Sonnenbergstraße 13 · D-70184 Stuttgart
Tel. 07 11 / 21 037 00 · Fax 07 11 / 21 037 53

Run-Time Testing with Sequence LTL

Projekt VerSyKo - Technical Report

Version	1.0	**Datum**	2011-12-15
Status	☒ IP1 ☐ IP2 ☐ IR ☐ SR ☐ IW ☐ PD ☐ RL		
Autor	Martin Sulzmann, Axel Zechner, Ramin Hedayati	Unterschrift	
Freigabe	N.N.	Unterschrift	

Abstract

Mission critical system must meet high safety standards and require thorough verification/testing of their software components. Unfortunately, formal notations such as linear temporal logic are often not used due to the lack of domain-specific LTL abstractions. We introduce a domain-specific variant of LTL, referred to as SEQUENCE LTL where properties are specified as simple temporal sequences of boolean propositions connected by temporal arrow combinators. We have applied SEQUENCE LTL for run-time testing in several real-world mission-critical embedded system applications. Finite trace matching with SEQUENCE LTL allows for a natural explanation of why a trace could be matched and not matched. Such information is highly useful for error-debugging and discovering if the test cases sufficiently cover a SEQUENCE LTL property. We believe that SEQUENCE LTL supports a better understanding of LTL properties for non-experts and opens up the opportunity for further applications of formal methods in industry.

1 Introduction

Linear temporal logic (LTL) [9] is a powerful formalism for the concise specification of complex, temporal interaction patterns and has numerous applications to verify the static and dynamic behavior of software systems.

In static verification we rely on a model checker to verify that the model, typically represented as a Büchi automata, satisfies the LTL specification. For most real-world software systems, model checking is not feasible due to state explosion. To make the model checking problem tractable, we would need to build an abstraction of the actual system.

The alternative is to give up on static verification and rely on run-time testing. At run-time, the system produces a trace log. The trace log is then matched against the LTL test property. The advantage of run-time verification is that there is no need to abstract the system's behavior. Of course, we can only detect the presence of a system's incorrect behavior but not its absence. This disadvantage is generally accepted. Run-time testing is the predominant method in industry to validate and verify software systems.

Unfortunately, LTL is not widely used in industry because domain experts such as system engineers simply lack the expertise in specifying LTL formulas. Even for experts, the specification of LTL properties derived out of textual descriptions can be a daunting task. One of the main obstacles that hinder the more wide spread adoption of LTL in industry is the lack of suitable domain-specific LTL abstractions.

In this paper, we introduce a domain-specific variant of LTL, referred to as SEQUENCE LTL, which has the following two novel aspects:

1. SEQUENCE LTL is applied in several real-world applications for run-time testing of mission-critical embedded systems. By imposing some restrictions in expressiveness, SEQUENCE LTL supports a better understanding of LTL properties for non-experts and fosters the dialogue between formal method experts, test engineers and domain experts.

2. Matching a finite trace against a LTL specification normally yields a yes/no answer only. That is, either the trace can be matched or a match does not exist. In case of SEQUENCE LTL, we produce a detailed explanation of why a trace could be matched and not matched against a SEQUENCE LTL formula. This information can be used for error-debugging and discovering if the test cases sufficiently cover a SEQUENCE LTL property.

In the following, we briefly elaborate on these two points.

SEQUENCE LTL Key Ideas

Intuitive LTL specification. SEQUENCE LTL statements are specified as temporal sequences of boolean propositions connected by temporal arrow combinators such as $\cdot \xrightarrow{+} \cdot$, $\cdot \xrightarrow{n} \cdot$, and $\cdot \xrightarrow{U+} \cdot$. Boolean propositions are of the form

$$Engine = On \;\; \&\& \;\; Key = Pressed$$

which states that the engine is on and the key is pressed. For the moment, we abbreviate propositions by letters A, B etc. The meaning of temporal arrow combinators is as follows. $A \xrightarrow{+} B$ denotes that A holds now and after that B holds eventually. $A \xrightarrow{n} B$ denotes that A holds now and B holds after exactly n steps where $n > 0$. $A \xrightarrow{U+} B$ denotes that A holds at least once until B holds. We postpone the description of some further combinators till later.

Here's a more complex SEQUENCE LTL statement making use of the above combinators:

$$A \xrightarrow{+} B \xrightarrow{2} C \xrightarrow{U+} D$$

The above states that after proposition A, within one or more steps, proposition B must hold. Exactly two steps after B, proposition C holds. C holds as long as we reach a step where proposition D holds.

The 'temporal arrow' notation enforced by SEQUENCE LTL guarantees that statements in SEQUENCE LTL can be directly connected to the common diagrammatic explanation of the semantics of LTL formulas in terms of program traces. In every temporal sequence the proposition to left must occur in the trace before any proposition to right. In our experience, this makes it easier to reason about the correctness of a LTL specification and is of great help when discussing LTL specifications with domain experts such as system engineers which have no formal training in LTL. For comparison, here's the equivalent LTL formula written in the common notation:

$$A \wedge (next\ (\diamond\ (B \wedge next\ next\ (C \wedge (C\ until\ D)))))$$

In SEQUENCE LTL, we trade clarity for expressiveness. Temporal sequences cannot be arbitrarily nested, rather, sequences must be written in right-associative normal form. For example, in SEQUENCE LTL we disallow nested sequences such as $(A \xrightarrow{U+} B) \xrightarrow{U+} C$. The meaning of such statements can be hard to interpret even for LTL experts. Therefore, such statements are forbidden in SEQUENCE LTL. In our practical experience, this leads to a clearer and more transparent specification of temporal properties.

As a further restriction, we require that negation and conjunction only to appear among boolean propositions. That is, $(A \xrightarrow{U+} B) \wedge (C \xrightarrow{3} D)$ is disallowed. The motivation for this restriction is that the test property shall be as atomic as possible and capture only a single trace which has possibly some choice points. Simultaneous traces, i.e. conjunction among temporal sequences, indicates that the test property should be broken up into more elementary parts.

Operators choice $\vee$ and always $\Box$ (a.k.a. globally) can be used arbitrarily to compose SEQUENCE LTL statements.

Constructive Finite Trace Matching. In run-time testing, we are given a finite trace and wish to check if the trace matches the property. Suppose, the system under test yields a finite trace, e.g.

$$[(1; A), (2; A), (3; B), (4; D)]$$

which states that in cycle 1 and 2 proposition A holds, in cycle 3 proposition B and so on.

Matching the above trace against

$$A \xrightarrow{+} B \xrightarrow{2} C \xrightarrow{U+} D$$

yields the annotated formula

$$A_1^{\surd} \xrightarrow{+} B_3^{\surd} \xrightarrow{2} (C \xrightarrow{U+} D)^{?}$$

The annotation $\cdot^{\surd}$ indicates that the proposition matches an element of the trace whereas the subscript indicates the position (cycle number) in the trace. The annotation $\cdot^{?}$ tells us that the sub-formula $C \xrightarrow{U+} D$ could not be tried because the trace ended prematurely for the following reason. After matching B the above formula tells us to take two further steps. But then we have already reached the end of the trace. Hence, we cannot try $C \xrightarrow{U+} D$.

The question is how to evaluate formulas which contain $\cdot^{?}$ annotations. Here, we follow a 'weak' finite trace LTL semantics [2] where in doubt we report match success. Our reason for adopting a weak finite trace LTL semantics is that we want to reduce the number of false negatives. A match shall only fail if we can provide a concrete counter-example. Hence, we interpret $\cdot^{?}$ as a successful match. For the above example, we therefore report success.

We generally demand that for each test property specified as a SEQUENCE LTL formula there is at least one test case, i.e. trace, which has a match not containing $\cdot^{?}$ annotations. This guarantees that either we have a concrete counter-example trace falsifying the property, or showing that the property holds for a specific example trace.

Contributions and Outline of this Report

In summary, we make the following contributions:

- We introduce SEQUENCE LTL, explain its semantic by translation to LTL and motivate its use via several real-world examples (Section 3).
- We give an algorithm for matching a finite trace against a SEQUENCE LTL formula. The algorithm is constructive and yields detailed explanation of success and failure of matching (Section 4). We also can give feedback to the user whether the SEQUENCE LTL test property is sufficiently covered by the test cases provided (Section 4.2).
- We have fully implemented the approach in the functional language Haskell. The SEQUENCE LTL language is embedded as a domain-specific language extension in Haskell. We report on our experiences in applying SEQUENCE LTL for run-time testing of several real-world projects in the area of mission-critical embedded systems (Section 5).

In the up-coming section, we give an overview of our run-time testing set-up. Related work is discussed in Section 6. We conclude in Section 7.

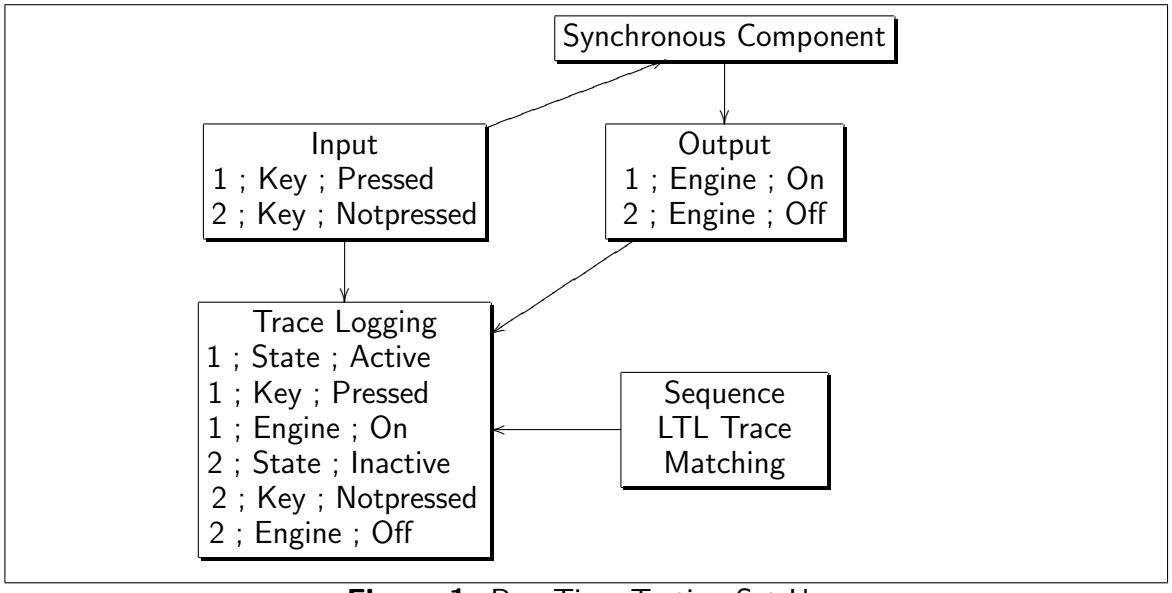

Figure 1: Run-Time Testing Set-Up

2 Run-Time Testing Set-Up

Our current application domain of SEQUENCE LTL is for off-line, run-time testing of mission-critical embedded systems. The diagram in Figure 1 gives an overview of our off-line run-time testing set-up. We consider testing of synchronous components which read and write its input and output at fixed cycle intervals. The component is stimulated by setting input sensors to specific values at designated cycle points. Notation `1 ; Key ; Pressed` states that in the first cycle the input key is pressed. For each cycle and input we obtain some output. In our example, in the first cycle the output sensor `Engine` is set to the value `On`. Besides input and output sensor values there can also be some internal state. In each cycle, we log these values in a program trace which is stored off-line in a file. For example, in the first cycle, the component's state is active, the input key is pressed and as output the engine is turned on. The stored program trace is then matched against the test property specified as a SEQUENCE LTL statement to check if the component violates some properties.

In principle, our approach can also be used for on-line run-time testing at the system level. But all our current examples involve a single synchronous component where the program trace is analyzed off-line.

3 SEQUENCE LTL

Figure 2 describes the syntax of SEQUENCE LTL statements. Sequence connectors $\cdot \xrightarrow{n} \cdot$, $\cdot \xrightarrow{+} \cdot$ and $\cdot \xrightarrow{U+} \cdot$ we have already seen in the introduction. The conditional connectors $\cdot \overset{n}{\Longrightarrow} \cdot$, $\cdot \overset{+}{\Longrightarrow} \cdot$ could be expressed in terms of $\cdot \xrightarrow{n} \cdot$ and $\cdot \xrightarrow{+} \cdot$. For example, $P \overset{n}{\Longrightarrow} S$ is logically equivalent to $!P \vee (P \xrightarrow{n} S)$. For reasons of coverage, see up-coming Section 4.2, we treat the conditional connectors as primitives.

SEQUENCE LTL:

x	$::=$	i	Input sensor
	$\mid$	o	Output sensor
	$\mid$	s	Internal state
v	$::=$	$0 \mid 1 \mid ...$	Integer values
	$\mid$	$On \mid Off \mid ...$	Enum values
e	$::=$	$x \mid v \mid e + e \mid ...$	Expressions
A	$::=$	$e = e \mid e > e \mid e \leq e \mid ...$	Atomic propositions
P	$::=$	$A \mid P \&\& P \mid P \mid\mid P$	Boolean propositions
	$\mid$	$!P \mid P \implies P$	
S	$::=$	P	
	$\mid$	$S \vee S$	Choice
	$\mid$	$\Box\, S$	Always
	$\mid$	$P \xrightarrow{+} S$	One or more steps
	$\mid$	$P \overset{+}{\Longrightarrow} S$	Conditional steps
	$\mid$	$P \xrightarrow{n} S$	n steps, $n > 0$
	$\mid$	$P \overset{n}{\Longrightarrow} S$	Conditional n steps, $n > 0$
	$\mid$	$P \xrightarrow{U+} S$	At least once until

Figure 2: SEQUENCE LTL Syntax

Below are some further derived connectors which we use frequently in our application:

$$P \xrightarrow{(n,m)} S \quad \text{Within } n \text{ and } m \text{ steps, } n > 0,\ m - n > 0$$
$$= P \xrightarrow{n} S \vee ... \vee P \xrightarrow{m} S$$

$$P \overset{(n,m)}{\Longrightarrow} S \quad \text{Conditional version}$$
$$= P \overset{n}{\Longrightarrow} S \vee ... \vee P \overset{m}{\Longrightarrow} S$$

$$P \xrightarrow{U(n,m)} S \quad \text{Min } n, \text{ max } m \text{ times, } n > 0,\ m - n > 0$$
$$= \underbrace{P \xrightarrow{1} ...P \xrightarrow{1}}_{n \text{ times}}(S \vee P \xrightarrow{1} S \vee ... \vee \underbrace{P \xrightarrow{1} ...P \xrightarrow{1}}_{m-n \text{ times}} S)$$

$$P \overset{U(n,m)}{\Longrightarrow} S \quad \text{Conditional variant}$$
$$= \underbrace{P \overset{1}{\Longrightarrow} ...P \overset{1}{\Longrightarrow}}_{n \text{ times}}(S \vee P \overset{1}{\Longrightarrow} S \vee ... \vee \underbrace{P \overset{1}{\Longrightarrow} ...P \overset{1}{\Longrightarrow}}_{m-n \text{ times}} S)$$

The meaning of SEQUENCE LTL statements is explained by translation to standard LTL expressions. For completeness, we repeat the definition of standard LTL formulas.

$$\begin{array}{rcl} L & ::= & A \mid B \mid ... \mid L \wedge L \mid L \vee L \mid !L \\ & \mid & next\ L \mid \diamond\ L \mid L\ until\ L \mid \Box\ L \end{array}$$

The translation function from SEQUENCE LTL to standard LTL is defined by structural induction over SEQUENCE LTL statements.

$[\![\cdot]\!]$ :: SEQUENCE LTL $\rightarrow$ Standard LTL

$$\begin{array}{lcl} [\![A]\!] & = & A \\ [\![P_1 \&\& P_2]\!] & = & [\![P_1]\!] \wedge [\![P_2]\!] \\ [\![P_1 || P_2]\!] & = & [\![P_1]\!] \vee [\![P_2]\!] \\ [\![!P]\!] & = & ![\![P]\!] \\ [\![S_1 \vee S_2]\!] & = & [\![S_1]\!] \vee [\![S_2]\!] \\ [\![P \xrightarrow{+} S]\!] & = & [\![P]\!] \wedge next\ (\diamond\ [\![S]\!]) \\ [\![P \stackrel{+}{\Longrightarrow} S]\!] & = & [\![!P]\!] \vee ([\![P]\!] \wedge next\ (\diamond\ [\![S]\!])) \\ [\![P \xrightarrow{n} S]\!] & = & [\![P]\!] \wedge next^n\ ([\![S]\!]) \\ [\![P \stackrel{n}{\Longrightarrow} S]\!] & = & [\![!P]\!] \vee ([\![P]\!] \wedge next^n\ ([\![S]\!])) \\ [\![P \xrightarrow{U+} S]\!] & = & [\![P]\!] \wedge next\ ([\![P]\!]\ until\ [\![S]\!]) \end{array}$$

In the above, we use the short-hand notation $next^n\ L$:

$$next^1\ L = next\ L \quad next^{n+1}\ L = next\ (next^n\ L)$$

Many of the LTL patterns [6] found in the literature can be re-phrased in terms of SEQUENCE LTL. However, as motivated in the introduction, SEQUENCE LTL doesn't support nested LTL statements such as

$$((A\ until\ B) \wedge (C\ until\ D))\ until\ E$$

We summarize these observations in the following propositions.

Proposition 3.1 LTL patterns in [6] involving propositional statements can be expressed in terms of SEQUENCE LTL.

Proposition 3.2 There are LTL statements which cannot be expressed in terms of SEQUENCE LTL.

Due to space limitations, we omit the formal details for the above propositions.

In our experience, the restrictions imposed on SEQUENCE LTL do not represent a serious limitation. We currently employ SEQUENCE LTL for the specification of run-time tests in the following three real-world applications. For each application, we give a sample textual requirement and its natural translation to SEQUENCE LTL:

- Railway level-crossing:

  If a train enters the safe-guarding section the road traffic must be stopped within 30 to 60 cycles.

$$\square \left(\begin{array}{l} Train = Absent \\ \quad \overset{1}{\Longrightarrow} Train = Detected \\ \qquad \overset{U(30,60)}{\longrightarrow} \left(\begin{array}{c} TrainSig = Go \\ \Longrightarrow \\ RoadSig = Stop \end{array} \right) \end{array} \right)$$

- Motor engine start-stop system (MESS):

  Via a key, the driver can switch on and off the MESS system. The MESS system becomes active on the rising edge of a key stroke.

$$\square \left(\begin{array}{l} (Key = Off \ \ \&\& \ \ Status = Active) \\ \quad \overset{1}{\Longrightarrow} (Key = On \implies Status = InActive) \end{array} \right)$$

- Central control system in the area of land defense systems.

  If the fire button is pressed for longer than 100 cycles, the alarm shall be turned on.

$$\square \ (FireButton = Pressed \overset{U(100,100)}{\Longrightarrow} Alarm = On)$$

4 Run-Time Testing

The component under test yields a finite trace which is of the following form:

V	::=	$[x_1 = v_1, ..., x_n = v_n]$	Sensor/State assignment
T	::=	$[V_1, ..., V_m]$	Finite trace

We use Haskell notation for lists and also employ various helper functions whose declarations are given below:

$[]$	Empty list/trace
$V : T$	Trace with head V and tail T
length xs	Length of a list xs
head xs	Head of a list xs
drop i xs	Drop i elements from xs, yields $[]$ if there are not enough elements
take i xs	Take i elements from xs, yields $[]$ if there are not enough elements
repeat x	List with x repeated infinitely often

Our task is to check if the finite trace T violates any SEQUENCE LTL formula S. In first step, we match T against S which is formalized in terms of judgments $T \vdash S \rightsquigarrow M$

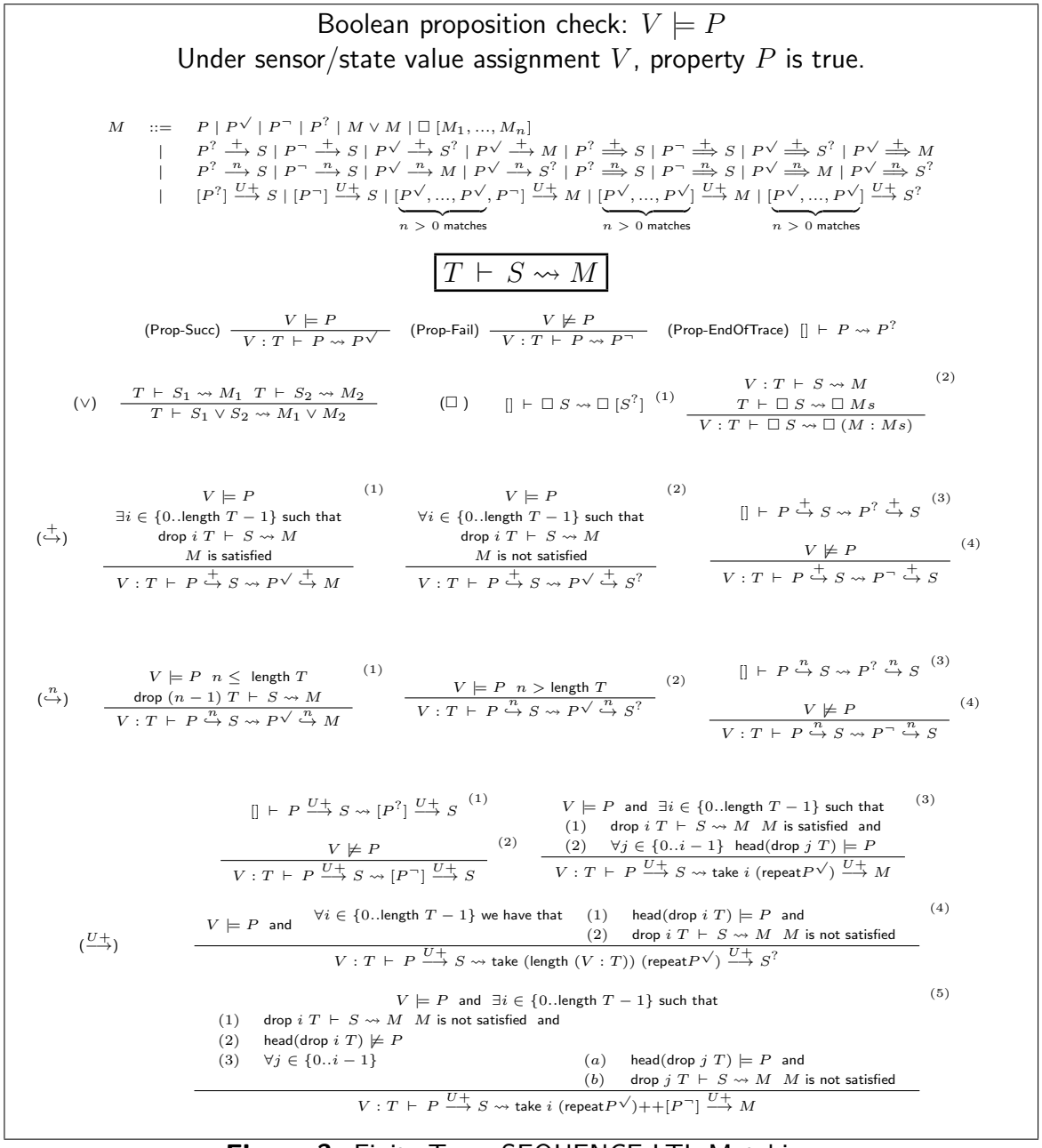

Figure 3: Finite Trace SEQUENCE LTL Matching

where M is an annotated formula derived from S describing the match result. Annotations are of the following form:

$.^{\surd}$	Success annotation
$.^{\neg}$	Failure annotation
$.^{?}$	Premature end of trace

The possible matchings and the matching judgment $T \vdash S \rightsquigarrow M$ are defined in Figure 3. We use numbers to distinguish the various sub-cases of a rule group. For

$$
\begin{array}{l}
eval(\cdot) :: M \to \{True, False\} \\
eval(P^{\surd}) = True \quad eval(P^{\neg}) = False \quad eval(P^{?}) = False \\
eval(M_1 \vee M_2) = eval(M_1) \vee eval(M_2)
\end{array}
$$

$$
\begin{array}{ll}
eval(\Box\ [M^{?}]) = True & eval(\Box\ (M : Ms)) = \\
 & \quad eval(M) \wedge eval(\Box\ Ms) \\
eval(P^{?} \xrightarrow{+} S) = False & eval(P^{\neg} \xrightarrow{+} S) = False \\
eval(P^{\surd} \xrightarrow{+} S^{?}) = True & eval(P^{\surd} \xrightarrow{+} M) = eval(M) \\
eval(P^{?} \Longrightarrow^{+} S) = False & eval(P^{\neg} \Longrightarrow^{+} S) = True \\
eval(P^{\surd} \Longrightarrow^{+} S^{?}) = True & eval(P^{\surd} \Longrightarrow^{+} M) = eval(M) \\
eval(P^{?} \xrightarrow{n} S) = False & eval(P^{\neg} \xrightarrow{n} S) = False \\
eval(P^{\surd} \xrightarrow{n} S^{?}) = True & eval(P^{\surd} \xrightarrow{n} M) = eval(M) \\
eval(P^{?} \Longrightarrow^{n} S) = False & eval(P^{\neg} \Longrightarrow^{n} S) = True \\
eval(P^{\surd} \Longrightarrow^{n} S^{?}) = True & eval(P^{\surd} \Longrightarrow^{n} M) = eval(M) \\
eval(\_ \xrightarrow{U+} S^{?}) = False & eval(ps \xrightarrow{U+} M) = \\
 & \quad evalPs(ps) \wedge eval(M)
\end{array}
$$

$$
\begin{array}{lcl}
evalPs(\cdot) & :: & [P_1, ..., P_n] \to \{True, False\} \\
evalPs([]) & = & True \\
evalPs(P^{\surd} : ps) & = & evalPs(ps) \\
evalPs(P^{\neg} : ps) & = & False
\end{array}
$$

Figure 4: SEQUENCE LTL Match Satisfiability

example, we write ($\Box$ -(1)) and ($\Box$ -(2)) to refer to the two sub-cases of the rule group ($\Box$). ($\hookrightarrow$) are rule schemas where $\hookrightarrow$ is place-holder for $\longrightarrow$ and $\Longrightarrow$ which have identical matching rules.

In Figure 4, we define the outcome of matching in terms of the $eval(\cdot)$ which takes an annotated formula and either yields true (match success) or false (match failure). We define that M is satisfied iff $eval(M) = True$.

Some of the function patterns are overlapping, e.g. consider the case for $\xrightarrow{U+}$. We therefore assume that the cases are tried from top to bottom and from left to right.

We take a look at some of the details.

4.1 Finite Trace SEQUENCE LTL Matching

Matching yields an annotated SEQUENCE LTL formula, For simplicity, we don't record cycle numbers in a match. Our implementation provides such details. We even record which parts of a boolean formula contributed to success and failure.

We briefly discuss the possible matchings, i.e. annotated SEQUENCE LTL formulas, obtained via $\cdot \vdash \cdot \rightsquigarrow \cdot$ and their evaluation via $eval(\cdot)$ by considering the various cases.

P: A proposition is either true or false or can't be tried because the trace ended prematurely. P itself is not a match result but appears within in unmatched formula.

Therefore, there's no need to define the $eval(P)$ case.

$S_1 \vee S_2$: We match both branches but only require that one branch is successful.

$\Box\ S$: We record a match for each step. That is, we obtain a list of matches. Each step must be successful for the overall match to be successful.

$P \xrightarrow{+} S$: If the match yields $P^? \xrightarrow{+} S$, the trace ended prematurely. For this case, we report match failure because we couldn't check the leading proposition P. In case of $P^\neg \xrightarrow{+} S$, the boolean proposition failed and therefore we report match failure. $P^\surd \xrightarrow{+} S^?$ means that the proposition succeeded but S couldn't be matched successfully because the trace ended prematurely. As motivated in the introduction, we employ a weak finite LTL semantics to reduce the amount of false negatives. Therefore, we report match success. Finally, for $P^\surd \xrightarrow{+} M$ we consult the match M to decide about success or failure.

$P \xRightarrow{+} S$: This case is similar to $P \xrightarrow{+} S$ with the exception that if P yields true, we report match success.

Cases $P \xrightarrow{n} S$ and $P \xRightarrow{n} S$ are analogous to the above.

$P \xrightarrow{U+} S$: We record a match each time P holds. We assume that the judgment rules for ($\xrightarrow{U+}$) in Figure 3 are tried from top to bottom. The possible match outcomes are as follows. In case of $[P^?] \xrightarrow{U+} S$ the trace ended prematurely. Like in case of $P^? \xrightarrow{+} S$, we report match failure. $[P^\neg] \xrightarrow{U+} S$ indicates failure in the first step and also yields match failure. In case of

$$[\underbrace{P^\surd, ..., P^\surd}_{n > 0 \text{ matches}}, P^\neg] \xrightarrow{U+} M$$

we reach a step where we could neither successfully match P nor S. Hence, we again report match failure. The case

$$[\underbrace{P^\surd, ..., P^\surd}_{n > 0 \text{ matches}}] \xrightarrow{U+} M$$

indicates that we could match P in each step until we could match S. Hence, we report success. The last remaining case is

$$[\underbrace{P^\surd, ..., P^\surd}_{n > 0 \text{ matches}}] \xrightarrow{U+} S^?$$

In this case we could match P in each step but couldn't find match for S because the trace ended prematurely. In our design, we assume a strong until semantics and therefore we report match failure.

Here's a sample match derivation. For simplicity, we abbreviate sensor/state assignments by letters.

$$\frac{\dfrac{[C] \vdash C \rightsquigarrow C^\surd}{[B, D, C] \vdash B \xRightarrow{2} C \rightsquigarrow B^\surd \xRightarrow{2} C^\surd}}{[A, A, B, D, C] \vdash A \xrightarrow{+} B \xRightarrow{2} C \rightsquigarrow A^\surd \xrightarrow{+} B^\surd \xRightarrow{2} C^\surd}$$

The individual derivation steps are as follows. In the first derivation step, trace $[A, A, B, D, C]$ matches $A \xrightarrow{+} B \overset{2}{\Longrightarrow} C$ by application of rule ($\xrightarrow{+}$-(1)). The leading proposition A in $A \xrightarrow{+} B \overset{2}{\Longrightarrow} C$ matches the first element in $[A, A, B, D, C]$. According to rule ($\xrightarrow{+}$-(1)) we must find a suffix of $[A, A, B, D, C]$ which matches $B \overset{2}{\Longrightarrow} C$. The suffix here is $[B, D, C]$. That is we drop one element. Next, we apply ($\overset{2}{\Longrightarrow}$-(1)). Finally, we reach the case (Prop-Succ).

In fact, there's another possible match as shown by the below derivation.

$$\frac{[A, B, D, C] \vdash B \overset{2}{\Longrightarrow} C \rightsquigarrow B^{\neg} \overset{2}{\Longrightarrow} C}{[A, A, B, D, C] \vdash A \xrightarrow{+} B \overset{2}{\Longrightarrow} C \rightsquigarrow A^{\checkmark} \xrightarrow{+} B^{\neg} \overset{2}{\Longrightarrow} C}$$

The difference is that during the first derivation step using rule ($\xrightarrow{+}$-(1)) we select the suffix $[A, B, D, C]$ to match $B \overset{2}{\Longrightarrow} C$ which then leads to the subsequent derivation step

$$[A, B, D, C] \vdash B \overset{2}{\Longrightarrow} C \rightsquigarrow B^{\neg} \overset{2}{\Longrightarrow} C$$

where we use ($\overset{2}{\Longrightarrow}$-(4)).

From the above, we conclude that our matching relation $\cdot \vdash \cdot \rightsquigarrow \cdot$ is indeterministic. For a given trace T and SEQUENCE LTL formula S, we possibly obtain distinct matchings M_1 and M_2 where $T \vdash S \rightsquigarrow M_1$ and $T \vdash S \rightsquigarrow M_2$. Both matchings are possibly satisfiable. See the above example. The indeterminism is due to connectors $\xrightarrow{+}$, $\overset{+}{\Longrightarrow}$ and $\xrightarrow{U+}$ which possibly allow us to make an arbitrary number of $n > 0$ steps in the future.

This situation is similar to the regular expression matching case where we also may obtain several matchings unless we impose a deterministic matching policy such as greedy or POSIX matching.

In our setting, we favor matchings which show that either we have a concrete counter-example trace falsifying the property, or showing that the property holds for a specific example trace. If either of these conditions are met, we say that a match sufficiently covers a SEQUENCE LTL formula.

4.2 Matching Coverage

Figure 5 formalizes coverage of annotated SEQUENCE LTL obtained through finite trace matching. $cov(\cdot)$ yields true if either we have a concrete counter-example trace falsifying the property, or showing that the property holds for a specific example trace. The definition of $cov(\cdot)$ is derived from $eval(\cdot)$. We have underlined the differences. Compared to $eval(\cdot)$ we always interpret $^?$ as a failure case. In conditional connector statements, we require that pre-conditions must be successful. In case of $ps \xrightarrow{U+} M$, we either have a concrete counter-example or the statement holds for this specific trace. Hence, we immediately report true. Next, we consider some examples to illustrate coverage.

$$
\begin{array}{ll}
cov(\cdot) :: M \to \{True, False\} & \\
cov(P^{\surd}) = True \quad cov(P^{\neg}) = \underline{True} \quad cov(P^{?}) = False & \\
cov(M_1 \vee M_2) = cov(M_1) \vee cov(M_2) & \\
cov(\Box\ [M^{?}]) = \underline{False} & cov(\Box\ (M : Ms)) = \underline{cov(M) \vee cov(\Box\ Ms)} \\
cov(P^{?} \xrightarrow{+} S) = False & cov(P^{\neg} \xrightarrow{+} S) = \underline{True} \\
cov(P^{\surd} \xrightarrow{+} S^{?}) = \underline{False} & cov(P^{\surd} \xrightarrow{+} M) = cov(M) \\
cov(P^{?} \Longrightarrow^{+} S) = False & cov(P^{\neg} \Longrightarrow^{+} S) = \underline{False} \\
cov(P^{\surd} \Longrightarrow^{+} S^{?}) = \underline{False} & cov(P^{\surd} \Longrightarrow^{+} M) = cov(M) \\
cov(P^{?} \xrightarrow{n} S) = False & cov(P^{\neg} \xrightarrow{n} S) = \underline{True} \\
cov(P^{\surd} \xrightarrow{n} S^{?}) = \underline{False} & cov(P^{\surd} \xrightarrow{n} M) = cov(M) \\
cov(P^{?} \Longrightarrow^{n} S) = False & cov(P^{\neg} \Longrightarrow^{n} S) = \underline{False} \\
cov(P^{\surd} \Longrightarrow^{n} S^{?}) = \underline{False} & cov(P^{\surd} \Longrightarrow^{n} M) = cov(M) \\
cov(\_ \xrightarrow{U+} S^{?}) = False & cov(ps \xrightarrow{U+} M) = \underline{True}
\end{array}
$$

Figure 5: SEQUENCE LTL Match Coverage

In our first example, we repeat the two matchings for trace $[A, A, B, D, C]$ and formula $A \xrightarrow{+} B \Longrightarrow^{2} C$ we have seen earlier

$$[A, A, B, D, C] \vdash A \xrightarrow{+} B \Longrightarrow^{2} C \rightsquigarrow A^{\surd} \xrightarrow{+} B^{\surd} \Longrightarrow^{2} C^{\surd}$$

$$[A, A, B, D, C] \vdash A \xrightarrow{+} B \Longrightarrow^{2} C \rightsquigarrow A^{\surd} \xrightarrow{+} B^{\neg} \Longrightarrow^{2} C$$

Coverage for both matchings is as follows

$$
\begin{array}{ll}
 & cov(A^{\surd} \xrightarrow{+} B^{\surd} \Longrightarrow^{2} C^{\surd}) \\
= & cov(B^{\surd} \Longrightarrow^{2} C^{\surd}) \\
= & cov(SuccC) \\
= & True
\end{array}
$$

$$
\begin{array}{ll}
 & cov(A^{\surd} \xrightarrow{+} B^{\neg} \Longrightarrow^{2} C) \\
= & cov(B^{\neg} \Longrightarrow^{2} C) \\
= & False
\end{array}
$$

That is, for the first match function $cov(\cdot)$ yields true but for the second match $cov(\cdot)$ yields false.

In SEQUENCE LTL we can make false statements in temporal sequences but it's impossible to find a match for such formulas where $cov(\cdot)$ yields true. For example, consider

$$A \xrightarrow{1} (!B \vee B) \Longrightarrow^{2} C$$

The proposition $(!B \vee B)$ is false. For trace $[A, C, C]$ we find the match

$$[A, C, C] \vdash A \xrightarrow{1} (!B \vee B) \overset{2}{\Longrightarrow} C \rightsquigarrow A^{\surd} \xrightarrow{1} (!B \vee B)^{\neg} \overset{2}{\Longrightarrow} C$$

Function $cov(\cdot)$ yields false. It's impossible to find matching trace where $cov(\cdot)$ yields true. All propositions must be derivable from some sensor/state value assignment V. Clearly, $V \not\models P$ for any P which is equivalent to false. Hence, the resulting match is $P^{\neg}$ for which $cov(\cdot)$ yields false.

Our next example investigates the differences obtained in coverage when expressing the conditional connector $\overset{n}{\Longrightarrow}$ in terms of $\xrightarrow{n}$. For example, consider the following matchings

$$[B, B] \vdash A \overset{1}{\Longrightarrow} B \rightsquigarrow A^{\neg} \overset{1}{\Longrightarrow} B$$
$$[B, B] \vdash !A \vee (A \xrightarrow{1} B) \rightsquigarrow !A^{\surd} \vee (A^{\neg} \xrightarrow{1} B)$$

In the first case, by using $\overset{1}{\Longrightarrow}$ function $cov(\cdot)$ yields false. In the second case, $A \overset{1}{\Longrightarrow} B$ is expressed in terms of $!A \vee (A \xrightarrow{1} B)$. The coverage computation steps are as follows.

$$\begin{aligned} & cov(!A^{\surd} \vee (A^{\neg} \xrightarrow{1} B)) \\ = \; & cov(!A^{\surd}) \vee cov(A^{\neg} \xrightarrow{1} B) \\ = \; & True \vee False \\ = \; & True \end{aligned}$$

This coverage result may be somewhat unexpected but is obviously correct considering the shape of the formula $!A \vee (A \xrightarrow{1} B)$.

We draw the following conclusions.

- We favor matchings for which $cov(\cdot)$ yields true. For a given trace and SEQUENCE LTL formula, we therefore calculate all matchings and then select the match for which $cov(\cdot)$ yields true (if any such match exists).
- Coverage may vary for logically equivalent SEQUENCE LTL formulas. It's the users responsibility to model the desired coverage criterion. In our applications, we always use $\overset{n}{\Longrightarrow}$.

5 Practical Experiences

We have fully implemented SEQUENCE LTL as an embedded domain-specific language extension in Haskell [8] and we make use of the may extensions found in the Glasgow Haskell Compiler [7]. Haskell is also used for the implementation of the matching and coverage algorithms introduced in Section 4 as well as further algorithms for e.g. rendering the annotated formulas obtained from matching as HTML reports.

Below is the Haskell formulation of the earlier motor engine start-stop system example from Section 3.

```
always $
 (prop $ Key .=. Off) .&&. (prop $ Status .=. Active)
      .==>. 1 $
          (prop $ Key .=. On)
              .=>.
          (prop $ Status .=. InActive)
```

For reasons of space, we only provide some quantitative figures of our SEQUENCE LTL applications and leave a detailed discussion to future work.

- Railway level-crossing: 12 SEQUENCE LTL formulas.
- Motor engine start-stop system: 9 SEQUENCE LTL formulas.
- Central control system in the area of land defense systems: 220 SEQUENCE LTL formulas.

Test cases, i.e. input sequences to generate program traces, for each of the three application are currently written by hand. In future work, we consider generating test cases automatically out of SEQUENCE LTL specifications.

Initially, we used standard LTL but as motivated in the introduction our experiences with standard LTL are rather negative. LTL specifications are often incomprehensible for system and test engineers. One of the most important advantages of SEQUENCE LTL is the ability to build an annotated formula which constructively explains the result of finite trace matching. This feature is useful for several reasons.

Firstly, explaining the success and failure of matching a trace against a test property is helpful when debugging the application. Secondly, our test tool's outputs can manually be verified. This is an important requirement when it comes to formal certification of software where the tools used to build/verify the software must be trusted. For example, the software development standards RTCA/DO-178B [11] and IEC 61508-3[1] require for higher criticality levels that either the applied tools are formally certified, or the tool's outputs can be manually verified.

6 Related Work

Temporal Logic Formalisms. The idea of making the specification of temporal properties more accessible for non-experts is not new. For example, the work in [5] introduces a graphical notation to specify propositions which must hold at certain temporal points (interval). An advantage of our work is that we can explain the semantics of SEQUENCE LTL via a simple translation to LTL. The graphical interval logics allows to explicitly specify negative, i.e. forbidden traces which we have not considered yet.

LSCs [4] are another approach for graphical specification of interactive communicating systems which semantic also allows for comprehensible explanation of traces. However,

the focus of verification for embedded systems does not lie on communication but on the specification of properties on sensors and actors as addressed by SEQUENCE LTL.

The work in [6] identifies typical LTL patterns which can be used as building blocks for more complex specifications. Initially, we considered LTL patterns for the specification of test properties. For our application domain, run-time testing, we found it very difficult to provide reasonable explanations using LTL patterns. By composing patterns the resulting LTL property gets quickly fairly complicated which makes reasonable success/failure explanation a tricky business.

Finite Trace LTL Matching. There are various prior work which study finite trace matching algorithms, e.g. see [10], and the design space of the semantics of finite trace LTL matching, e.g. see [2].

As far as we know, none of the above works address explanation of matching in terms of annotated formulas and coverage of matching which we consider crucial for our application domain, run-time testing of mission critical systems.

Matching Coverage. The work in [12] proposes a coverage criteria which resembles the MC/DC coverage criteria [3].

There's clearly scope to improve our current coverage criteria, for example, by ensuring that the basic conditions in a boolean propositions SEQUENCE LTL statement have been taken all possible outcomes. We would also like to link coverage of SEQUENCE LTL statements with model coverage of our applications.

7 Conclusion

The most distinguishing feature of SEQUENCE LTL is the ability to provide for a constructive and naturally to understand explanation of finite trace matching. We currently use SEQUENCE LTL in three real-world application. Our current experiences as well as feedback from system/test engineers is very positive.

For our current application domains, the SEQUENCE LTL formalism appears to be sufficiently expressive in power. In future work, we would like to explore further application domains of SEQUENCE LTL and study extensions of SEQUENCE LTL if necessary. There are several other directions we plan to pursue in future work, e.g. automated test case generation from SEQUENCE LTL formulas and a more detailed investigation of coverage.

References

[1] IEC 61508-3. Functional safety of electrical / electronic / programmable electronic safety-related systems Part 3: Software requirements, 2010.

[2] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing ltl semantics for runtime verification. J. Log. and Comput., 20:651–674, 2010.

[3] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. Software Engineering Journal In Software Engineering Journal, 9:193–200, 1994.

[4] W. Damm and D. Harel. Lscs: Breathing life into message sequence charts. Technical report, Jerusalem, Israel, Israel, 1998.

[5] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. ACM Trans. Softw. Eng. Methodol., 3:131–165, April 1994.

[6] 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, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[7] Glasgow Haskell compiler home page. `http://www.haskell.org/ghc/`.

[8] Simon Peyton Jones, editor. Haskell 98 Language and Libraries – The Revised Report. Cambridge University Press, Cambridge, England, 2003.

[9] Amir Pnueli. The temporal logic of programs. In 18th Annual Symposium on Foundations of Computer Science, pages 46–57. IEEE, 1977.

[10] Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. Automated Software Engg., 12:151–197, 2005.

[11] RTCA/DO-178B. Software considerations in airborne systems and equipment certification, 1992.

[12] Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage metrics for requirements-based testing. In Proceedings of the 2006 international symposium on Software testing and analysis, ISSTA '06, pages 25–36, New York, NY, USA, 2006. ACM.