation for $(\top \ \mathrm{U}\ \varphi) \wedge (\varphi \vee \rhd_{\sim c}\varphi)$. The definition used here is semantically equivalent but shorter.

Chapter 3

Discussion of existing specification formalisms

This chapter provides an overview of existing temporal specification languages and formalisms and discusses their strengths and drawbacks. From the discussion we then derive several design goals for a new specification language.

3.1 Temporal logic

Temporal logic, as presented in the previous chapter, can be seen as a starting point for the development of most formal temporal specification languages. In the following we concentrate on LTL, i. e., on a linear view of time.

3.1.1 Advantages

Clearly defined semantics. Being based on logic, LTL has clearly defined semantics and can be reduced to a small set of core operators, which makes it suitable for formal proofs. This is a big advantage over semi-formal or informal specification methods.

Existing tool support. The semantics of LTL is well accepted, and it is understood by a broad range of verification tools. For instance, the SMV [McM] and SPIN [Hol97] model checkers use specifications based on LTL and CTL syntax. For these reasons LTL is apt as a smallest common denominator for connecting different tools, a task that is hard to perform with a language whose semantics is disputable or subject to change.

3.1.2 Drawbacks

Low level of abstraction. LTL operators allow the specification of requirements only on a rather low level of abstraction. Consider for example the requirement

"A proposition p becomes true at most two times
before the occurrence of end condition r".

A translation to Ltl of this rather simple property yields

$$(\Diamond r) \rightarrow ((\neg p \wedge \neg r) \text{ U } (r \vee ((p \wedge \neg r) \text{ U } (r \vee ((\neg p \wedge \neg r) \text{ U } (r \vee ((p \wedge \neg r) \text{ U } (r \vee (\neg p \text{ U } r)))))))))).$$

This formula does not reflect the initial requirement in an intuitive way. Understanding and changing such a formula is difficult.

Some research papers and tools define Ltl with a minimalistic set of operators in order to simplify proofs and implementation, at the cost of usability. For example, past operators are often disregarded, although they allow more succinct formulae [Mar03], and neither SMV nor SPIN provides the weak until operator W.

Unnecessary repetition of sub-expressions. Many Ltl-based tools do not allow the definition of abbreviations for recurrent sub-expressions. This is particularly painful as Ltl offers only a small set of operators and all other operators have to be constructed from this basic set, which usually involves repeating sub-expressions in the formula.

In the formula mentioned above, the proposition p appears five and the proposition r even ten times. Working with this kind of formula becomes really complicated if p or r are not atomic propositions but more complex expressions, because then one first has to find out which parts of the formula form the sub-expressions.

Proneness to errors. Writing Ltl formulae is difficult, not only for novices but also for experienced users. Even relatively simple requirements sometimes contain pitfalls that lead to defects in the specification. Consider for instance the property

"s precedes p after q".

We can find the following mapping to Ltl in [DAC99]:

$$(\Box \neg q) \vee \Diamond (q \wedge (\neg p \text{ W } s))$$

At first sight, this formula looks correct ("either q never holds or, when q becomes true, there is no p before an s"). Nevertheless, the formula contains a 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. 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. The minimalistic set of operators available in Ltl however intensifies the danger, as it forces users to build complex, error-prone formulae for even very simple requirements, as can be seen above.

Symbolic instead of named operators. Many verification tools, like SMV or SPIN, use an LTL syntax based on symbols rather than textual operator names. This saves time for experienced users who are familiar with logics and who want to write down a formula quickly. It might also simplify parsing a little. However, the resulting formulae are much more difficult to understand for novices, because the gap between a requirement in natural language and the formal specification is wider. Compare for example the formula

```
(<>b) -> ([]p)
```

and a specification in a formal pseudo-language

```
if eventually b then always p
```

to the requirement in simple natural language

"If eventually `b` occurs then `p` has to be true all the time".

The second specification resembles much more the requirement in natural language.

3.2 Pattern-based approaches

A *pattern* provides a solution to a recurrent problem, often including notes about its advantages, drawbacks, and alternatives. It enables inexperienced users to profit from expert knowledge. Furthermore, a pattern system can help to establish a common terminology for a domain.

Design patterns have had great success in software engineering [GHJV94].

3.2.1 Specification Patterns and the Bandera language

Dwyer et al. propose a system of patterns for property specification [DAC99], The patterns are based on a survey of real-world specifications and describe frequently found properties. They consist of *requirements* (e. g., "absence"—a condition is false or "response"—an event triggers another one) that can be expressed under different *scopes*. The available scopes are "globally", "before an event r", "after an event q", "between two events r and q" and "after-until", which is similar to "between" but does not require the occurrence of the second event. For each combination of requirement and scope, corresponding parameterisable formulae in various formalisms (including LTL) are provided. For instance, the "absence-of-p-before-r" pattern is expressed in LTL as

$$(\Diamond r) \rightarrow (\neg p \text{ U } r).$$

Dwyer et al. convincingly argue that scopes are needed in many real-world specifications but supported by very few languages. However, specification patterns as defined by Dwyer et al. cannot be nested: only propositional formulae may be used as parameters. Adding a new requirement to the pattern system means having to manually write an LTL formula for each scope.

The Bandera Specification Language [CDHR01] uses a textual representation of the specification patterns. A compiler that translates such specifications into LTL is part of the Bandera system.

3.2.2 Real-time specification patterns

Similar to the specification pattern system by Dwyer et al. [DAC99], Konrad and Cheng define a pattern system for the specification of real-time properties [KC05]. They identify five frequently used real-time requirements (e. g., "maximum duration" – a condition p may not last more than c time units) and provide a translation under five scopes into various formalisms, including MTL. They also provide a structured grammar that can be used to express the property by a standardised English sentence. A mapping to TLTL is not included but possible.

3.2.3 Discussion

Higher level of abstraction. Specification patterns deal with several of the problems of LTL by raising the level of abstraction. This reduces the complexity for the average user, as they no longer have to write LTL by hand. In particular, expressing a requirement under the "before"-scope becomes much easier. Referring to the name of the pattern makes clear what the intention of the specification is and frees the reader from the need to understand an LTL formula first. The problem of repeating sub-expressions vanishes if the patterns are used within a system that automatically instantiates a pattern for a given set of parameters, as for instance the Bandera system.

Limited expressiveness. Although many requirements can be expressed by one of the patterns or a variation, the expressiveness of the pattern system is limited to known patterns. Users who face a new problem have to go back to handwritten LTL formulae.

Furthermore, the five scopes proposed for the pattern system cover only a few situations. For instance, scopes could be defined to include or exclude the delimiting events (i. e., to form closed or open intervals). Various interpretations are also possible for the case that a delimiting event never occurs. Dwyer et al. describe (in the notes section of their pattern system) how to modify the formulae to match these situations. However, this forces the user to either work with a big predefined pattern catalogue that contains all combinations of variations, or to modify the formulae by hand. The latter requires good understanding of LTL and breaks up the pattern as a "black box" with a clearly defined meaning and an error-free translation.

Limited freedom in the combination of patterns. The specification patterns proposed by Dwyer et al. cannot be freely combined. While propositional formulae may always be inserted as parameters, the use of temporal formulae can lead to problems and often requires manual modification of the pattern. In particular, nesting two patterns is usually not possible. This limits the expressiveness of the pattern system and forces users to write or modify LTL formulae by hand for more complicated requirements.

3.3 Specification languages from the hardware domain

In the domain of hardware design, formal methods are used in industrial practice. Various domain-specific high-level specification languages have therefore been developed, two of which—namely Sugar/PSL and ForSpec—are presented in this section.

Both of them have higher expressiveness than Ltl and hence cannot be translated completely into Ltl. Usually they are used together with specialised, proprietary tools. Nevertheless, they deserve examination for our aim of defining a high-level specification language that is translatable into Ltl/Tltl, as the set of operators they provide reflects practical experience with the needs of the users. Also, some parts of those languages may be expressible in Ltl although the language as a whole is not.

The feature that usually prohibits translation of a language into Ltl is the inclusion of unrestricted regular expressions. Wolper has shown in [Wol83] that there are properties that can be expressed using regular expressions but not in Ltl. Languages that nevertheless include regular expressions are usually translated directly into a Büchi automaton that is generated during model checking anyway.

3.3.1 Sugar/PSL

The Sugar/PSL language is a high level specification language aimed mainly at hardware design. Sugar [BBDE$^+$01] was developed since 1994 at IBM Haifa as a "syntactic sugaring" of the branching time logic Ctl. It was selected as a candidate for a standard property specification language by the Accellera Formal Verification Technical Committee in 1998. Sugar 2.0 was based on a linear view of time while keeping branching time as an optional extension. It became the Accellera Property Specification Language PSL [Acc04]. After some minor corrections and modifications in version 1.1, PSL is undergoing standardisation by the IEEE [FMW05].

PSL is structured into four *layers* (boolean, temporal, verification and modelling). The boolean layer provides operators for propositional logic, while the operators of the temporal layer are used to combine propositional formulae 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.

In order to provide compatibility to different verification tools, PSL comes in different *flavors* (SystemVerilog, Verilog, VHDL and GDL). Each flavor has a slightly different syntax (mainly for the boolean and modelling layer) that is adapted to the underlying hardware description language.

PSL provides a rich set of operators for reasoning over boolean conditions (e. g., bit vector operations) and for regular expressions (named SEREs). A so-called clocking operator allows to state that an expression is evaluated only in steps where its clocking condition holds. The PSL abort operator can be used to model resets: it evaluates a pending expression to false on occurrence of an 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. PSL does not contain temporal past operators and no means for specifying real-time constraints.

PSL is usually 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. A translation of PSL into LTL is possible only for a subset of the language [TS05].

3.3.2 ForSpec/OpenVera$^{\text{TM}}$Assertions

The ForSpec Temporal Logic (FTL) [AFF$^+$02] is a temporal specification language developed at Intel and based on a linear view of time. It is tailored to the formal verification of hardware design. Much like Sugar/PSL, ForSpec provides regular and clocked expressions as well as accept and reject operators for modelling resets. ForSpec also contains limited support for references to the past. It does not address real-time properties. It has been included into the OpenVera$^{\text{TM}}$Assertions language [Syn02, Syn03]. ForSpec cannot be translated completely into LTL, as it provides regular expressions.

3.3.3 Discussion

Hardware-specific features. Both Sugar/PSL and ForSpec contain various features that are tailored to the verification of hardware design, like bit vector operations and a clocking operator. Sugar/PSL even defines some of its syntax depending on the underlying hardware description language. These features are superfluous or even disturbing when using these languages as general purpose specification formalisms.

Limited compatibility to LTL. Both languages include regular expressions, which prohibit a complete translation into LTL. Restricting regular expressions to a translatable subset seems feasible, but would probably remove a large set of operators.

Lack of scope operators. Neither one of the languages provides good support for scope statements.

Sugar/PSL contains the operator `next_event`, which is similar to the "after"-scope defined by Dwyer et al., and the operator `before`, which is similar to the "existence-before" pattern. The operators exist in inclusive/exclusive and weak/strong variants. Applying scopes to arbitrary expressions is however not possible.

ForSpec provides the operators `followed_by` and `triggers` that are similar to the "after"-scope. A "before"-operator is missing.

No support for real-time constraints. Neither Sugar/PSL nor ForSpec provide operators for the specification of real-time constraints.

3.4 Other approaches

3.4.1 EAGLE

EAGLE [BGHS04b] 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 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. EAGLE also allows the specification of real-time constraints. A subset with equal expressiveness as LTL can be defined [BGHS04a].

3.4.2 Temporal Rover assertions

The Temporal Rover [Dru00] is a verification tool that checks temporal logic formulae included as assertions in the source code of a program. The input syntax is based on LTL and MTL and provides both textual (`Always`) and symbolic (`[]`) syntax. It also contains two operators that allow statements about the repetition of events.

The author is not aware of a compiler that translates Temporal Rover specifications into pure LTL. Implementing such a tool seems however possible.

3.4.3 Discussion

EAGLE. The fix-point semantics employed in EAGLE is rather difficult to understand for novices without a mathematical background. Therefore, EAGLE does not seem to be a promising basis for developing an easy-to-use specification language.

Temporal Rover assertions. Temporal Rover assertions resemble LTL in many aspects and do not add much comfort for the user.

3.5 Design goals for a new specification language

In the following we define a number of design goals for a new specification language that shall remedy the weaknesses mentioned above.

High level of abstraction. The language shall provide a sufficiently high level of abstraction. This can improve readability and reduce the probability of introducing errors. It can also avoid repetition of sub-expressions. For instance, the language should support scope statements.

Similarity to natural language or programming languages. Specifications shall be easy to read and to understand. This can be achieved if they resemble natural language as much as possible without becoming ambiguous. Similarities to programming languages ease the understanding of the language for engineers.

Avoiding implicit assumptions. Assuming a default meaning for an ambiguous statement can shorten specifications, but may lead to misinterpretations. Because temporal specifications are usually small and rather complex, it is more important to find as many errors as possible during compilation than to reduce the number of characters a user has to type. Therefore, users shall be forced to write down their requirements in an explicit and unambiguous way.

Extendability. The language shall provide the possibility to define macros in a flexible way and to thereby extend the set of available operators. This enables the user to adapt the language to different domains.

Real-time support. The language shall allow the specification of real-time constraints, needed in particular for the verification of reactive systems. The timed subset shall be clearly separated from the untimed subset, so that users are free to choose whether they do or do not want to use real-time.

Translatability into LTL and TLTL. The untimed subset of the language shall be completely translatable into LTL because a wide range of tools accepts LTL formulae as input. The timed language shall be translatable into TLTL.

In order to make the practical usage of the language possible, the formulae that result from the compilation process must be about as efficiently to check as average handwritten formulae.

Chapter 4

Features of the Salt language

A new specification language called Salt is developed in this thesis. Salt stands for **S**tructured **A**ssertion **L**anguage for **T**emporal Logic. This chapter presents the main features of Salt and describes in detail how a mapping to Ltl can be obtained for two non-trivial aspects, namely the scope operators and simplified regular expressions. For a complete language reference, see the Salt manual in the appendix. The manual also contains a translation schema that defines the formal semantics of Salt, a tutorial and various example specifications.

All but section 4.4 of this chapter refers to the untimed subset of Salt, which can be translated into Ltl. Section 4.4 deals with the timed extension and its translation into Tltl.

4.1 General structure

Assertions

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

Salt mainly uses textual operators, so that the frequently used Ltl formula

$$\Box(p \rightarrow \Diamond q)$$

would be written as

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

Layering

Basically, the Salt language consists of the following three layers, each covering different specification aspects:

- *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. Similar to the temporal layer, it is divided into a future and a past fragment.

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

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

4.2 Propositional layer

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

Atomic propositions

Boolean propositions are the atomic elements from which Salt expressions are built. They usually correspond to boolean variables, signals or expressions of the system under test.

Referring to boolean variables is simple: every identifier that appears in a Salt specification and that was not previously defined as a macro or a formal parameter is treated as an atomic proposition. This means it appears in the output as written in the specification.

Also more complex expressions, such as predicates over integer numbers, can be employed as boolean propositions, as long as a subsequent tool can evaluate them to either true or false. Such complex expressions must be enclosed in `""` in a Salt specification and appear unchanged in the output. For example,

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

is a valid Salt specification and results in the output

```
LTLSPEC state!=ERROR
```

The ability to process arbitrary text as atomic propositions makes the Salt language independent from the domain of application and the models and tools that influence the rest of the verification process. The Salt compiler can be customised using proposition parser plugins performing checks and/or transformations on the atomic propositions. This allows to detect syntax errors rapidly. For instance, a custom proposition parser could check that all propositions are valid Java expressions.

Boolean operators

The following boolean operators can be used in SALT:

Boolean operator	SALT operator
$\neg$	`!` or **`not`**
$\wedge$	`&` or **`and`**
$\vee$	`|` or **`or`**
$\rightarrow$	`->` or **`implies`** or **`if-then-else`**
$\leftrightarrow$	`<->` or **`equals`**

The conditional operators **`if-then`** and **`if-then-else`** tend to make specifications easier to read, because **`if-then-else`** constructs are familiar to programmers of almost every language. Similar operator appear in the ForSpec language. With the **`if-then`** operator, the example from above[1] could be rewritten as

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

4.3 Temporal layer

This section deals with the heart of the SALT language, the temporal layer. The temporal layer consists of a future and a symmetrical past fragment. Although past operators do not add expressiveness [GPSS80], they can help to write formulae that can be understood more easily and processed more efficiently [Mar03].

SALT provides a past operator for every future operator presented here, including operators like **`occurring`** or the scope operators. Future operators are translated using only LTL future operators, and past operators are translated using only LTL past operators. This leaves the user the choice whether they do or do not want to use past operators (some model checkers, for instance SMV, allow the use of past operators in LTL formulae, while others like SPIN do not).

In the following, we concentrate on the future fragment only.

4.3.1 Standard LTL operators

SALT provides the common LTL operators. Untimed SALT therefore has the same expressiveness as LTL.

LTL operator	SALT
U	**`until`**
W	**`until weak`**
R	**`releases`**
$\Box$	**`always`**
$\Diamond$	**`eventually`**
$\circ$	**`next`**

[1]The example uses operators from both the temporal and the propositional layer.

4.3.2 Extended operators

SALT also provides a number of extended operators that express frequently used requirements. Some are rather simple abbreviations, but yield more intuitive specifications (e. g., **`never`**). Other operators provide a concise way to specify requirements that are awkward to express in LTL, like for example the fact that an event occurs a certain number of times in the future.

- **`never`**. The **`never`** operator is similar to **`always`** but requires that a formula never holds. Sugar/PSL provides a similar operator.
- Extended **`until`**. SALT provides an extended version of the LTL U operator. The users can specify whether they want it to be *exclusive* (φ has to hold until the moment ψ occurs) or *inclusive* (φ has to hold until and during the moment ψ occurs) [2]

  They can also chose whether the end condition is *required* (must eventually occur), *weak* (may or may not occur) or *optional* (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, the 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 [3].

  A similar operator can be found in Sugar/PSL and—with less flexibility—in Temporal Rover assertions.

4.3.3 Counting quantifiers

SALT provides two operators **`occurring`** and **`holding`** allowing to specify that an event has to occur a certain number of times in the future. **`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, expressing, e. g., 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

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

The corresponding SALT specification is written as

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

The **`occurring`** operator is somewhat similar to the `Repeated` operator of Temporal Rover assertions.

[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.

[3]Note that this operator refers only to states at certain positions in the sequence, not to real-time constraints.

4.3.4 Exception operators

SALT includes the exception operators **rejecton** and **accepton** that interrupt the evaluation of a formula on occurrence of an exceptional 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.

Exception operators can be useful, for example, when specifying a communication protocol that requires certain messages to be exchanged 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 1.1.

The translation of the exception operators uses an intermediate language SALT--, in which most other SALT operators have already been replaced by LTL expressions. Exception operators cannot be translated by simple pattern replacement, but require weaving the abort condition into the whole subexpression. This can be done more easily if the set of operators in the expression is small. The formal semantics of LTL enriched with exception operators operators (called Reset-LTL) is explored in [ABKV03], and—from a slightly different point of view—in [EFH$^+$03].

4.3.5 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 in 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**. Scope operators in SALT can be used with arbitrary formulae, even with nested scope operators. It is furthermore possible to specify whether the delimiting events are part of the interval (*inclusive*) or not (*exclusive)* and whether the occurrence of the delimiting events is strictly required or not. Figure 4.1 illustrates the different scopes.

Scopes appear as an important issue in the specification pattern system and the Bandera language. However, the 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. The flexibility of the scope operators in SALT is a distinguishing feature of the language, and the author does not know of another specification language that offers a similar degree of freedom.

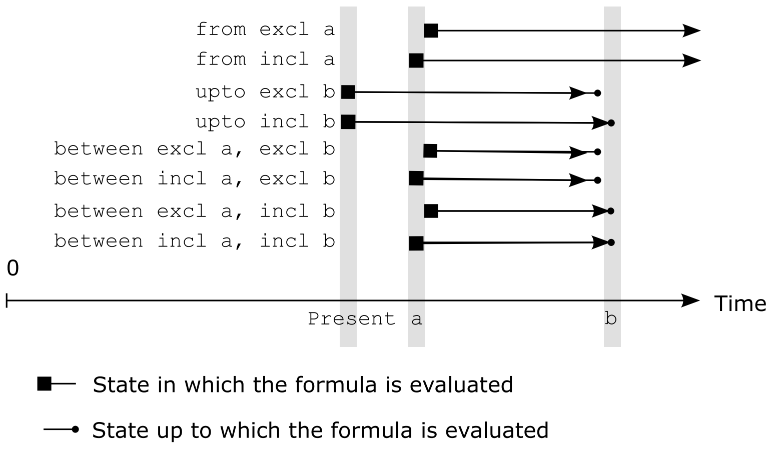


Figure 4.1: Inclusive and exclusive semantics of scope operators

Challenging properties of the **`upto`** operator

While it is possible to implement a translation of the **`from`** operator into LTL relatively straightforward, the **`upto`** operator proves to be more difficult, as we will see in the following example.

A specification

```
assert always φ upto b
```

expresses the fact that φ must hold always until the occurrence of end condition b. A naïve translation into LTL would be

$$\varphi \mathrm{~W~} b.$$

This translation is in order for a purely propositional φ, but might be wrong when temporal operators are used within φ. 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 this naïve translation, although `s` occurs after the end condition `b`, which clearly violates our intuitions. In order to express the intuitively expected semantics, the negated end condition b has to be inserted into the U and ○ statements of φ in various places, for example like this:

$$(p \rightarrow (\neg b \mathrm{~U~} (\neg b \wedge s))) \mathrm{~W~} b$$

Dwyer et al. outline this procedure in the notes of their specification pattern system [DAC99]. It is however a lengthy and error-prone work if done manually.

In the following section, we therefore define the stop operator that allows us to represent specifications like the example from above correctly and to automate their translation. The stop operator is introduced when translating **`upto`** and **between** operators into SALT-- (i. e., it never appears in a SALT specification directly). It is then transformed into an LTL expression in a similar way as the **rejecton** and **accepton** operators.

The example from above yields a SALT-- formula like

$$((p \rightarrow \Diamond s) \text{ stop } b)\mathrm{W}b$$

with stop b expressing that $(p \rightarrow \Diamond s)$ shall not take into account states after the occurrence of b.

Formal derivation of a translation into LTL

The semantics of the stop operator can be imagined as evaluating the argument φ on a sequence that has been cut on occurrence of the end condition b. A $\Diamond$ operator whose argument has not been satisfied yet when reaching the end of the sequence evaluates to false. Consequently, a $\Box$ operator whose argument has not been falsified yet evaluates to true. This is exactly the semantics of LTL on sequences truncated under the neutral view, which are defined in [EFH$^+$03] as follows (with w being a truncated sequence $\neq \varepsilon$ and $i < |w|$):

$$
\begin{array}{lcl}
w, i \models p & \iff & p \in w_i \\
w, i \models \neg\varphi & \iff & w, i \not\models \varphi \\
w, i \models \varphi \wedge \psi & \iff & w, i \models \varphi \text{ and } w \models \psi \\
w, i \models \circ\varphi & \iff & i < |w| - 1 \text{ and } w, i+1 \models \varphi \\
w, i \models \varphi \text{ U } \psi & \iff & \exists j | i \leq j < |w| : (w, j \models \psi) \text{ and} \\
 & & \quad \forall k | i \leq k < j : (w, k \models \varphi)
\end{array}
$$

We introduce two variants of stop: an *exclusive* one, which limits the evaluation of φ to the time before the step when b occurs, and an *inclusive* one, which takes this step still into account. However, there is a problem with the exclusive stop: If the end condition occurs immediately, we have to evaluate φ on an empty sequence. Empty sequences have a meaning under the weak/strong view, but not under the neutral view. It is not clear whether p $\text{stop}_{\text{excl}}$ b should be true or false if p and b occur simultaneously. SALT addresses this problem by requiring the user to specify explicitly for an exclusive **upto** which semantics they want, unless it can be concluded from the nature of φ.

Definitions. We define $w_{[a...b[}$ as the section of w that starts at position a (inclusive) and goes up to position b (exclusive) and $w_{[a...b]}$ as the section of w that starts at position a (inclusive) and goes up to position b (inclusive). Instead of an index b, we can also use the occurrence of an end condition c_{end} as delimiter of the section, and write $w_{[a...c_{end}[}$ resp. $w_{[a...c_{end}]}$ for the section of sequence w beginning at position a (inclusive) and going up to the position where c_{end} occurs (exclusive resp. inclusive). Only c_{end} at positions $\geq a$ are considered. If $b = \infty$ or $b \geq |w|$ or if c_{end} never occurs, the section is equal to w^a.

Inclusive semantics. We first deal with the inclusive variant of stop and develop a translation schema that replaces the operator and its arguments by an Ltl expression. As a base, we start with the semantics defined above and rewrite it for sections of sequences delimited by two indices a and b (inclusive). It is required that $a \leq b$, so that the section is not empty.

$$\begin{array}{lcl}
w_{[a...b]} \models p & \iff & p \in w_a \\
w_{[a...b]} \models \neg\varphi & \iff & w_{[a...b]} \not\models \varphi \\
w_{[a...b]} \models \varphi \wedge \psi & \iff & w_{[a...b]} \models \varphi \wedge w_{[a...b]} \models \psi \\
w_{[a...b]} \models \circ\varphi & \iff & a+1 \leq b \wedge w_{[a+1...b]} \models \varphi \\
w_{[a...b]} \models \varphi \text{ U } \psi & \iff & \exists j | a \leq j \leq b : (w_{[j...b]} \models \psi) \wedge \\
 & & \quad \forall k | a \leq k < j : (w_{[k...b]} \models \varphi)
\end{array}$$

The semantics for $\circ$ and U can be rewritten by changing the comparisons between positions into simple tests of equality.

$$\begin{array}{lcl}
w_{[a...b]} \models \circ\varphi & \iff & a \neq b \wedge w_{[a+1...b]} \models \varphi \text{ }^{4} \\
w_{[a...b]} \models \varphi \text{ U } \psi & \iff & \exists j | a \leq j : (w_{[j...b]} \models \psi) \wedge \\
 & & \quad (\forall k | a \leq k < j : (w_{[k...b]} \models \varphi \wedge k \neq b))^{5}
\end{array}$$

The equality tests can then be expressed by tests on the occurrence of an end condition c_{end} at position b using

$$a = b \iff c_{end} \in w_a \qquad \text{and} \qquad a \neq b \iff c_{end} \notin w_a$$

$a \leq b$ is ensured automatically as we consider only c_{end} at positions $\geq a$.

$$\begin{array}{lcl}
w_{[a...c_{end}]} \models \circ\varphi & \iff & (c_{end} \notin w_a) \wedge (w_{[a+1...b]} \models \varphi) \\
w_{[a...c_{end}]} \models \varphi \text{ U } \psi & \iff & \exists j | a \leq j : (w_{[j...c_{end}]} \models \psi) \wedge \\
 & & \quad (\forall k | a \leq k < j : ((w_{[k...c_{end}]} \models \varphi) \wedge \\
 & & \quad (c_{end} \notin w_k)))
\end{array}$$

The right side of the equations can be expressed using Ltl operators and yield the following inductively defined translation function $\mathrm{T}(\varphi)$ for the inclusive stop operator:

$$\begin{array}{lcl}
\mathrm{T}(p \text{ stop}_{\text{incl}}\ c_{end}) & = & p \\
\mathrm{T}((\neg\varphi) \text{ stop}_{\text{incl}}\ c_{end}) & = & \neg\mathrm{T}(\varphi \text{ stop}_{\text{incl}}\ c_{end}) \\
\mathrm{T}((\varphi \wedge \psi) \text{ stop}_{\text{incl}}\ c_{end}) & = & \mathrm{T}(\varphi \text{ stop}_{\text{incl}}\ c_{end}) \wedge \mathrm{T}(\psi \text{ stop}_{\text{incl}}\ c_{end}) \\
\mathrm{T}((\circ\varphi) \text{ stop}_{\text{incl}}\ c_{end}) & = & (\circ\mathrm{T}(\varphi \text{ stop}_{\text{incl}}\ c_{end})) \wedge \neg c_{end} \\
\mathrm{T}((\varphi \text{ U } \psi) \text{ stop}_{\text{incl}}\ c_{end}) & = & (\mathrm{T}(\varphi \text{ stop}_{\text{incl}}\ c_{end}) \wedge \neg c_{end}) \text{ U} \\
 & & \quad \mathrm{T}(\psi \text{ stop}_{\text{incl}}\ c_{end})
\end{array}$$

Exclusive semantics. The definition of the exclusive stop operator works similarly. We start with the semantics of Ltl on sections of sequences delimited by two indices a (inclusive) and b (exclusive). It is required that $a < b$, so that at least w_a is contained in the section.

[4] $a \leq b$ together with $a \neq b$ ensures that $a + 1 \leq b$.

[5] $a \leq b$ together with $\forall k | a \leq k < j : k \neq b$ ensures that $j \leq b$.

$$
\begin{array}{lll}
w_{[a...b[} \models p & \Longleftrightarrow & p \in w_a \\
w_{[a...b[} \models \neg\varphi & \Longleftrightarrow & w_{[a...b[} \not\models \varphi \\
w_{[a...b[} \models \varphi \wedge \psi & \Longleftrightarrow & w_{[a...b[} \models \varphi \wedge w_{[a...b[} \models \psi \\
w_{[a...b[} \models \circ\varphi & \Longleftrightarrow & a+1 < b \wedge w_{[a+1...b[} \models \varphi \\
w_{[a...b[} \models \varphi \text{ U } \psi & \Longleftrightarrow & \exists j | a \leq j < b : (w_{[j...b[} \models \psi) \wedge \\
 & & \quad \forall k | a \leq k < j : (w_{[k...b[} \models \varphi)
\end{array}
$$

In a similar way as before we rewrite the comparisons between positions into tests of equality and replace the equality tests by tests on the occurrence of end condition c_{end} at position b. It is required that $c_{end} \notin w_a$ in order to ensure that the sequence is not empty ($a < b$).

$$
\begin{array}{lll}
w_{[a...c_{end}[} \models \circ\varphi & \Longleftrightarrow & (c_{end} \notin w_{a+1}) \wedge (w_{[a+1...b[} \models \varphi) \\
w_{[a...c_{end}[} \models \varphi \text{ U } \psi & \Longleftrightarrow & \exists j | a \leq j : (w_{[j...c_{end}[} \models \psi) \wedge (c_{end} \notin w_j) \wedge \\
 & & \quad (\forall k | a \leq k < j : ((w_{[k...c_{end}[} \models \varphi) \wedge \\
 & & \quad (c_{end} \notin w_k)))
\end{array}
$$

We are now ready to define the translation function for the exclusive stop operator. However, we have to deal with the problem of empty sequences, as the above formulae are valid only for $c_{end} \notin w_a$. SALT resolves this problem by constraining the argument of **`upto`** statements:

- weak variant: the users can specify explicitly that they want the expression to be true if c_{end} occurs immediately. In this case, a $\vee c_{end}$ is added. It therefore does not matter what the rest of the formula evaluates to if c_{end} is true.
- strong variant: the users can specify explicitly that they want the expression to be false if c_{end} occurs immediately. In this case, a $\wedge\neg c_{end}$ is added. It therefore does not matter what the rest of the formula evaluates to if c_{end} is true.
- normal variant: if none of the above is specified, the argument of **`upto`** is required to be one of
  - $\varphi \text{ U } \psi$ (the above definition of U automatically evaluates to false if c_{end} occurs immediately, and thereby fulfils the intuitively expected semantics)
  - $\varphi \wedge \psi$ (if φ and ψ both recursively meet the same requirements)
  - $\neg\varphi$ (if φ recursively meets the same requirements)
  - $\Box\varphi$, $\Diamond\varphi$ (automatically show the desired semantics when expressed using U)

The translation for the exclusive stop operator is defined as follows:

$$
\begin{aligned}
\mathrm{T}(p \text{ stop}_{\text{excl}}\ c_{end}) &= p\\
\mathrm{T}((\neg\varphi) \text{ stop}_{\text{excl}}\ c_{end}) &= \neg\mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ c_{end})\\
\mathrm{T}((\varphi \wedge \psi) \text{ stop}_{\text{excl}}\ c_{end}) &= \mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ c_{end}) \wedge \mathrm{T}(\psi \text{ stop}_{\text{excl}}\ c_{end})\\
\mathrm{T}((\circ\varphi) \text{ stop}_{\text{excl}}\ c_{end}) &= \circ(\mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ c_{end}) \wedge \neg c_{end})\\
\mathrm{T}((\varphi \;\mathrm{U}\; \psi) \text{ stop}_{\text{excl}}\ c_{end}) &= (\mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ c_{end}) \wedge \neg c_{end}) \;\mathrm{U}\\
&\qquad (\mathrm{T}(\psi \text{ stop}_{\text{excl}}\ c_{end}) \wedge \neg c_{end})
\end{aligned}
$$

4.3.6 Regular expressions

Regular expressions provide a convenient way to express complex patterns of events. They appear in many specification languages like Sugar/PSL, ForSpec or EAGLE. Furthermore, the concept of regular expressions is known to many programmers, as it appears in text searching[6] or in the programming language Perl.

However, regular expressions can not be generally translated into Ltl, as the regular languages are a real superset of the languages expressible using Ltl [Wol83]. Other specification languages, like for instance Sugar/PSL, can ignore this fact as they are translated directly into an automaton representation, which has a higher expressiveness than Ltl [TS05].

Still, many interesting properties can be expressed by regular expressions that are translatable into Ltl, and therefore Salt includes support for such simplified regular expressions.

Expressiveness of regular expressions

Regular expressions over an alphabet Σ are normally defined using the operators . (concatenation), $\cup$ (union) and $*$ (Kleene star). The complement operator $\overline{}$ is sometimes added, although all regular languages can be expressed without it.

A regular language is expressible by an Ltl formula if and only if the language is *star-free*. This has been shown by proving equal expressiveness of First-Order Monadic Logic of Order to Ltl [Kam68, GPSS80] and regular star-free languages [MP71]. Star-freeness means that the language can be described by a regular expression using concatenation, union and complement, but not the Kleene star. Note that there are regular expressions using the Kleene star that still describe a star-free language. In particular, the Kleene closure of any subset of Σ is star-free, because it can be expressed as $\Gamma* = \overline{\overline{\emptyset}\Sigma \setminus \Gamma\overline{\emptyset}}$.

Another way to ensure star-freeness is to require that the corresponding finite automaton (DFA) is counter-free. A counter for a string u in a DFA is a sequence $q_0, q_1, \ldots, q_{m-1}$ of distinct states with $\delta * (q_i, u) = q_{i+1}$ for all $i < m$ with $m > 1$ and $q_m = q_0$ [MP71].

While various algorithms exist for translating regular expressions into automata, the translation of star-free regular expressions into Ltl formulae is less explored. Zuck presents such an algorithm [Zuc86], which however has non-elementary complexity.

[6]e. g., Menu Search – Search in the Eclipse Workbench (`http://www.eclipse.org`).

Regular expressions in Salt

Salt regular expressions provide concatenation (`;`), union (`|`) and Kleene star operators (`*`), but no complement operator. The argument of a Kleene star is required to be a purely 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. It seems that many relevant properties can be expressed conveniently without the complement operator (Sugar/PSL does not provide a complement operator either).

Additionally, Salt provides operators that do not increase the expressiveness of its regular expressions, but make their use more convenient:

- The overlapping sequence operator `:` is inspired by the Sugar/PSL language 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 to specify that an expression has to hold at least, at most, exactly or 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[7].

Formal derivation of a translation into Ltl

Let ρ be a regular expression using ., $\cup$ and $*$ over propositions with the constraint that the argument of any Kleene star is a purely propositional formula.

ρ can be transformed using the equivalences

$$(\rho_1.\rho_2).\rho_3 \iff \rho_1.(\rho_2.\rho_3) \quad \text{and} \quad (\rho_1 \cup \rho_2).\rho_3 \iff (\rho_1.\rho_3) \cup (\rho_2.\rho_3)$$

so that the left argument of a concatenation operator is either a purely propositional formula or a Kleene star expression. When considering regular expressions that match finite prefixes of infinite sequences, as we do for Salt regular expressions, a trailing Kleene star expression can be replaced by $\top$ (the language it defines includes ε, which is a prefix to any sequence).

Now, the semantics of a regular expression ρ on a sequence w of propositions is defined as follows:

$$\begin{array}{lcl}
w \models_{RE} p & \iff & p \in w_0 \\
w \models_{RE} p.\rho & \iff & w \models_{RE} p \text{ and } w^1 \models_{RE} \rho \\
w \models_{RE} p*.\rho & \iff & \exists i | 0 \leq i : w^i \models_{RE} \rho \text{ and } \forall j | 0 \leq j < i : w^j \models_{RE} p \\
w \models_{RE} \rho \cup \psi & \iff & w \models_{RE} \rho \text{ or } w \models_{RE} \psi
\end{array}$$

[7]This is true for regular expressions made of propositional formulae. The last element of a Salt regular expression is however allowed to be an arbitrary Salt expression. This includes expressions like `(always a)`, which does ensure that `a` remains true forever on an *infinite* sequence.

The same semantics can be expressed using LTL operators and yield the following translation function $\mathrm{T}(\varphi)$:

$$\begin{aligned}
\mathrm{T}(p) &= p \\
\mathrm{T}(p.\rho) &= p \wedge \circ\mathrm{T}(\rho) \\
\mathrm{T}(p * .\rho) &= p\ \mathrm{U}\ \mathrm{T}(\rho) \\
\mathrm{T}(\rho_1 \cup \rho_2) &= \mathrm{T}(\rho_1) \vee \mathrm{T}(\rho_2)
\end{aligned}$$

The translation schema actually used by the SALT compiler is slightly more complicated due to the additional operators and optimisation reasons.

4.4 Timed layer

SALT contains a timed extension that allows the specification of real-time constraints. Timed operators are translated using the event predicting and event recording operator of TLTL, a timed variant of LTL. For better readability, the translation schema uses an intermediate language containing the extended TLTL operators defined in 2.5.

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

- **next timed**[$\sim c$]φ
  states that the next occurrence of φ is within the time bounds $\sim c$. This corresponds to the TLTL operator $\rhd_{\sim c}\varphi$.
- φ **until timed**[$\sim c$] ψ
  states that φ is true until the next occurrence of ψ, and that this occurrence of ψ is within the time bounds $\sim c$. 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 $\sim c$.

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

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

could have three different meanings:

1. p has to hold during the next 3 time units or until the occurrence of b.

2. p has to hold during the next 3 time units and b is not allowed to occur during this time.

3. p has to hold during the next 3 time units regardless of whether b occurs.

A look at the real-time specification pattern system [KC05] shows that this kind of ambiguity actually occurs: the maximum duration pattern ($\neg\Diamond\Box_{\leq c} p$) under the "before" scope follows interpretation 1, while the bounded invariance pattern ($\Box(p \rightarrow \Box_{\leq c} s)$) follows interpretation 2.

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

The Salt formulae yielding the correct semantics for the above example are:

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

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

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

4.5 Macros and parameterised expressions

Salt allows user-defined macros and parameterised sub-expressions. The use of macros can help to make a specification easier to understand, because complicated sub-expressions can be externalised and accessed by a name that explains what the sub-expression stands for. Sub-expressions that are used several times have to be written down only once and can even be instantiated with different concrete values:

```
define processends(name) := eventually $name$_end
assert processends(main)  -- yields eventually main_end
assert processends(sockethandler)
assert processends(eventhandler)
```

User-defined macros in Salt can be called in the same ways as built-in operators. Within certain limits[8], this allows the user to extend the Salt language with their own operators. For example, the macro in the following example is called in infix notation:

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

[8]For instance, no custom counting quantifier operators can be defined by the user.

Iteration operators allow to instantiate a parameterised sub-expression or macro with a list of values provided by the user.

```
-- A list with input_1 | input_2 | input_3:
assert someof enumerate[1..3] as i in input_$i$
```

```
-- A list containing !(always a) & !(always !b) &
   !(always c):
assert noneof list [a, !b, c] as i in always i
```

User-defined macros and iteration over parameterised expressions are a part of many high-level specification languages, for instance Sugar/PSL, although not all offer a similar flexibility and comfort as SALT for calling macros.

Chapter 5

Implementation of the SALT compiler

This chapter describes how the compiler that translates the SALT language into LTL/TLTL has been implemented.

5.1 Architecture considerations

This section describes the different options that were considered for the architecture of the SALT compiler. The final choice is then explained in section 5.2.

5.1.1 General conditions

The following issues had to be considered:

- The SALT compiler is a prototype. It will probably rather be used in research projects than in a professional environment. Therefore, usability and error handling are not primary goals, although they should not be neglected either.
- The SALT language was developed in an evolutionary way and might undergo further changes in the future. Therefore, the compiler architecture had to allow for easy and fast adaptations to the compiler.
- Compilation speed is unimportant, because temporal specifications are short. Compilation time is small compared to the time needed to create the specification and to run the formula through a model checker. The same applies to memory space requirements.
- Macro expansion capabilities are required. As the specification language was supposed to include user-defined macros, a solution that provided macro expansion at little cost was advantageous.

5.1.2 Front end

For the preprocessing and parsing parts of the compiler, the following choices had to be made.

External preprocessor or internal macro processing. One possibility to support user-defined macros is the use of an existing preprocessor, like the C preprocessor or the M4 general purpose macro preprocessor. This avoids reinvention of the wheel, but it restricts the syntax of the language in what concerns macros and possibly also token types.

LALR(k) versus LL(k) parser. Most parser generators available produce either LALR(k) parsers (bottom-up parsers, e. g., lex/yacc) or LL(k) parsers (top-down parsers, e. g., ANTLR). LALR(k) usually is more flexible in what concerns the grammar of the language and the generated syntax tree. However, a recursive descent LL(k) parser can rely on semantic predicates for disambiguation. This allows the parser for example to use the knowledge that `respondsto` has been defined as a binary macro to process `x respondsto y` correctly. A LALR(k) parser would only be able to see three consecutive identifier tokens. Parsing speed was not an issue, as explained above.

5.1.3 Transformation and back end

For the transformation of the abstract syntax tree (AST) and the final output, the following alternatives were considered.

- Object-oriented AST transformation. The most obvious way to implement a compiler back end is to define a class hierarchy for the nodes of the AST and to implement tree transformations manually, possibly supported by a tree transformation framework (like the one included in ANTLR) or object-oriented techniques like the visitor pattern [GHJV94]. This solution is easy to understand, flexible and powerful. It supports the introduction of additional features like advanced error handling quite well. Various implementation languages and platforms are available.

- XSLT engine. XML documents represent data in a tree-like structure. XSLT [Cla99] provides a way to describe XML transformations in a declarative way. Therefore, it seems an alternative to perform AST transformations using XSLT. However, XSLT transformations reach there limits when implementing complex transformations like the translation of exception operators. Furthermore, XSLT files are originally intended for text data and therefore rather difficult to read and understand when applied to syntax trees.

- Haskell AST transformation. Haskell is a mature functional programming language that provides user-defined data structures and pattern matching. This makes it well suited for the rapid development of simple and complex tree transformation code. When a Haskell interpreter is used instead of a precompiled program, macro expansion can be handled by defining the macros as user-defined functions, thus eliminating the need for an external preprocessor. Advanced error handling proves to be a bit difficult in Haskell.

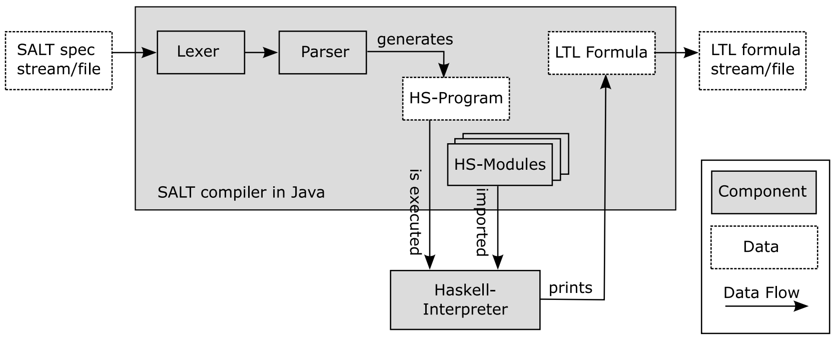


Figure 5.1: SALT compiler architecture

5.2 Compiler architecture

The architecture of the SALT compiler can be seen in figure 5.1. The specification is read by a lexer/parser combination. No external preprocessor is used. Both lexer and parser are Java classes generated using ANTLR [PQ95] and therefore belong to the group of recursive descent LL(k) parsers. The parser does not build a syntax tree but directly generates a Haskell program that contains the specification as well as the function calls necessary for its translation. The actual translation functions are imported from other Haskell modules. User-defined macros are translated into function definitions. The compiler then calls a Haskell interpreter to run the program. This is completely transparent to the user. When the program is run, it translates the specification and prints the resulting LTL formula, which is read by the compiler's main program and written to the chosen output channel.

This architecture allowed for a rapid and flexible development of the SALT compiler. If an industrial-quality compiler should be necessary in the future for a stable SALT language specification, the use of Haskell would probably be abandoned in favour of a completely object-oriented architecture, leading to an easier to install compiler and the possibility of better error handling.

5.3 Compilation phases

Compilation consists of the following phases [1]:

5.3.1 SALT specification parsing

The input to the compiler is the SALT specification as written by the user, which is read by a lexer/parser combination. The parser generates a Haskell program that contains the specification as well as the function calls necessary for its translation. Each assertion from the original specification is represented by one expression in the Haskell program. The main function of the program calls the

[1]Compilation phases might not be completely separated during an actual execution due to lazy evaluation.

translation functions for each of those expressions. The compiler then invokes a Haskell interpreter to run the program.

5.3.2 Macro expansion

Macro definitions have been translated by the parser into Haskell function definitions. Macro calls have been translated into function calls. During the execution of the Haskell program, all macros are therefore automatically expanded without the need of a macro preprocessor. Several high-level SALT operators (e. g., **`occurring`** and the iteration operators) are represented internally like macros. Their function definitions reside in one of the imported modules, and all uses of these operators are automatically replaced by expressions made of simpler SALT operators. This allows to keep the set of core SALT operators small.

5.3.3 Construction of the core SALT AST

For each core SALT element, there is a corresponding data constructor that allows the construction of an abstract syntax tree (AST) node. For example, the data constructor `SALT.Until` expects one parameter specifying whether the node represents a future or past operator, and two parameters representing the two child expressions. After the first step of the execution of the Haskell program, one AST has been built for each assertion.

5.3.4 Translation into SALT--

The compiler then traverses the SALT ASTs in left-right-top-down direction and translates each node into corresponding SALT-- AST nodes. SALT-- contains all LTL as well as the acc, rej and stop operators. acc and rej correspond to the SALT exception operators. stop is introduced during the translation of **`upto`** and **`between`**. Pattern matching on the SALT ASTs is required for the translation of most operators. For example, the translation of the regular expression repetition operator `*` depends on the sequence operator that forms its parent node. The result of this translation step are ASTs built from SALT-- nodes.

5.3.5 Translation into LTL

Eliminating the non-LTL operators from the ASTs requires weaving the end conditions into a whole sub-tree. The compiler traverses the ASTs in bottom-up direction. When it reaches a non-LTL operator, it calls a weaving function that traverses the sub-tree and inserts the end condition in all appropriate places. This translation step results in ASTs built from LTL nodes only.

5.3.6 LTL output

Finally, the compiler traverses the LTL ASTs and prints the formula in the desired output syntax [2]. This might require the replacement of several nodes that represent an operator that does not exist in the output syntax. For example, no W operator exists in SMV syntax. It is therefore replaced by an equivalent

[2]For instance, boolean or is written as `||` when using the SPIN model checker and as `|` when using SMV.

expression, which is chosen according to the size relation of its two child expressions. The use of illegal operators, like past operators in SPIN syntax, is detected here.

5.4 Optimisation

The use of optimised, context-dependent translation patterns as well as a final optimisation step performing local changes help reducing the size of the generated formulae.

For example, most verification tools do not support the W operator directly. Therefore, W has to be expressed using other operators in the Ltl output step. The Salt compiler choses between the two equivalent expressions

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

depending on whether φ or ψ is more complex. In most cases, the first expression, which is less intuitive for humans, yields better technical results.

An important optimisation that is performed on the final Ltl formula replaces $\Box(\varphi \text{ W } \psi)$ by $\Box(\varphi \vee \psi)$.

5.5 Error handling

The Salt compiler aims at delivering all error messages, regardless of their nature, to the user in a uniform way.

Errors during the lexing and parsing process are forwarded directly to the compiler main program. The parser also performs a number of simple semantic checks, like relating macro uses to macro definitions.

During the generation of the Haskell program, a source info (SI) node constructor call is added for each element of the future AST. The SI node contains the line and column number of the corresponding source code element and allows the following translation steps to output error messages containing a source code reference.

Errors that occur during the Haskell part of the translation result in the generation of error nodes. Each sub-expression that can not be properly translated is replaced by such a node containing the related source code position and the error message. During traversal of the ASTs, error nodes are copied to the new AST. The final Ltl output function searches the tree for error nodes. If it finds any, it prints the corresponding error messages (after removing duplicates) instead of the formula output. This procedure is necessary because Haskell as a functional programming language does not allow side effects, i. e., forbids storing error messages in a separate place during translation.

5.6 Compilation process example

This section illustrates the compilation of an example specification through all compilation phases. SI nodes (used for detailed error messages) have been left out for readability reasons. ASTs are displayed in a way similar to how Haskell would print them using the `show` function.

1. Salt specification:[3]

```
define respondsto(x, y) := y -> (eventually x)
assert (nextn[2] (/a;!a/ respondsto b)) upto incl
   weak c
```

2. Haskell program:

```
module Main where
import SALT
...
respondsto x y = (SALT.Impl y ((SALT.Eventually
   Future x)))
...
_assertion_1 = (SALT.UpTo Future ((SALTMacros.nextn
   Future (SALT.Exactly 2) ((Main.respondsto
   (SALT.Sequence Future (SALT.Ident "a") (SALT.Not
   (SALT.Ident "a"))) (SALT.Ident "b")))))
   (SALTMacros.inclusive (SALTMacros.weak
   (SALT.Ident "c"))))
...
main = Main.process_assertion (_assertion_1) >>
   Prelude.putStr ""
...
process_assertion a = LTL2SMV.printLTL
   Common.WithoutTimed Common.WithPast
   Common.WithNext (OptimizeLTL.optimizeLTL
   (RLTL2LTL.convertRLTL2LTL
   (SALT2RLTL.convertSALT2RLTL a)))
```

3. Salt AST:

```
UpTo Future (Next Future (Next Future (Impl (Ident
   "b") (Eventually Future (Sequence Future (Ident
   "a") (Not (Ident "a"))))))) (Inclusive (Weak
   (Ident "c")))
```

4. Salt-- AST:

```
StopOnIncl Future (Next Future (Next Future (Impl
   (Ident "b") (Eventually Future (And (Ident "a")
   (Next Future (Not (Ident "a")))))))) (Ident "c")
```

5. Ltl AST:

```
And (Next Future (And (Next Future (Impl (Ident "b")
   (Until Future (Not (Ident "c")) (And (Ident "a")
   (And (Next Future (Not (Ident "a"))) (Not (Ident
   "c"))))))) (Not (Ident "c")))) (Not (Ident "c"))
```

[3]The aim of this specification is to comprise many different operators, not to specify a meaningful requirement.

6. LTL specification in output syntax (SMV):

```
LTLSPEC (X ((X (b -> (!c U (a & ((X !a) & !c))))) &
    !c)) & !c
```

5.7 Compiler features

Platforms. The standalone SALT compiler runs on both Windows and Linux platforms. It can be downloaded together with a manual from `http://salt.in.tum.de`. A web interface has been created for users who want to try SALT without installing a Haskell interpreter.

Parameterisable propositional layer. The SALT language allows arbitrary quoted text as atomic propositions. In order to make this feature more reliable and allow for instance only valid Java expressions, custom proposition parsers can be provided to the compiler via a plugin mechanism.

Flexible output syntax. The SALT compiler can generate LTL formulae for both SMV and SPIN model checkers (with a slightly different syntax). Custom output formats can be implemented using a plugin-mechanism.

For timed specifications, the compiler can produce either pure TLTL with $\rhd_{\sim c}$ and $\lhd_{\sim c}$ as the only timed operators, or extended TLTL with the additional timed operators $\mathrm{U}_{\sim c}$, $\mathrm{W}_{\sim c}$, $\Box_{\sim c}$ and $\Diamond_{\sim c}$ as well as the corresponding past operators. Extended TLTL is much easier to read than pure TLTL and has similarities to MTL.

Operator restrictions. The compiler can be set to forbid the use of past operators, if the user wants pure future formulae. It can also be set to forbid the use of next operators (or operators that are translated using next operators), which leads to stutter-invariant formulae.

Chapter 6

Experimental results

This chapter presents and discusses some experimental results on the correctness and efficiency of compiler-generated LTL formulae compared to handwritten ones, and on the kind of blowup in the size of the LTL formula that results from a linear growth of a SALT specification.

6.1 Compiler correctness testing

Automated tests are an important tool in a software development process. A test framework was set up in order to run automated tests on the SALT compiler.

6.1.1 Methods

Each test consists of a SALT specification that is translated into an LTL formula φ_1 by the compiler. The equivalence of this formula to a handwritten LTL formula φ_2 is then tested with a model checker, in this case NuSMV [CCGR99]. This is done by generating a SMV [McM] specification of a system that contains all boolean variables that appear in the formula and assigns arbitrary values to the variables in each step. If the formula

$$\varphi_1 \leftrightarrow \varphi_2$$

can be proven to be true for this system, the formulae are equivalent. If the formula

$$\Box(\varphi_1 \leftrightarrow \varphi_2)$$

can be proven to be true, the formulae are congruent, i. e., they are equivalent in each state (see [Pnu77]).

For some tests, it is useful to define invariants in SALT, like

```
assert always (occurring[>=3] a ) <-> !(occurring[<3] a)
```

In these cases, the φ_2 to be met is simply $\top$.

The whole testing procedure was implemented as a JUnit [BG00] test case class.

6.1.2 Specification Patterns tests

Dwyer et. al. define a system of specification patterns [DAC99]. The patterns consist of *requirements* that can be expressed under different *scopes*. For each combination, an LTL formula is provided.

In order to gain test cases for the compiler, the requirements were written as SALT expressions and inserted into five different SALT specifications that represent the different scopes. Each combination was tested against the corresponding LTL formula from the pattern catalogue using the procedure described above.

Requirement	SALT **expression**
Absence	**`never`** `p`
Existence	**`eventually`** `p`
Universality	**`always`** `p`
Precedence	`!p` **`until weak`** `s`
Response	**`always`** `(p -> (`**`eventually`** `s))`
Bounded existence	**`occurring`** `[<=2] p`
2-1 response chain	**`always`** `(/s;*;t/ -> /s;*;t:*:p/)`
2-1 precedence chain	`!p` **`until weak`** `/s&!p;!p*;t/`

1-2 constr. resp. chain A	**`always`** `(p -> (`**`eventually`**
	`(`**`req`** `/s;*;t/` **`upto excl weak`** `z)))`

1-2 constr. resp. chain B	**`always`** `(p -> (`**`eventually`**
	`(/s;*;t/` **`upto incl weak`** `z)))`

Each requirement was inserted into each of the five scopes:

Scope	SALT **expression**
Globally	**assert** (φ)
Before `r`	**assert** (φ)**upto excl opt** `r`
After `q`	**assert** (φ)**from incl opt** `q`
Between `q` and `r`	**assert always** ((φ)**between** **incl opt** `q`, **excl opt** `r`)
After `q` until `r`	**assert always** ((φ)**between** **incl opt** `q`, **excl weak** `r`)

When running these tests for the first time, several of the test cases failed. All failures that were not results of bugs in the compiler or misinterpretations of the requirements could be explained by mistakes or inconsistencies in the LTL formulae from the pattern catalogue:

- The After-Precedence pattern contained an error.
- The Between-Existence and After-Until-Existence patterns showed a behaviour that was inconsistent with other pattern formulae in the case of an immediately occurring end condition `r`.
- The Constrained Response Chain pattern had errors (missing checks for end condition `r`) in the scopes Before, Between, After-Until as well as an error in After scope (probably a typo—missing $\Diamond$). Furthermore, the formulae provided did not check for absence of the constraint `z` in the last

state of the pattern. This resulted in writing two different SALT expressions, one (version A) reflecting the behaviour of the pattern formulae and the other (version B) reflecting the probably intended behaviour.

- The Response Chain pattern in the scopes Before, Between and After-Until contained errors (missing checks for end condition `r`).

After correction of these problems in the LTL formulae, all tests were successful and served as regression tests during the further development of the SALT compiler. The fact that mistakes in the pattern catalogue could be found is an indication that SALT indeed does raise the level of abstraction and is hence less error-prone than LTL.

6.1.3 Other tests

Other test cases include example specifications found in [DAC99]. For these examples, a SALT specification was written following the informal description of the requirement and compared to the LTL formula provided in the example. Several of the examples were erroneous or inaccurate.

Finally, a number of test cases was handwritten in order to cover as much compiler code as possible. These test cases include invariants like

```
assert always (occurring[>=3] a) <-> !(occurring[<3] a)
```

as well as equivalence tests like the specification

```
assert a until weak b
```

which is tested to be equal to

$$(a \text{ U } b) \vee (\Box a).$$

6.2 Compiler efficiency

In many cases, the use of a high-level language that is translated into a lower-level language by a compiler yields bigger, less efficient results than a manual translation. This drawback is however compensated by a gain in development, debugging and maintenance time. Most modern compilers perform optimisations on the generated code that improve its quality significantly. Heavily optimising compilers can even produce better code than the average programmer.

As the time required for model checking can be exponential with respect to the size of the formula to be checked, efficiency was an import issue for the development of the SALT compiler. One might suspect that generated formulae are bigger and less efficient to check than handwritten ones. We will however see in this section that this is not the case.

6.2.1 Methods

In order to quantify the efficiency of the SALT compiler, existing LTL formulae were compared to the formulae generated by the SALT compiler from a corresponding SALT specification. This was done for two data sets: the specification

Specification Patterns (N=50)	**BA [Fri]**	**BA [Odd]**	**U**	**X**	**Bool.**
Increase	2	6	2	0	26
No change	24	28	35	46	17
Decrease	24	16	13	4	7

Example Specifications (N=26)	**BA [Fri]**	**BA [Odd]**	**U**	**X**	**Bool.**
Increase	1	0	0	0	13
No change	15	12	18	26	11
Decrease	10	14	8	0	2

Figure 6.1: Comparison of generated to handwritten formulae: in how many cases does a parameter increase or decrease?

pattern system [DAC99] (50 specifications) and a collection of real-world example specifications, mostly from the survey data of [DAC99] (26 specifications). The increase or decrease of the generated formula compared to the handwritten one 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 $\circ$ in the formula.
- Boolean: Number of boolean leafs, i. e., variable references and constants. This is a good parameter for estimating the length (not necessarily the complexity) of the formula.

6.2.2 Results

Figure 6.1 shows for how many of the specifications of the two data sets a parameter increases, decreases or remains the same when the specification is translated with the SALT compiler (instead of a manual translation into LTL). Figure 6.2 quantifies the average increase or decrease of each parameter. We can see that the formulae generated by the SALT compiler are in many cases bigger in size (number of boolean leafs), but use less temporal operators and also yield a smaller BA. Often there is not much difference between generated and handwritten formulae. Only few specifications yield smaller automata for the handwritten formulae.

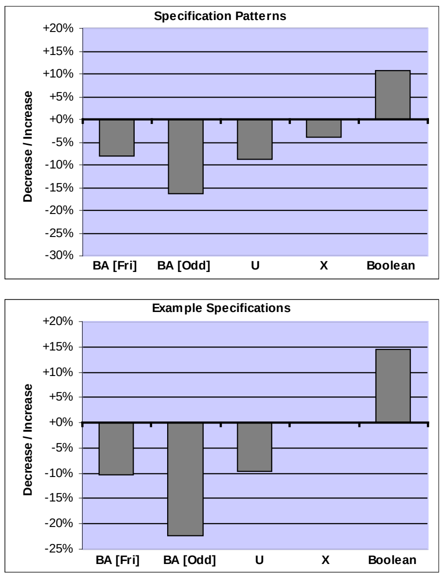


Figure 6.2: Average size increase/decrease of generated formulae

6.2.3 Interpretation

Using Salt to write specifications does not increase model checking cost. On the contrary, it often even leads to more efficient formulae.

The author of this text sees two reasons for these surprising results. The Salt compiler performs a number of optimisations, as explained in 5.4. For example, when translating a W it 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. When φ and ψ have about the same complexity, the first expression normally leads to smaller automata. However, humans tend to always use the second expression (or a similar one), because it is more intuitive to understand ("φ is true until ψ becomes true, or φ is true all the time.").

The second reason for the gap between generated and handwritten formulae is that—apparently—the generation of BA is still not as good as it could be. As the handwritten formula and the generated one are equivalent, a more advanced generator might be able to produce the same automaton, or at least a similar one, for both formulae.

6.3 Blowup introduced by Salt operators

Providing powerful operators, like **`occurring`** or **`upto`**, can be problematic if users are not aware of the increase in complexity that the use of those operators may entail. They might be surprised when changing a single integer number in the specification causes their model checking tool to reach its limits. Also, they might accidentally use a powerful operator to express a simple property, and thereby cause a huge overhead in complexity.

In the following, we will measure the growth that a linear change in a SALT specification causes in the translated LTL formula for various examples, and see that most SALT operators are relatively good-natured.

6.3.1 Methods

The following specifications were used for the test. Each of them contains a complexity parameter *i*, either directly as argument to a counting quantifier operator or through repeating a sub-expression *i* times.

- **`occurring`** operator example:

```
assert occurring[i] (p implies eventually q)
```

- **`holding`** operator example:

```
assert holding[i] (p implies eventually q)
```

- Nested **`upto`** operator example:

```
assert (weak ... (weak (p implies eventually q) before
    excl weak b1) ... before excl weak bi)
```

- Nested **`rejecton`** and **`accepton`** operators example:

```
assert ((... (( (p implies eventually q) rejecton r1)
   accepton a1) ... rejecton ri) accepton ai)
```

- Regular expression bounded repetition operator example:

```
assert /a*[<=i];b/
```

- Regular expression sequence operator example:

```
assert /true; a1*; b1; ... ai*; bi/
```

- **`until`** nested in the left argument example:

```
assert (... ((p implies eventually q) until u1) ...
   until ui)
```

For the test, *i* was set to values between 0 and 5. Each of the resulting specifications was then translated into an LTL formula and analysed using the same criteria as in 6.2.

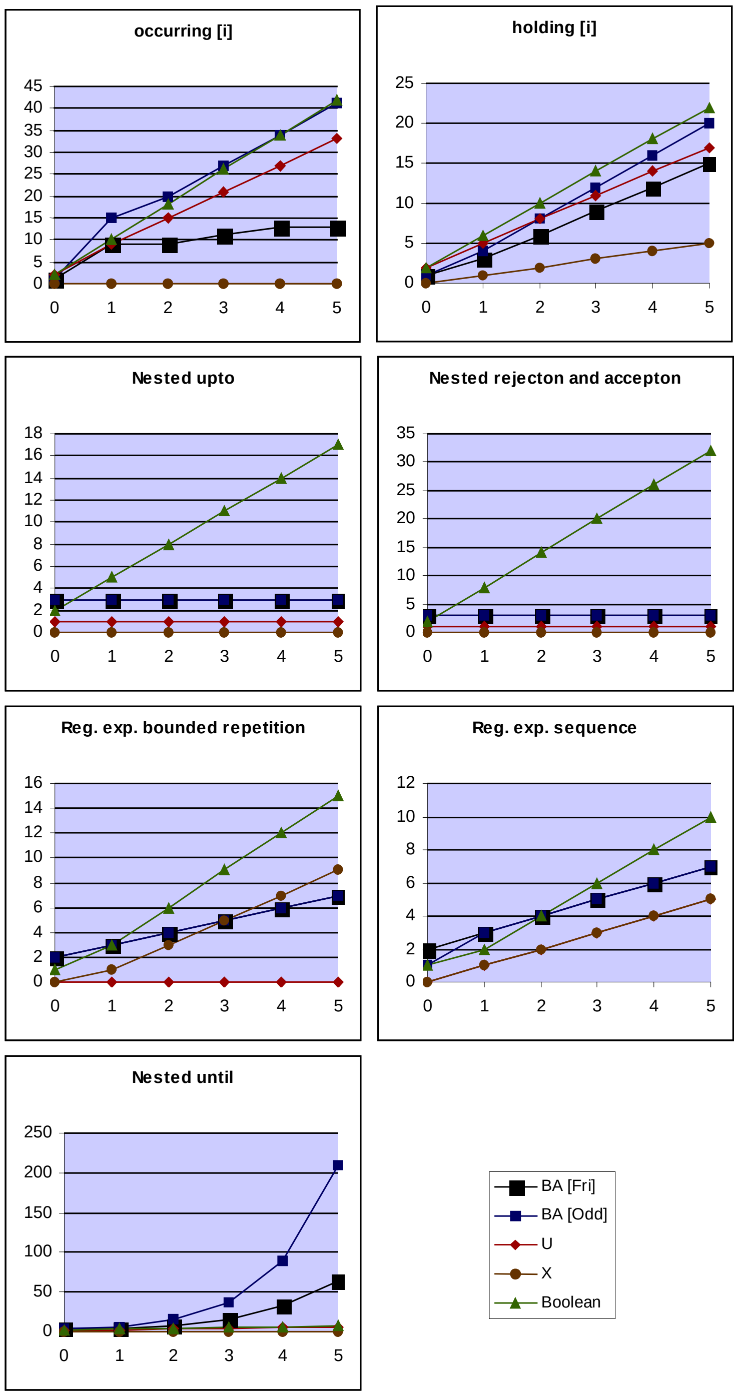


Figure 6.3: Blowup introduced by SALT operators

6.3.2 Results

The results can be seen in figure 6.3, with the complexity parameter i on the x-axis.

The **occurring** and **holding** operators cause a linear growth of all parameters in relation to the repetition counter i. For the **occurring** operator, this growth is even somewhat compensated by the BA generation.

Nested **upto** and **rejecton**/**accepton** operators cause a linear growth of the length of the formula, while the number of temporal operators and the size of the corresponding BA remain constant.

Regular expression sequence and repetition operators cause a more or less linear growth of all parameters in relation to the length of the sequence respectively the repetition counter i.

The nested **until** operator example yields a linear increase of the number of temporal operators but an exponential blowup of the size of the BA.

6.3.3 Interpretation

The Salt operators studied in the example all turn out to be relatively good-natured in what concerns the growth of the most important parameter, the size of the BA. Some operators only increase the size of the boolean expressions (which looks complicated but does not affect the BA generation). Others do add temporal operators, but create mainly a chain of operators where nested operators appear in the right argument of an U. This kind of formula apparently yields approximately linear growth of the BA, a rather harmless effect when compared to the worst-case behaviour that the nested **until** example (nested in the left argument) shows. The author has the impression that examples that—like the last one—generate huge formulae from small specifications often express requirements that are inherently complex to represent in Ltl.

Chapter 7

Conclusion

This thesis presented SALT, a high-level temporal specification language. SALT has been designed to be easily understandable and usable for both software engineers and verification experts. It provides a higher level of abstraction than LTL and permits its users to define their own operators via a macro mechanism, thus extending the language and raising the level of abstraction further.

Among the most advanced features of the SALT language are the scope operators. The author does not know of another specification language that provides such operators with similar flexibility, although their importance has been recognised. Other features, like regular expressions, exception operators or counting quantifiers are known from specification languages like Sugar/PSL. The contribution that the SALT language provides here is the complete translatability of these features into LTL and a compiler that actually performs this task. This allows the users to choose from a broad set of verification tools.

Experimental results show that using SALT instead of writing LTL specifications by hand does not deprave the efficiency of the subsequent verification tools. On the contrary, the SALT compiler often produces more efficient formulae than the average engineer.

Although the development of SALT was guided by the analysis of real-world example specifications and the features of other specification languages, the language surely contains flaws and misses possible features. Therefore, user feedback is now absolutely required and very much appreciated. Many of the LTL example specifications examined during the development of the SALT language were rather trivial formulae; only some of them specified complex requirements. The author hopes that the SALT language will encourage more people to use formal methods and to test more complicated requirements. This will allow the language to mature and grow together with its more and more complex applications.

The SALT compiler can be downloaded together with a manual from the website `http://salt.in.tum.de`. An interactive web interface that permits to translate a SALT specification without having to install the compiler can be found on the same site. Furthermore, the SALT compiler has been prototypically integrated into AUTOFOCUS [HSSS96], a modelling and verification tool.

An interesting issue that lies beyond the scope of this thesis but could be explored in future research is the integration of formal methods into a development process. Which parts of a system should be checked using formal methods? Which other techniques can be employed? Should temporal specifications be written as part of a requirements document, or should they be created from a textual representation of the requirements? When is the best time to perform the transition from an informal representation to a formal specification? Finding an answer to these questions involves weighing benefits against drawbacks, and is therefore hard to do on a purely theoretical level. A thorough empirical analysis using a real-world project could provide evidence on how software engineering can profit best from the application of formal methods.

Bibliography

[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.

[ABLS05] Oliver Arafat, Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification revisited. Technical Report TUM-I0518, Institut für Informatik, Technische Universität München, October 2005.

[Acc04] Accellera. *Property Specification Language Reference Manual Version 1.1*, 2004.

[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.

[AFH94] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. A determinizable class of timed automata. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV)*, pages 1–13, London, UK, 1994. Springer-Verlag.

[AFH96] Rajeev Alur, Tomas Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.

[AH92] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 74–106, London, UK, 1992. Springer.

[AH94] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *Journal of the ACM*, 41:181–204, 1994.

[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.

[BCM05] Patricia Bouyer, Fabrice Chevalier, and Nicolas Markey. On the expressiveness of TPTL and MTL. In R. Ramanujam and

Sandeep Sen, editors, *Proceedings of the 25th Conference on Fundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Lecture Notes in Computer Science. Springer, December 2005.

[BDG$^+$04] Guillaume Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Arnaud Venet, Willem Visser, and Rich Washington. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in Systems Design*, 25(2-3):167–198, 2004.

[BG00] Kent Beck and Erich Gamma. *Test-infected: programmers love writing tests*, pages 357–376. Cambridge University Press, New York, NY, USA, 2000.

[BGHS04a] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with LTL in EAGLE. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004.

[BGHS04b] 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.

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

[BR05] Therese Berg and Harald Raffelt. Model checking. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[CCGR99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)*, pages 495–499, London, 1999. Springer.

[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.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[Cla99] James Clark. XSL transformations (XSLT) version 1.0. Technical report, W3C, 1999.

[CM05] S. Colin and L. Mariani. Run-time verification. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, chapter 18. Springer, 2005.

[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.

[Dru00] D. Drusinsky. The Temporal Rover and the ATG Rover. *Lecture Notes in Computer Science*, 1885:323–329, 2000.

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

[EFH$^+$03] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 27–39, July 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.

[Hol90] G. J. Holzmann. *Design and validation of computer protocols*. Prentice Hall, 1990.

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

[HR04] Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology*, 6:158–173, August 2004.

[HSSS96] Franz Huber, Bernhard Schätz, 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.

[KC05] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *Proceedings of the 27th international conference on Software Engineering (ICSE)*, pages 372–381, New York, NY, USA, 2005. ACM Press.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2:255–299, 1990.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

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

[McM] Ken McMillan. *SMV Language Reference*. Cadence Berkeley Labs, Berkeley, CA, USA.

[MP71] Robert McNaughton and Seymour A. Papert. *Counter-Free Automata*. Number 65 in MIT research monographs. The MIT Press, 1971.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.

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

[PQ95] T. J. Parr and R. W. Quong. ANTLR: a predicated-LL(k) parser generator. *Software – Practice and Experience*, 25(7):789–810, 1995.

[RS97] Jean-François Raskin and Pierre-Yves Schobbens. State clock logic: A decidable real-time logic. In *Proceedings of the International Workshop on Hybrid and Real-Time Systems (HART)*, pages 33–47, London, UK, 1997. Springer.

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

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

[Syn02] Synopsys, Inc. Synopsys releases OpenVera 2.0 language with new assertions. Press release, 2002.

[Syn03] Synopsys, Inc. *OpenVera^{TM} Language Reference Manual: Assertions Version 2.3*, 2003.

[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.

[Var01] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 1–22, London, UK, 2001. Springer.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and Seung Joon Park. Model checking programs. In *Proceedings of the 15th IEEE international conference on Automated software engineering (ASE)*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, 1986.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

[Wol02] Pierre Wolper. Constructing automata from temporal logic formulas: a tutorial. *Lectures on formal methods and performance analysis: first EEF/Euro summer school on trends in computer science*, pages 261–277, 2002.

[Zuc86] Lenore D. Zuck. *Past Temporal Logic.* PhD thesis, Weizmann Institute, 1986.

List of Figures

List of acronyms

AST	Abstract Syntax Tree. See section 5.3.
BA	Büchi Automaton. See [BR05].
CTL	Computational Tree Logic. See section 2.3 and [CE82].
DFA	Deterministic Finite Automaton.
LTL	Linear Temporal Logic. See section 2.3 and [Pnu77, MP92].
MITL	Metric Interval Temporal Logic. See section 2.5 and [AFH96].
MTL	Metric Temporal Logic. See section 2.5 and [Koy90].
SALT	Structured Assertion Language for Temporal Logic. See chapter 4.
SMV	Symbolic Model Verifier. See section 2.1 and [McM].
SPIN	Simple Promela INterpreter. See section 2.1 and [Hol97].
TLTL	Timed Linear Temporal Logic. See section 2.5 and [D'S03].
TPTL	Timed Propositional Temporal Logic. See section 2.5 and [AH94].

Appendix

SALT

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

Language Reference and Compiler Manual

Version 1.0 — April 2006

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

Appendix A

General Information

A.1 The SALT language

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

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

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

A.2 Licensing and Contact

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

Developers: Jonathan Streit

Contributions from: Andreas Bauer, Martin Leucker

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

Third party software used: ANTLR parser generator by Terence Parr

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

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

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

A.3 Typographical conventions

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

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

A.4 SALT version history

0.7 Preliminary version

1.0 Revised manual, AUTOFOCUS support

Appendix B

Installation

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

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

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

Installation procedure

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

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

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

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

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

Appendix C

Usage

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

C.1 Command line parameters

The following parameters can be provided to the compiler:

- *file*
  Sets the input file to be read. The SALT specification will be read from the given file. If no input file is provided, the specification is read from standard in (normally the console). This allows the usage of Unix pipes.
- `-f "`*spec*`"`
  Processes a specification from the command line. This option allows to translate small specifications provided on the command line.
- `-o` *file*
  Sets the output file to be used. The result is written to the given file. If this option is not present, the result will be written to standard out (normally the console). This allows the usage of Unix pipes.
- `-e`
  Switches to embedded SALT mode. This is useful when a SALT specification forms a part of another file, for example an SMV model. The compiler searches the input for SALT specifications delimited by `BEGINSALT` and `ENDSALT`. The rest of the file is copied to the output, with the SALT specifications replaced by the resulting LTL formulae. See chapter D for an example.
- `-parser` *module*
  Enables a custom proposition parser plugin. The proposition parser is called to check and/or transform atomic propositions. For information on how to provide a custom implementation see section C.2.

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

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

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

- `-printer` *module*
  Enables a custom printing function plugin. The printing function is used to print the final LTL formula in the desired output syntax. For information on how to provide a custom implementation see section C.2.

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

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

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

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

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

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

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

- `-nonext`
  Don't allow next operators. The use of the **`next`** operator as well as of other SALT operators that are translated using the **`next`** operator (e. g., regular expressions) produces an error message. This ensures that the formula is stutter-invariant.
- `-v`
  Choose verbose mode. The compiler will output some more status messages.
- `-h` or `-?`
  Show help screen.

C.2 Extending the SALT compiler

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

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

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

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

Appendix D

Getting started with Salt

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

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

D.1 First steps

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

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

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

When we run the compiler on the specification, we obtain

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

which corresponds to the Ltl formula

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

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

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

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

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

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

BEGINSALT
  assert never erroroccured
ENDSALT
```

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

D.2 SALT for LTL users

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

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

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

The following three expressions all have the same meaning:

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

D.3 Scope operators

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

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

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

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

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

The correct specification looks like this:

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

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

D.4 Regular expressions

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

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

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

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

D.5 Exception operators

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

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

D.6 Macros

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

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

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

D.7 Iteration operators

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

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

D.8 Further steps

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

Appendix E

Salt Language Reference

This chapter describes the Salt language in detail. Formal semantics are defined in chapter F via a translation schema. See chapter D for a tutorial if you want to learn Salt and chapter G for some example specifications.

E.1 General structure

Assertions

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

Syntax

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

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

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

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

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

Figure E.1: Salt syntax: general structure

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

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

Layers

The SALT language consists of three layers:

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

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

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

E.2 Propositional layer

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

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

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

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

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

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

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

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

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

Figure E.2: SALT syntax: propositional layer

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

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

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

Quoted propositions make the SALT language independent from the domain it is used in and the models and tools that influence the rest of the verification process. In order to make the use of complex propositions more reliable, custom proposition parser plugins can be provided to check and/or transform atomic propositions that appear in a SALT specification (see section C.2).

Example:

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

yields the output

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

Parameterised propositions. Inside a boolean proposition, function parameters or iteration variables may appear between `$$`. During translation they are replaced by the value of the parameter. See section E.5 for details on parameterised expressions.

Example:

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

yields the output

```
LTLSPEC main_started & !main_error
```

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

Custom proposition parser plugins can define their own behaviour for proposition checking (see section C.2).

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

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

E.3 Temporal layer

The temporal layer is the heart of the SALT language. It allows to express temporal properties by combining propositional expressions with temporal operators. The temporal layer consists of a future fragment and a completely symmetrical past fragment. For the sake of brevity, this section presents only future operators explicitly. The corresponding past operators are introduced in E.3.6.

E.3.1 Simple temporal operators

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

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

Extended `until`

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure E.3: SALT syntax: temporal layer

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

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

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

Extended **`next`**

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

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

E.3.2 Scope operators

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

The **upto** operator

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

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

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

The **from** operator

φ **from exclusive required** a
φ **from exclusive optional** a
φ **from inclusive required** a
φ **from inclusive optional** a

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

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

The **between** operator

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

Start and end conditions

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

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

Example:

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

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

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

Inclusive and exclusive semantics

`upto` can either be *exclusive* (the step when b occurs is not taken into account any more) or *inclusive* (the step when b occurs is the last step of the denoted interval). To specify which behaviour is meant, the condition b must be prefixed with the modifier keyword **`exclusive`** or **`inclusive`**. The same applies to the **`from`** operator: inclusive means that the step when a occurs is the step φ is evaluated. Exclusive means that φ is evaluated in the next step after a. The **`between`** operator requires its start and end condition to be prefixed by **`inclusive`** or **`exclusive`**. A **`between`** operator with an exclusive start condition looks for occurrences of the end condition only from the next step on, i. e., it ignores b if it occurs together with the start condition a. The abbreviations **`incl`** and **`excl`** may be used instead of the long keywords. Figure E.4 provides a visualisation of inclusive/exclusive semantics.

Behaviour for empty time intervals

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

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

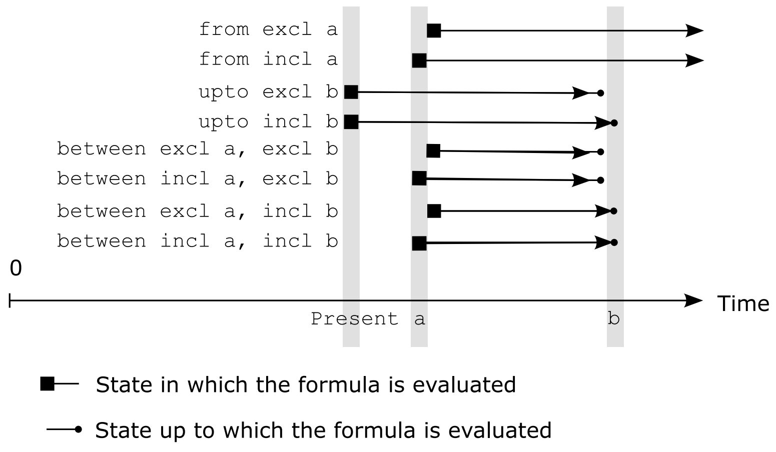


Figure E.4: Inclusive and exclusive semantics of scope operators

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

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

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

Details and special cases

All three operators may be freely nested.

Note that although **upto** and **from** seem to be very similar, there is a crucial difference: The **from** operator evaluates its argument from a certain step (where the condition a holds for the first time) on toward the future. The **upto** operator evaluates its argument at the current step, but on a future limited by the occurrence of the end condition b. See figure E.4 for a visualisation.

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

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

Examples

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

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

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

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

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

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

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

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

E.3.3 Exception operators

```
φ rejecton b
φ accepton b
```

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

Examples:

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

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

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

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

The semantics of **rejecton** and **accepton** have similarities to those of the **upto** operator (see E.3.2). However, **rejecton** and **accepton** evaluate any pending formula to false resp. true on occurrence of the abort condition, while the result of an **upto** depends on the nature of φ.

Example:

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

rejecton and **accepton** do not have corresponding past operators (see E.3.6). They influence both future and past operators in φ.

E.3.4 Regular expressions

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

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

Two sequence operators are available:

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

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

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

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

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

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

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

Examples of allowed sequences:

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

Examples of forbidden sequences:

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

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

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

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

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

E.3.5 Counting quantifiers

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

`occurring` operator

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

`holding` operator

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

E.3.6 Past operators

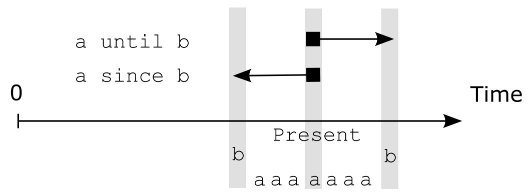
SALT supports past operators, that reason about past states instead of future ones. For every future operator there is a corresponding past operator, as shown in figure E.5. Future operators are translated using only LTL future operators, and past operators are translated using only LTL past operators. This leaves the users the choice whether they do or do not want to use past operators.

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

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

Figure E.5: Overview of the future operators and their corresponding past operators

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



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

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

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

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

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

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

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

E.4 Timed layer

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

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

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

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

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

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

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

Figure E.6: SALT syntax: timed layer

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

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

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

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

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

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

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

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

Examples:

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

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

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

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

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

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

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

For example,

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

could have three different meanings:

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

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

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

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

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

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

E.5 Macros and parameterised expressions

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

E.5.1 Parameterised expressions

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

Parameters can be used in two ways within an expression: First, they can be employed directly as part of an expression. Secondly, they can be used *within* an atomic proposition, e. g., as part of a variable name. This is expressed by referencing the parameter between `$$` in the atomic proposition (see section E.2).

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

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

E.5.2 Macros

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

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

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

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

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

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

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

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

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

<formal_parameter> ::= <identifier>

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

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

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

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

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

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

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

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

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

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

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

Figure E.7: Salt syntax: macros and parameterised expressions

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

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

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

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

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

Example:

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

E.5.3 Iteration

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

The general syntax for an iteration expression is

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

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

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

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

- **`enumerate`** `[`n`..`m`]`
  creates a list containing the numbers between n and m. This list can be used to create for example a parameterised list of boolean variables with a common base name.
- *list* **`without`** φ
  removes the element φ (which must be an element of *list*) from the list.
- *list* **`with`** φ
  adds φ to the list.

Iteration operators.

- **`allof`**
  combines the instantiated expressions with a logical and, i. e., *all of* them have to be true in order to make the whole expression true.
- **`noneof`**
  combines the negated instantiated expressions with a logical and, i. e., *none of* them is allowed to be true in order to make the whole expression true.
- **`someof`**
  combines the instantiated expressions with a logical or, i. e., *some of* them have to be true in order to make the whole expression true.
- **`exactlyoneof`**
  requires *exactly one of* the instantiated expressions to be true and all the others to be false in order to make the whole expression true.

Examples:

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

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

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

Appendix F

Translation schema

This chapter describes how the SALT language is translated into LTL and TLTL and thereby defines the formal semantics of SALT.

The translation of past operators is left out for brevity, unless stated otherwise. It follows the same schema as the translation of the future operators. The translation of timed operators is described in section F.6. The other sections of this chapter refer to untimed SALT.

Translation is done in several steps:

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

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

The LTL operators used during translation are:

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

And for timed expressions additionally:

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

The SALT-- operators used are:

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

F.1 Replacement of non-core SALT operators

F.1.1 never

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

F.1.2 releases

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

F.1.3 nextn

$$\begin{array}{lll}
\mathrm{T}(\texttt{nextn[=}n\texttt{]}\varphi) & = & \\
\quad\quad \text{if } n = 0\text{:} & & \mathrm{T}(\varphi) \\
\quad\quad \text{else:} & & \circ\mathrm{T}(\texttt{nextn[=}n-1\texttt{]}\varphi) \\
\mathrm{T}(\texttt{nextn[}n\texttt{..}m\texttt{]}\varphi) & = & \mathrm{T}(\texttt{nextn[=}n\texttt{]}(\texttt{nextn[<=}m-n\texttt{]}\varphi)) \\
\mathrm{T}(\texttt{nextn[<=}n\texttt{]}\varphi) & = & \\
\quad\quad \text{if } n = 0\text{:} & & \mathrm{T}(\varphi) \\
\quad\quad \text{else:} & & \varphi \vee \circ\mathrm{T}(\texttt{nextn[<=}n-1\texttt{]}\varphi) \\
\mathrm{T}(\texttt{nextn[<}n\texttt{]}\varphi) & = & \mathrm{T}(\texttt{nextn[<=}n-1\texttt{]}\varphi) \\
\mathrm{T}(\texttt{nextn[>=}n\texttt{]}\varphi) & = & \mathrm{T}(\texttt{nextn[=}n\texttt{]}\Diamond\varphi) \\
\mathrm{T}(\texttt{nextn[>}n\texttt{]}\varphi) & = & \mathrm{T}(\texttt{nextn[>=}n+1\texttt{]}\varphi)
\end{array}$$

F.1.4 **`occurring`**

$\mathrm{T}(\texttt{\textbf{occurring}}\texttt{[=}n\texttt{]}\varphi) \quad =$
- if $n = 0$: $\neg\Diamond\mathrm{T}(\varphi)$
- if $n = 1$: $\neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge ((\mathrm{T}(\varphi)\ \mathrm{W}\ \neg\Diamond\mathrm{T}(\varphi)))$[1]
- else: $\neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge (\mathrm{T}(\varphi)\ \mathrm{U}\ (\neg\mathrm{T}(\varphi)\wedge \mathrm{T}(\texttt{\textbf{occurring}}\texttt{[=}n-1\texttt{]}\varphi))))$

$\mathrm{T}(\texttt{\textbf{occurring}}\texttt{[}n\texttt{..}m\texttt{]}\varphi) \quad =$
- if $n = 0$: $\mathrm{T}(\texttt{\textbf{occurring}}\texttt{[<=}m\texttt{]}\varphi)$
- if $n = 1$: $\neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge (\mathrm{T}(\varphi)\ \mathrm{W}\ (\neg\mathrm{T}(\varphi)\wedge \mathrm{T}(\texttt{\textbf{occurring}}\texttt{[<=}m-1\texttt{]}\varphi))))$[1]
- else: $\neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge (\mathrm{T}(\varphi)\ \mathrm{U}\ (\neg\mathrm{T}(\varphi)\wedge \mathrm{T}(\texttt{\textbf{occurring}}\texttt{[}n-1\texttt{..}m-1\texttt{]}\varphi))))$

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

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

$\mathrm{T}(\texttt{\textbf{occurring}}\texttt{[>=}n\texttt{]}\varphi) \quad =$
- if $n = 0$: $\top$
- if $n = 1$: $\Diamond\mathrm{T}(\varphi)$
- else: $\Diamond(\mathrm{T}(\varphi) \wedge (\Diamond(\neg\mathrm{T}(\varphi)\wedge \mathrm{T}(\texttt{\textbf{occurring}}\texttt{[>=}n-1\texttt{]}\varphi))))$

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

F.1.5 **`holding`**

$\mathrm{T}(\texttt{\textbf{holding}}\texttt{[=}n\texttt{]}\varphi) \quad =$
- if $n = 0$: $\neg\Diamond\mathrm{T}(\varphi)$
- if $n = 1$: $\neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge \circ_W \neg\Diamond\mathrm{T}(\varphi)))$[2]
- else: $\neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi) \wedge \circ\mathrm{T}(\texttt{\textbf{holding}}\texttt{[=}n-1\texttt{]}\varphi))$

$\mathrm{T}(\texttt{\textbf{holding}}\texttt{[}n\texttt{..}m\texttt{]}\varphi) \quad =$
- if $n = 0$: $\mathrm{T}(\texttt{\textbf{holding}}\texttt{[<=}m\texttt{]}\varphi)$
- if $n = 1$: $\neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi)\wedge \circ_W\mathrm{T}(\texttt{\textbf{holding}}\texttt{[<=}m-1\texttt{]}\varphi))$[2]
- else: $\neg\mathrm{T}(\varphi)\ \mathrm{U}\ (\mathrm{T}(\varphi)\wedge \circ\mathrm{T}(\texttt{\textbf{holding}}\texttt{[}n-1\texttt{..}m-1\texttt{]}\varphi))$

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

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

$\mathrm{T}(\texttt{\textbf{holding}}\texttt{[>=}n\texttt{]}\varphi) \quad =$
- if $n = 0$: $\top$
- if $n = 1$: $\Diamond\mathrm{T}(\varphi)$[2]
- else: $\Diamond(\mathrm{T}(\varphi) \wedge \circ\mathrm{T}(\texttt{\textbf{holding}}\texttt{[>=}n-1\texttt{]}\varphi))$

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

---

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

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

F.1.6 Regular expressions, part I

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

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

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

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

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

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

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

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

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

F.1.7 Iteration operators

The iteration operators are translated as follows:

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

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

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

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

---

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

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

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

F.2 Translation of core SALT into SALT--

F.2.1 `until`

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

F.2.2 upto

$\mathrm{T}(\varphi\ \texttt{\textbf{upto excl req}}\ b)$		$=$	
	if $\mathrm{T}(\varphi) = \Box\psi$:		$(\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b$
	if $\mathrm{T}(\varphi) = \neg\Diamond\psi$:		$(\neg\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b$
	else:		$(\Diamond b) \wedge (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b)$[5]
$\mathrm{T}(\varphi\ \texttt{\textbf{upto excl opt}}\ b)$		$=$	
	if $\mathrm{T}(\varphi) = \Diamond\psi$:		$\neg((\neg\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b)$
	else:		$(\Diamond b) \rightarrow (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b)$[5]
$\mathrm{T}(\varphi\ \texttt{\textbf{upto excl weak}}\ b)$		$=$	$(\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b)$
$\mathrm{T}(\texttt{\textbf{req}}\ \varphi\ \texttt{\textbf{upto excl req}}\ b)$		$=$	
	if $\mathrm{T}(\varphi) = \Box\psi$:		$\neg b \wedge ((\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b)$
	if $\mathrm{T}(\varphi) = \neg\Diamond\psi$:		$\neg b \wedge ((\neg\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b)$
	else:		$(\Diamond b) \wedge \neg b \wedge (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b)$[5]
$\mathrm{T}(\texttt{\textbf{req}}\ \varphi\ \texttt{\textbf{upto excl opt}}\ b)$		$=$	
	if $\mathrm{T}(\varphi) = \Diamond\psi$:		$\neg((\neg\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b)$
	else:		$(\Diamond b) \rightarrow (\neg b \wedge (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b))$[5]
$\mathrm{T}(\texttt{\textbf{req}}\ \varphi\ \texttt{\textbf{upto excl weak}}\ b)$		$=$	$\neg b \wedge (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b)$
$\mathrm{T}(\texttt{\textbf{weak}}\ \varphi\ \texttt{\textbf{upto excl req}}\ b)$		$=$	
	if $\mathrm{T}(\varphi) = \Box\psi$:		$(\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b$
	if $\mathrm{T}(\varphi) = \neg\Diamond\psi$:		$(\neg\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b$
	else:		$(\Diamond b) \wedge (b \vee (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b))$[5]
$\mathrm{T}(\texttt{\textbf{weak}}\ \varphi\ \texttt{\textbf{upto excl opt}}\ b)$		$=$	
	if $\mathrm{T}(\varphi) = \Diamond\psi$:		$b \vee \neg((\neg\psi\ \mathrm{stop}_{\mathrm{excl}}\ b)\ \mathrm{U}\ b)$
	else:		$(\Diamond b) \rightarrow (b \vee (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b))$[5]
$\mathrm{T}(\texttt{\textbf{weak}}\ \varphi\ \texttt{\textbf{upto excl weak}}\ b)$		$=$	$b \vee (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{excl}}\ b)$
$\mathrm{T}(\varphi\ \texttt{\textbf{upto incl req}}\ b)$		$=$	$(\Diamond b) \wedge (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{incl}}\ b)$
$\mathrm{T}(\varphi\ \texttt{\textbf{upto incl opt}}\ b)$		$=$	$(\Diamond b) \rightarrow (\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{incl}}\ b)$
$\mathrm{T}(\varphi\ \texttt{\textbf{upto incl weak}}\ b)$		$=$	
	if $\mathrm{T}(\varphi) = \Box\psi$:		$\neg(\neg b\ \mathrm{U}\ \neg(\psi\ \mathrm{stop}_{\mathrm{incl}}\ b))$
	if $\mathrm{T}(\varphi) = \neg\Diamond\psi$:		$\neg(\neg b\ \mathrm{U}\ (\psi\ \mathrm{stop}_{\mathrm{incl}}\ b))$
	else:		$(\mathrm{T}(\varphi)\ \mathrm{stop}_{\mathrm{incl}}\ b)$[5]

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

F.2.3 from

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

F.2.4 between

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

F.2.5 Exception operators

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

F.2.6 Regular expressions, part II

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

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

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

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

The sequence operators are then translated as follows:

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

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

F.3 Translation of SALT-- into LTL

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

F.3.1 acc

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

The translation of $\rightarrow$, $\leftrightarrow$, W and $\circ_W$ is done using the corresponding LTL equivalents in F.5.

F.3.2 rej

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

The translation of $\rightarrow$, $\leftrightarrow$, W and $\circ_W$ is done using the corresponding LTL equivalents in F.5.

F.3.3 stop$_{\text{incl}}$

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

The past stop operators are translated in a similar way as the future stop operators, but affecting only past operators. The translation of W, $\rightarrow$ and $\leftrightarrow$ is done using the corresponding LTL equivalents in F.5.

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

F.3.4 $\mathbf{stop_{excl}}$

$\mathrm{T}(b \text{ stop}_{\text{excl}}\ s)$	$=$	b
$\mathrm{T}((\neg\varphi) \text{ stop}_{\text{excl}}\ s)$	$=$	$\neg\mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s)$
$\mathrm{T}((\varphi \wedge \psi) \text{ stop}_{\text{excl}}\ s)$	$=$	$\mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s) \wedge \mathrm{T}(\psi \text{ stop}_{\text{excl}}\ s)$
$\mathrm{T}((\varphi \vee \psi) \text{ stop}_{\text{excl}}\ s)$	$=$	$\mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s) \vee \mathrm{T}(\psi \text{ stop}_{\text{excl}}\ s)$
$\mathrm{T}((\varphi \text{ U } \psi) \text{ stop}_{\text{excl}}\ s)$	$=$	$(\neg s \wedge \mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s)) \text{ U } (\neg s \wedge \mathrm{T}(\psi \text{ stop}_{\text{excl}}\ s))$
$\mathrm{T}((\varphi \text{ W } \psi) \text{ stop}_{\text{excl}}\ s)$	$=$	$\mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s) \text{ W } (s \vee \mathrm{T}(\psi \text{ stop}_{\text{excl}}\ s))$
$\mathrm{T}((\circ\varphi) \text{ stop}_{\text{excl}}\ s)$	$=$	$\circ(\neg s \wedge \mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s))$
$\mathrm{T}((\circ_W\varphi) \text{ stop}_{\text{excl}}\ s)$	$=$	$\circ(s \vee \mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s))$
$\mathrm{T}((\Box\varphi) \text{ stop}_{\text{excl}}\ s)$	$=$	$\mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s) \text{ W } s$
$\mathrm{T}((\Diamond\varphi) \text{ stop}_{\text{excl}}\ s)$	$=$	$(\neg s) \text{ U } (\neg s \wedge \mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s))$
$\mathrm{T}((\varphi \text{ S } \psi) \text{ stop}_{\text{excl}}\ s)$	$=$	$\mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s) \text{ S } \mathrm{T}(\psi \text{ stop}_{\text{excl}}\ s)$[7]
$\mathrm{T}((\bullet\varphi) \text{ stop}_{\text{excl}}\ s)$	$=$	$\bullet\mathrm{T}(\varphi \text{ stop}_{\text{excl}}\ s)$[7]

The past stop operators are translated in a similar way as the future stop operators, but affecting only past operators. The translation of $\rightarrow$ and $\leftrightarrow$ is done using the corresponding Ltl equivalents in F.5.

F.4 Optimisation

The following equivalences are used for optimisation:

$\top \text{ U } \varphi$	$\Longleftrightarrow$	$\Diamond\varphi$
$\neg\Diamond\neg\varphi$	$\Longleftrightarrow$	$\Box\varphi$
$\Box\Box\varphi$	$\Longleftrightarrow$	$\Box\varphi$
$\Diamond\Diamond\varphi$	$\Longleftrightarrow$	$\Diamond\varphi$
$\neg\varphi \text{ U } \varphi$	$\Longleftrightarrow$	$\Diamond\varphi$
$\Box(\varphi \text{ W } \psi)$	$\Longleftrightarrow$	$\Box(\varphi \vee \psi)$
$\varphi \text{ W } (\varphi \wedge \psi)$	$\Longleftrightarrow$	$\neg(\neg\psi \text{ U } \neg\varphi)$
$(\varphi \vee \psi) \text{ U } \psi$	$\Longleftrightarrow$	$\varphi \text{ U } \psi$

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

F.5 Operator replacement

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

$$\Box\varphi \iff \neg(\top \text{ U } \neg\varphi)$$

$$\Diamond\varphi \iff \top \text{ U } \varphi$$

$$\circ_W \psi \iff \neg\circ(\neg\varphi)$$

$$\varphi \text{ W } \psi \iff \begin{cases} \text{if } |\psi| \leq |\varphi|^8\text{:} & \neg(\neg\psi \text{ U } (\neg\varphi \wedge \neg\psi)) \\ \text{else:} & (\varphi \text{ U } \psi) \vee \Box\varphi \end{cases}$$

$$\neg(\neg\varphi \text{ U } \neg\psi) \iff \varphi \text{ R } \psi$$

F.6 Translation of timed operators

F.6.1 Timed Salt into timed Salt--

T(**`next timed`**`[`$\sim c$`]`φ)	$=$	$\triangleright_{\sim c}\mathrm{T}(\varphi)$
T(φ **`until timed`**`[`$\sim c$`]` ψ)	$=$	$\mathrm{T}(\varphi)\ \mathrm{U}_{\sim c}\ \mathrm{T}(\psi)$
T(φ **`until timed`**`[`$\sim c$`]` **`weak`** ψ)	$=$	$\mathrm{T}(\varphi)\ \mathrm{W}_{\sim c}\ \mathrm{T}(\psi)$
T(φ **`until timed`**`[`$\sim c$`]` **`excl req`** ψ)	$=$	$\mathrm{T}(\varphi)\ \mathrm{U}_{\sim c}\ \mathrm{T}(\psi)$
T(φ **`until timed`**`[`$\sim c$`]` **`excl opt`** ψ)	$=$	$(\Diamond_{\sim c}\mathrm{T}(\psi)) \rightarrow (\mathrm{T}(\varphi)\ \mathrm{U}_{\sim c}\ \mathrm{T}(\psi))$
T(φ **`until timed`**`[`$\sim c$`]` **`excl weak`** ψ)	$=$	$\mathrm{T}(\varphi)\ \mathrm{W}_{\sim c}\ \mathrm{T}(\psi)$
T(φ **`until timed`**`[`$\sim c$`]` **`incl req`** ψ)	$=$	$\mathrm{T}(\varphi)\ \mathrm{U}_{\sim c}\ (\mathrm{T}(\varphi) \wedge \mathrm{T}(\psi))$
T(φ **`until timed`**`[`$\sim c$`]` **`incl opt`** ψ)	$=$	$(\Diamond_{\sim c}\mathrm{T}(\psi)) \rightarrow (\mathrm{T}(\varphi)\ \mathrm{U}_{\sim c}\ (\mathrm{T}(\varphi) \wedge \mathrm{T}(\psi)))$
T(φ **`until timed`**`[`$\sim c$`]` **`incl weak`** ψ)	$=$	$\mathrm{T}(\varphi)\ \mathrm{W}_{\sim c}\ (\mathrm{T}(\varphi) \wedge \mathrm{T}(\psi))$
T(**`timed`**`[`$\sim c$`]` φ **`releases`** ψ)	$=$	$\mathrm{T}(\psi)\ \mathrm{W}_{\sim c}\ (\mathrm{T}(\psi) \wedge \mathrm{T}(\varphi))$
T(**`always timed`**`[`$\sim c$`]` φ)	$=$	$\Box_{\sim c}\mathrm{T}(\varphi)$
T(**`eventually timed`**`[`$\sim c$`]` φ)	$=$	$\Diamond_{\sim c}\mathrm{T}(\varphi)$

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

F.6.2 Timed SALT-- into extended TLTL

acc (`accepton`):

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

rej (`rejecton`):

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

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

F.6.3 Extended TLTL into pure TLTL

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

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

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

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

Appendix G

Examples

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

Simple specification

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

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

Specification using `until`

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

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

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

Scheduler specification

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

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

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

Precedence specification

This example makes use of macros and past operators.

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

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

Elevator specification

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

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

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

Input channel iteration specification

This example makes use of iteration operators.

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

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

Response pattern specification

This example makes use of regular expressions.

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

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

Real-time example

This example uses the timed extension of Salt.

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

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

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

Appendix H

Index