Parse Tree Structure in LTL Requirements Diagnosis

Ingo Pill*, Thomas Quaritsch†, Franz Wotawa*

*Institute for Software Technology, TU Graz, Inffeldgasse 16b/II, 8010 Graz, Austria
†HTL Pinkafeld, Meierhofplatz 1, 7423 Pinkafeld, Austria

This author was with the Institute for Software Technology when contributing

Accepted for publication by IEEE, copyrights by IEEE - link to final version will be added when available

Abstract—Automated assistance in ensuring a product’s reliability and functional correctness is certainly a powerful asset, but also requires us to express our expectations in a formal way as accessible to our algorithms and tools. In recent work, we showed for specifications in Pnueli’s “Temporal Logic of Programs” LTL how to diagnose such a specification if we find that it does not catch our intent, i.e., when some behavior expected to satisfy it actually violates it (and vice versa). In this paper, we show how to improve this process in that we exploit structural data in the form of a specification’s parse tree for our diagnostic reasoning. We discuss the achieved effects for the example of the well-known model-based diagnosis algorithm HS-DAG and report on corresponding experimental results that show our reasoning’s attractiveness.

I. INTRODUCTION

Automated reasoning that assists us in getting our products right is certainly an important asset for minimizing development times and achieving the desired quality for our increasingly complex solutions. While a product’s quality is certainly crucial for nuclear applications or space exploration projects, also widely available products like cars demand for the detailed consideration of quality aspects, e.g., related to safety concerns. Both, customers and public authorities, the latter via standards like IEC 61508¹ and its adaptation ISO 26262² (Automotive Safety Integrity Level (ASIL) as used in the automotive industry), require us to deliver our best in terms of quality.

The adoption of automated reasoning, however, requires us to express our expectations in a precise, formal manner as accessible by our algorithms and tools. Such formal requirements (specifications) then may drive a system’s creation, testing, verification, and potentially even its synthesis. The demand for means to get our requirements perfectly right is thus obvious, but in industry, still up to 50% of product defects, and up to 80% of rework efforts (that can consume up to 50% of total development costs) can be traced back to requirement defects [1], [2]. So how to verify and debug formal specifications then?

Complementing research on specification coverage and vacuity [3], [4], formal requirements analysis [5] and corresponding tools like RAT* [6] offer development and investigation workflows that support verifying invariants or corner case behavior with a specification, as well as support designers in simulating and inspecting a specification’s detailed behavior. The step of locating a fault(s) is however still manual. In [7] we proposed for requirements in Pnueli’s “Temporal Logic of Programs” (Linear Temporal Logic) LTL [8] a model-based diagnosis approach for scenarios where we face some behavior unexpectedly satisfying (or contradicting) the requirements. As explained in more detail in the Preliminaries (Section II), this means isolating those specific requirements parts (in the form of operator occurrences in an LTL specification’s parse tree) that, if we assume that we should have specified something different in their stead, can explain the encountered issue. An immediate advantage is that these diagnoses are at specification level, i.e., the various operators and formulae a designer used to write her specification, rather than reporting possibly missing or faulty edges and states in an automaton or SAT encoding as usually derived for computing with LTL specifications.

For instance, let us briefly consider an automatic parking brake (aps) of a modern car. Obviously, under no circumstances shall it release the wheels without the user requesting it. One safety requirement then could be that if the car is not moving, the wheels should be and stay blocked until the driver either pushes the gas pedal in order to drive, or pushes the break pedal gaining manual control over the brakes. In a logic-like style this could read like always ((aps_active and stop) \(\rightarrow\) (brake until (gas_pedal or brake_pedal))). For LTL, we should use the syntactic sugar "weak until" \(W\) instead of the core operator "until" \(U\) (see Section II) to actually specify this requirement, since using \(U\) would require one of the pedals to be pushed eventually - which is not our intent since we could also turn the car off for completing a scenario. When using "U", for a violating scenario where the fault is visible, our approach described in [7] would offer then a diagnosis containing this very operator.

In Section III, we discuss how a specification’s structure and subformulae can affect the search space for diagnoses as mentioned above, and propose ways how to actually exploit our findings dynamically in an actual search for the effect of minimizing consumed resources.

Our reasoning is general enough to be adopted for various diagnosis algorithms (see [9] for a comparison of some relevant algorithms) or even manual debugging, where we discuss in Section IV the effects achieved for an
implementation of our ideas for the well-known and still competitive [9] HS-DAG diagnosis algorithm [10], [11]. Our corresponding experimental results show the attractiveness of our reasoning. We conclude in Section V with a brief discussion of the current as well as related work, and give directions of our research agenda.

II. Preliminaries

For our introductions of LTL [8] and a trace, we assume a finite set of atomic propositions AP (about variables/signals etc.) that induces alphabet Σ = 2^AP.

Definition 1. Let AP be a finite set of atomic propositions, and δ as well as φ LTL formulae. Then an LTL formula is defined inductively as follows [8]:

- for any p ∈ AP, p is an LTL formula
- ¬φ, φ ∧ δ, φ ∨ δ, Xφ, and φ U δ are LTL formulae

Definition 2. A parse tree (syntax tree) T(φ) = (V_φ, v_φ, E_φ, l(v ∈ V_φ)) for an LTL formula φ is a directed, vertex-labeled tree, where

- V_φ is the set of vertices s.t. for each subformula ψ in φ there is exactly one vertex (v_ψ) labeled with ψ,
- l(v) is a labeling function for vertices v ∈ V_φ s.t. l(v_ψ) = ψ,
- v_φ ∈ V is the single root vertex,
- and E is T’s set of edges, s.t. for v_ψ1, v_ψ2 ∈ V_φ, e = (v_ψ1, v_ψ2) is in E, iff ψ2 is an operand of ψ1.

LTL is defined in the context of infinite words over some alphabet, where we consider traces defined via explicit finite sequences as returned, e.g., by model-checkers. Such a finite sequence of length k+1 can describe a single infinite word only in the form of a lasso-shape (with a cycle looping back from the last step k to 0 ≤ i ≤ k). The other option would be to consider the sequence as a prefix, such that it refers to the (entire) set of infinite words that extend it.

Definition 3. An infinite trace τ is an infinite word over letters from some alphabet Σ of the form τ = (τ_0τ_1...τ_iτ_{i+1}...τ_k)ω with i, k ∈ N, 0 ≤ i ≤ k, τ_i ∈ Σ for any 0 ≤ i ≤ k, and (...)ω denoting infinite repetition of the corresponding (sub-)sequence. With τ_i, we refer to τ’s suffix starting with τ_i.

With ⊤ denoting logic True and ⊥ denoting logic False, the popular LTL operators δ R σ, F φ, G φ, and δ W σ are actually syntactic sugar for common formulae ¬((¬δ) U (¬σ)), ⊤ U φ, ⊥ R φ, and δ U σ V G δ respectively. While one can certainly deal with these operators via rewriting them, we report their direct semantics in order to enhance, e.g., accessibility of our examples. Indeed, please give special attention to the weak until operator W that (in contrast to the “standard” until operator U) does not require the second operand to become eventually true, and should have been used by the designer for expressing the aps example’s requirement discussed in the introduction.

Definition 4. Given an infinite word (or trace) τ and an LTL formula φ, τ(=τ^ω) satisfies φ, denoted as τ ⊨ φ, under the following conditions

- τ_i ⊨ p iff p ∈ τ_i
- τ_i ⊨ ¬φ iff τ_i ⊭ φ
- τ_i ⊨ δ ∧ σ iff τ_i ⊨ δ and τ_i ⊨ σ
- τ_i ⊨ δ ∨ σ iff τ_i ⊨ δ or τ_i ⊨ σ
- τ_i ⊨ Xφ iff τ_{i+1} ⊨ φ
- τ_i ⊨ δ U σ iff ∃ j ≥ i, τ_j ⊨ σ and ∀ i ≤ m < j, τ_m ⊭ δ
- τ_i ⊨ δ R σ iff ∀ j ≥ i, τ_j ⊨ σ or ∃ i ≤ m < j, τ_m ⊨ δ
- τ_i ⊨ F φ iff ∃ j ≥ i, τ_j ⊨ φ
- τ_i ⊨ G φ iff ∀ j ≥ i, τ_j ⊨ φ
- τ_i ⊨ δ W σ iff τ_i ⊨ σ U σ or τ_i ⊨ G δ

Reiter formalized in [11] a consistency-oriented diagnosis theory for reasoning about defective system components. That is, given a system S’s set of components COMP, assumption predicates AB(c_i) for all c_i ∈ COMP catching whether c_i behaves abnormally, a system description SD defining S’s correct behavior ¬AB(c_i) ⇒ NominalBehavior(c_i), and some actual observations OBS about S’s behavior, S is considered to be at fault iff Θ = SD U OBS U {¬AB(c_i)|c_i ∈ COMP} is unsatisfiable.

In our requirements diagnosis scenario, SD will be a SAT-encoding of a given LTL specification φ, and COMP will be φ’s individual subformulae as of the nodes in φ’s parse tree. OBS will be a trace of all of φ’s signals (atomic propositions). But let us focus on Reiter’s theory first.

A minterm in the assumptions AB(c_i) defines a specific “health” state of the considered system, and a diagnosis Δ is a subset-minimal set of components that if we assume them to be faulty (AB(c_i) = ⊤) makes the observations become consistent with the expected behavior of the system as defined by SD for the resulting health state.

Definition 5. A diagnosis for (SD, COMP, OBS) is a subset-minimal set Δ ⊆ COMP such that SD U OBS ∪ {¬AB(c_i)|c_i ∈ COMP \ Δ} is satisfiable.

If Θ = ⊥, Reiter proposed to compute the set of diagnoses as minimal hitting sets of the sets of conflicting assumptions ¬AB(c_i). That is, if such a set Δ hits all the conflicts, and no Δ’ ⊂ Δ does so, then Δ is a diagnosis. Note that hitting all the minimal conflicts is sufficient (since the non-minimal ones are then hit by default).

Definition 6. A conflict C for (SD, COMP, OBS) is a set C ⊆ COMP such that SD U OBS U {¬AB(c_i)|c_i ∈ C} is unsatisfiable. iff there is no C’ ⊂ C, such that C’ is a conflict, then C is a minimal conflict.

Greiner et al. proposed in [10] an improved/corrected algorithmic variant using a directed acyclic graph (DAG) for steering the search space exploration. Those DAG’s nodes n labeled with ”w” give the diagnoses via their h(n):

Definition 7. (HS-DAG) Let D be a growing node- and edge-labeled DAG with some initial and unlabeled root node n_0. Process unlabeled nodes in D in breadth-first order as
follows, where for some node $n$, $h(n)$ is defined as the set of edge labels on the path in $D$ from root node $n_0$ to node $n$ ($h(n_0) = \emptyset$).

1) (Closing) If there is a node $n'$ such that $h(n') \subset h(n)$, and which is labeled with “$\psi$” ($h(n)$ is a hitting set), then close node $n$. Neither will a label be computed for $n$, nor will any successor nodes generated. Proceed with the next node.

2) If for all $C_i \in CS$: $C_i \cap h(n) \neq \emptyset$, then label $n$ with “$\psi$”. Otherwise label $n$ with some $C_j$: $C_j$ is the first set in CS s.t. $C_j \cap h(n) = \emptyset$.

3) (Pruning) If a priorly unused set $C_i$ was used to label node $n$, attempt to prune $D$. That is, for nodes $n'$ labeled with some $C_j \in CS$ such that $C_i \subset C_j$ do as follows:
   a) Relabel $n'$ with $C_i$. Then, for any $c_i$ in $C_j \setminus C_i$, the edge labeled $c_i$ originating from $n'$ is no longer allowed. The node connected by this edge and all of its descendants are removed, except for those nodes with another ancestor that is not being removed. Note that this step may eliminate the very node $n$ currently being processed.
   b) Interchange the sets $C_j$ and $C_i$ in CS. (Note that this has the same effect as eliminating $C_j$ from CS.)

If $n$ was removed, proceed with the next unlabeled node.

4) If $n$ was labeled with some $C_i \in CS$, generate for each $c_i \in C_i$ a new edge originating in $n$ and labeled with $c_i$. If there is a node $n'$ in $D$ such that $h(n') = h(n) \cup \{c_i\}$, then let the edge labeled $c_i$ point to $n'$. Hence, $n'$ will have more than one parent. Otherwise, generate a new node $m$ as destination for the edge. This new node $m$ will be processed (labeled and expanded) after all new nodes $n_i$ in the same generation as $n$ (s.t. $|h(n_i)| = |h(n)|$) have been processed.

5) If there is no further unlabeled node, return DAG $D$.

While Def. 7 might suggest to compute the set CS of conflicts $C_i$ a priori, CS can be easily computed on-the-fly. That is, we can implement Step 2 via checking the satisfiability of $\Theta = SD \cup OBS \{ \neg AB(c_i) | C_i \in COMP \setminus h(n) \}$. If $\Theta$ is found to be unsatisfiable, and there is no priorly computed conflict not hit by $h(n)$, we retrieve from the utilized solver a new conflict as an (ideally subset-minimal) unsatisfiable core in the predicates $\neg AB(c_i)$ (this will be node $n$’s label). The advantage of computing CS-on-the-fly becomes most evident when we are interested only in diagnoses of limited cardinality (e.g. up to triple faults). The algorithm is exhaustive enough to derive a set of conflicts sufficient for this purpose, but which is most likely only a subset of the complete CS. Consequently it saves on computationally expensive theorem prover calls as needed to derive conflicts. Relying on the same basic theory, today we can choose from a wide selection of diagnosis algorithms that compute diagnoses as of Reiter’s Def. 5 (see [9] for some comparison of selected variants).

### TABLE I. UNFOLDING RATIONALES AND CNF CLAUSES FOR LTL OPERATORS

<table>
<thead>
<tr>
<th>$\varphi$</th>
<th>Unfolding rationales</th>
<th>I Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T/L$</td>
<td>$\phi_i \leftrightarrow T/L$</td>
<td>$\varphi_i$</td>
</tr>
<tr>
<td>$\delta \land \sigma$</td>
<td>$\phi_i \leftrightarrow (\delta \land \sigma)$</td>
<td>$(b_1) \varphi_i \land \sigma$</td>
</tr>
<tr>
<td>$\delta \lor \sigma$</td>
<td>$\phi_i \leftrightarrow (\delta \lor \sigma)$</td>
<td>$(b_2) \varphi_i \lor \sigma$</td>
</tr>
<tr>
<td>$\delta \land \phi_i$</td>
<td>$\phi_i \leftrightarrow (\delta \land \phi_i)$</td>
<td>$(b_3) \varphi_i \land \sigma \lor \phi_i$</td>
</tr>
<tr>
<td>$\delta \lor \phi_i$</td>
<td>$\phi_i \leftrightarrow (\delta \lor \phi_i)$</td>
<td>$(b_4) \varphi_i \lor \sigma \lor \phi_i$</td>
</tr>
<tr>
<td>$\neg \delta \land \phi_i$</td>
<td>$\phi_i \leftrightarrow (\neg \delta \land \phi_i)$</td>
<td>$(c_1) \neg \phi_i \land \sigma \lor \phi_i$</td>
</tr>
<tr>
<td>$\neg \delta \lor \phi_i$</td>
<td>$\phi_i \leftrightarrow (\neg \delta \lor \phi_i)$</td>
<td>$(c_2) \neg \phi_i \lor \sigma \lor \phi_i$</td>
</tr>
<tr>
<td>$\sigma \land \phi_i$</td>
<td>$\phi_i \leftrightarrow (\sigma \land \phi_i)$</td>
<td>$(c_3) \varphi_i \land \sigma \land \phi_i$</td>
</tr>
<tr>
<td>$\sigma \lor \phi_i$</td>
<td>$\phi_i \leftrightarrow (\sigma \lor \phi_i)$</td>
<td>$(c_4) \varphi_i \lor \sigma \land \phi_i$</td>
</tr>
<tr>
<td>$X \delta$</td>
<td>$\phi_i \leftrightarrow \delta_{i+1}$</td>
<td>$(c_5) \varphi_i \land \sigma \land \phi_i$</td>
</tr>
<tr>
<td>$\delta U \sigma$</td>
<td>$\phi_i \rightarrow (\sigma \land (\delta \land \phi_{i+1}))$</td>
<td>$(d_1) \varphi_i \rightarrow \sigma \land \delta \land \phi_{i+1}$</td>
</tr>
<tr>
<td>$\delta W \sigma$</td>
<td>$\phi_i \rightarrow (\sigma \land (\delta \lor \phi_{i+1}))$</td>
<td>$(d_2) \varphi_i \rightarrow \sigma \land \delta \lor \phi_{i+1}$</td>
</tr>
<tr>
<td>$\delta W \sigma$</td>
<td>$\phi_i \rightarrow (\sigma \land (\delta \land \phi_{i+1}))$</td>
<td>$(d_3) \varphi_i \rightarrow \sigma \land \delta \land \phi_{i+1}$</td>
</tr>
<tr>
<td>$\delta W \sigma$</td>
<td>$\phi_i \rightarrow (\sigma \land (\delta \lor \phi_{i+1}))$</td>
<td>$(d_4) \varphi_i \rightarrow \sigma \land \delta \lor \phi_{i+1}$</td>
</tr>
<tr>
<td>$\delta W \sigma$</td>
<td>$\phi_i \rightarrow (\sigma \land (\delta \land \phi_{i+1}))$</td>
<td>$(d_5) \varphi_i \rightarrow \sigma \land \delta \land \phi_{i+1}$</td>
</tr>
<tr>
<td>$\delta W \sigma$</td>
<td>$\phi_i \rightarrow (\sigma \land (\delta \lor \phi_{i+1}))$</td>
<td>$(d_6) \varphi_i \rightarrow \sigma \land \delta \lor \phi_{i+1}$</td>
</tr>
<tr>
<td>$\delta W \sigma$</td>
<td>$\phi_i \rightarrow (\sigma \land (\delta \land \phi_{i+1}))$</td>
<td>$(d_7) \varphi_i \rightarrow \sigma \land \delta \land \phi_{i+1}$</td>
</tr>
<tr>
<td>$\delta W \sigma$</td>
<td>$\phi_i \rightarrow (\sigma \land (\delta \lor \phi_{i+1}))$</td>
<td>$(d_8) \varphi_i \rightarrow \sigma \land \delta \lor \phi_{i+1}$</td>
</tr>
<tr>
<td>$\delta W \sigma$</td>
<td>$\phi_i \rightarrow (\sigma \land (\delta \land \phi_{i+1}))$</td>
<td>$(d_9) \varphi_i \rightarrow \sigma \land \delta \land \phi_{i+1}$</td>
</tr>
<tr>
<td>$\delta W \sigma$</td>
<td>$\phi_i \rightarrow (\sigma \land (\delta \lor \phi_{i+1}))$</td>
<td>$(d_{10}) \varphi_i \rightarrow \sigma \land \delta \lor \phi_{i+1}$</td>
</tr>
</tbody>
</table>

Now let us briefly rehearse how Reiter’s theory was adopted for our approach at LTL specification diagnosis given behavioral samples (traces) [7], where in the next section we show how to improve on it.

**Definition 8.** [7] In the context of a given infinite trace with length $k$ and loop-back time-step $l$, $E_1(\psi)$ encodes an LTL formula $\psi$ using the clauses presented in Table I, where we instantiate for each subformula $\varphi$ a new variable over time, denoted $\varphi_i$ for time instance $i$. Note that we assume $k$ and $l$ to be known inside $E_1$ and $R$.

$$E_1(\varphi) = \begin{cases} R(\varphi) \land E_1(\delta) \land E_1(\sigma) & \text{for } \varphi = \delta \circ_1 \sigma \\ R(\varphi) \land E_1(\delta) & \text{for } \varphi = \circ_2 \delta \\ R(\varphi) & \text{else} \end{cases}$$

with $\circ_1 \in \{\land, \lor, U\}$, $\circ_2 \in \{\neg, X\}$ and $R(\varphi)$ defined as the conjunction of the corresponding clauses in Table I.

**Definition 9.** [7] For a given infinite trace $\tau$ with given $k$, $E_2(\tau) = \bigwedge_{0 \leq i \leq k} \left[ \bigwedge_{p_i \in \tau} \bigwedge_{0 \leq i \leq k} \neg p_i \right]$ encodes the signal values as specified by $\tau$.

**Theorem 1.** [7] Assume an updated Table I, where each clause $c$ is extended to $\delta \land \phi_i \lor c$, and an assignment to all assumptions $op_{\varphi}$ on a specification $\varphi$’s various subformulas $\psi$’s correctness. An encoding $E_{WFM}(\varphi, \tau) = E_1(\varphi) \land E_2(\tau) \land \phi_0$ of an LTL formula $\varphi$ and a trace $\tau$ as of Definitions 8 and 9 is satisfiable, $\text{SAT}(E_{WFM}(\varphi, \tau), \text{iff } \tau \models \varphi$ under assumptions $op_{\varphi}$.
Using Theorem 1, we can derive for unexpected counterexamples or witnesses \( \tau \) via \( E_{WFM}(\varphi, \tau) \) or \( E_{WFM}(\neg \varphi, \tau) \) corresponding diagnoses in a specification \( \varphi \)'s operators / subformulae [7]. Note that multiple occurrences of a syntactic construct are individual subformulæ, that in turn can be at fault individually. We apologize for the typo that slipped into the final version of the cited paper [7], so that \( \tilde{E}_{WFM}(\varphi, \tau) \) was missing \( \wedge \varphi_0 \) in our theorem. We would like to point out that in contrast to [7], Table I reports clauses also for syntactic sugar.

III. A Parse Tree Can Help with the Diagnoses

In this section, we identify domain-related knowledge that can speed up LTL requirements diagnosis orthogonal to advancements of diagnosis algorithms themselves.

When diagnosing digital circuits, their structure can help in this respect [12]. That is, if we consider a circuit’s signal flow from the inputs to the outputs, we can find dominating components that overrule others (this set is called a dominator’s cone) in that a dominator might be closer to the relevant outputs. Focusing on these gates in the diagnosis process first [13], [14], we can exploit this (e.g. restrict the search space), and then afterwards we can “refine” retrieved diagnoses also with the elements in a dominator’s cone. While Pnueli’s “Temporal Logic of Programs” LTL is often used also for describing circuits, a dominator’s cone. While Pnueli’s “Temporal Logic of Programs” LTL is often used also for describing circuits, it is apparent that such cones are actually unsuitable for our requirements diagnosis concept. For example, for the scenario of a specification we have a single relevant output, i.e., the evaluation of specification \( \varphi \) at time step 0 that determines whether the trace satisfies the specification. Thus there would be a single maximal cone that does not help us regarding an optimization.

Nevertheless, as shown in the following, a specification’s parse tree does contain data that can be exploited. The motivating idea is as follows:

**Proposition 1.** If some subformula \( \psi \) from specification \( \varphi \) can resolve an issue, i.e., when assuming \( AB(\psi) \) (\( \overline{\psi} \) in our encoding), then so can all the superformulæ \( \delta \) of \( \psi \) (in that we could also assume \( AB(\delta) / \overline{\psi} \) instead).

**Proof:** (sketch) Informally, from the semantics of LTL operators as of Def. 4 it is apparent that the evaluation of an LTL formula at time step \( i \) depends on its own evaluation in the future (like for \( X \)) and the evaluation of its subformulæ in the current and/or future time steps, like for \( \vee \) or \( U \). This is evident also in our encoding and our clauses as given in Table I. If we assume \( AB(\psi) \) (\( \overline{\psi} \) in our encoding) expressing that we should have written something different in \( \psi \)'s stead, the behavior of \( \psi \) is unknown and left unconstrained. If now, by flipping \( \neg AB(\psi) \) to \( AB(\psi) \), the observations become consistent with \( SD \) under the updated assumptions s.t. \( \varphi_0 \) becomes true, then we can choose the same evaluation of \( \delta \) over time when assuming \( AB(\delta) \) (then the behavior of \( \delta \) is unconstrained) s.t. \( \varphi_0 \) can become true also then.

Formally, this means that for specification \( \varphi \), trace \( \tau \), and \( E_{WFM}(\varphi, \tau) \) as of Theorem 1, we have that if assuming \( \overline{\psi} \) for some subformula \( \psi \) makes \( E_{WFM}(\varphi, \tau) \) satisfiable, so does assuming \( \overline{\psi} \) for any superformula \( \delta \) of \( \psi \). That is, from Theorem 1, Def. 8, Def. 9, and Table I, we know that assuming \( \overline{\psi} \) frees the values \( \psi_0 \) in \( E_{WFM}(\varphi, \tau) \) due to \( \overline{\psi} \) satisfying \( R(\psi) \), i.e., \( \overline{\psi} \vee c \) for all corresponding clauses \( c \) for \( \psi \) in Table I. For \( E_{WFM}(\varphi, \tau) \) becoming true when assuming \( \overline{\psi} \), there is a satisfying assignment for all variables \( \delta \), s.t. \( \delta \) is a superformula of \( \psi \). Consequently, \( E_{WFM}(\varphi, \tau) \) is also satisfiable when assuming \( \overline{\psi} \) (instead) for any superformula \( \delta \) of \( \psi \).

In terms of conflicts as used to characterize diagnosis problems in Reiter’s diagnosis theory (see Sec. II), Proposition 1 can be formulated as follows.

**Corollary 1.** If some subformula \( \psi \) of specification \( \varphi \) can resolve a specific conflict \( C_i \), then so can \( \delta \) such that \( \delta \) is a superformula of \( \psi \) in specification \( \varphi \).

From Cor. 1, the following corollary follows directly.

**Corollary 2.** Some superformula \( \delta \) of subformula \( \psi \) in specification \( \varphi \) can resolve at least those conflicts that \( \psi \) can resolve.

The imminent question now is what does this all mean when considering Reiter’s diagnosis theory that defines and computes diagnoses as the minimal hitting sets of a diagnosis problem’s minimal conflicts?

Hitting a minimal conflict \( C_i \) as of Def. 6 actually resolves the considered conflict since one of the conflicting facts (in our case assumptions \( \neg AB(\psi) \)) is retracted and the subset left is no conflict (since \( C_i \) was a minimal conflict). In general, this is not the case for a non-minimal \( C_i \), since we could hit it with a \( \psi \) that is not part of the minimal conflict(s) contained in \( C_i \) s.t. \( C_i \setminus \psi \) would still be a conflict. Thus, while hitting and resolving is the same for a minimal conflict, it is not for a non-minimal one. Nevertheless, for minimal conflicts we have the following corollaries derived from Corollaries 1 and 2.

**Corollary 3.** If some subformula \( \psi \) of specification \( \varphi \) can hit/resolve a specific minimal conflict \( C_i \), then so can its superformulæ \( \delta \).

**Corollary 4.** Some superformula \( \delta \) of subformula \( \psi \) of specification \( \varphi \) can hit / resolve at least those minimal conflicts that \( \psi \) can resolve.

These observations certainly have an influence on the structure of a diagnosis problem’s minimal conflicts. That is, if a subformula \( \psi \) of specification \( \varphi \) is part of a minimal conflict \( C_i \) (so that \( \psi \) can resolve \( C_i \)), also \( \psi \)'s superformulæ have to be contained in \( C_i \).

**Proposition 2.** If a minimal conflict \( C_i \) contains some subformula \( \psi \), then it contains also all its superformulæ.

**Proof:** (sketch) The validity of this is easy to see. Let us assume that superformula \( \delta \) of \( \psi \) is not in \( C_i \). This would contradict Prop. 1 and Cor. 4, in that \( \delta \) can resolve/hit (at least) the minimal conflicts that \( \psi \) can hit (including \( C_i \)).

In the following, we show how our propositions and the corollaries can speed up a diagnosis process by dynamically...
deriving new “knowledge” from already computed data. That is, for instance, from some diagnosis \( \Delta \), we can derive further diagnoses \( \Delta' \) via the following lemma.

**Lemma 1 (Infer-up).** For a diagnosis \( \Delta = \{\psi_1, \ldots, \psi_n\} \) and some \( \psi_i \in \Delta \) with \( \delta \) a superformula of \( \psi_i \), the set \( \Delta' = (\Delta \setminus \{\psi_i\}) \cup {\delta} \) is a diagnosis as well.

**Proof:** (sketch) According to Prop. 1 / Corrs. 2 and 4, a superformula \( \delta \) of some \( \psi_i \) can resolve at least those conflicts that \( \psi_i \) can resolve (can hit at least the same minimal conflicts as \( \psi_i \)). Thus replacing \( \psi_i \) with \( \delta \) in \( \Delta \) constructs a set that can still resolve all conflicts (hits all the minimal conflicts). In order to derive a diagnosis that per Def. 5 has to be subset-minimal, we have to remove all subformulas of \( \delta \) from \( \Delta \), which, according to Prop. 1 / Corrs. 1 and 3, is not a problem concerning the set of (minimal) conflicts resolved (hit) by \( \Delta / \Delta' \).

This can help in the search space exploration, as approved hypotheses (diagnoses, or in the context of HS-DAG (see Sec. II) consistent sets \( h(n) \)) show the validity of others. For HS-DAG, for instance, we derive in Section IV a corresponding strategy for expanding a node, labeling also a consistent node’s siblings as consistent (with \( \vee^+ \)) if their edge labels refer to superformae of the subformula at hand. For a diagnosis approach using a SAT solver to compute diagnoses directly, like \([14]\), this could save further calls (or internal loop iterations) for computing additional diagnoses (see \([9]\) for an experimental comparison of several relevant diagnosis algorithms).

In the scope of Lemma 1, but thinking downwards in the parse tree, due to Corollary 4 we have that replacing \( \psi_i \in \Delta \) with one of its subformules (or adding it) obviously cannot grow the set of minimal \( \psi_i \)’s hit.

**Corollary 5 (Infer-down).** For some set \( \Delta = \{\psi_1, \ldots, \psi_n\} \) such that \( \Theta = SD \cup OBS \cup \{\neg AB(c_i) | c_i \in COMP \} \) is unsatisfiable, and a subformula \( \delta \) of some \( \psi_i \in \Delta \), we have that \( \Theta' \) for \( \Delta' = (\Delta \setminus \{\psi_i\}) \cup {\delta} \) is unsatisfiable as well.

Considering this corollary, an HS-DAG strategy similar to the one above could be fathomed, inferring the inconsistency of a DAG node. The effects however would be hardly noticeable, due to the conflict cache. That is, a set not hit by \( h(n) = \Delta' \) is certainly among the previously computed ones, i.e., would have been registered previously for \( \Delta \) (otherwise Prop. 2 would be violated). Thus, unlike the one above, this strategy cannot save an expensive theorem prover call. For a direct SAT solver setup, from an abstract point of view, a domain-dependent strategy in the solver could exploit this when internally learning of a conflict.

The following variant of Corollary 5, such that we do not replace \( \psi_i \) by subformula \( \delta \), but would add \( \delta \) to \( \Delta \), while valid for the same reasons, does allow us to prune the search space in terms of subformae of all the \( \psi \in \Delta \).

**Lemma 2 (Prune-down).** For some set \( \Delta = \{\psi_1, \ldots, \psi_n\} \) such that \( SD \cup OBS \cup \{\neg AB(c_i) | c_i \in COMP \} \) is unsatisfiable, and a subformula \( \delta \) of some \( \psi_i \in \Delta \), the set \( \Delta' = \Delta \cup \{\delta\} \) is unsatisfiable. Furthermore, \( \Delta' \) and any of its supersets cannot be a diagnosis.

**Proof:** (sketch) Obviously, \( \Delta' \) hits exactly those minimal conflicts \( C_i \) also hit by \( \Delta \) (according to Prop. 2), so that \( SD \cup OBS \cup \{\neg AB(c_i) | c_i \in COMP \} \setminus \Delta' \) cannot be satisfiable. Furthermore, by Prop. 1 / Corrs. 2 and 4, \( \psi_i \) hits (at least) all the minimal conflicts that \( \delta \) hits, so that one could remove \( \delta \) from \( \Delta' \) and any of its supersets without affecting the set of minimal conflicts hit by them respectively. Thus neither \( \Delta' \), nor any of its supersets can be a diagnosis that, per Def. 5, has to be subset-minimal.

Please note that HS-DAG perfectly implements this reasoning. That is, when retrieving in Step 2 a label for some non-leaf node \( n \), it asks for some \( C_i \) not hit by \( h(n) \). A corresponding \( C_i \) thus cannot contain a subformula of some \( \psi_i \in h(n) \) due to Prop. 2. For a direct diagnosis setup this could be addressed in the process of assigning values to all the variables (we could also add clauses that if \( \psi_i \) is part of a solution then its subformulae should not be).

Interestingly enough, a diagnosis’ definition allows us to apply some parts of this reasoning also for superformae.

**Lemma 3 (Prune-up).** For some \( \Delta = \{\psi_1, \ldots, \psi_n\} \) such that \( SD \cup OBS \cup \{\neg AB(c_i) | c_i \in COMP \} \setminus \Delta \) is unsatisfiable, adding some \( \delta \) \( (\Delta' = \Delta \cup \{\delta\}) \) that is a superformula of some \( \psi_i \in \Delta \) cannot yield a diagnosis.

**Proof:** By Proposition 1 / Corollaries 2 and 4 we know that \( \delta \) hits (at least) all the minimal conflicts that \( \psi_i \) hits, so that one could remove \( \psi_i \) from \( \Delta' \) without affecting the set of conflicts hit. Since diagnoses have to be subset-minimal as of Def. 5, \( \Delta' \) cannot be a diagnosis. Obviously, adding elements to \( \Delta' \) cannot resolve the issue at hand, so that also no superset of \( \Delta' \) can be a diagnosis.

The effect of Lemma 3 is that when some part of a solution is established (e.g. some \( h(n) \) for HS-DAG, or a partial assignment when computing diagnoses directly with a SAT-solver like in \([14]\)), we can remove from further search in this branch all superformae of any \( \psi_i \) in the partial solution (up to the point where we remove \( \psi_i \), e.g., during some back-tracking step in the SAT solver).

Summarizing, Lemmas 1 to 3 allow us to dynamically focus the search for diagnoses via easily derived positive or negative “data”. That is, we showed that, occasionally, we can derive a set \( \Delta' \)’s (in)consistency from that of some other \( \Delta \). Defined by the current context of a partial solution, we can furthermore prune from a branch’s search space formulae \( \psi \in COMP \) that are related to the partial solution via the requirements’ parse tree. In Section IV, we discuss how to implement our reasoning in HS-DAG.

Note that while Proposition 2 refers to minimal conflicts only, it still allows us to prune non-minimal conflicts (sheding some of the non-minimal/unnecessary components). The basic idea behind this is that if for some \( \psi_i \) in conflict \( C_i \) not all of its superformulae are also in \( C_i \), then \( \psi_i \) is obviously not part of a minimal conflict contained in \( C_i \). Thus \( \psi_i \) can be discarded from \( C_i \) since it is enough to hit/resolve the minimal conflicts. Obviously, this is handy if, for one or the other reason, one has to use a theorem prover whose returned conflicts are not always minimal.
Algorithm 1: Pruning conflict sets in HS-DAG.

**Requires:** $n$ — HS-DAG node being expanded

**Requires:** $T$ — parse tree of diagnosed specification $\varphi$

1. 
   **procedure** PRUNEUP($n$, $T$):
   
   1.  **procedure** CLEARMARKS($T$)
   2.  $C \leftarrow \ell(n)$
   3.  **for all** $\psi \in h(n)$
   4.  **while** $\psi \neq \text{NULL} \land \psi$ not marked
   5.  $\psi \leftarrow \text{PARENT}($\psi, $T)$
   6.  $C \leftarrow C \setminus \{\psi\}$
   7.  **mark** $\psi$
   8.  **return** $C$

Lemma 4. For some conflict $C = \{\psi_1, \ldots, \psi_n\}$, its subset $C' = C \setminus \{\psi_i\}$ of any of $\psi_i$’s superformulae $\delta$ is not in $C$ is also a conflict and contains all the essential $\psi_i$ in the minimal conflicts contained in $C$.

**Proof:** (sketch) For a minimal conflict, due to Prop. 2 there is no $\psi_i$ that can be pruned. For a non-minimal one, considering Prop. 2, we do not remove any $\psi_i$ that is part of a minimal conflict contained in $C$. Consequently, $C'$ is a conflict and contains the same minimal conflicts as $C$, s.t. only non-essential data was pruned.

With this filter, e.g., for HS-DAG exploring some branches and the related pruning steps can be avoided (the latter discarding the unnecessary data otherwise).

IV. The Effects Achieved for HS-DAG

While our reasoning helps also in manual debugging, we illustrate in this section the effects for the well-known HS-DAG diagnosis algorithm. Lemmas 1 and 3 were implemented by Algs. 2 and 1, where HS-DAG implements Lemma 2 by construction, as mentioned before. Since the PicoSAT solver used in the setup (see Sec. IV-A) returns minimal conflicts, we did not implement Lemma 4.

Prior to expanding an inconsistent HS-DAG node in Step 4 (see Section II), we call the procedure PRUNEUP (Algorithm 1). It discards from $n$’s intended label the superformulae of all $\psi_i \in h(n)$. Since conflicts may comprise multiple (overlapping) “chains” to the parse tree’s root, we mark those parse tree nodes already examined.

We assume now that HS-DAG expands an inconsistent node by iterating over its label/conflict $C$, considering first those subformulae farthest from $v_T$ in the parse tree $T$. Whenever a node $n$ is found to be consistent (i.e. $h(n)$ is labeled “✓”) in Step 2, we call Algorithm 2 (INFERENCEUP) that under certain conditions then labels siblings $n'$ with “✓” whose incoming edges are labeled with superformulae (of $h(n)$’s last edge’s label). In lines 7 to 8, we make the corresponding subset-check whether there is a subset in $h(n')$ that is a diagnosis, such that $n'$ should be closed. Again, we mark those parse tree nodes already considered.

Our first example for visualizing the effects was adopted from [5], and we diagnosed it also in [7]. It features a two line arbiter with request lines $r_1$ and $r_2$ and the corresponding grant lines $g_1$ and $g_2$. Its specification consists of the following four requirements: $R_1$ demanding that requests on both lines must be granted eventually, $R_2$ ensuring that no simultaneous grants are given, $R_3$ ruling out any initial grant before a request, and, finally, the faulty $R_4$ that aims at preventing additional grants until there are new incoming requests.

- $R_1: \forall i \in \{1, 2\}: G(r_i) \rightarrow F g_i$
- $R_2: G(\neg g_1 \land g_2)$
- $R_3: \forall i \in \{1, 2\}: (\neg g_i \lor r_i)$
- $R_4: \forall i \in \{1, 2\}: G(g_i) \rightarrow X (\neg g_i \lor r_i)$

Testing her specification, a designer defines an unexpected error in the HS-DAG of the following four requirements:

Algorithm 2: Inferring further diagnoses in HS-DAG.

**Requires:** $n$ — consistent HS-DAG node

**Requires:** $T$ — parse tree of diagnosed specification $\varphi$

**Requires:** $\psi$ — subformula that led to $n$

1. **procedure** INFERENCEUP($n$, $T$, $\psi$):
   2. $C \leftarrow \ell($\text{PARENT}($n$)) \triangleright parent node’s conflict
   3. $\delta \leftarrow \text{PARENT}($\psi, $T$) \triangleright parent in the parse tree
   4. **while** $\delta \neq \text{NULL}$
   5. if $\delta$ is not marked $\land \delta \in C$ then
   6. $n' \leftarrow \text{GET PSibling}(h(n) \setminus \psi) \cup \{\delta\}$
   7. if $\exists m$ s.t. $h(m) \subseteq h(n') \land \ell(m) = \checkmark$ then
   8. $\ell(n') \leftarrow \checkmark$
   9. else
   10. $\ell(n') \leftarrow \checkmark$
   11. **mark** $\delta$
   12. **else**
   13. **break**
   14. $\delta \leftarrow \text{PARENT}($\delta, $T$)

Testing her specification, a designer defines an unexpected failing witness, i.e., a trace that should satisfy the specification but violates it. This trace $\tau = \tau_0\tau_1(\downarrow)$ features consecutive (and instantly granted) single requests for both request/grant line pairs s.t. $\tau_0 = r_1 \land g_1 \land \neg r_2 \land \neg g_2$ and $\tau_1 = \neg r_1 \land \neg g_1 \land r_2 \land g_2$.

Similar to the parking brake example in the introduction, and as pointed out in [5], the problem is the until operator $\neg g_i \lor r_i$ in requirement $R_4$ that should be replaced by its weak version $\neg g_i \lor W r_i$. While the idea of both operators is that $\neg g_i$ should hold until $r_i$ holds, the weak version does not require $r_i$ to hold eventually, while the “strong” one does. Thus, $R_4$ in its current form repeatedly requires requests that are not provided by $\tau$, and which is presumably not in the designer’s intent.

In Table II, we can see the effects achieved with our reasoning. Our old approach [7], when using HS-DAG, obtained for this scenario 31 diagnoses, including the one pinpointing to wrong until operators in both instances of $R_4$. It issued 34 theorem prover (SAT solver) calls in total, when building its DAG with 44 nodes. Activating PRUNEUP, the number of nodes constructed by HS-DAG could be reduced from 44 to 38 (some minimum number of nodes is needed to represent the 31 diagnoses) at a negligible (<1%) run-time penalty. For the on-the-fly run with INFERUP activated, the number of consistency checks (Step 2) that require a theorem prover call could be cut down from 31 to 13 (~58%) since we could infer 18 diagnoses. This resulted in a run-time reduction (over 100
TABLE II. HS-DAG STATISTICS FOR THE ARBITER EXAMPLE USING NO/PRUNEUp/INFERUp/PRUNEUp+INFERUp OPTIMIZATION

<table>
<thead>
<tr>
<th></th>
<th>NO</th>
<th>PRUNEUp</th>
<th>INFERUp</th>
<th>PRUNEUp+INFERUp</th>
</tr>
</thead>
<tbody>
<tr>
<td># HS-DAG nodes</td>
<td>44</td>
<td>38</td>
<td>44</td>
<td>38</td>
</tr>
<tr>
<td># TP consistency checks</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td># TP conflict computations</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td># inferred diagnoses</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td># diagnoses</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>run-time (sec.)</td>
<td>0.3801</td>
<td>0.3821</td>
<td>0.2029</td>
<td>0.2033</td>
</tr>
</tbody>
</table>

(a) HS-DAG tree, inferred nodes shaded, pruned “edges” dashed. (b) Parse tree, the two conflicts are shaded/double lined.

Fig. 1. HS-DAG run for the arbiter formula.

runs) of more than 46%, even for this simple example. Using INFERUp and PRUNEUp aggregates the advantages, offering an attractive run-time as well as fewest nodes.

In Fig. 1, we show the created DAG for an even smaller example. Since PRUNEUp only affects DAG levels with \(|h(n)| > 1\), considering a single line only, we purposefully injected a second fault in \(R_4\) by replacing (in the rewritten implication) the logic OR with a logic AND, resulting in the formula \(\varphi = G(\neg g_1 \land X(\neg g_1 U r_1))\). The considered trace consisted of a single request and grant at time step 0: \(\tau = r_0 (\bot)^w\) with \(r_0 = r_1 \land g_1\). In Figure 1, we depict that specification \(\varphi\)’s parse tree as well as the derived conflicts on the right, and the example’s resulting DAG on the left. This DAG is constructed as follows: For each \(\psi_i\) in the root node’s labeling conflict \(\{\psi_2, \psi_8, \psi_9\}\), an edge and destination node is constructed, and we start processing for the nodes \(n\) with \(h(n) = \{\psi_2\}\) (inconsistent) and \(h(n) = \{\psi_8\}\) (consistent). Then, we deduce with INFERUp that \(h(n) = \{\psi_9\}\) is consistent as well (\(\psi_9\) is a superformula of \(\psi_8\)). From the inconsistent node’s labeling conflict \(\{\psi_4, \psi_5, \psi_6, \psi_7\}\), we skip (PRUNEUp) \(\psi_6\) and \(\psi_7\) in the expansion (being superformulae of some \(\psi \in h(n) = \{\psi_2\}\)). Then, after finding that \(h(n) = \{\psi_2, \psi_6\}\) is consistent, we deduce with INFERUp that \(h(n) = \{\psi_2, \psi_7\}\) is consistent.

A. Experimental results

We adopted our Python (CPython 2.7.1) HS-DAG implementation that we used in [7], and ran our tests on an early 2011 MacBook Pro (Intel Core i5 2.3GHz, 4GB RAM, SSD) with Mac OS X 10.6, the GUI and swapping disabled, and using a RAM-drive for the file system. As test samples, we generated random LTL formulae as suggested in [15]. That is, with \(N = |\varphi|/3\) variables and a uniform distribution of LTL operators. We injected triple faults in order to derive \(\varphi_m\) from \(\varphi\), and using our LTL encoding, we derived some trace \(\tau (k = 100, l = 50)\) via solving \(\tau \land \varphi \land \neg \varphi_m\). Verifying whether the set of injected faults is indeed a diagnosis for \(\varphi\) and \(\tau\) ensured that the injected faults do not mask each other in the context of \(\tau\).

For the results in Fig. 2, we generated 10 random diagnosis problems (as outlined above) for any \(|\varphi|\) in \([50, 100, \ldots, 300]\), ran HS-DAG ten times with diagnosis cardinality limits of 1, 2 and 3 (single, double and triple faults) with our various optimizations applied, and plotted average values. For the single fault diagnosis runs (solid lines), we observe a run-time reduction of up to approx. 60% due to INFERUp. The run-time benefit diminishes with rising maximum diagnosis cardinality, when, intuitively, the number of diagnoses (and thus inferable nodes) grows slower than the total number of DAG nodes. While PRUNEUp shows virtually no influence on the run-time, Figure 2b depicts its impact on the number of DAG nodes constructed for a specific problem. Growing with rising maximum diagnosis cardinality, a reduction of up to 23% was possible for \(|\varphi| = 200\) and \(|\Delta| \leq 3\).

Summarizing, while PRUNEUp could achieve a significant DAG node reduction for large diagnosis cardinalities (i.e., in unbounded runs), INFERUp could substantially reduce run-times for the more practical, low-bound case.

V. Discussion and Conclusions

In this paper, we derived several structural insights about requirements expressed in Pnueli’s “Temporal Logic of Programs” LTL [8] that allow us to improve on our diagnostic reasoning [7] for LTL specifications when we need an explanation in the form of possibly faulty specification parts (subformulae) for some unexpected issues as sketched in the introduction.

With us focusing on LTL requirements diagnosis, similar ideas have been exploited previously for other domains. For example, in the context of circuit diagnosis, the concepts of dominators and cones are exploited for circuit abstraction and a diagnosis speed up. Originating in the field of program analysis using control flow graphs [16], [17] and later adopted for the analysis of digital circuits [12], a dominating component can “overrule” the dominated ones (referred to as its corresponding cone) because, e.g., it is “closer to the output”. As dominators for an arbitrary graph structure can be computed in linear time [18], approaches such as [13], [14] focus their diagnostic search on those gates first. The resulting top level diagnoses are then refined by creating further potential diagnoses with dominators replaced by gates from their cone.

While cones are not directly exploitable for our requirements diagnosis concept (remember that we would get a single (maximal) cone if applied to a static LTL parse tree or raise complexity unnecessarily when temporally unfolding it) [19], [20] pursue the notion of (reverse) dominance for their SAT-based RTL debugging, resulting in implied (non-)solutions. We showed that for consistency-based diagnosis, when using HS-DAG, we can achieve a speed-up of about factor two also for our problem domain, using implied (inferred) solutions. On the other hand,
we showed that the implication of non-solutions does not speed up HS-DAG’s diagnosis process when using a conflict set cache. Instead, we could optimize HS-DAG’s search strategy in the context of a domination relation by pruning the conflicts depending on the current tree context. The latter resulted in up to 23% fewer DAG nodes in our tests.

We expect our reasoning to be attractive also for similar (temporal) formula-based description formats, like the Property Specification Language (PSL [21]) that extends LTL, which we will investigate in future work. While we reported run-time effects achieved for HS-DAG, our findings do not depend on a specific diagnosis algorithm. Implementations for further diagnosis approaches (other than HS-DAG) would allow us quantify the achieved effects also for, e.g., approaches like [14], [9] that derive diagnoses directly from a solver, and will require us to define specific interfaces to deliver our findings to a solving engine.

ACKNOWLEDGMENT

The work reported in this paper was supported by the European Commission under FP-7 agreement number 608770 (project “eDAS”) and is based on earlier work funded by the Austrian Science Fund (FWF): P22959-N23 (“MoDiaForTeD”). The authors would like to thank the anonymous reviewers of the DX workshop for their valuable comments on a prior version of this paper [22].

REFERENCES