Constructive Finite Trace Analysis with Linear Temporal Logic

Martin Sulzmann and Axel Zechner

Informatik Consulting Systems AG, Germany
{martin.sulzmann,axel.zechner}@ics-ag.de

Abstract. We consider linear temporal logic (LTL) for run-time testing over limited time periods. The technical challenge is to check if the finite trace produced by the system under test matches the LTL property. We present a constructive solution to this problem. Our finite trace LTL matching algorithm yields a proof explaining why a match exists. We apply our constructive LTL matching method to check if LTL properties are sufficiently covered by traces resulting from tests.

1 Introduction

Linear temporal logic (LTL) [4] 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.

Our interest here is in the application of LTL for run-time testing. Specifically, our focus is on off-line testing where the system produces a *finite* trace log. The trace log is obtained by stimulation of the system by a test case. The resulting traces are then matched against some LTL formulas which express test properties.

There exists several prior works which study finite LTL trace matching, e.g. see [3, 5]. The problem is that existing algorithms for finite trace matching only yield yes/no answers. That is, either the answer is yes and the trace could be matched, or the answer is no and there is no match. In our view this is often not sufficient. For example, we wish to have a more detailed explanation why a trace could be matched or why is there no match.

Our novel idea is to apply a constructive algorithm for finite trace matching where the algorithm yields a *proof* in case of a successful match. Proofs are representations of parse trees (a.k.a. derivation trees) and provide detailed explanations why there is a match. Thus, we can for example inspect some suspicious test cases which succeeded unexpectedly. There are several further advantages of representing the result of finite LTL trace matching in terms of proofs.

Proofs provide for independent verification of the test results. This is important in case we apply a finite trace LTL trace matching tool in the context of a formal software certification process such as DO-178B [6] where the tools output either must be formally certified or alternatively are manually verifiable. Formal tool certification is often too cost-intensive and requires a potential costly re-certification in case of software changes. Based on the proof representation it is straightforward to verify the test results manually.

Via proofs it is also easy to accommodate for various matching semantics, e.g. weak or strong [2]. The advantage here is that we don't need to re-run the entire matching algorithm if for example we favor a weak semantics. We simply compute the proof and then afterwards we choose the appropriate proof interpretation, e.g. either weak or strong.

Proofs allow us to check to what extent the LTL properties are covered by tests (traces). For example, if some pre-condition is never satisfied the LTL property may be vacuously true but clearly the LTL property is then not fully covered. One of our new contributions is a method to *check* for a fixed set of LTL properties and traces if the LTL properties are sufficiently covered. This complements earlier works [7] which shows how to *generate* traces to sufficiently cover a given set of LTL properties. In practice, automatic generation of test cases is often not possible due to the lack of a formal test model on which we could apply a model checker. Therefore, tests are written by hand. The ability to check coverage of LTL properties is clearly a big win to evaluate the quality of a test suite.

The Key Ideas. We consider an example to highlight the key ideas of our work. We assume a finite trace of the form $[a, b, c, d, d, e, f]$ where letters $a - f$ stand for atomic propositions representing the inputs and outputs of the system under test recorded at some specific measuring points. The test property is specified via the LTL formula $(a \vee b)$ *until* $(c \wedge \mathit{next}\ (\diamond(e \vee f)))$.

Our constructive matching algorithm generates the following proof term

$$[\mathit{inl}\ a^{\checkmark}, \mathit{inr}\ b^{\checkmark}]\ \mathit{until}_{\mathit{prf}}\ (c^{\checkmark}, \mathit{fwd}_{\mathit{next}}\ \ \mathit{fwd}_{\diamond}\ \mathit{fwd}_{\diamond}\ \mathit{stop}_{\diamond}\ e^{\checkmark}) \qquad (1)$$

whose graphical representation is as follows

$$\begin{tikzcd}
 & \text{until} \arrow[dl, "1"'] \arrow[d, "2"] \arrow[r] & {} \arrow[d, "3"] \arrow[dr, "3"] & & & & \\
{} \arrow[d, "\mathit{inl}"] & {} \arrow[d, "\mathit{inr}"] & \text{c} & {} \arrow[r, "4_n"] & {} \arrow[r, "5_\diamond"] & {} \arrow[r, "6_\diamond"] & \text{e} \\
\text{a} & \text{b} & & & & &
\end{tikzcd}$$

Edges are labeled with numbers where numbers refer to specific trace positions. Subscripts $_n$ and $_\diamond$ tell us whether the next trace position is reached due to *next* or $\diamond$. Label *inl* (*inr*) indicates matching against the left (right) component of choice ($\vee$). Leaf nodes represent matched atomic propositions. Based on this representation it is now quite clear that the trace matches the LTL formula.

There are further possible matches, i.e. proofs:

$$[\mathit{inl}\ a^{\checkmark}, \mathit{inr}\ b^{\checkmark}]\ \mathit{until}_{\mathit{prf}}\ (c^{\checkmark}, \mathit{fwd}_{\mathit{next}}\ \ \mathit{fwd}_{\diamond}\ \mathit{fwd}_{\diamond}\ \mathit{fwd}_{\diamond}\ \mathit{stop}_{\diamond}\ f^{\checkmark}) \qquad (2)$$

$$[\mathit{inl}\ a^{\checkmark}, \mathit{inr}\ b^{\checkmark}]\ \mathit{until}_{\mathit{prf}}\ (c^{\checkmark}, \mathit{fwd}_{\mathit{next}}\ \ \mathit{fwd}_{\diamond}\ \mathit{fwd}_{\diamond}\ \mathit{fwd}_{\diamond}\ \mathit{fwd}_{\diamond}\ (\diamond(e \vee f))^{?}) \qquad (3)$$

In proof (2) we match f instead of e in $\diamond(e \vee f)$. Proof (3) represents a match where we reach the end of trace without having matched the sub-formula $\diamond(e \vee f)$. This proof is of course quite silly because we know there is a match without $^{?}$.

The point is that proofs can possibly represent 'partial' matches. That is, subformulas could not be matched due to a prematurely ending trace.

In particular, we are interested in 'shortest' proof. Informally, for a shortest proof the longest derivation path from the root to a leaf is minimal among all other proofs. Proof (1) is the shortest for our running example. Based on shortest proofs we can check if a test suite satisfies the *unique first cause* (UFC) coverage criteria [7]. Briefly, a test suite achieves UFC coverage of a set of requirements expressed as LTL formulas if each condition in an LTL formula has been shown to affect the formula's outcome as the unique first cause for some trace.

For our running example, we find that UFC coverage is *not* achieved. Condition f affects the formula's outcome (the formula is satisfied for this trace). But clearly f is not the unique first cause because in the trace $[a, b, c, d, d, e, f]$ there is the 'earlier' condition e due to which the formula is satisfied as well. This is easy to see by inspecting the shortest proof (1). $e^{\surd}$ appears in the proof (1) but $f^{\surd}$ is absent.

Via the additional trace $[a, b, c, d, d, f, e]$ we achieve UFC coverage. In the shortest proof

$$[\mathit{inl}\ a^{\surd}, \mathit{inr}\ b^{\surd}]\ \mathit{until}_{\mathit{prf}}\ (c^{\surd}, \mathit{fwd}_{\mathit{next}}\ \ \mathit{fwd}_{\diamond}\ \mathit{fwd}_{\diamond}\ \mathit{stop}_{\diamond}\ f^{\surd})$$

resulting from matching the above trace against the LTL formula we find $f^{\surd}$.

Based on the above observations, we can re-phrase the UFC coverage criteria as follows. To achieve UFC coverage, for each condition a in an LTL formula there must exist a shortest proof in which $a^{\surd}$ appears.

Summary of Contributions and Outline of Paper.

– We give a constructive explanation of finite trace LTL matching where the particular finite trace semantics can be chosen a-posteriori (Section 2).
– We provide for an efficient algorithm to compute shortest proofs (Section 3).
– We re-phrase the UFC coverage criteria in terms of shortest proofs (Section 4).

Related work is discussed in Section 5.

2 Constructive Finite Trace LTL Matching

We formalize matching of finite trace T against an LTL formula L, see Figure 1. A trace T is a finite list of atomic propositions where we represent atomic propositions by letters a, b etc. In our actual implementation, propositions are compositions of more elementary basic conditions, e.g. $\mathit{Key} == \mathit{On} \land \mathit{Speed} > \mathit{100}$. For brevity, we ignore this level of detail and only consider *atomic* propositions.

We use the standard LTL syntax. In our formulation, we assume that formulas are in negation normal form. For brevity, we omit the negation operator $\neg$ and assume that the negation of a proposition is simply represented as an atomic proposition, e.g. a.

The matching relation among finite traces and LTL formulas is described in terms of judgments of the form $T \vdash L \leadsto P$. The judgment $T \vdash L \leadsto P$ states that trace T matches formula L and the additional parameter P represents a proof for a match. We generally assume that for the matching relation $\cdot \vdash \cdot \leadsto \cdot$, T and L are input values and P is the output in case of a successful match.

LTL in negation normal form:

B	$::= a \mid b \mid ...$	Atomic propositions
L	$::= B \mid \mathit{True} \mid \mathit{False} \mid L \wedge L \mid L \vee L$	Boolean layer
	$\mid\ \mathit{next}\ L \mid \diamond L \mid L\ \mathit{until}\ L \mid \Box L$	Temporal layer

Finite trace:

T	$::= []$	Empty list/trace
	$\mid\ B : T$	Trace with head B and tail T

Proofs:

$$P \ ::= B^{\checkmark} \mid \mathit{True}^{\checkmark} \mid L^{?} \mid \mathit{inl}\ P \mid \mathit{inr}\ P \mid (P,P)$$
$$\mid\ \mathit{fwd}_{next}\ \ P \mid \Box\ Ps \mid \mathit{stop}_{\diamond}\ P \mid \mathit{fwd}_{\diamond}\ P \mid Ps\ \mathit{until}_{prf}\ P$$
$$Ps ::= [] \mid P : Ps$$

Short-hand for list of proofs: $[P_1, ..., P_n] \ = \ P_1 : ... : P_n : []$

$$\boxed{T \vdash L \rightsquigarrow P}$$

$$\text{(True)} \quad T \vdash \mathit{True} \rightsquigarrow \mathit{True}^{\checkmark} \qquad \text{(Base)} \quad \frac{B' = B}{B' : T \vdash B \rightsquigarrow B^{\checkmark}}$$

$$\text{(EndOfTrace)} \quad \frac{L \neq \mathit{True}}{[] \vdash L \rightsquigarrow L^{?}} \qquad (\mathit{next}) \quad \frac{T \vdash L \rightsquigarrow P}{B : T \vdash \mathit{next}\ L \rightsquigarrow \mathit{fwd}_{next}\ \ P}$$

$$(\vee\text{-Left}) \quad \frac{B : T \vdash L_1 \rightsquigarrow P_1}{B : T \vdash L_1 \vee L_2 \rightsquigarrow \mathit{inl}\ P_1} \qquad (\vee\text{-Right}) \quad \frac{B : T \vdash L_2 \rightsquigarrow P_2}{B : T \vdash L_1 \vee L_2 \rightsquigarrow \mathit{inr}\ P_2}$$

$$(\wedge) \quad \frac{B : T \vdash L_1 \rightsquigarrow P_1 \quad B : T \vdash L_2 \rightsquigarrow P_2}{B : T \vdash L_1 \wedge L_2 \rightsquigarrow (P_1, P_2)}$$

$$(\diamond\text{-Stop}) \quad \frac{B : T \vdash L \rightsquigarrow P}{B : T \vdash \diamond L \rightsquigarrow \mathit{stop}_{\diamond}\ P} \qquad (\diamond\text{-Fwd}) \quad \frac{T \vdash \diamond L \rightsquigarrow P}{B : T \vdash \diamond L \rightsquigarrow \mathit{fwd}_{\diamond}\ P}$$

$$(\Box\text{-1}) \quad \frac{\begin{array}{c} B : T \vdash L \rightsquigarrow P_1 \\ T \vdash \Box L \rightsquigarrow \Box\ Ps \end{array}}{B : T \vdash \Box L \rightsquigarrow \Box\ (P_1 : Ps)} \qquad (\Box\text{-2}) \quad \frac{[B] \vdash L \rightsquigarrow P_1}{[B] \vdash \Box L \rightsquigarrow \Box\ [P_1]}$$

$$(\mathit{until}\text{-1}) \quad \frac{B : T \vdash L_2 \rightsquigarrow P_2}{B : T \vdash L_1\ \mathit{until}\ L_2 \rightsquigarrow []\ \mathit{until}_{prf}\ P_2}$$

$$(\mathit{until}\text{-2}) \quad \frac{B : T \vdash L_1 \rightsquigarrow P_1 \quad T \vdash L_1\ \mathit{until}\ L_2 \rightsquigarrow Ps\ \mathit{until}_{prf}\ P_1}{B : T \vdash L_1\ \mathit{until}\ L_2 \rightsquigarrow P_1 : Ps\ \mathit{until}_{prf}\ P_2}$$

$$(\mathit{until}\text{-3}) \quad \frac{[B] \vdash L_1 \rightsquigarrow P_1}{[B] \vdash L_1\ \mathit{until}\ L_2 \rightsquigarrow [P_1]\ \mathit{until}_{prf}\ L_2^{?}}$$

Fig. 1. Constructive Finite Trace LTL Matching

Figure 1 contains the rules to derive judgments $T \vdash L \leadsto P$. A derivation with final judgment $T \vdash L \leadsto P$ is essentially a (up-side-down) tree where the leaves (judgments) are connected to base rules (EndOfTrace), (True) and (Base) and intermediate nodes are connected to the other rules. In essence, P represents a compact representation of the derivation tree.

The proof $B^{\checkmark}$ states that proposition B could be verified. Such proofs are derived via rule (Base) which states that B matches the head of the trace. $\mathit{True}^{\checkmark}$ is a proof for the always *True* formula which matches any trace. See rule (True).

Rule (EndOfTrace) and the corresponding proof $L^?$ indicate that L is unmatched because the trace ended prematurely. Depending on the interpretation of the match, i.e. proof, we can consider $L^?$ either as a valid match or not and can thus accommodate a weak or strong LTL matching semantics.

Rules ($\vee$-Left) and ($\vee$-Right) deal with matching a non-empty trace against a disjunction of formulas. Proofs *inl* P and *inr* P indicate which branch of a choice formula ($\vee$) could be matched. We use pairs (P, P) to represents proofs for matching a conjunction of formulas. See rule ($\wedge$).

Proof $\mathit{fwd}_{\mathit{next}}\ P$ represents a match for a *next* L formula. See rule (*next*).

Proof $\mathit{stop}_{\diamond}\ P$ indicates that the eventually ($\diamond$) quantified formula could be matched at the present trace position whereas proof $\mathit{fwd}_{\diamond}\ P$ shows that we have to make a step forward in the future to find a match. See rules ($\diamond$-Stop) and ($\diamond$-Fwd).

For example, here's a derivation making use of rule ($\diamond$-Stop). For clarity, each derivation step is annotated with the respective rule applied (read upwards).

$$\dfrac{\dfrac{a = a}{[a,a] \vdash a \leadsto a^{\checkmark}}\ (\text{Base})}{[a,a] \vdash \diamond a \leadsto \mathit{stop}_{\diamond}\ (a^{\checkmark})}\ (\diamond\text{-Stop})$$

In case of the always (a.k.a. globally) operator $\Box$ the quantified LTL formula must hold at each position of the trace. The corresponding proof $\Box\ Ps$ contains therefore a list of proofs Ps where each individual proof represents a proof for a particular position. All of these proofs are collected in a list. See rule ($\Box$-1). In the last step, we reach the empty trace. The resulting proof $(\Box L)^?$ is ignored. See rule ($\Box$-2). For example, consider the following sample derivation.

$$\dfrac{\dfrac{a = a}{[a] \vdash a \leadsto a^{\checkmark}}\ (\text{Base}) \qquad \dfrac{\Box a \neq \mathit{True}}{[] \vdash \Box a \leadsto (\Box a)^?}\ (\text{EndOfTrace})}{[a] \vdash \Box a \leadsto \Box\ [a^{\checkmark}]}\ (\Box\text{-2})$$

Similarly to $\Box$, the proof for *until* uses a list to represent the sub-proof for the left operand. See rules (*until*-1) and (*until*-2). In our formulation, we also build a match/proof in case the right operand can never be matched but the left operand is matched at each position. See rule (*until*-3).

In summary, a proof P represents a compact representation of a derivation where a trace T matches an LTL formula L. That is, from the shape of P we can conclude which rules have been applied to build the derivation and we can reconstruct the entire derivation. In fact, each proof implies a trace and a formula such that the trace matches the formula. This is easy to see, by viewing P as the input and T and L as outputs of the matching relation $T \vdash L \rightsquigarrow P$. It follows:

Lemma 1 (Proofs represent Derivations). *(1) Let $T \vdash L \rightsquigarrow P$ be the final judgment of a derivation. Then, proof P exactly tells us which rules have been applied and in which order.*

(2) Let P be a proof. Then, $T \vdash L \rightsquigarrow P$ is derivable for some trace T and formula L.

A-Posteriori Weak Interpretation of Proofs. The general problem with LTL and finite traces is how to deal with cases where the trace ends prematurely. For example, consider the proof resulting from the derivation

$$\dfrac{\dfrac{a \neq \mathit{True}}{[] \vdash a \rightsquigarrow a^?}\ (\text{EndOfTrace})}{[a] \vdash \mathit{next}\ a \rightsquigarrow \mathit{fwd}_{\mathit{next}}\ \ (a^?)}\ (\mathit{next})$$

In the context of testing with LTL, it is likely that some test cases (traces) are too short for some test properties (LTL formulas). To avoid false negatives, we favor a weak interpretation of proofs.

Definition 1 (Weak Proof Interpretation). *We say that formula L is* weakly matched *by trace T, written $T \vdash_{\mathit{weak}} L$, iff there exists a proof P such that $T \vdash L \rightsquigarrow P$.*

We find that the formula in the above example is weakly matched.

Similarly, we can give a strong proof interpretation following the strong finite trace semantics introduced in [2].

Definition 2 (Strong Proof Interpretation). *We say that formula L is* strongly matched *by trace T, written $T \vdash_{\mathit{strong}} L$, iff there exists a proof P such that $T \vdash L \rightsquigarrow P$ and P does not contain any term of the form $\cdot^?$.*

In case of a weak proof interpretation, it is important to check that the LTL properties are sufficiently covered by test cases. As motivated in the introduction, we use shortest proofs for coverage checking. Next, we formalize shortest proofs.

Shortest Proofs. Figure 2 defines the size of a proof and a formula. The size of the proof is the longest possible path from the root to a leaf. Leafs $B^{\checkmark}$ have size 1 whereas leafs which contain $\mathit{True}^{\checkmark}$ represent trivial matches and therefore we set their size to 0.

For example, consider the proofs obtained by matching $[a, a]$ against $\mathit{next}\ a \vee \Diamond a$:

$$[a, a] \vdash \mathit{next}\ a \vee \Diamond a \rightsquigarrow \mathit{inr}\ (\mathit{stop}_{\Diamond}\ a^{\checkmark}) \quad [a, a] \vdash \mathit{next}\ a \vee \Diamond a \rightsquigarrow \mathit{inl}\ (\mathit{fwd}_{\mathit{next}}\ \ a^{\checkmark})$$

where $\mathit{size}(\mathit{inr}\ (\mathit{stop}_{\Diamond}\ a^{\checkmark})) = 1 < 2 = \mathit{size}(\mathit{inl}\ (\mathit{fwd}_{\mathit{next}}\ \ a^{\checkmark}))$.

$$\boxed{\mathit{size} : P \mapsto \mathbb{N}}$$

$$\begin{array}{llll}
\mathit{size}(B^{\checkmark}) & = 1 & \mathit{size}(\mathit{inl}\ P) & = \mathit{size}(P) \\
\mathit{size}(\mathit{True}^{\checkmark}) & = 0 & \mathit{size}(\mathit{inr}\ P) & = \mathit{size}(P) \\
\mathit{size}(L^{?}) & = \mathit{size}(L) & \mathit{size}(P_1, P_2) & = \mathit{max}(\mathit{size}(P_1), \mathit{size}(P_2)) \\
\mathit{size}(\mathit{stop}_{\diamond}\ P) & = \mathit{size}(P) & \mathit{size}(\mathit{fwd}_{\diamond}\ P) & = 1 + \mathit{size}(P)
\end{array}$$

$$\begin{array}{ll}
\mathit{size}(\mathit{fwd}_{\mathit{next}}\ \ P) & = 1 + \mathit{size}(P) \\
\mathit{size}(\Box[P_1, ..., P_n]) & = 1 + \mathit{max}_{i=1}^{i \leq n}((i-1) + \mathit{size}(P_i)) \\
\mathit{size}([P_1, ..., P_n]\ \mathit{until}_{\mathit{prf}}\ \ P) & = 1 + \mathit{max}(n + \mathit{size}(P), \mathit{max}_{i=1}^{i \leq n}((i-1) + \mathit{size}(P_i)))
\end{array}$$

$$\boxed{\mathit{size} : L \mapsto \mathbb{N}}$$

$$\begin{array}{llll}
\mathit{size}(B) & = 1 & \mathit{size}(\mathit{True}) & = 1 \\
\mathit{size}(\mathit{False}) & = 1 & \mathit{size}(L_1 \wedge L_2) & = \mathit{size}(L_1) + \mathit{size}(L_2) \\
\mathit{size}(L_1 \vee L_2) & = \mathit{size}(L_1) + \mathit{size}(L_2) & \mathit{size}(\mathit{next}\ L) & = 1 + \mathit{size}(L) \\
\mathit{size}(\diamond L) & = 1 + \mathit{size}(L) & \mathit{size}(L_1\ \mathit{until}\ L_2) & = 1 + \mathit{size}(L_1) + \mathit{size}(L_2) \\
\mathit{size}(\Box L) & = 1 + \mathit{size}(L) & &
\end{array}$$

Fig. 2. Size of Proofs and Formulas

In case of a proof for $\wedge$ the size of the overall proof is determined by the maximum of the size of the sub-proofs. Similarly, we compute the maximum of the sub-proofs for $\Box$ and *until* . The difference compared to $\wedge$ is that for $\Box$ and *until* we add $i - 1$ to take into account the iterations through the trace. The additional 1+ in e.g. $\mathit{size}(\Box L) = 1 + \mathit{size}(L)$ ensures to unambiguously select among proofs for $\Box$ and *until* and proofs for some unfolding of $\Box$ and *until* for a specific trace. For example, consider

$$[a] \vdash \Box a \vee a \leadsto \mathit{inl}\ \Box\ [a^{\checkmark}] \qquad [a] \vdash \Box a \vee a \leadsto \mathit{inr}\ a^{\checkmark}$$

For trace $[a]$, proposition a is essentially the unfolded version of $\Box a$. We strictly favor the unfolded version by adding 1 in case of $\Box$. For the above, we find that

$$\mathit{size}(\mathit{inr}\ a^{\checkmark}) = 1 < 2 = \mathit{size}(\mathit{inl}\ \Box\ [a^{\checkmark}])$$

The only remaining ambiguity arises in pathological cases such as matching $[a]$ against $a \vee a$ and matching $[a]$ against $\mathit{next}\ b \vee \mathit{next}\ c$. In the first case, we find two identical matches by either choosing the left or right branch. In the second case, the trace ended prematurely and we end up with unresolved formulas of equal size in the left and right branch.

To resolve such un-ambiguities, we favor the "left-most" proof in case of several shortest proofs. For brevity, we omit a formal definition and only provide the intuition. We say a proof P_1 is *left-most* w.r.t. some other proof P_2 iff along the longest paths from the root of P_1 and P_2 we find that P_1's path takes earlier a left turn than P_2's path.

Definition 3 (Shortest Left-Most Proof). *Let $T \vdash L \leadsto P$. We say that P is the* shortest left-most proof *w.r.t. trace T and formula L iff for any other proof P' such that $T \vdash L \leadsto P'$ we have that either*

– *$size(P) < size(P')$, or*
– *$size(P) = size(P')$ and P is left-most w.r.t. P'.*

Obviously, there exists other strategies to make the matching relation deterministic. For example, instead of choosing the left-most proof among the shortest proofs, we could choose the shortest proof among the left-most proofs.

Definition 4 (Left-Most Shortest Proof). *Let $T \vdash L \rightsquigarrow P$. We say that P is the* left-most shortest proof *w.r.t. trace T and formula L iff P is a left-most proof and for any other proof left-most proof P' such that $T \vdash L \rightsquigarrow P'$ we have that $size(P) < size(P')$.*

For example, proof $inl\ \Box\ [a^{\checkmark}]$ is the left-most shortest proof for trace $[a]$ and $\Box a \vee a$. But as shown above, this proof is not the shortest left-most.

3 Deterministic Matching with Derivatives

We first develop an algorithm to compute the left-most shortest match. Based on that we then derive an algorithm for computing the shortest left-most match.

The straightforward approach to obtain the left-most shortest match would be to employ a back-tracking algorithm where we interpret the judgments in Figure 1 as Prolog clauses. However, such an approach easily leads to undesirable high run-time behavior.

For example, consider the trace $[a, ..., a, c]$ and the formula $\diamond(a \wedge \diamond b)$. In each step, besides the last step, we can match a and then seek for b which cannot be matched. Thus, we end up with a quadratic run-time behavior where we would expect that a linear scan of the trace ought to be sufficient. This situation is similar to the regular expression for which it is well-known that a back-tracking matching algorithm easily leads to exponential run-time behavior.

To avoid unnecessary back-tracking, we seek for a matching algorithm which strictly guarantees to make progress towards computing a proof. The basic idea is to reduce the matching problem $B : T \vdash L$ to the simpler problem $T \vdash L\backslash B$ where formula $L\backslash B$ is obtained from L by consuming the current head B of the trace. The formula $L\backslash B$ is referred to as the *derivative* of L with respect to B and can be obtained by structural induction over the shape of L. The concept of derivatives, originally developed for regular expressions [1], also applies to linear temporal logic as first shown in [3] We extend this idea to compute the left-most shortest and shortest left-most match.

One of the challenges we face is to build the proof of the original formula out of the proof of the derivative. Roughly, we attack this challenge as follows. For a trace $[B_1, ..., B_n]$, we build the sequence of derivatives $L \rightarrow_{f_1} L\backslash B_1 \rightarrow_{f_2} ... \rightarrow_{f_n} L\backslash B_1...\backslash B_n$. The purpose of the f_i's will be explained shortly. By using Boolean laws we check if the final formula $L\backslash B_1...\backslash B_n$ yields true. If yes, we can build a proof P. The proof for the original formula L is obtained by applying the proof transformers f_i. In each derivative step, we compute a proof transformer function f_i which tells us how to build the proof of the original formula given the proof of the derivative. Thus, we obtain the proof of the initial formula L by application of $(f_1 \circ\ ... \circ\ f_n)\ P$. Next, we formalize this idea.

Computing the Left-Most Shortest Match. Figure 3 defines judgments

Expressions and functions over proofs:

$$\begin{array}{rcll} e & ::= & P & \text{Proofs} \\ & \mid & \mathsf{case}\ P\ \mathsf{of}\ \overline{P \to P} & \text{Case expression} \\ f & ::= & \lambda P.e & \text{Functions with input pattern } P \\ & \mid & \bot & \text{Undefined} \end{array}$$

$$\boxed{L \backslash B \vdash_d \ (L \mathbin{|} P \to P)}$$

$$(\mathrm{True}_d) \quad \mathit{True} \backslash B \vdash_d \ (\mathit{True} \mathbin{|} \lambda \mathit{True}^{\checkmark}.\mathit{True}^{\checkmark}) \qquad (\mathrm{False}_d) \quad \mathit{False} \backslash B \vdash_d \ (\mathit{False} \mathbin{|} \bot)$$

$$(\text{Succ-B}_d) \ \frac{B' = B}{B' \backslash B \vdash_d \ (\mathit{True} \mathbin{|} \lambda \mathit{True}^{\checkmark}.B^{\checkmark})} \qquad (\text{Fail-B}_d) \ \frac{B' \neq B}{B' \backslash B \vdash_d \ (\mathit{False} \mathbin{|} \bot)}$$

$$(\vee_d) \ \frac{\begin{array}{c} L_1 \backslash B \vdash_d \ (L_1' \mathbin{|} f_1) \\ L_2 \backslash B \vdash_d \ (L_2' \mathbin{|} f_2) \\ f = \lambda P.\ \mathsf{case}\ P\ \mathsf{of} \\ \quad \mathit{inl}\ P' \to \mathit{inl}\ (f_1\ P') \\ \quad \mathit{inr}\ P' \to \mathit{inr}\ (f_2\ P') \end{array}}{(L_1 \vee L_2) \backslash B \vdash_d \ (L_1' \vee L_2' \mathbin{|} f)} \qquad (\wedge_d) \ \frac{\begin{array}{c} L_1 \backslash B \vdash_d \ (L_1' \mathbin{|} f_1) \\ L_2 \backslash B \vdash_d \ (L_2' \mathbin{|} f_2) \\ f = \lambda (P_1, P_2).(f_1\ P_1, f_2\ P_2) \end{array}}{(L_1 \wedge L_2) \backslash B \vdash_d \ (L_1' \wedge L_2' \mathbin{|} f)}$$

$$(\mathit{next}_d) \quad (\mathit{next}\ L) \backslash B \vdash_d \ (L \mathbin{|} \lambda P.\mathit{fwd}_{\mathit{next}}\ \ P$$

$$(\diamond_d) \ \frac{\begin{array}{c} L \backslash B \vdash_d \ (L' \mathbin{|} f') \\ f = \lambda P.\ \mathsf{case}\ P\ \mathsf{of} \\ \quad \mathit{inl}\ P' \to \mathit{stop}_{\diamond}\ (f\text{‘}\ P') \\ \quad \mathit{inr}\ P' \to \mathit{fwd}_{\diamond}\ P' \end{array}}{(\diamond L) \backslash B \vdash_d \ (L' \vee \diamond L \mathbin{|} f)} \qquad (\Box_d) \ \frac{\begin{array}{c} L \backslash B \vdash_d \ (L', f') \\ f = \lambda (P, P').\ \mathsf{case}\ P'\ \mathsf{of} \\ \quad \Box\ Ps \to \Box\ (f'\ P : Ps) \\ \quad (\Box L)^{?} \to \Box\ [f'\ P] \end{array}}{(\Box L) \backslash B \vdash_d \ (L' \wedge \Box L \mathbin{|} f)}$$

$$(\mathit{until}_d) \ \frac{\begin{array}{l} \qquad L_1 \backslash B \vdash_d \ (L_1', f_1) \quad L_2 \backslash B \vdash_d \ (L_2', f_2) \\ f = \lambda P.\ \mathsf{case}\ P\ \mathsf{of} \\ \quad \mathit{inl}\ P' \qquad\qquad\qquad\quad\ \to []\ \mathit{until}_{\mathit{prf}}\ f_2\ P' \\ \quad \mathit{inr}\ (P_1, P_2\ \mathit{until}_{\mathit{prf}}\ P_3) \to ((f_1\ P_1) : P_2)\ \mathit{until}_{\mathit{prf}}\ P_3 \\ \quad \mathit{inr}\ (P_1, P_2^{?}) \qquad\qquad\ \to [f_1\ P_1]\ \mathit{until}_{\mathit{prf}}\ (P_2^{?}) \end{array}}{(L_1\ \mathit{until}\ L_2) \backslash B \vdash_d \ (L_2' \vee (L_1' \wedge (L_1\ \mathit{until}\ L_2)) \mathbin{|} f)}$$

Fig. 3. Building Derivatives and Proof Transformers

$L \backslash B \vdash_d \ (L' \mathbin{|} f)$ which build the derivative $L' = L \backslash B$ and also a proof transformation function which transforms a proof for L' into a proof for L.

The base cases (True_d), (False_d), (Succ-B_d) and (Fail-B_d) are straightforward. False formulas are represented by $\bot$, the undefined proof transformer. As we will see, false formulas and their $\bot$ proofs only appear in intermediate steps. They will be eventually discarded because they are not derivable in our matching rule system.

$$\boxed{L \vdash_p P}$$

$$(\text{True}_p) \quad \mathit{True} \vdash_p \mathit{True}^{\checkmark} \qquad (\wedge) \quad \frac{L_1 \vdash_p P_1 \quad L_2 \vdash_p P_2}{(L_1 \wedge L_2) \vdash_p (P_1, P_2)}$$

$$(\vee\text{-Left}_p) \quad \frac{L_1 \vdash_p P_1}{(L_1 \vee L_2) \vdash_p \mathit{inl}\ P_1} \qquad (\vee\text{-Right}_p) \quad \frac{\begin{array}{c}\text{there is no } P_1 \text{ such that } L_1 \vdash_p P_1 \\ L_2 \vdash_p P_2\end{array}}{(L_1 \vee L_2) \vdash_p \mathit{inl}\ P_2}$$

$$(\text{Base}_p) \quad B \vdash_p B^? \qquad (\diamond_p) \quad \diamond L \vdash_p (\diamond L)^? \qquad (\mathit{next}_p) \quad \mathit{next}\ L \vdash_p (\mathit{next}\ L)^?$$

$$(\Box_p) \quad \Box L \vdash_p (\Box L)^? \qquad (\mathit{until}_p) \quad L_1\ \mathit{until}\ L_2 \vdash_p (L_1\ \mathit{until}\ L_2)^?$$

Fig. 4. Building the Final Left-Most Proof (weak version)

Rules $(\vee_d)$ and $(\wedge_d)$ are defined by structural induction and contain no surprises. In rule (next_d), we simply drop the *next* $\cdot$ operator.

More interesting is rule $(\diamond_d)$. The derivative of $\diamond L$ w.r.t. B is $(L\backslash B) \vee \diamond L$ where $L\backslash B$ is the derivative of L w.r.t. B. As we will see, we favor the 'left-most' match and therefore we first try to find a match for L at the current position B and only in case of failure we will continue with the next position by trying again $\diamond L$. The proof transformation function f checks if a proof is found in either the left or right component of the resulting derivative and then applies the proof transformer resulting from $L\backslash B$ to construct a proof for $\diamond L$.

In rule $(\Box_d)$, we assume that eventually $\Box L$ is matched against the empty trace which then results in the proof $(\Box L)^?$. Therefore, the second case when building the proof for $\Box L$ given the proof for $L\backslash B \wedge \Box L$.

In rule (until_d), the derivative for $L_1\ \mathit{until}\ L_2$ is $L_2\backslash B \vee (L_1\backslash B \wedge (L_1\ \mathit{until}\ L_2))$ and expresses that we either immediately satisfy L_2, or we must further unroll the until formula. The resulting proof transformer f covers all the until cases (1-3) we have seen in Figure 1.

Figure 4 builds a proof for the final LTL formula. Any unmatched LTL formula is considered as possibly true. Recall that we postpone the decision of how to interpret proofs. The rules strictly favor the left-most match. For example, see rules $(\vee\text{-Left}_p)$ and $(\vee\text{-Right}_p)$.

We have now everything at hand to formalize the derivative-based algorithm for matching a trace against a formula.

Definition 5 (Derivatives Matching Algorithm). *Let L be an LTL formula, T be a finite trace of the form $[B_1, ..., B_n]$ and P be a proof. We say that P is the derivative matching result of matching T against L, written $T \vdash_d L \rightsquigarrow P$, iff*

– *$L\backslash B_1 \vdash_d (L_1 \mid f_1)$,...,$L_{n-1}\backslash B_n \vdash_d (L_n \mid f_n)$ for some L_1,...,L_n and f_1, ...,f_n, and*
– *$L_n \vdash_p P'$ for some P', and*

– $P = (f_1 \circ \ ... \circ \ f_n) \ P'$.

For example, consider trace $[a, a]$ and formula $\mathit{next}\ a \vee \diamond a$. We first build the derivatives of $L = \mathit{next}\ a \vee \diamond a$:

$$L \backslash a = \underbrace{a \vee (\mathit{True} \vee \diamond a)}_{L_1} \qquad L_1 \backslash a = \underbrace{\mathit{True} \vee (\mathit{True} \vee (\mathit{True} \vee \diamond a))}_{L_2}$$

The proof transformers connected to the derivative steps are as follows:

$$L \backslash a \vdash_d \ L_1 \,\mathbf{|}\, \underbrace{\begin{array}{l} \lambda P.\ \mathsf{case}\ P\ \mathsf{of} \\ \quad \mathit{inl}\ P' \rightarrow \ \mathit{inl}\ (\mathit{fwd}_{\mathit{next}}\ \ P') \\ \quad \mathit{inr}\ P' \rightarrow \ \mathsf{case}\ P'\ \mathsf{of} \\ \qquad\qquad\quad \mathit{inl}\ P'' \rightarrow \mathit{stop}_\diamond\ a^{\checkmark} \\ \qquad\qquad\quad \mathit{inr}\ P'' \rightarrow \mathit{fwd}_\diamond\ P'' \end{array}}_{f_1}$$

$$L_1 \backslash a \vdash_d \ L_2 \,\mathbf{|}\, \underbrace{\begin{array}{l} \lambda P.\mathsf{case}\ P\ \mathsf{of}\ \ \mathit{inl}\ P' \rightarrow \ \mathit{inl}\ a^{\checkmark} \\ \qquad\qquad\qquad \mathit{inr}\ P' \rightarrow \mathsf{case}\ P'\ \mathsf{of} \\ \qquad\qquad\qquad\qquad\quad \mathit{inl}\ P'' \rightarrow \mathit{inl}\ \ \mathit{True}^{\checkmark} \\ \qquad\qquad\qquad\qquad\quad \mathit{inr}\ P'' \rightarrow \qquad\qquad \mathsf{case}' P''\ \mathsf{of} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{inl}\ P''' \rightarrow \mathit{stop}_\diamond\ \ \mathit{True}^{\checkmark} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{inr}\ P''' \rightarrow \mathit{fwd}_\diamond\ P''' \end{array}}_{f_2}$$

For the final formula, we find

$$\mathit{True} \vee (\mathit{True} \vee (\mathit{True} \vee \diamond a)) \vdash_p \ \mathit{inl}\ \ \mathit{True}^{\checkmark}$$

We now transform the final proof into a proof of the initial formula by applying the proof transformers connected to the derivative steps:

$$(f_1 \circ f_2)(\mathit{inl}\ \ \mathit{True}^{\checkmark}) = \mathit{inl}\ (\mathit{fwd}_{\mathit{next}}\ \ a^{\checkmark})$$

The proof on the right is the proof for the original formula $\mathit{next}\ a \vee \diamond a$. This proof is also the left-most shortest proof. This result holds in general.

In a first step, we verify that the proof transformer connected to the derivative computes the proof of the original formula given the proof of the derivative.

Lemma 2 (Derivatives Matching Correctness). *Let* $L \backslash B \vdash_d \ (L' \,\mathbf{|}\, f)$ *and* $T \vdash L' \rightsquigarrow P'$. *Then,* $B : T \vdash L \rightsquigarrow P$ *for some* P *such that* $f\ P' = P$.

Proof. (Sketch) By induction over the structure of L and the derivation $T \vdash L' \rightsquigarrow P'$. For example, consider $L_1\ \mathit{until}\ L_2$. Case (until_d) applies. By assumption we have that $T \vdash \ L_2' \vee (L_1' \wedge (L_1\ \ \mathit{until}\ L_2)) \rightsquigarrow P''$. For brevity, we only consider the case $P'' = \mathit{inl}\ P'$. Thus, we conclude that $T \vdash \ L_2' \rightsquigarrow P'$ (1). From the premise of case (until_d), we conclude $L_2 \backslash B \vdash_d \ (L_2', f_2)$ (2). By induction hypothesis applied to (1) and (2) we conclude that $B : T \vdash \ L_2 \rightsquigarrow f_2\ P'$. Via rule rule $(\mathit{until}\text{-}1)$ we conclude that $B : T \vdash \ L_1\ \mathit{until}\ L_2 \rightsquigarrow []\ \mathit{until}_{\mathit{prf}}\ f_2\ P'$. By construction we find that $f\ \mathit{inl}\ P' = []\ \mathit{until}_{\mathit{prf}}\ f_2\ P'$ and thus we are done.

The other cases can be proven similarly.

The following result follows immediately by construction.

Lemma 3 (Correctness of Final Left-Most Proof). *Let $L \vdash_p P$. Then $[] \vdash L \rightsquigarrow P$ and P is the left-most shortest proof w.r.t. $[]$ and L.*

The composition of the individual proof transformers clearly yields a valid proof of the original formula. We further know that the final proof is the left-most shortest proof. The important observation is that the derivatives matching step $L \backslash B \vdash_d (L' \mid f)$ preserves left-most shortest proofs. That is, if the proof P' of L' is left-most shortest, then it follows that proof $f\ P'$ of L is also left-most shortest. Thus, we obtain the following result.

Theorem 1 (Left-Most Shortest Derivatives Matching Correctness). *Let $T \vdash_d L \rightsquigarrow P$ for some trace T, LTL formula L and proof P. Then, $T \vdash L \rightsquigarrow P$ and P is the left-most shortest proof w.r.t. T and L.*

Computing the Shortest Left-Most Match. We are now interested in the *shortest* match. There are several adjustments we need to make to the derivative-based matching algorithm:

- (1) We must aggressively simplify formulas by using Boolean laws such as $L \vee \mathit{True} = L$. Thus, we favor formulas which evaluate as early as possible to *True* and the resulting proofs are shorter.
- (2) The simplifications must be applied in intermediate derivative matching steps.
- (3) We currently built the left-most shortest final proof. Here, we need some additional rules to guarantee that we built the shortest left-most final proof.

To motivate (1) and (2) we consider formula $\mathit{next}\ a \vee a$ and trace $[a, a]$. For brevity, we only consider the resulting derivatives which are:

$$(\mathit{next}\ a \vee a)\backslash a = a \vee \mathit{True} \qquad (a \vee \mathit{True})\backslash a = \mathit{True} \vee \mathit{True}$$

From $\mathit{True} \vee \mathit{True}$ we obtain the final proof $\mathit{inl}\ \mathit{True}^{\surd}$. Application of the proof transformers connected to derivatives then leads to $\mathit{inl}\ \mathit{fwd}_{\mathit{next}}\ a^{\surd}$. This is the left-most shortest proof but clearly not the shortest left-most proof which is $\mathit{inr}\ a^{\surd}$.

To obtain the shortest proof we must apply simplifications also in intermediate steps. For our example, in the first derivative matching step we simplify $a \vee \mathit{True}$ to *True*. The subsequent derivative step $\mathit{True}\backslash a = \mathit{True}$ then yields the final formula *True* which has the final proof $\mathit{True}^{\surd}$. Application of the proof transformers connected to the derivative matching and simplification step then leads to $\mathit{inr}\ a^{\surd}$. This is the shortest left-most proof we were looking for.

Next, we provide the details of the simplification step in terms of judgments $L \vdash_s (L \mid P \to P)$. Similar to the derivative matching step, each simplification step yields a proof transformer which builds a proof of the original formula given a proof of the simplified formula. The simplification rules are specified in Figures 5 and 6.

Figure 5 contains the standard Boolean simplifications concerning $\vee$ etc. In the LTL context, (Boolean) simplification also need to be applied 'below' LTL

$$\boxed{L \vdash_s (L \mid P \to P)}$$

$(\mathrm{T}_s)\quad \mathit{True} \vdash_s (\mathit{True} \mid \lambda P.P) \qquad (\mathrm{F}_s)\quad \mathit{False} \vdash_s (\mathit{False} \mid \bot) \qquad (\mathrm{B}_s)\quad B \vdash_s (B \mid \lambda P.P)$

$(\vee\text{-}1_s)\quad (\mathit{True} \vee L) \vdash_s (\mathit{True} \mid \lambda P.\mathit{inl}\ P) \qquad (\vee\text{-}2_s)\quad (L \vee \mathit{True}) \vdash_s (\mathit{True} \mid \lambda P.\mathit{inr}\ P)$

$(\vee\text{-}3_s)\quad (\mathit{False} \vee L) \vdash_s (L \mid \lambda P.\mathit{inr}\ P) \qquad (\vee\text{-}4_s)\quad (L \vee \mathit{False}) \vdash_s (L \mid \lambda P.\mathit{inl}\ P)$

$$(\vee\text{-}5_s)\quad \frac{\begin{array}{c} L_1 \neq \mathit{False} \text{ and } L_1 \neq \mathit{True} \text{ and } L_2 \neq \mathit{False} \text{ and } L_1 \neq L_2 \\ L_1 \vdash_s (L_1' \mid f_1) \quad L_2 \vdash_s (L_2' \mid f_2) \\ \begin{array}{l} f = \lambda P.\ \mathsf{case}\ P\ \mathsf{of} \\ \qquad \mathit{inl}\ P' \to \mathit{inl}\ (f_1\ P') \\ \qquad \mathit{inr}\ P' \to \mathit{inr}\ (f_2\ P') \end{array} \end{array}}{(L_1 \vee L_2) \vdash_s ((L_1' \vee L_2') \mid f)}$$

$$(\vee\text{-}6_s)\quad \frac{L \vdash_s (L' \mid f') \quad f = \lambda P.\mathit{inl}\ (f'\ P)}{(L \vee L) \vdash_s (L' \mid f)}$$

$(\wedge\text{-}1_s)\ (\mathit{True} \wedge L) \vdash_s (L \mid \lambda P.(\mathit{True}^{\checkmark}, P)) \quad (\wedge\text{-}2_s)\ (L \wedge \mathit{True}) \vdash_s (L \mid \lambda P.(P, \mathit{True}^{\checkmark}))$

$(\wedge\text{-}3_s)\quad (\mathit{False} \wedge L) \vdash_s (\mathit{False} \mid \bot) \qquad (\wedge\text{-}4_s)\quad (L \wedge \mathit{False}) \vdash_s (\mathit{False} \mid \bot)$

$$(\wedge\text{-}5_s)\quad \frac{\begin{array}{c} L_1 \neq \mathit{True} \text{ and } L_1 \neq \mathit{False} \text{ and } L_2 \neq \mathit{True} \text{ and } L_2 \neq \mathit{False} \text{ and } L_1 \neq L_2 \\ L_1 \vdash_s (L_1' \mid f_1) \quad L_2 \vdash_s (L_2' \mid f_2) \end{array}}{(L_1 \wedge L_2) \vdash_s ((L_1' \wedge L_2') \mid \lambda(P_1, P_2).(f_1\ P_1, f_2\ P_2))}$$

$$(\wedge\text{-}6_s)\quad \frac{L \vdash_s (L' \mid f') \quad f = \lambda P.(f'\ P, f'\ P)}{(L \wedge L) \vdash_s (L' \mid f)}$$

Fig. 5. Simplifications and Proof Transformers I

operators. For example, consider *next* $(a \vee \mathit{True})$ which shall be simplified to *next True*. For such simplifications, we apply the rules in Figure 6.

In rule $(\diamond_s)$, we make use of the short-hand notation $\mathit{fwd}_\diamond{}^n(P)$:

$$\mathit{fwd}_\diamond{}^0(P) = P \qquad \mathit{fwd}_\diamond{}^{n+1}(P) = \mathit{fwd}_\diamond\ (\mathit{fwd}_\diamond{}^n(P))$$

The proof transformation function f in this rule distinguishes between the case that a proof for L could be found, resp. the trace ended prematurely. In the first case, we follow the chain of $\mathit{fwd}_\diamond$ steps until we reach $\mathit{stop}_\diamond\ P$ which is then replaced by $\mathit{stop}_\diamond\ (f'\ P)$. In case the trace ended, represented by some proof $L''^{?}$, we use the original (non-simplified) formula L to represent the proof $\mathit{fwd}_\diamond{}^n(L^?)$ for $\diamond L$.

In rule $(\Box_s)$, we apply the proof transformer f' to each of the sub-proofs. The exception is in case of a sub-proof of the form $^?$. This must be the last sub-proof. Like in case of rule $(\diamond_s)$, we use the original (non-simplified) formula

Helper: $\quad adj\ f\ L = \lambda P.\mathsf{case}\ P\ \mathsf{of}\ \begin{array}{l}(L'')^? \to L^? \\ P' \to f\ P'\end{array}$

$$(\diamond_s)\quad \frac{L \vdash_s (L' \mid f') \qquad \begin{array}{l} f = \lambda P.\ \mathsf{case}\ P\ \mathsf{of} \\ \quad fwd_\diamond{}^n(stop_\diamond\ P) \to fwd_\diamond{}^n(stop_\diamond\ (f'\ P)) \\ \quad fwd_\diamond{}^n((L'')^?) \to fwd_\diamond{}^n(L^?) \end{array}}{(\diamond L) \vdash_s ((\diamond L') \mid f)}$$

$$(\Box_s)\quad \frac{L \vdash_s (L' \mid f') \quad f'' = adj\ f'\ L}{(\Box L) \vdash_s ((\Box L') \mid \lambda\Box\ [P_1, ..., P_n].\Box[f''\ P_1, ..., f''\ P_n])}$$

$$(until_s)\quad \frac{L_1 \vdash_s (L_1' \mid f_1) \quad L_2 \vdash_s (L_2' \mid f_2)}{(L_1\ until\ L_2) \vdash_s \left((L_1'\ until\ L_2') \,\Bigg|\, \begin{array}{l}\lambda[P_1, ..., P_n]\ until_{prf}\ P. \\ {[f_1\ P_1, ..., f_1\ P_n]}\ until_{prf}\ ((adj\ f_2)\ P_2)\end{array}\right)}$$

$$(next_s)\quad \frac{L \vdash_s (L' \mid f)}{(next\ L) \vdash_s ((next\ L') \mid \lambda fwd_{next}\ \ P.(adj\ f\ L)\ P)}$$

Fig. 6. Simplifications and Proof Transformers II

L to represent the last sub-proof of $\Box L$. For brevity, we make use of the helper function $adj\ f\ L$ to either apply f or simply return $L^?$. This helper function is also used in rules $(until_s)$ and $(next_s)$.

We always assume that simplification rules are applied aggressively by traversing an LTL formula from top to bottom and from left to right. Then, the following result follows.

Lemma 4 (Simplification Correctness and Preservation of Shortest Left-Most). *Let $L \vdash_s (L' \mid f)$ and $T \vdash L' \rightsquigarrow P'$ such that P' is the shortest left-most proof w.r.t. T and L'. Then, $T \vdash L \rightsquigarrow P$ for some P such that $f\ P' = P$ and P is the shortest left-most proof w.r.t. T and L.*

We yet need to address (3) from above. For example, consider formula $next\ (next\ a) \lor next\ b$ and trace $[c]$. The final formula is $next\ a \lor b$. The current final proof construction algorithm in Figure 4 yields $inl\ (next\ a)^?$ but the shortest final proof is $inr\ a^?$.

Hence, we extend Figure 4 with two additional rules. We write $L \vdash_{p_s} P$ to denote proof construction form formulas using the extended set of rules.

$$(\lor\text{-L}^?{}_p)\quad \frac{\begin{array}{c}L_1 \vdash_{p_s} L_1'^? \quad L_2 \vdash_{p_s} L_2'^? \\ size(L_1'^?) \leq size(L_2'^?)\end{array}}{(L_1 \lor L_2) \vdash_{p_s} inl\ L_1'^?} \qquad (\lor\text{-R}^?{}_p)\quad \frac{\begin{array}{c}L_1 \vdash_{p_s} L_1'^? \quad L_2 \vdash_{p_s} L_2'^? \\ size(L_2'^?) < size(L_1'^?)\end{array}}{(L_1 \lor L_2) \vdash_{p_s} inr\ L_2'^?}$$

The above rules apply if both branches of a 'choice' formula are unmatched. Otherwise, we will apply the existing rules ($\lor$-Left$_p$) and ($\lor$-Right$_p$). Thus, $(next\ a \lor b) \vdash_{p_s} inr\ a^?$.

Lemma 5 (Correctness of Final Shortest Proof). *Let $L' \vdash_s (L \mid f)$ and $L \vdash_{p_s} P$. Then $[] \vdash L \rightsquigarrow P$ and P is the shortest left-most proof w.r.t. $[]$ and L.*

The definition of the shortest-left most match algorithm follows addressing the above points (1-3).

Definition 6 (Shortest Left-Most Match Algorithm). *Let L be an LTL formula, T be a finite trace of the form $[B_1, ..., B_n]$ and P be a proof. We define $T \vdash_{d_{slm}} L \rightsquigarrow P$ iff*

– *$L \vdash_s (L' \mid f')$, $L \backslash B_1 \vdash_d (L_1 \mid f_1)$,*
  $L_1 \vdash_s (L'_1 \mid f'_1)$, $L'_1 \backslash B_2 \vdash_d (L_2 \mid f_2)$,
  ...,
  $L_{n-1} \vdash_s (L'_{n-1} \mid f'_{n-1})$, $L'_{n-1} \backslash B_n \vdash_d (L_n \mid f_n)$,
  for some $L', L_1, L'_1 ..., L_n$ and $f', f_1, f'_1 ..., f_n$, and

– *$L_n \vdash_s (L'_n \mid f'_n)$, $L'_n \vdash_{p_s} P'$ for some P', L'_n, f'_n, and*

– *$P = (f_1 \circ f'_1 \circ \; ... \circ \; f_n \circ f'_n) \; P'$.*

Theorem 2 (Computing the Shortest Left-Most Proof). *Let $T \vdash_{d_{slm}} L \rightsquigarrow P$ for some trace T, LTL formula L and proof P. Then, we have that $T \vdash L \rightsquigarrow P$ and P is the shortest left-most proof w.r.t. T and L.*

The above result provides the basis for checking coverage of a set of requirements expressed as LTL formulas.

4 Checking LTL Coverage by Inspecting Proofs

We repeat the *unique first cause* (UFC) coverage condition proposed in [7]: A test suite achieves UFC coverage of a set of requirements expressed as temporal formulas, if: (1) every basic condition in any formula has taken on all possible outcomes at least once and (2) each basic condition has been shown to affect the formula's outcome as the unique first cause. A condition a is the unique first cause (UFC) for ϕ along a path π if, in the first state along π in which ϕ is satisfied, it is satisfied because of a.

Condition (1) essentially corresponds to the MC/DC coverage criteria. In our formulation, we ignore this level of detail here because we only consider atomic propositions at the Boolean propositional level.

The important point is that condition (2) can be characterized precisely in terms of shortest left-most proofs. Roughly, conditions a in some test property L must be covered by some shortest left-most proof P. That is, $a^{\surd}$ in P. To unambiguously distinguish among several occurrences of a, e.g. as in $a \vee a$, we attach distinct labels k to conditions a, written a_k. For example, $a_1 \vee a_2$. Thus, we can re-phrase the unique first cause coverage condition as follows.

Definition 7 (Unique First Cause Coverage Revisited Condition). *A test suite is a set $\{T_1, ..., T_n\}$ of traces and a set $\{L_1, ..., L_m\}$ of LTL test properties.*

We say that a test suite satisfies the unique first cause coverage revisited condition *iff for all test properties L_i and for all atomic condition a_k in L_i we find some trace T_j such that $T_j \vdash L_i \rightsquigarrow P$ for some P where P is the shortest left-most proof and $a_k^{\surd}$ is in P.*

Based on Theorem 2 it immediately follows that the Unique First Cause Coverage Revisited Condition is checkable.

5 Related Work and Conclusion

There are various prior works which study finite trace matching algorithms, e.g. see [3, 5], and the design space of the semantics of finite trace LTL matching, e.g. see [2]. To the best of our knowledge, we are the first to study constructive finite trace matching. Such a matching approach has several advantages as discussed in the introduction.

Of particular interest is the application of checking coverage of LTL test properties. Our focus here is the UFC coverage condition introduced in [7]. We can give a precise definition of the UFC condition in terms of shortest left-most proofs and thus we can easily check if a test suite satisfies the UFC condition.

The LTL matching and coverage approach as described has been fully implemented and is in actual use in some mission-critical embedded system applications. We check coverage of LTL properties w.r.t. manually written test cases. As our implementation language we use Haskell which fits very well the rewriting nature of our matching algorithms. We incorporate several optimizations such as hash consing for efficient comparison etc. Haskell's lazy evaluation strategy is of advantage in case of larger formulas with short proofs. Thanks to laziness we only need to evaluate the necessary parts. Due to space constraints, we postpone a more detailed description of our implementation and experiences from several industrial case studies to some future work.

Another interesting topic is the issue of providing sensible explanation whys a trace does not match the formula. Currently, we simply return the *first* failure position in the trace and the formula. We believe that often there can be better, e.g. *shortest*, explanations. This is something we will pursue in future work.

Acknowledgements

We thank the reviewers for their comments.

References

1. J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.
2. 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.
3. 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.
4. Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.
5. Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engg.*, 12:151–197, 2005.
6. RTCA/DO-178B. Software considerations in airborne systems and equipment certification, 1992.
7. 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.