Constructive Trace Validation with Flat Temporal Logic Patterns

Martin Sulzmann[1] and Axel Zechner[2]

[1] Hochschule Karlsruhe - Technik und Wirtschaft
martin.sulzmann@hs-karlsruhe.de
[2] Informatik Consulting Systems AG, Germany
axel.zechner@ics-ag.de

Abstract. We consider the problem of verifying if a program trace is valid with respect to a linear temporal logic (LTL) specification. Existing methods only provide yes/no answers. Such answers are unsatisfactory and not helpful to precisely track down the source of a failure. We show that by restricting LTL formulas to flat, non-nested and right-associative forms, we obtain a method which provides detailed explanations about the outcome of trace validation in a form comprehensible for the user. For example, we obtain information which parts of the specification have been exercised in case of success and which parts have been violated in case of failure. The approach has been fully implemented and is applied in some real-world projects for testing of synchronously executed software components.

1 Introduction

We consider an instance of run-time testing where the system under test yields a finite trace and validity of the trace is verified by checking if a linear temporal logic (LTL) specification [12] matches the trace. Our interest is in providing detailed explanations *why* trace validation has succeeded/failed.

Specifically, we seek for detailed explanations which establish precise connections between the trace *and* the LTL formula. In case of failure, we can thus much easier localize which parts of the specification have been violated. In case of success, we can check if a specification is sufficiently covered. For example, if a pre-condition is never satisfied, the specification clearly holds, but not all parts of the specification may be sufficiently covered. Clearly, such detailed information, is essential for debugging of failed tests and evaluating the quality of tests (e.g. coverage of the specification).

Our approach to provide more detailed explanations is to view finite trace validation as a parsing problem. In some prior work [15], we showed how to generate witnesses in case a match for a LTL formula exists. A witness represents a successful parse tree explaining in detail which subparts of the LTL formula match which subparts of the trace. A short-coming of our prior work is that we didn't consider the case of failed trace validation nor did we appropriately cover the case where an extension of the trace could possibly lead to success. These two cases are clearly of high practical relevance.

Another point is that the full power of LTL is rarely used in practice. We argue that by restricting the the set of allowable temporal patterns, the results of trace validation become much more comprehensible for the user.

In this paper, we address the above issues Specifically, we make the following contributions:

- We introduce FlatLTL, a LTL variant inspired by [3, 2] where formulas may only be composed of flat, non-nested and right-associative LTL statements. We explain the semantics of FlatLTL by translation to LTL and motivate its use via several real-world examples (Section 2).
- We explain the outcome of trace validation for FlatLTL via the following cases: (a) why there is a match, (b) why there is no match, or (c) why there could possibly be a match (Section 3.2).
- We develop precise notions of orderings among matches in case there are ambiguous results (Section 3.3).
- We introduce an algorithm to compute matches according to a fixed ordering strategy (Section 3.4).
- We relate our approach to earlier works [7] on weak and strong finite trace LTL semantics (Section 4).

Further related work is discussed in Section 5. We conclude in Section 6.

The constructive trace validation approach is fully implemented and integrated into a complete tool-chain for implementation, verification and testing of synchronously executed software components. The appendix gives an overview. Our implementation as well as an example from the Automotive area are freely available via

`http://www.home.hs-karlsruhe.de/~suma0002/SEQLTL.html`

2 Temporal Specifications with FlatLTL

Syntax and Semantics FlatLTL statements are specified as temporal sequences of boolean propositions connected by temporal arrow combinators such as $\cdot \xrightarrow{+} \cdot$, $\cdot \xrightarrow{n} \cdot$, and $\cdot \xrightarrow{U+} \cdot$. Following [3, 2], we impose the restriction that temporal sequences must be flat. That is, on the 'left' of an arrow combinator, we may only specify Boolean propositions which we abbreviate by letters A, B and so on in the following examples.

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.

Each connector $\xrightarrow{n}$, $\xrightarrow{+}$ and $\xrightarrow{U+}$ must take at least one step into the future. We believe that this condition is quite useful when specifying a temporal chain of events as in

$$A \xrightarrow{+} B \xrightarrow{+} C$$

The above suggests that B happens after A and C happens after B. This is guaranteed by the semantics of $\xrightarrow{+}$. Of course, we can represent the common

'eventually' operation via $A \xrightarrow{*} B = (A \xrightarrow{+} B) \vee (A \wedge B)$ where we have made explicit that A and B can happen at the same time.

Here's a more complex FlatLTL 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. That is, $\xrightarrow{U+}$ expresses a form of 'until' where we demand that C holds at least once.

Signals and propositions:

x	$::= i$	Input signal
	$\mid o$	Output signal
	$\mid s$	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	$::= True \mid e = e \mid e > e \mid e \leq e \mid ...$	Atomic propositions
P	$::= A \mid P \&\& P \mid P \mid\mid P \mid !P$	Boolean propositions

FlatLTL:

S	$::= P$	Boolean proposition
	$\mid S \vee S$	Choice
	$\mid \Box\ S$	Always
	$\mid P \xrightarrow{+} S$	One or more steps
	$\mid P \xrightarrow{n} S$	n steps, $n > 0$
	$\mid P \xrightarrow{U+} S$	At least once until

$$L ::= A \mid B \mid ... \mid L \wedge L \mid L \vee L \mid !L \mid next\ L \mid \diamond\ L \mid L\ until\ L \mid \Box\ L$$

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

$$\begin{aligned}
[\![A]\!] &= A \\
[\![P_1 \&\& P_2]\!] &= [\![P_1]\!] \wedge [\![P_2]\!] \\
[\![P_1 \mid\mid P_2]\!] &= [\![P_1]\!] \vee [\![P_2]\!] \\
[\![!P]\!] &= ![\![P]\!] \\
[\![S_1 \vee S_2]\!] &= [\![S_1]\!] \vee [\![S_2]\!] \\
[\![\Box\ S]\!] &= \Box\ [\![S]\!] \\
[\![P \xrightarrow{+} S]\!] &= [\![P]\!] \wedge next\ (\diamond\ [\![S]\!]) \\
[\![P \xrightarrow{n} S]\!] &= [\![P]\!] \wedge next^n\ ([\![S]\!]) \\
[\![P \xrightarrow{U+} S]\!] &= [\![P]\!] \wedge next\ ([\![P]\!]\ until\ [\![S]\!])
\end{aligned}$$

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

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

Fig. 1: FlatLTL Syntax and Semantics

Figure 1 summarizes the syntax of FlatLTL statements and gives the translation to standard LTL expressions.

Practical Examples We consider FlatLTL specifications for three real-world applications. For each application, we give a sample textual requirement and its natural translation to FlatLTL. The above examples make use of some derived combinators, see Figure 2, which we frequently use in our applications.

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

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

- Motor start-stop system automatic (MSA):
  Via a key, the driver can switch on and off the MSA system. The MSA system becomes active on the rising edge of a key stroke.

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

- Central control system (CCS) in the area of land defense systems.
  If the fire button is pressed for longer than 100 cycles, the alarm shall be turned on.

$$\square \ (FireButton = Pressed \stackrel{U[100]}{\Longrightarrow} Alarm = On)$$

$$P \stackrel{n}{\Longrightarrow} S \ = \ !P \vee P \stackrel{n}{\longrightarrow} S$$

$$P \longrightarrow^n S \ = \ \underbrace{P \stackrel{1}{\longrightarrow} ... P}_{n \text{ times}} \stackrel{1}{\longrightarrow} S$$

$$P \Longrightarrow^n S \ = \ \underbrace{P \stackrel{1}{\Longrightarrow} ... P}_{n \text{ times}} \stackrel{1}{\Longrightarrow} S$$

$$P \stackrel{U(n,m)}{\longrightarrow} S \ = \ P \longrightarrow^n S \vee ... \vee P \longrightarrow^m S$$
Min n, max m times, $n > 0$, $m - n >= 0$

$$P \stackrel{U[n]}{\Longrightarrow} S \ = \ P \Longrightarrow^{n-1} (P \implies S)$$

Fig. 2: Derived FlatLTL Operators

Comparison to LTL Patterns Many of the LTL patterns [6] found in the literature can be re-phrased in terms of FlatLTL. For example, here is the LTL formula resulting from a pattern specification [6] to state that a flush (F) must happen in between a write (W) and a read (R)

$$\mathsf{InBetween}(\mathsf{W},\mathsf{F},\mathsf{R}) \quad := \quad (\mathsf{W} \wedge (\diamond\, \mathsf{R})) \implies !\mathsf{R}\ \mathit{until}\ \mathsf{F}$$

The equivalent FlatLTL representation is as follows.

$$\begin{aligned} W \overset{1}{\Longrightarrow} & (\Box\ !R) \\ & \vee (F\ \&\&\ !R) \xrightarrow{+} R \\ & \vee (!F\ \&\&\ !R) \xrightarrow{U+} (F\ \&\&\ !R) \xrightarrow{+} R \end{aligned}$$

Symbol $\Box$ denotes the always (globally) operator and $\overset{1}{\Longrightarrow}$ is the conditional variant of $\xrightarrow{1}$. The FlatLTL is clearly more verbose because we effectively enumerate all possible cases. We believe that this has some advantages!

The 'temporal arrow' notation enforced by FlatLTL guarantees that statements in FlatLTL 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 the left must occur in the trace before any proposition to the 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.

Expressiveness We review in more detail the restrictions imposed on FlatLTL. Conjunction among complex formulas are disallowed. Our reasoning is that simultaneous traces, i.e. conjunction among temporal sequences, often indicate that the property should be broken up into more elementary parts. However, disallowing conjunction among complex formulas does not affect the expressiveness of FlatLTL as stated by the following proposition.

Proposition 1 (Closed under Conjunction). *Let S_1 and S_2 be two FlatLTL formulas. Then, there exists a FlatLTL formula S_3 such that $[\![S_1 \wedge S_2]\!]$ and $[\![S_3]\!]$ are equivalent.*

A limiting factor is the restriction that to the left of an FlatLTL arrow combinator we may only specify Boolean propositions. As formally verified in [3], there are LTL statements which cannot be expressed in FlatLTL. However, to go beyond the expressive power of FlatLTL we require some LTL statement with an 'until' hierarchy which is nested by three at least. Hence, the LTL pattern

$$\mathsf{InBetween}(\mathsf{W1},\mathsf{F1},\mathsf{InBetween}(\mathsf{W2},\mathsf{F2},\mathsf{R2}))$$

could be converted to FlatLTL. This may no longer be the case if we would replace R2 by another pattern.

Proposition 2 (Dams[3]). *FlatLTL is strictly less expressive than LTL.*

We believe that the restrictions imposed by FlatLTL are acceptable. Cases which have an 'until' hierarchy of more than three appear to be rare in practice. We happily choose simplicity of the specification over expressiveness. The significant advantage of FlatLTL is that the results of trace validation can be explained in a comprehensible format. This is what we will discuss next.

3 Constructive Finite Trace Validation

3.1 Highlights

We are given a finite trace T and need to check if FlatLTL formula S matches the trace T. Our idea is to view the matching problem as a parsing problem where we may encounter the following three cases:

Witness for successful match: S matches T, i.e. T is an element of S's language and there is a witness in form of a successful parse tree.

Counter-example for failed match: There is no match, i.e. T is not an element and based on the 'failed' parse tree we can explain why there is no match.

Partial match: S possibly matches T but we cannot give any conclusive answer because the trace ended prematurely. We can explain this situation via an incomplete parse tree.

We illustrate each case via some examples.

Witness for successful match Suppose, the system under test yields a finite trace, e.g. $[A, A, B]$. That is, A holds in the first and second cycle and B in the third cycle. Matching against $\mathit{True} \xrightarrow{+} B$ yields the witness $\mathit{True}^{\checkmark} \xrightarrow{+_2} B^{\checkmark}$. A witness is an annotated FlatLTL formula where the annotation $\cdot^{\checkmark}$ indicates that the proposition matches an element of the trace and the subscript annotation in $\xrightarrow{+_2}$ indicates how many steps we advanced in the trace. From the witness we can derive why there is a match. Hence, a witness represents a compact representation of a parse tree.

Counter-example for failed match $A \xrightarrow{1} C$ can't be matched against the trace $[A, B]$. We produce a counter-example in the form $A^{\checkmark} \xrightarrow{1} C^{\neg}$. The annotation $^{\neg}$ indicates the position in the trace where the matching/parsing process encountered failure.

Partial match Matching $A \xrightarrow{+} B \xrightarrow{2} C \xrightarrow{U+} D$ against $[A, A, B, D]$ yields $A_1^{\checkmark} \xrightarrow{+_2} B^{\checkmark} \xrightarrow{2} (C \xrightarrow{U+} D)^{?}$. The annotation $\cdot^{?}$ tells us that the sub-formula $C \xrightarrow{U+} D$ could not be tried because the trace ended prematurely. This happens because after matching B we must take two further steps. But then we have already reached the end of the trace. Hence, we cannot try $C \xrightarrow{U+} D$.

Let's consider a more realistic example. The FlatLTL formula from above

$$
\begin{aligned}
W \xRightarrow{1} & (\Box\ !R) \\
& \vee (F\ \&\&\ !R) \xrightarrow{+} R \\
& \vee (!F\ \&\&\ !R) \xrightarrow{U+} (F\ \&\&\ !R) \xrightarrow{+} R
\end{aligned}
$$

Matching the trace $[W, F, F, F]$ against the above formula yields

1. $W^{\checkmark} \xrightarrow{1} \square\ [!R^{\checkmark}, !R^{\checkmark}, !R^{\checkmark}]$
2. $W^{\checkmark} \xrightarrow{1} (F\ \&\&\ !R)^{\checkmark} \xrightarrow{+_3} R^{?}$
3. $W^{\checkmark} \xrightarrow{1} !F^{\neg} \xrightarrow{U+} (F\ \&\&\ !R) \xrightarrow{+} R$

Each of the above corresponds to one of the 'choice' cases.

The first case represents a witness for a successful match. We use list notation in $\square\ [!R^{\checkmark}, !R^{\checkmark}, !R^{\checkmark}]$ to record the match results for each step in case of a globally property. Thus, we can see that the test case is successful because after some initial write there hasn't been any read.

The second case is an example of a partial match. The sub-match $W^{\checkmark} \xrightarrow{1} (F\ \&\&\ !R)^{\checkmark}$ shows that the flush happened immediately after the initial write. But then a read never took place because after three steps the trace has ended prematurely as indicated by $... \xrightarrow{+_3} R^{?}$.

The third case reports some failure because the atomic proposition $(!F\ \&\&\ !R)$ can't be satisfied. In our implementation, we take care to precisely identify failing and successful parts of Boolean propositions. Hence, we report $!F^{\neg}$ instead of $(!F\ \&\&\ !R)^{\neg}$. The sub-formula which succeeds the failing sub-formula is also included and therefore we find the failed match $W^{\checkmark} \xrightarrow{1} !F^{\neg} \xrightarrow{U+} (F\ \&\&\ !R) \xrightarrow{+} R$.

To summarize, based on the FlatLTL specification and the resulting match results we obtain detailed information of what happens during test execution. In this example, we can easily see that the the test case is possibly 'incomplete' because we haven't so far exercised all parts of the specification. An additional test case is needed to cover the property that there is a read after a flush and a write.

Next, we consider the formal details of how to represent the various match results and how to compute them.

3.2 Constructive Matching via Annotated FlatLTL Formulas

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

$$\begin{array}{ll} V ::= [x_1 = v_1, ..., x_n = v_n] & \text{Signal/State assignment} \\ T ::= [V_1, ..., V_m] & \text{Finite trace} \end{array}$$

where $[]$ denotes the empty list/trace and $V : T$ denotes the trace with head V and tail T. The helper function *length* computes the number of elements in a list/trace.

Trace validation performs a match of the trace against the FlatLTL formula which results in a parse tree (either successful, failed or incomplete). Parse trees are represented by annotated FlatLTL formulas which contain information which parts of the specification have been matched, violated or for which parts no conclusive answer exists. The complete syntax of annotated FlatLTL formulas is given in Figure 3.

We briefly discuss the purpose of the various syntactic forms. Evaluation of a proposition P yields either true or false which we represent by $P^{\checkmark}$ and $P^{\neg}$. The third possibility is that evaluation is not possible because the trace has ended

Annotated FlatLTL:

$$P_M ::= P^{\checkmark} \mid P^{\neg}$$
$$M ::= S^{?} \mid \Box\ [M_1, ..., M_n] \mid [P^{\checkmark}, P_{M_1}, ..., P_{M_j}] \xrightarrow{U+} M \mid P^{\neg} \xrightarrow{U+} S^{?}$$
$$\mid\ P^{\checkmark} \xrightarrow{+_i} M \mid P^{\neg} \xrightarrow{+} S^{?} \mid P^{\checkmark} \xrightarrow{n} M \mid P^{\neg} \xrightarrow{n} S^{?}$$

$$\boxed{T \vdash M}$$

$$(\mathrm{P}^{\checkmark})\ \frac{V \models P}{V : T \vdash P^{\checkmark}} \qquad (\mathrm{P}^{\neg})\ \frac{V \not\models P}{V : T \vdash P^{\neg}} \qquad (\mathrm{S}^{?})\ [] \vdash S^{?} \qquad (\mathrm{S}^{\neg})\ \frac{\gamma \in \{n, +, U+\} \quad V \not\models P}{V : T \vdash P^{\neg} \xrightarrow{\gamma} S^{?}}$$

$$(\xrightarrow{n})\ \frac{n >= 1 \quad V \models P \quad [V_n, ...V_{n+i}] \vdash M}{[V_0, V_1, ..., V_n, ...V_{n+i}] \vdash P^{\checkmark} \xrightarrow{n} M}$$

$$(\xrightarrow{+})\ \frac{n >= 1 \quad V \models P \quad [V_n, ..., V_{n+i}] \vdash M}{[V_0, V_1, ..., V_n, ..., V_{n+i}] \vdash P^{\checkmark} \xrightarrow{+_n} M}$$

$$(\xrightarrow{U+}\text{-1})\ \frac{T \vdash M}{T \vdash [] \xrightarrow{U+} M} \qquad (\xrightarrow{U+}\text{-2})\ \frac{\text{if } \delta =^{\checkmark} \text{ then } V \models P \text{ else } V \not\models P \quad T \vdash [P^{\delta_1}, ..., P^{\delta_i}] \xrightarrow{U+} M}{V : T \vdash [P^{\delta}, P^{\delta_1}, ..., P^{\delta_i}] \xrightarrow{U+} M}$$

$$(\Box\text{ -1})\ \frac{[V] \vdash M}{[V] \vdash \Box\ [M]} \qquad (\Box\text{ -2})\ \frac{V : T \vdash M_1 \quad T \vdash \Box\ [M_2, ..., M_n]}{V : T \vdash \Box\ [M_1, M_2, ..., M_n]}$$

Fig. 3: Traces and Annotated Formulas

prematurely. This case is represented by $P^{?}$. The form $P^{?}$ is a special instance of $S^{?}$ to indicate that a complex formula could not be evaluated.

The form $\Box\ [M_1, ..., M_n]$ arises in case of an $\Box\ S$ property. We record the match for each step which then results into the list of matches $[M_1, ..., M_n]$. Similarly, via the form $[P^{\checkmark}, P_{M_1}, ..., P_{M_j}] \xrightarrow{U+} M$ we record the complete sequence of matches for P in $P \xrightarrow{U+} S$ where we assume that we could match P at least once. The form $P^{\neg} \xrightarrow{U+} S^{?}$ captures immediate failure.

Matching the property $P \xrightarrow{+} S$ results into one of the following forms: $P^{\neg} \xrightarrow{+} S^{?}$, $P^{\checkmark} \xrightarrow{+_i} M$, and $(P \xrightarrow{+} S)^{?}$. The first form captures failure, the second form arises in case we could successfully match P and advance at least i-steps in the trace to reach the match M. The last form represents the case that the trace has ended prematurely. A similar observation applies to $\xrightarrow{n}$.

There is no representation for $\vee$ ('choice') because we always favor a specific match. For example, if matching against $S_1 \vee S_2$ yields the successful matches M_1 and M_2, we favor the 'shortest' successful match among M_1 and M_2. In case

of failure, we select the 'longest' failing match. If there are no successful and failed matches, we favor the 'longest partial' match. Details are described below.

Relation $T \vdash M$ in Figure 3 connects a trace T with an annotated formula M. In essence, the natural deduction style proof system in Figure 3 verifies that the trace agrees with the annotated formula.

As we have seen in Section 3.1, there may be several possible M's for a given trace T where each M stems from the same FlatLTL formula. For

$$\begin{aligned} W \overset{1}{\Longrightarrow} (\Box\ !R) \\ \vee (F\ \&\&\ !R) \xrightarrow{+} R \\ \vee (!F\ \&\&\ !R) \xrightarrow{U+} (F\ \&\&\ !R) \xrightarrow{+} R \end{aligned}$$

we find annotated formulas

1. $W^{\surd} \xrightarrow{1} \Box\ [!R^{\surd}, !R^{\surd}, !R^{\surd}]$
2. $W^{\surd} \xrightarrow{1} (F\ \&\&\ !R)^{\surd} \xrightarrow{+_3} R^{?}$
3. $W^{\surd} \xrightarrow{1} !F^{\neg} \xrightarrow{U+} (F\ \&\&\ !R) \xrightarrow{+} R$

where each of the above can be connected to trace $[W, F, F, F]$ via the relation in Figure 3.

3.3 Ordering among Annotated FlatLTL Formulas

Instead of reporting all possible match results, we select a specific match. Our current approach is as follows. We strictly favor successful matches over failure matches. If there is no success and no failure, then there must be a partial match.

Shortest witness In case of several successful matches, we favor the shortest successful match. For example, consider matching $(\mathit{True} \xrightarrow{+} B) \vee (\mathit{True} \xrightarrow{+} C)$ against $[A, A, B, C]$ where in some intermediate step we encounter the witnesses $\mathit{True} \xrightarrow{+_2} B^{\surd}$ and $\mathit{True} \xrightarrow{+_3} C^{\surd}$. $\mathit{True} \xrightarrow{+_2} B^{\surd}$ is the shorter witness when comparing the length of the parse trees.

Longest counter-example In case of counter-examples, we favor the *longest* instead of the *shortest* explanation. For example, consider $A \overset{1}{\Longrightarrow} C$ which is a short-hand for $!A \vee (A \xrightarrow{1} C)$. When matching this formula against trace $[A, B]$, we find counter-examples $(!A)^{\neg}$ and $A^{\surd} \xrightarrow{1} C^{\neg}$. We argue that counter-examples such as $(!A)^{\neg}$ are too short and therefore too trivial and in our experience don't provide much insight into the problem. Hence, we choose the longer counter-example $A^{\surd} \xrightarrow{1} C^{\neg}$. A similar observation applies to partial matches.

Longest partial example For example, consider $A \xrightarrow{+} B \xrightarrow{+} C \xrightarrow{+} D$ and trace $[A, B, C, B]$. Clearly, $A^{\surd} \xrightarrow{+_1} B^{\surd} \xrightarrow{+_1} C^{\surd} \xrightarrow{+} D^{?}$ and $A^{\surd} \xrightarrow{+_3} B^{\surd} \xrightarrow{+} (C \xrightarrow{+} D)^{?}$ are possible partial matches. But the former (longer partial match) is more informative than the latter.

We formalize the above. First, we introduce three helper functions *succ*, *fail* and *partial* in Figure 4. For a successful match, *succ* yields the trace index

$$
\begin{array}{ll}
succ(P^{\checkmark}) & = 1 \\
succ(P^{\checkmark} \xrightarrow{n} M) & = n + succ(M) \\
succ(P^{\checkmark} \xrightarrow{+_i} M) & = i + succ(M) \\
succ([P^{\delta_1}, ..., P^{\delta_i}] \xrightarrow{U+} M) & = \begin{cases} i + succ(M) & \text{if } \delta_i = {}^{\checkmark} \\ \bot & \text{otherwise} \end{cases} \\
succ(\square\ [M_1, ..., M_n]) & = \max_{i \in \{1,...,n\}} succ(M_i) \\
succ(\_) & = \bot \\
 & \\
fail(P^{\neg}) & = 1 \\
fail(P^{\neg} \xrightarrow{\gamma} M) & = 1 \qquad \gamma \in \{n, +, U+\} \\
fail(P^{\checkmark} \xrightarrow{n} M) & = n + fail(M) \\
fail(P^{\checkmark} \xrightarrow{+_i} M) & = i + fail(M) \\
fail([P^{\delta_1}, ..., P^{\delta_i}] \xrightarrow{U+} M) & = \begin{cases} i & \text{if } \delta_i = {}^{\neg} \\ i + fail(M) & \text{otherwise} \end{cases} \\
fail(\square\ [M_1, ..., M_n]) & = \min_{i \in \{1,...,n\}} fail(M_i) \\
fail(\_) & = \bot \\
 & \\
partial(S^{?}) & = 0 \\
partial(P^{\checkmark} \xrightarrow{n} M) & = n + partial(M) \\
partial(P^{\checkmark} \xrightarrow{+_i} M) & = i + partial(M) \\
partial([P^{\delta_1}, ..., P^{\delta_i}] \xrightarrow{U+} M) & = \begin{cases} i + partial(M) & \text{if } \delta_i = {}^{\checkmark} \\ \bot & \text{otherwise} \end{cases} \\
partial(\square\ [M_1, ..., M_n]) & = \max_{i \in \{1,...,n\}} partial(M_i) \\
partial(\_) & = \bot
\end{array}
$$

Extension of $+$, max and min with $\bot$:

$$
\begin{array}{c}
i + \bot = \bot \qquad \bot + \bot = \bot \qquad \bot + \bot = \bot \\
\max_{i \in I} n_i = \bot \text{ if } n_j = \bot \text{ for some } j \in I \\
\min_{i \in I} n_i = \bot \text{ if } n_j = \bot \text{ for some } j \in I
\end{array}
$$

Fig. 4: Trace Positions of Witnesses, Counter-examples and Partial Examples

position of the last sub-match. For a failed match, *fail* yields the trace index position of the earliest failed submatch. Similarly, *partial* yields the trace index position of the earliest partial submatch.

Helper functions are defined by structural recursion over the shape of annotated FlatLTL formulas. We assume that the cases are tried from top to bottom where default cases such as $succ(\_) = \bot$ apply if none of the other cases are applicable. The purpose of $\bot$ ("undefined") is to indicate that a certain case does not apply. For example, if during the computation of *succ* we encounter a failed or partial match, we rely on the value $\bot$ to cancel out the entire calculation of the path length. See the bottom part of Figure 4 where we extend some common operations with $\bot$. For example, $succ(A^{\checkmark} \xrightarrow{3} B^{?}) = \bot$ because $3 + \bot$ evaluates to $\bot$. Similar reasoning applies to *fail* and *partial*.

Thus, we can impose an ordering among annotated formulas. We define $M_1 \leq_s M_2$ iff $succ(M_1) \leq succ(M_2)$. We define $M_1 \leq_c M_2$ iff $fail(M_1) \leq fail(M_2)$. We define $M_1 \leq_p M_2$ iff $partial(M_1) \leq partial(M_2)$.

Proposition 3 (Total and Well-founded Ordering). *We consider the set E of all witnesses, counter-examples and partial examples which are in relation $T \vdash M$ for a fixed trace T. Then, we have that*

- $\leq_s$ *is a total and well-founded ordering among all witnesses in* E,
- $\leq_c$ *is a total and well-founded ordering among all counter-examples in* E, *and*
- $\leq_p$ *is a total and well-founded ordering among all partial examples in* E

We generally favor *shortest witnesses*, i.e. the minimal witness w.r.t. $\leq_s$. If there are no witnesses, we favor *longest counter-examples*, i.e. the maximal counter-example w.r.t. $\leq_c$. If there are neither witnesses nor counter-examples, we favor *longest partial examples*, i.e. the maximal partial example w.r.t. $\leq_p$.

3.4 Computation of Annotated FlatLTL Formulas

Next, we formalize how to derive specific matchings in terms of a matching relation $T \vdash S \rightsquigarrow M$ where T is the trace, S the FlatLTL formula and M the match result. We assume that T and S are input values. Figure 5 formalizes the various (natural deduction style) inference rules to derive judgments $T \vdash S \rightsquigarrow M$.

Rules ($\mathrm{P}^{\surd}$) and ($\mathrm{P}^{\neg}$) deal with atomic propositions. We write $V \models P$ to denote the Boolean proposition check. This check holds iff under signal/state value assignment V, property P is true. In case of a prematurely ending trace, rule ($\mathrm{S}^{?}$) applies. Rule ($\mathrm{S}^{\neg}$) deals with the case that the leading proposition of a temporal sequence formula has failed. In our implementation we take care to identify subparts which are responsible for success/failure. For example, suppose under some V atomic proposition P_1 is satisfied but not P_2. Then, our implementation reports $[V] \vdash P_1 \&\& P_2 \rightsquigarrow P_1^{\surd}$.

Rules ($\xrightarrow{n}$) and ($\xrightarrow{1}$) deal with bounded FlatLTL formulas. Unbounded formulas making use of the operators $\xrightarrow{+}$ and $\xrightarrow{U+}$ are reduced to the bounded case by trying out all finite combinations. See rules ($\xrightarrow{+}$) and ($\xrightarrow{U+}$). For example, $\xrightarrow{+}$ is reduced to the finite cases $\xrightarrow{1}$,...,$\xrightarrow{n}$. The final resulting match is annotated with the trace position i of the match resulting from the finite combinations.

In case of $\xrightarrow{U+}$, formula $P \longrightarrow^{n} S$ is a short-hand for $P \xrightarrow{1} ...P \xrightarrow{1} S$. That is, for n steps P holds followed by S. By construction, we can guarantee that in the final resulting match $[P^{\surd}, P^{\delta_1}, ..., P^{\delta_i}] \xrightarrow{U+} M$ we have that either each δ_k equals $\surd$, or all but the last δ_k are equal to $\surd$. We assume that P could be matched at least once. The immediate failure case $P^{\neg} \xrightarrow{U+} S^{?}$ is covered by rule ($\mathrm{S}^{\neg}$).

Rules ($\Box$ -1) and ($\Box$ -2) deal with matching against 'globally' properties. We record the matchings in each step.Rule ($\vee$) deals with matching 'choice' where we favor shortest successful matches over longest failed match. If there are neither successful nor failed matches, we favor the longest partial match. Recall the definitions in Section 3.3.

The matching rules in Figure 5 imply a straightforward algorithm to compute a match based on a trace and a formula. The algorithm is decidable and the match computed is unique.

$$\boxed{T \vdash S \rightsquigarrow M}$$

$$(\mathrm{P}^{\checkmark}) \quad \frac{V \models P}{V : T \vdash P \rightsquigarrow P^{\checkmark}} \qquad (\mathrm{P}^{\neg}) \quad \frac{V \not\models P}{V : T \vdash P \rightsquigarrow P^{\neg}}$$

$$(\mathrm{S}^{?}) \quad [] \vdash S \rightsquigarrow S^{?} \qquad (\mathrm{S}^{\neg}) \quad \frac{\gamma \in \{n, +, U+\} \quad V \not\models P}{V : T \vdash P \xrightarrow{\gamma} S \rightsquigarrow P^{\neg} \xrightarrow{\gamma} S^{?}}$$

$$(\xrightarrow{n}) \quad \frac{n > 1 \quad V \models P \quad T \vdash \mathit{True} \xrightarrow{n-1} S \rightsquigarrow \mathit{True}^{\checkmark} \xrightarrow{n-1} M}{V : T \vdash P \xrightarrow{n} S \rightsquigarrow P^{\checkmark} \xrightarrow{n} M}$$

$$(\xrightarrow{1}) \quad \frac{V \models P \quad T \vdash S \rightsquigarrow M}{V : T \vdash P \xrightarrow{1} S \rightsquigarrow P^{\checkmark} \xrightarrow{1} M}$$

$$(\xrightarrow{+}) \quad \frac{V \models P \quad \mathit{length}(T) = n \quad V : T \vdash P \xrightarrow{1} S \vee ... \vee P \xrightarrow{n+1} S \rightsquigarrow P^{\checkmark} \xrightarrow{i} M}{V : T \vdash P \xrightarrow{+} S \rightsquigarrow P^{\checkmark} \xrightarrow{+_i} M}$$

$$(\xrightarrow{U+}) \quad \frac{\begin{array}{c} V \models P \quad \mathit{length}(T) = n \quad \delta_1, ..., \delta_i \in \{^{\checkmark}, ^{\neg}\} \quad i \in \{0, ..., n\} \\ V : T \vdash P \longrightarrow^{1} S \vee ... \vee P \longrightarrow^{n+1} S \rightsquigarrow P^{\checkmark} \xrightarrow{1} P^{\delta_1} \xrightarrow{1} ... \xrightarrow{1} P^{\delta_i} \xrightarrow{1} M \end{array}}{V : T \vdash P \xrightarrow{U+} S \rightsquigarrow [P^{\checkmark}, P^{\delta_1}, ..., P^{\delta_i}] \xrightarrow{U+} M}$$

$$(\Box\text{ -1}) \quad \frac{[V] \vdash S \rightsquigarrow M}{[V] \vdash \Box\, S \rightsquigarrow \Box\, [M]}$$

$$(\Box\text{ -2}) \quad \frac{V : T \vdash S \rightsquigarrow M_1 \quad T \vdash \Box\, S \rightsquigarrow \Box\, [M_2, ..., M_n]}{V : T \vdash \Box\, S \rightsquigarrow \Box\, [M_1, M_2, ..., M_n]}$$

$$(\vee) \quad \frac{\begin{array}{c} V : T \vdash S_1 \rightsquigarrow M_1 ... V : T \vdash S_n \rightsquigarrow M_n \\ \text{where } M_i \text{ such that:} \\ \text{(a) } M_i \text{ is the shortest witness among } M_1, ..., M_n, \text{ or} \\ \text{(b) there is no witness, and} \\ M_i \text{ is the longest counter-example among } M_1, ..., M_n, \text{ or} \\ \text{(c) there are only partial examples, and} \\ M_i \text{ is the longest partial example among } M_1, ..., M_n \end{array}}{V : T \vdash S_1 \vee ... \vee S_n \rightsquigarrow M_i}$$

Fig. 5: Finite Trace FlatLTL Matching

Proposition 4 (Deterministic Result and Termination). *For any trace T and FlatLTL formula S there exists exactly one annotated formula M such that $T \vdash S \rightsquigarrow M$ and M can be computed in finite time. We find that M is either the shortest witness, or the longest counter-example, or the longest partial example if there are neither witnesses nor counter-examples.*

4 Connection to Finite LTL Trace Semantics

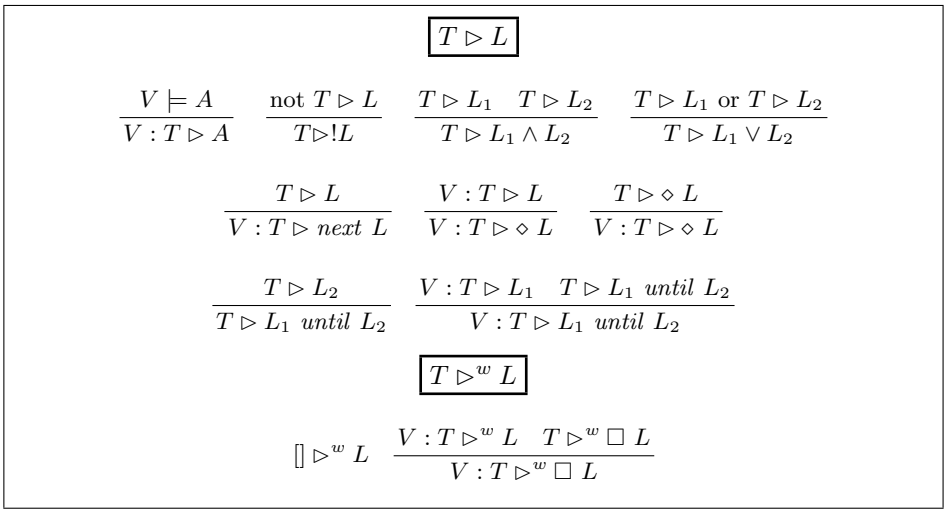

Fig. 6: Weak and Strong Finite Trace LTL Semantics

We relate the outcome of our trace validation algorithm to the finite trace LTL semantics described in [7]. Briefly, witnesses represent valid statements under a strong semantics and partial examples represent valid statements under a weak semantics. The formal details are as follows.

Figure 6 gives a recast of the semantics described in [7] in terms of (natural deduction style) proof system. Proof rules are composed of judgments $T \rhd L$ where T is a finite trace and L a LTL formula.

By default, we assume a *strong* semantics. For example, $[V] \rhd \mathit{next}\ L$ is not derivable. Hence, a globally property can never be satisfied under a strong semantics and therefore there are no rules which involve the $\Box$ operator.

Judgment $T \rhd^w L$ defines a *weak* semantics where we assume that all rules are inherited from $\cdot \rhd \cdot$ plus the two extra rules. Compared the formulations in [7], our formulation of a weak semantics is more liberal because we also accept statements such as $[] \rhd^w \diamond L$.

For witnesses, we can state that the translated FlatLTL formula is valid under a strong finite trace semantics. Recall that mapping function $[\![\cdot]\!]$ takes a FlatLTL formula and yields a plain LTL formula (see Figure 1).

Proposition 5 (Strong Finite Trace Interpretation for Witnesses). *Let T be a trace, S a FlatLTL formula and M an annotated formula such that $T \vdash S \rightsquigarrow M$ and $succ(M) \neq \bot$. Then, $T \rhd [\![S]\!]$.*

For partial examples, validity holds w.r.t. the weak finite trace semantics.

Proposition 6 (Weak Finite Trace Interpretation for Partial Examples). *Let T be a trace, S a FlatLTL formula and M an annotated formula such that $T \vdash S \rightsquigarrow M$ and $partial(M) \neq \bot$. Then, $T \rhd^w [\![S]\!]$.*

5 Related Work

Error Trace Analysis The work in [1] considers the issue of providing more detailed explanation in case of an erroneous trace. The cause of the error is localized by exploiting the existence of correct traces.

Our work assumes that a temporal logic specification is given. We use a constructive matching algorithm to establish precise connections between a trace and a temporal logic specification. The interpretation of match results is left to the user.

Providing further assistance such as suggesting possible fixes to the trace and/or temporal logic specification is something we haven't considered so far. Possibly, abductive inference methods [4] might be helpful.

Finite Trace LTL Matching This topic has been addressed by several earlier works e.g. see [9, 13]. None of these works address the issue of providing more detailed explanation to distinguish between the possible outcomes of matching such as successful, failed and partial matches.

Similar in spirit to our work is the work described in [10, 11]. LTL formulas are translated to automata following the approach in [8]. Then, during run-time verification, states are classified in satisfied, possibly violated and permanently violated states. Thus, feedback to the user can be given about the status of the monitored system.

The difference to our work is that we provide detailed feedback in terms of the original temporal specification as seen by the user. It's not clear to us if/how automata state information could be mapped to our notion of successful, partial and failure match. Specifically, we favor *shortest* successful matches and *longest* partial/failure matches which in our experience provide the most information content.

In our own prior work [15], we generated witnesses for the general LTL case but we found it very difficult to provide meaningful explanations for failed and partial matches. Based on the more specialized FlatLTL language, we can provide detailed explanations for failed and partial matches. We believe that it is worthwhile to trade detailed explanations for expressiveness.

Visual Temporal Logic Approaches It is a well accepted fact that temporal properties often become complex and are hard to debug. There are numerous works which introduce visual notations with the goal to make the specification of LTL properties more accessible for non-experts. For example, see [5, 14].

Our work is clearly inspired by the above works. For example, the concept of 'timeline' in [14] has some resemblance with our form of FlatLTL formulas. Our main focus is on explaining finite-trace matching against temporal formulas which has not discussed by any of the above works.

6 Conclusion

We have introduced a constructive trace validation method for a simplified LTL pattern language to which we refer to as FlatLTL. For our current application domain, the FlatLTL formalism appears to be sufficiently expressive in power.

In future work, we would like to explore further application domains of FlatLTL and study extensions of FlatLTL if necessary. There are several other directions we plan to pursue in future work such as automated test case generation from FlatLTL formulas and improving the display of match results in our implementation.

References

1. Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Proc. of POPL'03*, pages 97–105. ACM, 2003.
2. Werner Damm, Bernhard Josko, and Rainer Schlör. Specification and validation methods. chapter Specification and verification of VHDL-based system-level hardware designs, pages 331–409. Oxford University Press, Inc., New York, NY, USA, 1995.
3. Dennis Dams. Flat fragments of ctl and ctl*: Separating the expressive and distinguishing powers. *Logic Journal of the IGPL*, 7(1):55–78, 1999.
4. Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In *Proc. of PLDI'12*, pages 181–192. ACM, 2012.
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. Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proc. of CAV'03*, volume 2725 of *LNCS*, pages 27–39. Springer, 2003.
8. Dimitra Giannakopoulou and Klaus Havelund. Automata-based verification of temporal properties on running programs. In *Proc. of ASE'01*, pages 412–416. IEEE Computer Society, 2001.
9. Claude Jard and Thierry Jéron. On-line model checking for finite linear temporal logic specifications. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 189–196. Springer, 1990.
10. Fabrizio Maria Maggi, Marco Montali, Michael Westergaard, and Wil M. P. van der Aalst. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In *Proc. of BPM'11*, volume 6896 of *LNCS*, pages 132–147. Springer, 2011.
11. Fabrizio Maria Maggi, Michael Westergaard, Marco Montali, and Wil M. P. van der Aalst. Runtime verification of LTL-based declarative process models. In *Proc. of RV'11*, volume 7186 of *LNCS*, pages 131–146. Springer, 2012.
12. Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.
13. Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engg.*, 12:151–197, 2005.
14. Margaret H. Smith, Gerard J. Holzmann, and Kousha Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, pages 14–, Washington, DC, USA, 2001. IEEE Computer Society.
15. Martin Sulzmann and Axel Zechner. Constructive finite trace analysis with linear temporal logic. In *Proc. of TAP'12*, volume 7305 of *LNCS*, pages 132–148. Springer, 2012.

A FlatLTL Trace Matcher Tool

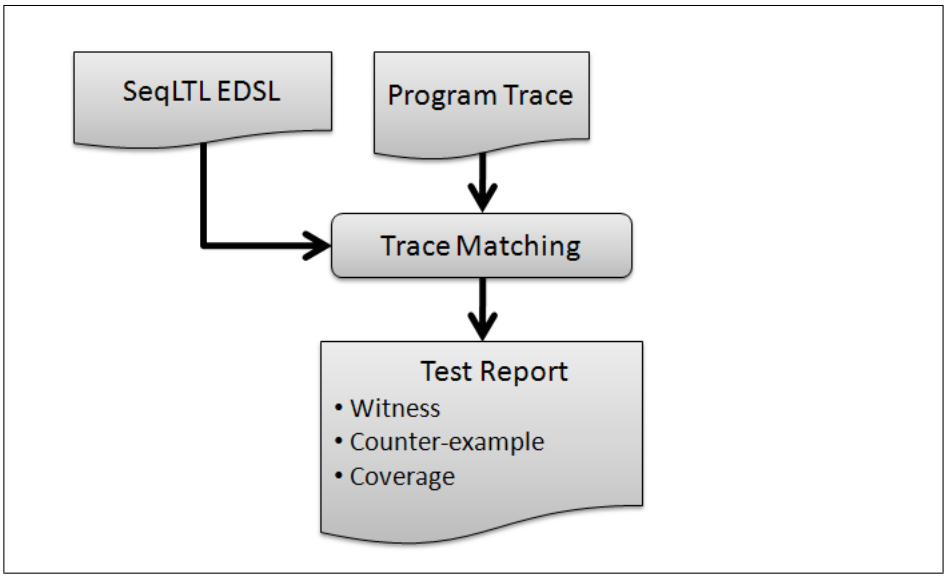

Fig. 7: FlatLTL Trace Matcher

Our constructive method for trace validation is integrated into a complete tool-chain for implementation, verification and testing of synchronously executed software components which read and write their input and output at fixed cycle intervals. Figure 7 gives a brief overview concerning the constructive trace validation part.

Test execution proceeds as follows. Test properties specified in our FlatLTL are matched against finite program traces. These traces are obtained by stimulating the system under test over a fixed time period. The whole process is automated via a tool and the test outcome is reported in HTML format which displays the match result.

FlatLTL EDSL The user interface to FlatLTL is an embedded domain-specific language (EDSL) (also known as internal DSL) in Haskell.

The following example is taken from the Automotive area where we model the requirements of a motor start-stop system automatic (MSA). We shall express the following:

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

The Haskell EDSL formalization `propMSA` is shown in Figure 8.

Compared to an external DSL which comes with its own parser, our EDSL relies on the host language syntax (here Haskell) to represent EDSL constructs. This results in a slightly "noiser" representation of FlatLTL properties.

```
-- Motor start-stop system automatic (MSA) test property
propMSA =
  always $
   (valueIn key .==. constE NotPressed  ./\.
    valueOut status .==. constE Active)
      .==>. 1 $
           (valueIn key .==. constE Pressed)
                  .=>.
           (valueOut status .==. constE InActive)

-- Derived combinator: conditional steps
-- if b holds then s holds in n steps from now on
(.==>.) b n s =  ((.!.) b) .\/. ((.-->.) b n s)
```

Fig. 8: FlatLTL EDSL MSA Example

For example, via EDSL primitive `valueIn` we access the values of input sensor signal `key` and via `valueOut` we access the values of output sensor signal `status`. Via `constE` we turn Haskell constants such as `Pressed`, `NotPressed` etc into EDSL constants.

The advantage of EDSLs is that it is straightforward to provide new language abstractions. For example, the combinator `.==>.` is expressed in terms of the primitive combinator `.-->.`. See Figure 8. Thus, we can keep the constructs of our FlatLTL language fairly small and can quickly customize our system to new application domains.

```
-- Central control system (CCS) test property
propCCS =
 always $
   (valueIn fireButton .==. constE Pressed)
         .=||=>. 100
            (valueOut alarm .==. constE On)

-- Derived combinator: Conditional duration
-- if b holds in n steps then in the nth step s holds
(.=||=>.) b n s
  | n == 1    = (.==>.) b 0 s
  | n > 1     = (.==>.) b 1 ((.=||=>.) b (n-1) s)
  | otherwise = error "(.=||=>.) invalid range"
```

Fig. 9: FlatLTL EDSL CCS Example

We consider another example taken from the area of land defense systems where we specify the properties of a central control system (CCS).

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

To formalize the above, we define a temporal operator to express that if an event holds for a given cycle period, then some other event must follow. The EDSL formalization of the above specification is then straightforward. See Figure 9.

Test property: Atomic parts not covered are underlined:

```
ALWAYS
  CHOICE
    • Not: (All:
                    • IN_GENMSA_KEY = NotPressed
                    • OUT_GENMSA_STATUS = MSA_Active)
    • (All:
               • IN_GENMSA_KEY = NotPressed
               • OUT_GENMSA_STATUS = MSA_Active)
               .--1-->
                  (Any:
                     • Not: _IN_GENMSA_KEY = Pressed_
                     • _OUT_GENMSA_STATUS = MSA_Inactive_ )
```

Test report:

Matched parts are shown in bold face
Match failures are underlined
Unmatched pares are shown in standard font style

```
ALWAYS
Seq:  [Step 1]
         Not: OUT_GENMSA_STATUS(MSA_Inactive) = MSA_Active
Seq:  [Step 2]
         Not: OUT_GENMSA_STATUS(MSA_Inactive) = MSA_Active
Seq:  [Step 3]
         Not: IN_GENMSA_KEY(Pressed) = NotPressed
Seq:  [Step 4]
         Not: IN_GENMSA_KEY(Pressed) = NotPressed
Seq:  [Step 5]
         (All:
            • IN_GENMSA_KEY(NotPressed) = NotPressed
            • OUT_GENMSA_STATUS(MSA_Active) = MSA_Active)
        .--1-->
           (Any:
              • Not: IN_GENMSA_KEY = Pressed
              • OUT_GENMSA_STATUS = MSA_Inactive)
```

Fig. 10: Test Report MSA Example

Test Report MSA Example Figure 10 shows a test report example for the MSA property. There are two parts: (1) The FlatLTL test property and (2) the result of matching the property against a trace. The trace is not shown but can be accessed via some hyperlinks. For display we use a simple textual ASCII representation. More fancy representation, e.g. using visual UML notation, is left for future work. In our actual tool, we use colors to highlight (partial) matches and failures. Here, we use bold face, underline etc.

Part (1) shows that not all parts of the FlatLTL formula have been exercised (i.e. matched) by this test case. Our tool also computes the overall coverage of a

formula across all test cases. Part (2) shows the match result. For a globally property, we report the match result for each step in the trace. As much as possible, we only display 'minimal' parts which are responsible for success/failure.

In the first step, we find that the preconditions (first part of choice) are not satisfied. More precisely, the match [3]

```
Not: OUT_GENMSA_STATUS(MSA_Inactive) = MSA_Active
```

shows that output sensor `MSA_STATUS` has the actual value `MSA_Inactive` but the expected value is `MSA_Active`.

In the last (sixth) step, we find an example of a partial match. All preconditions are satisfied but the actual conditions can't be verified because the trace has ended prematurely.

[3] Prefixes `OUT_GENMSA_`, `OUT_GENMSA_` and `MSA_` arise due to some internal naming conventions and can be easily changed by the user.