Model-based Fault Classification for Automotive Software

Mike Becker[1], Roland Meyer[1], Tobias Runge[1,2],
Ina Schaefer[1,2], Sören van der Wall[1], and Sebastian Wolff[3(✉)]

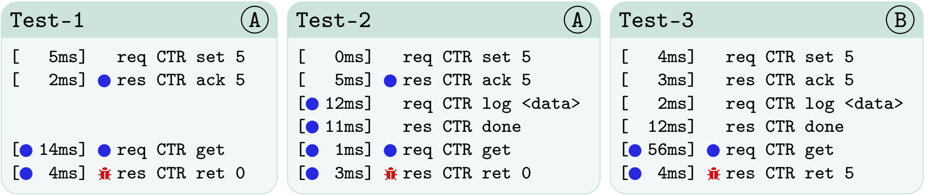
[1] TU Braunschweig, Germany,
{mike.becker,roland.meyer,s.van-der-wall}@tu-bs.de
[2] KIT Karlsruhe, Germany, {tobias.runge,ina.schaefer}@kit.edu
[3] New York University, USA, sebastian.wolff@nyu.edu

Abstract Intensive testing using model-based approaches is the standard way of demonstrating the correctness of automotive software. Unfortunately, state-of-the-art techniques leave a crucial and labor intensive task to the test engineer: identifying bugs in failing tests. Our contribution is a model-based classification algorithm for failing tests that assists the engineer when identifying bugs. It consists of three steps. (i) Fault localization replays the test on the model to identify the moment when the two diverge. (ii) Fault explanation then computes the reason for the divergence. The reason is a subset of messages from the test that is sufficient for divergence. (iii) Fault classification groups together tests that fail for similar reasons. Our approach relies on machinery from formal methods: (i) symbolic execution, (ii) Hoare logic and a new relationship between the intermediary assertions constructed for a test, and (iii) a new relationship among Hoare proofs. A crucial aspect in automotive software are timing requirements, for which we develop appropriate Hoare logic theory. We also briefly report on our prototype implementation for the CAN bus *Unified Diagnostic Services* in an industrial project.

Keywords: Fault Explanation · Fault Classification · Hoare Proofs.

1 Introduction

Intensive testing is the de-facto standard way of demonstrating the correctness of automotive software, and the more tests the higher the confidence we have in a system [42]. Model-based approaches have been instrumental in pushing the number of tests that can be evaluated, by increasing the degree of automation for the testing process. Indeed, all of the following steps are fully automated today: determining the test cases including the expected outcome, running them on the system, and comparing the outcome to the expectation [45]. Yet, there is a manual processing step left that, so far, has resisted automation. If the outcome of the test and the expectation do not match, the bug has to be identified. This is the moment the test engineer comes into play, and also the moment when automation strikes back. The bug will not only show up in one, but rather in



```
Test-1                      Ⓐ
[   5ms]   req CTR set 5
[   2ms] ● res CTR ack 5


[● 14ms] ● req CTR get
[●  4ms] 🐞 res CTR ret 0
```

```
Test-2                      Ⓐ
[   0ms]   req CTR set 5
[   5ms] ● res CTR ack 5
[● 12ms]   req CTR log <data>
[● 11ms]   res CTR done
[●  1ms] ● req CTR get
[●  3ms] 🐞 res CTR ret 0
```

```
Test-3                      Ⓑ
[   4ms]   req CTR set 5
[   3ms]   res CTR ack 5
[   2ms]   req CTR log <data>
[  12ms]   res CTR done
[● 56ms] ● req CTR get
[●  4ms] 🐞 res CTR ret 5
```

Figure 1. Traces of an ECU `CTR` with operations `set`, `get`, and `log`. Faults are marked with 🐞, relevant events with ●. Labels Ⓐ and Ⓑ indicate distinct causes for the faults.

a large number of test cases, and the engineer has to go through all of them to make sure not to miss a mistake. This is the problem we address: assist the test engineer when searching for bugs among a large number of failing tests.

Though our ideas may apply more broadly, we develop them in the context of hardware-in-the-loop testing for embedded controllers (ECUs) in the automotive industry [5]. The final ECU with its software is given to the test engineer as a black box. During testing, the ECU interacts with a (partly simulated) physical environment. This interaction is driven by a test suite derived from a test model. There are several characteristics that make hardware-in-the-loop testing substantially different from the earlier steps in the continuous integration and testing process (model/software/processor-in-the-loop testing). The first is the importance of timing requirements [2]. Second, the ECU with its software is a black-box. Indeed, in our setting it is provided by a supplier and the testing unit does not have access to the development model. Third, there is a test model capturing the product requirements document (PRD). It is a complex artifact that specifies the intended system behavior at a fine level of detail, including logical states, transitions, timing requirements, and message payloads. Indeed, *"testing automotive systems often requires test scenarios with a very precise sequence of time-sensitive actions"* [5]. As is good practice [5,42,45], the test model is different from the development model (it is even developed by a different company). Lastly, there are hundreds to thousands of tests, which is not surprising as it is known that real-time requirements *"are notoriously hard to test"* [45].

Example 1. Figure 1 illustrates the task at hand (ignore the ● marks for now). The figure shows three traces derived from the *Unified Diagnostic Services* [25]. A trace is a recording of the requests and responses resulting from executing a test case (pre-defined request sequence) on the ECU under test. Each line of the trace contains one message, carrying: (i) a time stamp indicating the time since the last message resp. the start, (ii) the type of message, `req` for requests and `res` for responses, (iii) an ECU identifier, the recipient for requests and the sender for responses, (iv) the name of an operation, e.g., `set`, and (v) optional payload.

In the first trace, the ECU with identifier `CTR` is requested to perform the `set` operation with value `5`. The ECU acknowledges that the operation was executed successfully, repeating value `5`. Subsequently, `CTR` receives a `get` request to which it responds with (returns) value `0`. The second trace additionally requests a `log` operation between `set` and `get`. In the third trace, `get` returns `5` instead of `0`.

The `get` responses in all traces are marked with 🐞 because they are faulty. Our example PRD requires `get` to return the value of the latest `set`, unless more than $50ms$ have passed since the latest (response to) `set`, in which case `0` has to be returned. Assume the PRD does not specify any influence of `log` on `set`/`get`, and vice versa. The first two traces expose the same fault, indicated by Ⓐ: the `set` appears to have been ignored. The last trace exposes a different fault, indicated by Ⓑ: `CTR` appears to have ignored that $50ms$ have passed. □

Our contribution is an algorithm that classifies failing test cases according to their causes. The algorithm expects as input the same information that is available to the test engineer: the test model and the traces of the failing tests. It consists of three steps: fault localization, fault explanation, and fault classification. The fault localization can be understood as replaying a trace on the model to identify the moment when the two diverge. In Example 1, this yields the 🐞 marks. The fault explanation then computes the reason for the divergence. The reason can be understood as a small set of messages in the trace that is sufficient for the divergence. In the example, this set is marked with ●. Even when removing the remaining messages, we would still have a bug. The fault classification groups together traces that are faulty for similar reasons. In the example, labels Ⓐ and Ⓑ.

Our approach relies on machinery from formal methods, following the slogan in [5]: *"more formal semantics are needed for test automation"*. Behind the fault localization is a symbolic execution [14,29]. The challenge here is to summarize loops in which time passes but no visible events are issued. We solve the problem with a widening approach from abstract interpretation [7]. Our fault explanation [3,18,19,20,28,40,52] is based on Hoare logic [6,44]. The challenge is to identify messages as irrelevant (for making the test fail), if they only let time pass but their effect is dominated by earlier parts of the test. We achieve this using a new relationship between the assertions in the Hoare proof that is constructed for the test at hand. The fault classification [50,51] equates Hoare proofs [38]. The challenge is again related to timing: the precise moments in which messages arrive will be different from test to test. We propose a notion of proof template that allows us to equate Hoare proofs only based on timing constraints satisfied by the underlying tests. The precise timing does not matter.

We implemented the classification in a project with the automotive industry, targeting the CAN bus *Unified Diagnostic Services*. The test model has all the features mentioned above: real time, messages, and numerical payloads. It is derived from a PRD with 350 pages of natural language and has 12k states and 70k transitions. Our approach is practical: in 24 minutes we process test suites of up to 1000 tests with an average of 40 and outliers of up to 2500 messages in length.

One may wonder why we classify tests at all. Since they are derived from a test model, why not group them by the functionality they test or coverage they achieve? The point is that functionality and coverage are only means of exposing faults [50]. The faults are what matters for the test engineer, and the same fault will show up in tests for different functions. Our experiments confirm this: we discover previously undetected faults in tests that targeted functions different from the failing one. We are particularly successful with faults involving timing, which are largely function independent and therefore admit a high degree of nondeterminism. Taking a step back, tests are designed by functionality or coverage, because it is hard to anticipate or even formulate possible faults in advance [45, 47,50,51]. Our explanation step makes the notion of a fault precise, and allows us to obtain the classification that the engineer needs for writing a test report.

Another question is whether we approach the problem from the wrong side. There is a large body of work on test suite minimization [36,50]. So why classify tests a posteriori when we could have executed fewer tests in the first place? The answer is that test suite minimization techniques are known to reduce the fault detection effectiveness, as demonstrated in the famous WHLM [48], WHMP [49], and Siemens [41] studies. This is unacceptable in the automotive sector.

A companion technical report containing missing details is available as [4].

2 Formal Model

We introduce a class of automata enriched by memory and clocks to model PRDs. A so-called *PRD automaton* is a tuple $\mathcal{A} = (Q, \rightarrow, S, E, V, C)$ with a finite set of states Q, a finite transition relation $\rightarrow$ among states, initial states $S \subseteq Q$, a finite set of events E, a finite set of memory variables V, and a finite set of clocks C. Variables and clocks are disjoint, $V \cap C = \varnothing$. Transitions take the form $p \xrightarrow{e,\, g,\, up} q$ with states $p, q \in Q$, event $e \in E$, guard g, and update up. Additionally, there are transitions $p \xrightarrow{\Delta,\, g,\, up} q$ that react on time progression, denoted by the special symbol $\Delta \notin E$. *Guards* are Boolean formulas over (in)equalities of memory variables, clocks, and constants. We assume a strict typing and forbid (in)equalities among memory variables and clocks. *Updates* are partial functions that may give new values to variables v, $up(v) \in \mathbb{Z}$, or reset clocks c, $up(c) = 0$. Lifting variable updates from values to terms (over variables) is straightforward.

The runtime behavior of PRD automata is defined in terms of labeled transitions between configurations. A *configuration* of $\mathcal{A}$ is a tuple $cf = (p, \varphi)$ consisting of a state $p \in Q$ and a total valuation $\varphi : V \rightarrow \mathbb{Z} \cup C \rightarrow \mathbb{R}_{\geq 0}$ of variables and clocks. The configuration is initial if $p \in S$ is initial (no constraints on φ).

Valuations φ are affected by the progression of time t and updates up. Progressing φ by t yields a new valuation $\varphi + t$, coinciding with φ on all variables v and advancing all clocks c by t, $(\varphi + t)(c) = \varphi(c) + t$. To apply up to φ, we introduce the *transformer* $[\![up]\!]$. It yields a new valuation $[\![up]\!](\varphi) = \varphi'$ such that

$$\varphi'(v) \;=\; up(v) \neq \bot \;?\; up(v) : \varphi(v) \qquad \text{and} \qquad \varphi'(c) \;=\; up(c) \neq \bot \;?\; 0 : \varphi(c)\;.$$

PRD automata $\mathcal{A}$ process finite traces $w = s_1 \dots s_n$ of events and time progressions, $s_i \in E \cup \mathbb{R}_{\geq 0}$. Events are instantaneous and time progressions make explicit the passing of time. A *basic run* $(p_1, \varphi_1) \xrightarrow{s_1} \cdots \xrightarrow{s_n} (p_{n+1}, \varphi_{n+1})$ of $\mathcal{A}$ on w is a sequence of steps where (p_1, φ_1) is initial. Steps $(p, \varphi) \xrightarrow{e} (q, \varphi')$ for events $e \in E$ are due to transitions in $\mathcal{A}$, so they satisfy the following two conditions:

(i) There is a transition $p \xrightarrow{e,\, g,\, up} q$ such that g is enabled. Enabledness means that φ is a model of g, written $\varphi \models g$.

(ii) The valuation φ' is induced by the transformer for up, $\varphi' = [\![up]\!](\varphi)$.

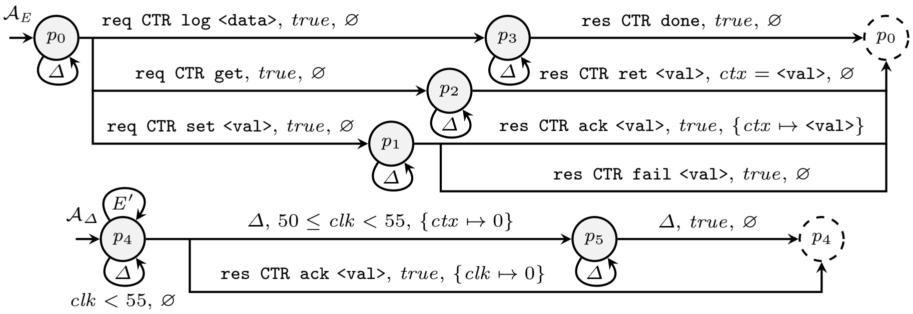

$\mathcal{A}_E$
p_0
Δ
req CTR log <data>, $true$, $\varnothing$
p_3
Δ
res CTR done, $true$, $\varnothing$
p_0
req CTR get, $true$, $\varnothing$
p_2
Δ
res CTR ret <val>, ctx = <val>, $\varnothing$
req CTR set <val>, $true$, $\varnothing$
p_1
Δ
res CTR ack <val>, $true$, $\{ctx \mapsto$ <val>$\}$
res CTR fail <val>, $true$, $\varnothing$
$\mathcal{A}_\Delta$
E'
p_4
Δ
$clk < 55$, $\varnothing$
Δ, $50 \leq clk < 55$, $\{ctx \mapsto 0\}$
p_5
Δ
Δ, $true$, $\varnothing$
p_4
res CTR ack <val>, $true$, $\{clk \mapsto 0\}$

Figure 2. Model $\mathcal{A}_E \times \mathcal{A}_\Delta$ for the ECU `CTR` from Example 1. Automaton $\mathcal{A}_E$ specifies operations `log`, `get`, and `set`. Automaton $\mathcal{A}_\Delta$ specifies how variable ctx is reset. We omit the guards $true$ and updates $\varnothing$ on Δ-loops. We use $E' \triangleq E\setminus\{$`res CTR ack <val>`$\}$.

Similarly, steps $(p, \varphi) \xrightarrow{t} (q, \varphi')$ taking time $t \in \mathbb{R}_{\geq 0}$ require:

(i) There is a Δ-transition $p \xrightarrow{\Delta,\, g,\, up} q$ enabled after waiting t time, $\varphi + t \models g$.

(ii) Valuation φ' is induced by clock progression plus up, $\varphi' = [\![up]\!](\varphi + t)$.

Finally, there are stuttering steps $(p, \varphi) \xrightarrow{0} (p, \varphi)$ which have no requirements.

Next, we lift basic runs to allow for multiple Δ-transitions during a single time progression t in w. This is needed to support complex behavior while waiting, as seen in Example 1. We rewrite w by splitting and merging time progressions. More precisely, we rewrite w into w' along these equivalences:

$$w_1.w_2 \equiv w_1.0.w_2 \qquad \text{and} \qquad w_1.t.w_2 \equiv w_1.t_1.t_2.w_2 \;\text{ if }\; t = t_1 + t_2\;. \qquad \text{(TEQ)}$$

Then, we say that $\mathcal{A}$ *has a run on* w if there is w' with $w' \equiv w$ so that $\mathcal{A}$ has a basic run on w'. The specification $\mathcal{L}(\mathcal{A})$ induced by $\mathcal{A}$ is the set of all traces w on which $\mathcal{A}$ has a run. Readers familiar with hybrid systems will observe that our rewriting produces finite decompositions only, thus excludes zeno behavior [1].

To simplify the exposition, we hereafter implicitly assume that traces w are normalized in the sense that every event is preceded and succeeded by exactly one time progression. This normalization is justified by the (TEQ) equivalences.

In practice, models have many transitions between two states in order to capture state changes that ignore parts of the event or accept a large number of possible values. To avoid PRD automata growing unnecessarily large, we use regular expressions instead of single events as transition labels. The automaton model presented so far naturally extends to such a lift. Our implementation integrates this optimization, see §7. For simplicity, we stick to vanilla automata hereafter.

Example 2. The automata $\mathcal{A}_E, \mathcal{A}_\Delta$ from Figure 2 specify `CTR` from Example 1. Automaton $\mathcal{A}_E$ addresses `get`, `log`, and `set`. The `set` request takes an arbitrary value `<val>` as a parameter. As discussed above, we use `<val>` as shorthand which can be translated on-the-fly into vanilla automata. The `set` request is always enabled and does not lead to updates. It may be followed by an `ack`, indicating success, or a `fail` response. If successful, variable ctx is updated to `<val>`. The reset of ctx after $50ms$ is implemented by $\mathcal{A}_\Delta$. Operations `get` and `log` are similar.

Automaton $\mathcal{A}_E$ does not specify any timing behavior, all its states have an always-enabled Δ-self-loop without updates. The timing behavior is specified by automaton $\mathcal{A}_\Delta$. It uses `ack` responses as a trigger to reset the timer clk and then waits until clk holds a value of at least 50. Once the threshold is reached, the Δ-transition from p_4 to p_5 setting ctx to 0 becomes enabled. Here, $\mathcal{A}_\Delta$ allows for slack: the reset must happen within $5ms$ once $50ms$ have passed. Within these $5ms$, $\mathcal{A}_\Delta$ may choose to cycle in p_4 without resetting or move to p_5 while resetting ctx. In practice, this kind of slack is common to account for the inability of hardware to execute after exactly $50ms$, as a guard like $\mathit{clk} \leq 50$ would require.

The overall specification of our example is the composition $\mathcal{A}_E \times \mathcal{A}_\Delta$. The cross-product is standard: a step can be taken only if both $\mathcal{A}_E$ and $\mathcal{A}_\Delta$ can take the step. We do not go into the details of operations over automata. □

3 Fault Localization

We propose a method for localizing faults in traces w. Intuitively, we do so by letting $\mathcal{A}$ run on w. If for some prefix $w'.s$ of w there is no step to continue the run, i.e., $w' \in \mathcal{L}(\mathcal{A})$ but $w'.s \notin \mathcal{L}(\mathcal{A})$, then s is a fault and $w'.s$ is its *witness*. Witnesses play an integral role in our approach: a Hoare proof for a witness yields a formal reason for the fault. In §4, we will refine this reason by extracting a concise explanation for the fault. This explanation then allows us to classify faults in §5.

Technically, identifying faults s in w is more involved. Establishing $w' \in \mathcal{L}(\mathcal{A})$ requires us to find $w'' \equiv w'$ and a basic run of $\mathcal{A}$ on w''. Establishing $w'.s \notin \mathcal{L}(\mathcal{A})$, however, requires us to show that there exists no basic run of $\mathcal{A}$ on $w'.s$ at all. It is not sufficient to show that the single basic run witnessing $w' \in \mathcal{L}(\mathcal{A})$ cannot be extended to $w'.s$. We have to reason over all $\tilde{w} \equiv w'.s$ and over all basic runs on them. To cope with this, we encode symbolically all such basic runs of $\mathcal{A}$ as a Hoare proof. The Hoare proof can be thought of as a certificate for the fault.

Interestingly, our techniques for fault localization (§3), explanation (§4), and classification (§5) do not rely on the exact form of Hoare proofs or how they are obtained—any valid proof will do. Hence, we prefer to stay on the *semantic level*. We discuss how to efficiently generate the necessary proofs in §6. Note that the timing aspect of our model requires us to develop novel Hoare theory in §6.

Symbolic Encoding We introduce a symbolic encoding to capture infinitely many configurations in a finite and concise manner.

A *symbolic configuration* is a pair $\mathit{cf}_\sharp = (p, F)$ where p is a state and F is a first-order formula. We use F to encode potentially infinitely many variable/clock valuations φ. We say F denotes φ if φ is a model for F, written $\varphi \models F$.

A *condition* P is a finite set of symbolic configurations. We write $(p, \varphi) \models P$ if there is $(p, F) \in P$ with $\varphi \models F$. We also write $P \sqsubseteq R$ if $\mathit{cf} \models P$ implies $\mathit{cf} \models R$ for all cf. If $P \sqsubseteq R$ and $R \sqsubseteq P$, we simply write $P = R$. The initial condition is $\mathit{Init} \triangleq \{\, (p, \mathit{true}) \mid p \in S \,\}$ and the empty condition is $\mathit{false} = \varnothing$. For simplicity, we assume that conditions contain exactly one symbolic configuration per state, as justified by the next lemma. With that assumption, checking $P \sqsubseteq R$ can be encoded as an SMT query and discharged by an off-the-shelf solver like `Z3` [35].

`Test-1` Ⓐ

$\{\mathit{true}\}$ `[ 5ms]` $\{\mathit{true}\}$ `req CTR set 5` $\{\mathit{true}\}$ `[ 2ms]` $\{\mathit{true}\}$ `res CTR ack 5`

$\{(p_0, p_4 : \mathit{ctx} \neq 0 \wedge \mathit{clk} < 32)\}$ `[14ms]` $\{(p_0, p_4 : \mathit{ctx} \neq 0 \wedge \mathit{clk} < 46)\}$ `req CTR get`

$\{(p_2, p_4 : \mathit{ctx} \neq 0 \wedge \mathit{clk} < 46)\}$ `[ 4ms]` $\{(p_2, p_4 : \mathit{ctx} \neq 0)\}$ `res CTR ret 0` $\{\mathit{false}\}$

`Test-2` Ⓐ

$\{\mathit{true}\}$ `[ 0ms]` $\{\mathit{true}\}$ `req CTR set 5` $\{\mathit{true}\}$ `[ 5ms]` $\{\mathit{true}\}$ `res CTR ack 5`

$\{(p_0, p_4 : \mathit{ctx} \neq 0 \wedge \mathit{clk} < 23)\}$ `[12ms]` $\{(p_0, p_4 : \mathit{ctx} \neq 0 \wedge \mathit{clk} < 35)\}$ `req CTR log <data>`

$\{(p_3, p_4 : \mathit{ctx} \neq 0 \wedge \mathit{clk} < 35)\}$ `[11ms]` $\{(p_3, p_4 : \mathit{ctx} \neq 0 \wedge \mathit{clk} < 46)\}$ `res CTR done`

$\{(p_0, p_4 : \mathit{ctx} \neq 0 \wedge \mathit{clk} < 46)\}$ `[ 1ms]` $\{(p_0, p_4 : \mathit{ctx} \neq 0 \wedge \mathit{clk} < 47)\}$ `req CTR get`

$\{(p_2, p_4 : \mathit{ctx} \neq 0 \wedge \mathit{clk} < 47)\}$ `[ 3ms]` $\{(p_2, p_4 : \mathit{ctx} \neq 0)\}$ `res CTR ret 0` $\{\mathit{false}\}$

Figure 3. Hoare proofs for `Test-1` and `Test-2`.

Lemma 1. *$P \cup \{(p, F), (p, G)\} = P \cup \{(p, F \vee G)\}$ and $P \cup \{(p, \mathit{false})\} = P$.*

Later, we will use conditions P below quantifiers $\exists \overline{x}.P$ and in the standard Boolean connectives $G \oplus P$ with formulas G. We lift those operations to conditions by pushing them into the symbolic configurations of P as follows:

$$\exists \overline{x}.\ P \triangleq \{(p, \exists \overline{x}.\ F) \mid (p, F) \in P\} \quad \text{and} \quad G \oplus P \triangleq \{(p, G \oplus F) \mid (p, F) \in P\}\,.$$

Finding Faults We localize faults in traces $w = s_1 \ldots s_n$. This means we check whether or not $\mathcal{A}$ has a run on w. To do so, we rely on a Hoare proof for w which takes the form

$$\{\, P_0 \,\}\ s_1\ \cdots\ \{\, P_{i-1} \,\}\ s_i\ \{\, P_i \,\}\ \cdots\ s_n\ \{\, P_n \,\}\ ,$$

where every triple $\{\, P_i \,\}\, s_i\, \{\, P_{i+1} \,\}$ is a Hoare triple. Intuitively, the Hoare triple means: every step for s_i starting in a configuration from P_i leads to a configuration in P_{i+1}. Hoare triples are defined to be insensitive to trace equivalence:

$$\models \{\, P \,\}\, s\, \{\, R \,\} \;:\Longleftrightarrow\; \forall \mathit{cf}, \mathit{cf}', w'.\;\; \mathit{cf} \models P \wedge s \equiv w' \wedge \mathit{cf} \xrightarrow{w'} \mathit{cf}' \implies \mathit{cf}' \models R\ .$$

If the condition is satisfied, we call the Hoare triple *valid*. For brevity, we write $\{\, P \,\}\, w'.s\, \{\, S \,\}$ if there is R so that $\{\, P \,\}\, w'\, \{\, R \,\}$ and $\{\, R \,\}\, s\, \{\, S \,\}$ are both valid. Strengthening resp. weakening the precondition P resp. postcondition R preserves validity: $P' \sqsubseteq P$ and $\models \{\, P \,\}\, s\, \{\, R \,\}$ and $R \sqsubseteq R'$ implies $\models \{\, P' \,\}\, s\, \{\, R' \,\}$.

Now, finding faults boils down to checking the validity of Hoare triples. It is easy to see that $\mathcal{A}$ has no run on $w'.s$ if and only if $\models \{\, \mathit{Init} \,\}\, w'.s\, \{\, \mathit{false} \,\}$.

Lemma 2. *If $\models \{\mathit{Init}\}\, w'\, \{P\}\, s\, \{\mathit{false}\}$ and $P \neq \mathit{false}$, then $w'.s$ witnesses fault s.*

Example 3. Figure 3 gives proofs that perform fault localization in `Test-1` and `Test-2` from Figure 1. The beginning of both traces is irrelevant for the fault, so *true* is used as precondition. Then, the conditions track the amount of time that passes in the form of an upper bound on clock clk. Since clk stays below $50ms$, variable ctx is never reset by $\mathcal{A}_\Delta$. Hence, `get` must not return 0. But because `get` does return 0 in the trace, we arrive at *false*—the response is a fault. □

The Hoare proof certifying witness $w'.s$ is input to the fault explanation and classification in the next sections. As stated earlier, we defer the generation of Hoare proofs (by means of strongest postconditions and weakest preconditions) to §6, as it is orthogonal to fault explanation and classification.

4 Fault Explanation

We analyze the Hoare proof generated in §3 which certifies the fault in a witness. Our goal is to extract the events that contribute to the fault and dispose of those that are irrelevant. The result will be another valid Hoare proof that concisely explains the fault. On the one hand, the explanation will help the test engineer understand the fault and ultimately prepare the test report alluded to in §1. On the other hand, explanations of distinct test cases may be similar in terms of our classification approach from §5 while the original test cases are not, thus improving the effectiveness of the classification.

To determine a concise explanation, assume the Hoare proof certifying the fault can be partitioned into $\{\,\mathit{Init}\,\}\; w_1 \;\{\,P\,\}\; w_2 \;\{\,R\,\}\; w_3 \;\{\,P_k\,\}$. If P denotes fewer configurations than R, $P \sqsubseteq R$, we say that w_2 is irrelevant (the events therein). To see this, consider some configuration $\mathit{cf} \models P$. Executing w_2 from cf leads to some $\mathit{cf}' \models R$ which in turn leads to the fault by executing w_3. However, $\mathit{cf} \models R$ already holds. So, we can just execute w_3 from cf to exhibit the fault—w_2 is irrelevant indeed.

When timing plays a role in the fault, one might not be able to establish the simple inclusion $P \sqsubseteq R$ because removing w_2 altogether also removes the time that passes in it. However, it might be this passing of time, rather than the events, that leads to the fault. Therefore, we also check if the events (and the events only) in w_2 are irrelevant. This is the case if waiting has the same effect as performing full w_2. Technically, we check the validity of the triple $\{\,P\,\}\; w_2|_{\mathbb{R}_{\geq 0}} \;\{\,R\,\}$. The projection $w_2|_{\mathbb{R}_{\geq 0}}$ removes all events E from w_2: $e|_{\mathbb{R}_{\geq 0}} = \epsilon$ and $t|_{\mathbb{R}_{\geq 0}} = t$. The validity of the triple captures our intuition: any configuration $\mathit{cf} \models P$ can simply wait (taking Δ-transitions) for the same amount as w_2 and arrive in $\mathit{cf}' \models R$ from which w_3 and the fault are executable—the removed events $w_2|_E$ are irrelevant.

We apply the above reasoning—both $P \sqsubseteq R$ as well as $\models \{\,P\,\}\; w_2|_{\mathbb{R}_{\geq 0}} \;\{\,R\,\}$—to all partitionings of the given proof to identify the irrelevant sequences. The remaining events and time progressions all contribute to the fault. The result is the most concise explanation of the fault.

Unfortunately, our pruning rules are not confluent, meaning that different sequences of *irrelevance checks* may lead to different explanations. A witness may have more than one explanation if two irrelevant sequences partially overlap. To see this, consider the following (special case) partitioning of the witness' proof

$$\begin{tikzcd}
\{\,\mathit{Init}\,\} \arrow[rr, bend left] \arrow[rr, bend right] & w_1 & \{\,P\,\} \arrow[rrrr, bend left] \arrow[rr, bend right] & w_2 & \{\,R\,\} \arrow[rrrr, bend right] & w_3 & \{\,P\,\} \arrow[rr, bend left] & w_4 & \{\,R\,\} \arrow[rr, bend left] \arrow[rr, bend right] & w_5 & \{\,\mathit{false}\,\}\ .
\end{tikzcd}$$

Here, we deem irrelevant $w_2.w_3$ and $w_3.w_4$. However, we cannot remove $w_2.w_3.w_4$ entirely because the resulting proof might not be valid, which requires $P \sqsubseteq R$.

Even removing the intersection w_3 of the irrelevant sequences may not produce a valid proof as $R \sqsubseteq P$ might not hold either. The same problems arise if only $(w_2.w_3)|_E$ and/or $(w_3.w_4)|_E$ is irrelevant. We argue that this is desired: the witness is, in fact, a witness for two different faults, explained by $w_1.w_4.w_5$ resp. $w_1.w_2.w_5$. Overall, we compute all explanations in case there are overlapping irrelevant sequences. While this gives exponentially many explanations in theory, we rarely find overlaps in practice.

Example 4. We give the fault explanation for the proof of `Test-2` from Figure 3. As expected, both events `req CTR log <data>` and `res CTR done` are irrelevant. The condition $P = \{\, (p_0, p_4 : ctx \neq 0 \land clk < 23) \,\}$ before the `log` request reaches condition $R = \{\, (p_0, p_4 : ctx \neq 0 \land clk < 47) \,\}$ after the `log` response. This remains true after removing both events. Indeed, $\{\, P \,\}$ `[12ms][11ms][ 1ms]` $\{\, R \,\}$ is a valid Hoare triple and thus justifies removing the events. □

5 Fault Classification

We propose a classification technique that groups together witnesses exhibiting the same or a similar fault. Grouping together similar faults significantly reduces the workload of test engineers when preparing a test report for a large number of failing tests since only one (representative) test case per group needs to be inspected. The input to our classification is a set W of witness explanations as constructed in §4. The result of the classification is a partitioning of W into disjoint classes $W = W_1 \uplus \cdots \uplus W_m$. The partitioning is obtained by factorizing W along an equivalence $\sim$ that relates witness explanations which have similar faults. If $\sim$ is effectively computable, so is the factorization. We focus on $\sim$.

Intuitively, two explanations are similar, and thus related by $\sim$, if comprised of the same sequence of Hoare triples, that is, the same sequence of events and intermediary assertions. This strict equality, however, does not work well when timing is involved. Repeatedly executing the same sequence of events is expected to observe a difference in timing due to fluctuations in the underlying hardware. Moreover, explanations have already been stripped by irrelevant sequences the events and duration of which might differ across explanations.

To make up for these discrepancies, we relate explanations that are equal up to *similar clocks*. Consider an (in)equality F over clocks C. We can think of F, more concretely its solutions, as a polytope $M \subseteq \mathbb{R}^{|C|}$. Then, two clock assignments $\varphi, \varphi' \in \mathbb{R}^{|C|}$ are similar if they agree on the membership in M. That is, φ and φ' are similar if $\varphi, \varphi' \in M$ or $\varphi, \varphi' \notin M$. The polytope M we consider will stem from the transition guards in $\mathcal{A}$. Similarity thus means that $\mathcal{A}$ cannot distinguish the two clock assignments—they fail for the same reason.

Clock similarity naturally extends to sets of polytopes. The set of polytopes along which we differentiate clock assignments is taken from a *proof template*. A proof template for a trace is a unique Hoare proof where placeholders are used instead of actual time progressions. Hence, the explanations under consideration are instances of the template, i.e., can be obtained by replacing the placeholders with the appropriate time progressions. More importantly, the template gives rise to a set of *atomic constraints* from which all polytopes appearing in the explanations can be constructed (using Boolean connectives). Overall, this means that two explanations are similar if the clocks they allow for are similar wrt. the polytopes of the associated proof template, meaning that $\mathcal{A}$ cannot distinguish them and thus fails for the same reason.

A proof template for events $e_1 \dots e_k$ is a Hoare proof of the form

$$\{\, \mathit{Init} \,\}\, u_0 \,\cdots\, \{\, P_{2i-1} \,\}\, e_i \,\{\, P_{2i} \,\}\, u_i \,\{\, P_{2i+1} \,\}\, \cdots\, u_k \,\{\, \mathit{false} \,\} \;\; .$$

This proof is a template because $\overline{u} = u_0, \dots, u_k$ are symbolic time progressions, i.e, they can be thought of as variables rather than actual values from $\mathbb{R}_{\geq 0}$. An instance of the template is a valid Hoare proof

$$\{\, \mathit{Init} \,\}\, t_0 \,\cdots\, \{\, R_{2i-1} \,\}\, e_i \,\{\, R_{2i} \,\}\, t_i \,\{\, R_{2i+1} \,\}\, \cdots\, t_k \,\{\, \mathit{false} \,\}$$

with actual time progressions $\overline{t} = t_0, \dots, t_k$ such that the P_i subsume the R_i for the given choice of symbolic time progressions, $R_i \sqsubseteq P_i[\overline{u} \mapsto \overline{t}]$.

For the classification to work, we require the following properties of templates:

(C1) the template is uniquely defined by the sequence $u_0.e_1 \dots e_k.u_k$, and

(C2) the symbolic configurations appearing in the P_i are quantifier-free.

The former property associates a unique template to every trace. This is necessary for a meaningful classification via templates. The latter property ensures that the atomic constraints we extract from the template (see below) will contain only clocks from C. This is necessary for equisatisfiability to be meaningful. In §6 we show that weakest preconditions generate appropriate templates.

An atomic clock constraint is an (in)equality over symbolic time progressions and ordinary clocks (from C). We write $\mathit{acc}(P)$ for all such constraints syntactically occurring in P. For P_i from the above proof template, $\mathit{acc}(P_i)$ is a set of building blocks from which the R_i of *all* instantiations can be constructed. Moreover, $\mathcal{A}$ cannot distinguish time progression beyond $\mathit{acc}(P_i)$, making them ideal candidates for judging similarity.

We turn to the definition of the equivalence relation $\sim$. To that end, consider two explanations α, β of the following form

$$\begin{aligned} \alpha: &\quad \{\, \mathit{Init} \,\} \cdots \{\, R_{2i-1} \,\}\, e_i \,\{\, R_{2i} \,\}\, t_i \,\{\, R_{2i+1} \,\} \cdots \{\, \mathit{false} \,\} \\ \beta: &\quad \{\, \mathit{Init} \,\} \cdots \{\, R'_{2i-1} \,\}\, e_i \,\{\, R'_{2i} \,\}\, t'_i \,\{\, R'_{2i+1} \,\} \cdots \{\, \mathit{false} \,\} \, . \end{aligned}$$

The events $e_1, \dots, e_k$ match in both explanations, but the time progressions $\overline{t}$ and $\overline{t'}$ may differ. (Explanations with distinct event sequences are never related by $\sim$.) Both explanations are instances of the same proof template σ,

$$\sigma: \quad \{\, \mathit{Init} \,\} \cdots \{\, P_{2i-1} \,\}\, e_i \,\{\, P_{2i} \,\}\, u_i \,\{\, P_{2i+1} \,\} \cdots \{\, \mathit{false} \,\} \, .$$

Now, for α and β to be similar, $\alpha \sim \beta$, we require the R_i and R'_i to satisfy the exact same atomic clock constraints appearing in P_i relative to the appropriate instantiation of the symbolic clock values. It is worth stressing that we require satisfiability, not logical equivalence, because we want the clocks to be similar, not equal. We write $\mathrm{SAT}(F)$ if F is satisfiable, that is, if there is an assignment φ

to the free variables in F such that $\varphi \models F$. Formally then, we have:

$$\alpha \sim \beta \qquad \text{iff} \qquad \forall i\ \forall F \in acc(P_i).\ \ \mathrm{SAT}(F[\overline{u} \mapsto \overline{t}]) \iff \mathrm{SAT}(F[\overline{u} \mapsto \overline{t'}])\ .$$

It is readily checked that $\sim$ is an equivalence relation, that is, is reflexive, symmetric, and transitive, as alluded to in the beginning. Transitivity, in particular, is desirable in our use case. First, it means that all explanations from a class W_i of W are pairwise similar, that is, exhibit the same fault. Second, the partitions are guaranteed to be disjoint. Finally, it allows for the partitioning of W to be computed efficiently (by tabulating the result of the SAT queries), provided the SAT queries are efficient for the type of (in)equalities used.

Lemma 3. *Relation $\sim$ is an equivalence relation.*

Example 5. We classify the explanations of `Test-1` and `Test-2`, which correspond to the proofs from Figure 3 with the `log` events removed (cf. Example 4). Both explanations agree on the sequence of events. Figure 4 gives their common template. The atomic clock constraints are $u_1 + u_2 < 50$, $clk + u_1 < 50$, and $clk + u_1 + u_2 < 50$. `Test-1` and `Test-2` are similar because each clock constraint is satisfiable after instantiating the symbolic time progressions with the values in the respective trace. Hence, our classification groups these explanations together, `Test-1` $\sim$ `Test-2`. □

`Template<Test-1, Test-2>` Ⓐ

$\{(p_1, p_4 : u_1 + u_2 < 50)\}$
`res CTR ack 5`
$\{(p_0, p_4 : ctx \neq 0 \wedge clk + u_1 + u_2 < 50)\}$
`[` u_2`ms]`
$\{(p_0, p_4 : ctx \neq 0 \wedge clk + u_1 < 50)\}$
`req CTR get`
$\{(p_2, p_4 : ctx \neq 0 \wedge clk + u_1 < 50)\}$
`[` u_1`ms]`
$\{(p_2, p_4 : ctx \neq 0)\}$
`res CTR ret 0`
$\{false\}$

Figure 4. Proof template for the explanations of `Test-1` and `Test-2`.

6 Hoare Proofs with Timing

For the techniques presented so far to be useful, it remains to construct Hoare proofs for traces w. Strongest postconditions and weakest preconditions are the standard way of doing so. The former yields efficient fault localization (§3). The latter satisfies the requirements for templates (§5). Moreover, interpolation between the two produces concise proofs beneficial for fault explanations (§4).

It is worth pointing out that the aforementioned concepts are well-understood for programs and ordinary automata. However, they have not been generalized to a setting like ours where timing plays a role. Indeed, works like [21,23,24,43] involve timing, but do not develop the Hoare theory required here.

Strongest Postconditions We compute the *post image*, that is, make precise how $\mathcal{A}$ takes steps from symbolic configurations. A step from a symbolic configuration (p, F) due to transition $p \xrightarrow{\Delta,\, g,\, up} q$ on time progression t can be taken if the guard is enabled after waiting for t time. After waiting, all clocks c are $c' = c + t$. This means before waiting we have $c = c' - t$. However, clocks are always non-negative, $c' - t \geq 0$. Overall, we replace in F all clocks by their old versions and enforce non-negativity, $F' = F[C \mapsto C - t] \wedge C \geq t$. It remains to check guard g and apply update up. It is easy to see that the set of valuations in F' satisfying g is precisely $G = F' \wedge g$. To perform a singleton update $\{\, x \mapsto y \,\}$, we capture the new valuation of x by the equality $x = y$. To avoid an influence of the update of x on other variables/clocks, we have to rewrite G to not contain x. This is needed as G might use x to correlate other variables/clocks—we want to preserve these correlations without affecting them. We use an existential abstraction that results in $G' = \exists z.\ G[x \mapsto z] \wedge x = y$. Then, the post image is (q, G'). For stuttering steps, we add the original configuration (p, F) to the post image. Steps due to events from E are similar.

We define a symbolic transformer that implements the above update of the symbolic encoding F to G' in the general case:

$$[\![g|\{\overline{x}\mapsto\overline{y}\}]\!]^\sharp_t(F) \ \triangleq\ \exists\overline{z}.\ (F[C\mapsto C-t]\wedge C\geq t\wedge g)[\overline{x}\mapsto\overline{z}]\wedge\overline{x}=\overline{y}\ ,$$

where $\overline{x}$ is short for a sequence $x_1, \ldots, x_m$ of variables/clocks. We arrive at:

$$post^\sharp_t(P) \triangleq \{\, (q, [\![g|up]\!]^\sharp_t(F)) \mid (p,F)\in P \wedge p\xrightarrow{\Delta,\,g,\,up} q \,\} \cup \big(t=0\ ?\ P\ :\ \varnothing\big)$$

$$post^\sharp_e(P) \triangleq \{\, (q, [\![g|up]\!]^\sharp_0(F)) \mid (p,F)\in P \wedge p\xrightarrow{e,\,g,\,up} q \,\}\ .$$

The post image is sound and precise in the sense that it captures accurately the steps the configurations denoted by P can take. The lemma makes this precise.

Lemma 4. *$cf' \models post^\sharp_s(P)$ iff there is $cf \models P$ with $cf \xrightarrow{s} cf'$.*

Example 6. We apply $post^\sharp$ to $P = \{(p_4, 49 \leq clk \leq 52)\}$ for $\mathcal{A}_\Delta$ from Figure 2. Recall that $\mathcal{A}_\Delta$ resets variable ctx within $5ms$ after clk has reached the $50ms$ mark. Indeed, $post^\sharp_1(P)$ for $1ms$ contains both the resetting and the non-resetting case: $(p_5, 50 \leq clk \leq 53 \wedge ctx = 0)$ and $(p_4, 50 \leq clk \leq 53)$.

The post image still lacks a way to commute with the (TEQ) congruences. While $post^\sharp_5(post^\sharp_1(P))$ witnesses the reset via condition $55 \leq clk \leq 58 \wedge ctx = 0$ for both p_4 and p_5, it is not equivalent to $post^\sharp_6(P)$, which is *false* since all transitions in p_4 are disabled for a full $6ms$ wait. □

While the post image captures the individual steps of basic runs on traces w, we have to consider the basic runs of all traces $w' \equiv w$ to generate a Hoare proof for w. Basically, the (TEQ) equivalences state that the time progressions between events can be split/merged arbitrarily. To that end, we define the strongest postcondition sp which inspects all basic runs simultaneously, intuitively, by rewriting according to the (TEQ) equivalences on-the-fly. (Note that normalization according to §2 avoids the merging case of (TEQ).) Then, for events e the strongest postcondition merely applies the post image to e. For time progressions t, the strongest postcondition considers all decompositions of t into fragments $t_1, \ldots, t_k$ that add up to t and applies the post image iteratively to all the t_i. This includes stuttering where 0 is rewritten to $0 \ldots 0$. If there are loops in $\mathcal{A}$, the strongest postcondition might need to consider infinitely many decompositions. We address this problem by enumerating decompositions of increasing length and applying to each decomposition a widening ∇ with the following properties: (i) the result of the widening is $\sqsubseteq$-weaker than its input,

$P_i \sqsubseteq \nabla(P_1, \cdots, P_k)$ for all i, and (ii) the widening stabilizes after finitely many iterations for $\sqsubseteq$-increasing sequences, $P_1 \sqsubseteq P_2 \sqsubseteq \cdots$ implies that there is k so that $\nabla(P_1, \cdots, P_i) = \nabla(P_1, \cdots, P_{i+1})$ for all $i \geq k$. We write $\nabla(P_i)_{i\in\mathbb{N}}$ and mean the stabilized $\nabla(P_1, \cdots, P_k)$. Given a widening, the strongest postcondition is:

$$sp_t(P) \triangleq \nabla \left(\exists t_1, \cdots, t_i.\ t = t_1 + \cdots + t_i \ \wedge\ post^\sharp_{t_i} \circ \cdots \circ post^\sharp_{t_1}(P)\right)_{i\in\mathbb{N}}$$

$$sp_e(P) \triangleq post^\sharp_e(P) \qquad sp_{s.w}(P) \triangleq sp_w \circ sp_s(P) \qquad sp(P,\, w) \triangleq sp_w(P)$$

where the $t_1, \ldots, t_i$ are fresh. Observe that the sequence of post images in sp_t is $\sqsubseteq$-increasing: one can always extend the decomposition by additionally waiting for 0 time, $post^\sharp_{t_i}(P) \sqsubseteq post^\sharp_0 \circ post^\sharp_{t_i}(P)$. The strongest postcondition considers all basic runs and ∇ overapproximates the reachable configurations. It is sound.

Lemma 5. *If $sp(P,\, w) \sqsubseteq R$, then $\models \{\, P \,\}\, w \,\{\, R \,\}$.*

For the lemma to be useful, an appropriate widening ∇ is required. In general, finding such a widening is challenging—after all, it resembles finding loop invariants—and for doing so we refer to existing works, like [8,12,13], to name a few. In practice, a widening may be obtained more easily. In case $\mathcal{A}$ is free from $\mathit{\Delta}$-cycles, stabilization is guaranteed after k iterations, where k is the length of the longest simple $\mathit{\Delta}$-path. If there are $\mathit{\Delta}$-cycles, stabilization is still guaranteed after k iterations if all $\mathit{\Delta}$-cycles are *idempotent*. A $\mathit{\Delta}$-cycle is idempotent if repeated executions of the cycle produce only configurations that already a single execution of the cycle produces. Interestingly, idempotency can be checked while computing the widening: if the $(k{+}1)$st iteration produces new configurations, idempotency does not hold. In our setting, idempotency was always satisfied. For the remainder of this paper, we assume an appropriate widening is given.

Weakest Preconditions We also compute weakest preconditions, the time-reversed dual of strongest postconditions. Our definition will satisfy the template requirements (C1) and (C2) from §5.

The *pre image* is the set of symbolic configurations that reach a given configuration in automaton $\mathcal{A}$. Consider some (q, G) and $p \xrightarrow{\mathit{\Delta},\, g,\, up} q$. The pre image first rewinds updates $up = \{\, \overline{x} \mapsto \overline{y} \,\}$ by replacing $\overline{x}$ with $\overline{y}$. Then, it adds a disjunct $H = G[\overline{x} \mapsto \overline{y}] \vee \neg g$. Adding the disjunct makes the pre image weaker; it does not affect soundness in Lemma 6 which ignores the *stuck* configurations denoted by $(p, \neg g)$. Finally, we rewind the clock progression t by replacing all clocks c in H with $c + t$. We arrive at the pre image $F = H[C \mapsto C + t]$. Transitions due to events are similar. We define a symbolic transformer to apply the above process:

$$\overline{[\![g | \{\overline{x} \mapsto \overline{y}\}]\!]}^\sharp_t(G) \ \triangleq\ (G[\overline{x} \mapsto \overline{y}] \vee \neg g)[C \mapsto C + t] \ .$$

To account for other transitions leaving p that are enabled in H, we compute the meet $\sqcap$ of the per-transition pre images. Intuitively, this intersects symbolic configurations on a per-state basis, ensuring that any configuration from the pre image either gets stuck or steps to one of the configurations we computed the pre image for. Technically, the meet $\sqcap$ for sets M of symbolic configurations is:

$$\bigsqcap M \triangleq \{\, (p, \textstyle\bigwedge_{(p,F)\in M} F) \ \mid\ p \in Q \,\} \ .$$

Notably, when considering the meet of M, we cannot understand M as a condition. This is because conditions treat symbolic configurations disjunctively and can be normalized by Lemma 1. However, the meet is not preserved under these transformations. We write $M_1 \sqcap M_2$ to mean $\bigsqcap(M_1 \cup M_2)$.

The discussion yields the following definition of the pre image:

$$pre^{\sharp}_{t}(P) \triangleq \bigsqcap\{(p, \overline{\llbracket g|up \rrbracket}^{\sharp}_{t}(G)) \mid (q, G) \in P \wedge p \xrightarrow{\Delta,\, g,\, up} q\} \sqcap \big(t = 0 \ ?\ P \ :\ \varnothing\big)$$

$$pre^{\sharp}_{e}(P) \triangleq \bigsqcap\{(p, \overline{\llbracket g|up \rrbracket}^{\sharp}_{0}(G)) \mid (q, G) \in P \wedge p \xrightarrow{e,\, g,\, up} q\} \ ,$$

capturing precisely the forced reachability in $\mathcal{A}$, as stated by the next lemma.

Lemma 6. *$cf \models pre^{\sharp}_{s}(P)$ iff for all cf', $cf \xrightarrow{s} cf'$ implies $cf' \models P$.*

Example 7. We apply $pre^{\sharp}$ to $P = \{(p_4, 49 \leq clk \leq 52)\}$ for $\mathcal{A}_{\Delta}$ from Figure 2. Computing $pre^{\sharp}_{1}(P)$ highlights the need for the meet. The Δ-loop on p_4 does not give $(p_4, 48 \leq clk \leq 51)$ as precondition. Instead, it is $(p_4, 48 \leq clk < 49)$ which is the result of $\{(p_4, 48 \leq clk \leq 51)\} \sqcap \{(p_4, clk \geq 54 \vee clk < 49)\}$. Indeed, $\mathcal{A}_{\Delta}$ reaches a non-P configuration via the resetting transition to p_5 if $clk = 49$. □

The weakest precondition $wp(s, R)$ denotes all configurations that either step to R under s or have no step at all. Technically, the weakest precondition repeatedly applies the pre image for all decompositions of time progressions. For termination, we again rely on the widening ∇. Since the pre image sequence is $\sqsubseteq$-decreasing, we turn it into an increasing sequence by taking complements. More precisely, we use the widening $\overline{\nabla}(P_1, \cdots, P_m) \triangleq \neg\nabla(\neg P_1, \cdots, \neg P_m)$. The weakest precondition is defined by:

$$wp_{t}(P) \ \triangleq\ \overline{\nabla}\left(\forall t_1, \cdots, t_i.\ t = t_1 + \cdots + t_i \implies pre^{\sharp}_{t_1} \circ \cdots \circ pre^{\sharp}_{t_i}(P)\right)_{i \in \mathbb{N}}$$

$$wp_{e}(P) \ \triangleq\ pre^{\sharp}_{e}(P) \qquad wp_{w.s}(P) \ \triangleq\ wp_{w} \circ wp_{s}(P) \qquad wp(w,\, P) \ \triangleq\ wp_{w}(P) \ .$$

Note that wp_t applies to ordinary time progressions t as well as symbolic time progressions u appearing in proof templates. The weakest precondition is sound.

Lemma 7. *If $P \sqsubseteq wp(w,\, R)$, then $\models \{\, P \,\}\, w \,\{\, R \,\}$.*

Concise Hoare Proofs The developed theory allows for an efficient way to produce concise Hoare proofs. We first apply strongest postconditions to generate an initial proof. Then, starting from the back, we apply weakest preconditions and interpolation [9] to simplify the initial proof. We make this precise.

Combining Lemmas 2 and 5 gives an effective way of finding faults in traces $w = s_1 \ldots s_n$ and extracting a witness: iteratively compute the strongest postcondition for increasing prefixes of w and check if the result is unsatisfiable. That is, compute $P = sp(s_1. \cdots .s_k,\, Init)$ and check if $P = false$. If so, then $\widehat{w} = s_1 \ldots s_k$ is a witness for fault s_k. Otherwise, continue with the prefix $s_1 \ldots s_k.s_{k+1}$ which can reuse the previously computed P: $sp(s_1 \ldots s_k.s_{k+1},\, Init) = sp(s_{k+1},\, P)$. As per Lemma 5, the approach gives rise to the valid Hoare proof

$$\{\, Init \,\}\ s_1\ \cdots\ \{\, P_i \,\}\ s_{i+1}\ \{\, P_{i+1} \,\}\ \cdots\ s_k\ \{\, false \,\} \quad \text{with} \quad P_{i+1} = sp(P_i,\, s_{i+1}) \ .$$

It is well-known that strongest postconditions produce unnecessarily complex proofs [34]. To alleviate this weakness, we use interpolation [9]. For two formulas F and G with $F \implies G$, an interpolant is a formula I with $F \implies I$ and $I \implies G$. The interpolant for conditions P and R with $P \sqsubseteq R$, denoted $I(P, R)$, results from interpolating the symbolic configurations in P with the corresponding ones in R. Interpolants exist in first-order predicate logic [9,32].

From the above *sp*- generated proof we construct an interpolated proof

$$\{\, \mathit{Init} \,\}\ s_1\ \cdots\ \{\, I(P_i, R_i) \,\}\ s_{i+1}\ \{\, I(P_{i+1}, R_{i+1}) \,\}\ \cdots\ s_k\ \{\, \mathit{false} \,\}$$

using *wp* as follows. Assume we already constructed, starting from the back, the interpolants $I(P_k, R_k)$ through $I(P_{i+1}, R_{i+1})$. Now, the goal is to obtain an interpolant I so that $\{\, I \,\}\ s_{i+1}\ \{\, I(P_{i+1}, R_{i+1}) \,\}$ is valid. The weakest precondition for the latest interpolant yields $R_i = \mathit{wp}(s_i,\, I(P_{i+1}, R_{i+1}))$. This gives a valid Hoare triple $\models \{\, R_i \,\}\ s_{i+1}\ \{\, I(P_{i+1}, R_{i+1}) \,\}$. Our goal is to interpolate P_i and R_i. If $P_i \sqsubseteq R_i$, we can interpolate P_i and R_i to obtain $I = I(P_i, R_i)$.[4] Otherwise, we simply choose $I = R_i$. By Lemma 7 together with $I \sqsubseteq R_i$, we know that $\models \{\, I \,\}\ s_{i+1}\ \{\, I(P_{i+1}, R_{i+1}) \,\}$ is valid. Overall, this constructs a valid proof.

7 Application in Automotive Software

We implemented and tested our approach on benchmarks provided by our project partner from the automotive industry. The implementation parses, classifies, and annotates traces of ECUs running the Unified Diagnostic Services (UDS). We turned a PRD with 350 pages of natural language specifying 23 services into a PRD automaton of 12.5k states and 70k transitions. We evaluated our tool on 1000 traces which are processed within 24 minutes. Our tool is implemented in `C#` and processes traces in the three stages explained below. It naturally supports multi-threading for the localization, explanation, and classification since they are agnostic to the (set of) other traces being analyzed.

Preprocessing Stage The first stage parses trace files and brings them into a shape similar to Figure 1. UDS specify a request-response protocol for ECUs communicating over a CAN bus. The traces are a recording of all messages seen on the bus during a test run. We found the preprocessing more difficult than expected, because the trace files have a non-standard format. These problems stem from the fact that our industrial partner creates tests partly manually and inserts natural language annotations. A useful type of annotation that we could extract are the positions deemed erroneous by the test environment.

Modeling Stage The second stage creates the test model, a PRD automaton as defined in §2. Modeling a natural language PRD is a non-trivial and time-consuming process. To translate the PRD into an automaton, we developed an API capable of programmatically describing services and their communication

[4] One can show that the inclusion $P_i \sqsubseteq R_i$ is always satisfied in our setting where Δ-cycles are idempotent and the widenings ∇ and $\overline{\nabla}$ simply enumerate all necessary decompositions of time progressions. Refer to [4] for a more general property.

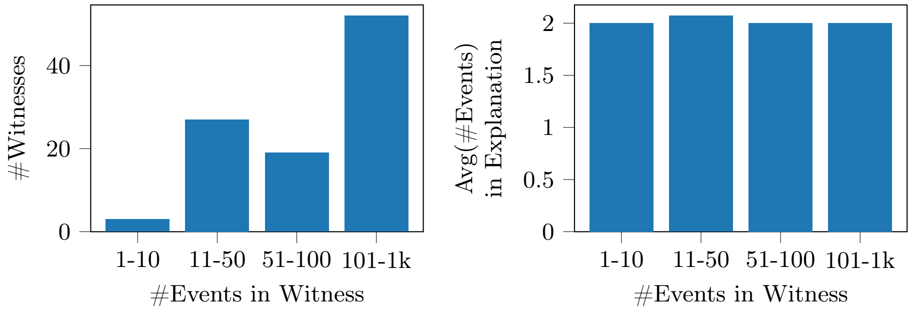


Figure 5. Statistics on witnesses: number (left) and average explanation length (right).

requirements. The API supports a declarative formulation of the communication requirements which it compiles down into an automaton. The compilation is controlled by a set of parameters because the PRD prescribes different behavior depending on the ECU version (and related static parameters). There are further high-level modeling constructs such as regular expressions, as alluded to in §2 and seen in Figure 2.

Unfortunately, not all requirements from the PRD are restricted to the trace: they may refer to events internal to the ECU that are not contained in the trace files. While our API and PRD automata are capable of expressing these requirements, the test environment is unable to detect them. To circumvent the problem of missing information, we over-approximated our model using non-determinism. That is, we simply allow our model to do any of the specified behaviors for unobservable internal events. A downside of this is that errors dependent on these events cannot be found during fault localization.

Analysis Stage The last stage performs fault localization (§3), explanation (§4), and classification (§5). We carefully inspected 86 traces curated by our industrial partner. The tests targeted one of the 23 services, yet they contain requests and responses to a multitude of services responsible for setting up the ECU configuration. The annotations of the test environment marked 100 faults, 95 of which are also found by our fault localization. Our tool finds and explains another 10 undetected faults, which totals to 105 fault explanations. The five faults missed by our localization are actually incorrect annotations by the test environment, which we will explain in a moment.

Figure 5 gives the lengths of the found witnesses and the average lengths of their explanations. The explanation lengths are closely tied to the kinds of faults in the test set. In our set, long witnesses tend to have a long prefix unimportant to the fault. This is reflected in the partitioning found by our classification.

The classification divides the faults into six partitions. We found that each partition belongs to one of the following three error types: (i) ECU responds too late (1+8); (ii) ECU fails to reset a variable upon restart (2); (iii) ECU responds when it should not (2+1+91). Here, 1+8 means we have two partitions and one with a single witness, one with eight equivalent witnesses. Each error type consists of at most two relevant events. Unrelated events in-between those two events are dropped by fault explanation. The relevant events are: (i) the request and the late response, (ii) the response event which revealed that the variable has not been reset, and (iii) the request and the incorrectly given response.

There are two partitions with error type (i). This is because the late response is given by another service and thus leads to different control flow in the automaton. Indeed, there might be distinct root causes: different services are likely controlled by different pieces of code. A similar reason produces three partitions of error type (iii). Interestingly, the singleton partition for (i) is completely missed by the test environment (no fault was marked). This supports our claim that the test environment only detects faults targeted by the tests and ignores other faults. The other partition of (i) was detected by the test environment by accident: in some traces, the ECU response is so late that the test environment incorrectly marks the response as missing. These incorrect marks represent no faults and are not considered by our localization. Instead, our localization actually detects the late responses and marks them correctly.

Our tool provides a partitioning file with direct links to the trace files. It also modifies the trace files to highlight the events related to the fault (cf. §4) and provides an intuitive explanation of the fault. As for the latter, the user is informed about the difference between the observed and the automaton-expected behavior. Our manual inspection showed no incorrect classification. That is, our tool has never grouped together traces which test engineers would deem caused by distinct faults. This is promising feedback because incorrect classification is dreaded: a single missed flaw of an ECU can cause large costs. Overall, we reduced the workload of manually inspecting 86 traces with 100 fault marks to inspecting six representative faults that expose more misbehavior than marked by the test environment.

8 Related Work

Fault Explanation Our work on fault explanation is related to minimizing unit tests in [30]: tests are pruned by removing the commands that are not contained in a backward slice from a failing instruction. With timing constraints, slicing does not work (every command is relevant), which is why we have developed our approach based on Hoare logic. The assertions provided by a Hoare proof have the additional advantage of being able to prune even dependent commands inside a slice (based on the relationship between intermediary assertions), which leads to higher reduction rates. Similar to our approach is the fault localization and explanation from [6,44]. That work also makes use of interpolation [33] and is able to strip infixes from a trace despite dependencies. Our fault localization can be understood as a generalization to a timed setting where every command contributes to the progression of time and therefore is delicate to remove.

A popular fault explanation approach that can be found in several variants in the literature [3,18,19,20,28,40,52] is Delta debugging: starting from a failing test, produce a similar but passing test, and take the difference in commands as an explanation of the fault. In [18,19,20,40,52], the passing test is found by repeatedly testing the concrete system [19], which is impossible in our in-vitro setting. In [3,18,28], a model checker resp. a solver is queried for a passing test resp. a satisfiable subset of clauses. Our Hoare proof can be understood as building up an alternative and valid execution. Different from a mere execution, however, intermediary assertions provide valuable information about the program state that we rely on when classifying tests.

The explanation from [26] divides a computation into fated and free segments, the former being deterministic reactions to inputs and the latter being inputs that, if controlled appropriately, avoid the fault and hence should be considered responsible for it. The segments are computed via rather heavy game-theoretic techniques, which would be difficult to generalize to timed systems. A more practical variant can be found in [46,53]. These works modify tests in a way that changes the evaluation of conditionals. Neither can we re-run tests in an in-vitro setting, nor would we be able to influence the timing behavior.

There is a body of literatur on statistical approaches to finding program points that are particularly prone to errors, see the surveys [47,50]. We need to pinpoint the precise as possible cause of a bug, instead.

Fault Classification Previous works on test case classification follow the same underlying principle [10,11,15,16,17,27,31,37,39]: devise a distance metric on test cases that is used to group them. The metrics are based on properties like the commonality/frequency of words in comments and variables in the code [11] or the correlation of tests failing/passing in previous test runs [15]. Symbolic execution has been used to derive more semantic properties based on the source code location of faults [31] and the longest prefix a failing trace shares with some passing trace [37]. The problem is that the suggested metrics are at best vague surrogates for the underlying faults. Using a model-based approach, we compare traces not against each other but against a ground truth (the PRD automaton).

Another related line of work is test case prioritization, test case selection, and test suite minimization [50]. Although formulated differently, these problems share the task of choosing tests from a predefined pool. Experiments have shown that manually chosen test suites outperform automatically selected ones in their ability to expose bugs [51]. To increase the number of tests that can be evaluated manually by an expert, the literature has proposed the use of clustering algorithms to group together tests with similar characteristics (so that the expert only has to evaluate clusters). The clustering is computed from syntactic information (a bitwise comparison of test executions). As argued before, we use semantic information and compute the classification wrt. a ground truth.

On the automatic side, [38] suggests the use of Hoare proofs to classify error traces. Our approach follows this idea and goes beyond it with the proposal of proof templates. Proof templates seem to be precisely the information needed to classify tests that are subject to real-time constraints. Harder et al. suggest to minimize test suites based on likely program invariants inferred from sample values obtained in test runs [22]. Hoare triples are more precise than invariants, even more so as we work with a ground truth rather than sample values.

Acknowledgements The results were obtained in the projects *"Virtual Test Analyzer I – III"*, conducted in collaboration with *IAV GmbH.* The last author is supported by a Junior Fellowship from the Simons Foundation (855328, SW).

Bibliography

[1] Abadi, M., Lamport, L.: An old-fashioned recipe for real time. In: REX Workshop. LNCS, vol. 600, pp. 1–27. Springer (1991)

[2] Alur, R., Dill, D.L.: A theory of timed automata. TCS **126**(2), 183–235 (1994)

[3] Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: POPL. pp. 97–105. ACM (2003)

[4] Becker, M., Meyer, R., Runge, T., Schaefer, I., van der Wall, S., Wolff, S.: Model-based fault classification for automotive software. CoRR **abs/2208.14290** (2022)

[5] Bringmann, E., Krämer, A.: Model-based testing of automotive systems. In: ICST. pp. 485–493. IEEE (2008)

[6] Christ, J., Ermis, E., Schäf, M., Wies, T.: Flow-sensitive fault localization. In: VMCAI. LNCS, vol. 7737, pp. 189–208. Springer (2013)

[7] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252. ACM (1977)

[8] Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL. pp. 84–96. ACM Press (1978)

[9] Craig, W.: Linear reasoning. A new form of the herbrand-gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957)

[10] Dickinson, W., Leon, D., Podgurski, A.: Finding failures by cluster analysis of execution profiles. In: ICSE. pp. 339–348. IEEE (2001)

[11] DiGiuseppe, N., Jones, J.A.: Concept-based failure clustering. In: SIGSOFT FSE. p. 29. ACM (2012)

[12] Dillig, I., Dillig, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. In: OOPSLA. pp. 443–456. ACM (2013)

[13] Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for esc/java. In: FME. LNCS, vol. 2021, pp. 500–517. Springer (2001)

[14] Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI. pp. 213–223. ACM (2005)

[15] Golagha, M., Lehnhoff, C., Pretschner, A., Ilmberger, H.: Failure clustering without coverage. In: ISSTA. pp. 134–145. ACM (2019)

[16] Golagha, M., Pretschner, A., Fisch, D., Nagy, R.: Reducing failure analysis time: An industrial evaluation. In: ICSE-SEIP. pp. 293–302. IEEE (2017)

[17] Golagha, M., Raisuddin, A.M., Mittag, L., Hellhake, D., Pretschner, A.: Aletheia: a failure diagnosis toolchain. In: ICSE (Companion Volume). pp. 13–16. ACM (2018)

[18] Groce, A.: Error explanation with distance metrics. In: TACAS. LNCS, vol. 2988, pp. 108–122. Springer (2004)

[19] Groce, A., Visser, W.: What went wrong: Explaining counterexamples. In: SPIN. LNCS, vol. 2648, pp. 121–135. Springer (2003)

[20] Guo, L., Roychoudhury, A., Wang, T.: Accurately choosing execution runs for software fault localization. In: CC. LNCS, vol. 3923, pp. 80–95. Springer (2006)

[21] Haase, V.: Real-time behavior of programs. IEEE Transactions on Software Engineering **SE-7**(5), 494–501 (1981)

[22] Harder, M., Mellen, J., Ernst, M.D.: Improving test suites via operational abstraction. In: ICSE. pp. 60–73. IEEE (2003)

[23] Haslbeck, M.P.L., Nipkow, T.: Hoare logics for time bounds - A study in meta theory. In: TACAS (1). LNCS, vol. 10805, pp. 155–171. Springer (2018)

[24] Hooman, J.: Extending hoare logic to real-time. Formal Aspects Comput. **6**(6A), 801–826 (1994)

[25] ISO: ISO 14229-1:2020 Road vehicles — Unified diagnostic services (UDS) — Part 1: Application layer. Standard ISO 14229-1:2020, International Organization for Standardization, Geneva, CH (2020)

[26] Jin, H., Ravi, K., Somenzi, F.: Fate and free will in error traces. In: TACAS. LNCS, vol. 2280, pp. 445–459. Springer (2002)

[27] Jordan, C.V., Hauer, F., Foth, P., Pretschner, A.: Time-series-based clustering for failure analysis in hardware-in-the-loop setups: An automotive case study. In: ISSRE Workshops. pp. 67–72. IEEE (2020)

[28] Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: PLDI. pp. 437–446. ACM (2011)

[29] King, J.C.: Symbolic execution and program testing. CACM **19**(7), 385–394 (1976)

[30] Leitner, A., Oriol, M., Zeller, A., Ciupa, I., Meyer, B.: Efficient unit test case minimization. In: ASE. pp. 417–420. ACM (2007)

[31] Liu, C., Han, J.: Failure proximity: a fault localization-based approach. In: SIGSOFT FSE. pp. 46–56. ACM (2006)

[32] Lyndon, R.: An interpolation theorem in the predicate calculus. Pacific Journal of Mathematics **9**, 129–142 (1959)

[33] McMillan, K.L.: Interpolation and sat-based model checking. In: CAV. LNCS, vol. 2725, pp. 1–13. Springer (2003)

[34] McMillan, K.L.: Interpolation and model checking. In: Handbook of Model Checking, pp. 421–446. Springer (2018)

[35] de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)

[36] Pan, R., Bagherzadeh, M., Ghaleb, T.A., Briand, L.C.: Test case selection and prioritization using machine learning: a systematic literature review. Empir. Softw. Eng. **27**(2), 29 (2022)

[37] Pham, V., Khurana, S., Roy, S., Roychoudhury, A.: Bucketing failing tests via symbolic analysis. In: FASE. LNCS, vol. 10202, pp. 43–59. Springer (2017)

[38] Podelski, A., Schäf, M., Wies, T.: Classifying bugs with interpolants. In: TAP@STAF. LNCS, vol. 9762, pp. 151–168. Springer (2016)

[39] Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., Wang, B.: Automated support for classifying software failure reports. In: ICSE. pp. 465–477. IEEE (2003)

[40] Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ASE. pp. 30–39. IEEE (2003)

[41] Rothermel, G., Harrold, M.J., Ostrin, J., Hong, C.: An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: ICSM. pp. 34–43. IEEE (1998)

[42] Schäuffele, J., Zurawka, T.: Automotive Software Engineering - Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen (6. Aufl.). Vieweg (2016)

[43] Schneider, F.B., Bloom, B., Marzullo, K.: Putting time into proof outlines. In: REX Workshop. LNCS, vol. 600, pp. 618–639. Springer (1991)

[44] Schwartz-Narbonne, D., Oh, C., Schäf, M., Wies, T.: VERMEER: A tool for tracing and explaining faulty C programs. In: ICSE (2). pp. 737–740. IEEE (2015)

[45] Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verification Reliab. **22**(5), 297–312 (2012)

[46] Wang, T., Roychoudhury, A.: Automated path generation for software fault localization. In: ASE. pp. 347–351. ACM (2005)

[47] Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Trans. Software Eng. **42**(8), 707–740 (2016)

[48] Wong, W.E., Horgan, J.R., London, S., Mathur, A.P.: Effect of test set minimization on fault detection effectiveness. Softw. Pract. Exp. **28**(4), 347–369 (1998)

[49] Wong, W.E., Horgan, J.R., Mathur, A.P., Pasquini, A.: Test set size minimization and fault detection effectiveness: A case study in a space application. J. Syst. Softw. **48**(2), 79–89 (1999)

[50] Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verification Reliab. **22**(2), 67–120 (2012)

[51] Yoo, S., Harman, M., Tonella, P., Susi, A.: Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In: ISSTA. pp. 201–212. ACM (2009)

[52] Zeller, A.: Isolating cause-effect chains from computer programs. In: SIGSOFT FSE. pp. 1–10. ACM (2002)

[53] Zhang, X., Gupta, N., Gupta, R.: Locating faults through automated predicate switching. In: ICSE. pp. 272–281. ACM (2006)