Explaining Violations of Properties in Control-Flow Temporal Logic

Joshua Heneage Dawes[1,2] and Giles Reger[1]

[1] University of Manchester, Manchester, UK
[2] CERN, Geneva, Switzerland

Abstract. Runtime Verification is the process of deciding whether a run of a program satisfies a given property. This work considers the more challenging problem of *explaining* why a run does or does not satisfy the property. We look at this problem in the context of CFTL, a low-level temporal logic. Our main contribution is a method for reconstructing representative execution paths, separating them into *good* and *bad* paths, and producing *partial parse trees* explaining their differences. This requires us to extend CFTL and our second contribution is a partial semantics used to identify the first violating observation in a trace. This is extended with a notion of *severity* of violation, allowing us to handle real-time properties sensitive to small timing variations. These techniques are implemented as an extension to the publicly available VYPR2 tool. Our work is motivated by results obtained applying VYPR2 to a web service on the CMS Experiment at CERN and initial tests produce useful explanations for realistic use cases.

1 Introduction

The Runtime Verification (RV) problem [5] is typically phrased as *given a run of a system (e.g. a trace) τ and a property φ, does τ satisfy φ?* Over the last 20 years many techniques and tools [6, 15, 16, 18, 20, 24] have been introduced to answer this question and most of these will answer *yes*, *no*, or some form of *maybe* but few attempt to explain *why* they return the given result. This work considers this challenge within the context of online monitoring of Control-Flow Temporal Logic [10, 11] (CFTL) properties but we argue that the approach generalises to the broader RV problem. As such, the technical details of the approach will use CFTL but we will comment on the general application of the idea throughout the paper. An advantage of using CFTL as a vehicle for this idea is that its semantics are already closely aligned with the control-flow of the monitored program, which will be useful when using this to explain violations.

CFTL is a low-level temporal logic with real-time constraints. Specifications in CFTL are written directly over program constructs (e.g. variable assignments and function calls) occuring within the scope of a single function. For example,

$$\forall t \in \mathsf{calls}(\texttt{save}) : \mathsf{duration}(t) \in [0, 10] \wedge \mathsf{dest}(t)(\texttt{result}) = 1$$

specifies that every local call to `save` should take no more than 10 seconds and the value of `result` afterwards should be 1. If a set of traces violate this property then it could either be due to the `save` call taking too long or the result being incorrect, we would like

to know which. Furthermore, if it were the former we would like to be able to quantify by *how much* we violate this time constraint. Lastly, but most importantly, there may be many calls to `save` in our function; we would like to know which one(s) are the source of the violations and which parts of the code contribute to the bad behaviour.

The general idea behind our approach is to take sets of violating and successful paths and abstract these to identify the parts of the function that influence the verdict. This requires two main steps. Firstly, we reconstruct paths through the monitored function from an observation trace. A key property of the monitoring approach for CFTL is that it admits an automatic and (in some sense) minimal instrumentation strategy. To reconstruct paths we modify this strategy to record sufficient information for full reconstruction. Secondly, once paths have been reconstructed, we generate *explanations* as generalised path objects (represented as context-free grammars) through a (symbolic) control flow graph of the monitored function. These explanations are further enhanced by a measure of the *severity* of time constraint violations. This work (and our previous work developing CFTL) is motivated by our experience working with engineers to apply runtime verification at the CMS Experiment at CERN.

We begin by introducing CFTL (Sec. 2) and an extension to partial traces (Sec. 3). We then describe a method for *reconstructing* paths (through the monitored function) from observation traces (Sec. 4), followed by our approach for producing *explanations* from sets of paths (Sec. 5). We then describe the implementation and an experiment demonstrating its use (Sec. 6). We conclude with related work and final remarks.

2 Control-Flow Temporal Logic (CFTL)

In this section we introduce the main concepts behind CFTL but refer the reader to previous work [9–11] for further details and formal definitions. CFTL is a linear-time temporal logic whose formulas reason over two central types of objects: *states*, instantaneous *checkpoints* in a program's runtime; and *transitions*, the computation that must happen to move between states.

2.1 CFTL formulas

CFTL specifications take prenex form e.g. a list of quantifiers followed by a boolean combination of *atoms*, which are expressions (possibly containing temporal operators) over states and transitions. Given a CFTL formula φ, we use Vars_φ to refer to the set of quantified variables in φ, $A(\varphi)$ to refer to the set of atoms in φ and, for $\alpha \in A(\varphi)$, we use $\mathsf{var}(\alpha)$ for the variable on which α is based. At the top-level, atoms are assertions over the value of variables in a state or duration of a transition. States and transitions within atoms either come from the outer quantification or from temporal operators that find the next state or transition satisfying a given constraint.

As another example of a CFTL formula consider

$$\forall q \in \mathsf{changes}(\texttt{var}) : q(\texttt{var}) = \texttt{True} \implies \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(\texttt{func}))) \in [0, 1].$$

which captures the property "for every change to `var`, if the new value is `True`, then the next call to the function `func` should take no more than 1 second".

2.2 What CFTL formulas mean

The semantics of CFTL formulas are defined over *dynamic runs*, which are sequences of observations with each observation relating directly to a point in the monitored function taken from the function's *symbolic control-flow graph*. It is important to note that a CFTL formula is defined in terms of elements in the control-flow graph and this is needed to understand the meaning of the formula. This close relation to the control-flow graph is what makes instrumentation points easy to define and helps later in explanation.

Symbolic Control-Flow Graphs. Given a program P (we assume a basic language with assignments, conditionals, and loops, see [10][3]), its Symbolic Control-Flow Graph $\mathsf{SCFG}(P)$ captures 1) the change in state generated by variable assignments and function calls in programs and 2) reachability in control-flow.

We associate with each node in the abstract syntax tree of P a unique *program point* and let Sym be a set of symbols representing variables and functions in P. Then, a *symbolic state* σ is a pair $\langle p, m\rangle$ where p is a program point and m is a map (partial function with finite domain) from symbols to the set of statuses $\{\text{changed}, \text{unchanged}, \text{called}, \text{undefined}\}$. We abuse notation and denote by $\sigma(x)$ the value to which m maps x. SCFGs are then directed graphs with symbolic states as vertices.

Definition 1. *A symbolic control-flow graph (SCFG) is a directed graph $\langle V, E, v_s\rangle$ with a finite set of symbolic states V, a finite set of edges $E \subseteq V \times V$, and an initial symbolic state $v_s \in V$.*

A symbolic state σ is *final* if it has no successors e.g. there is no edge $\langle \sigma, \sigma'\rangle$ in E. A *path* π through $\mathsf{SCFG}(P) = \langle V, E, v_s\rangle$ is a finite sequence of symbolic states $\sigma_1, \ldots, \sigma_k$ such that for every pair of adjacent symbolic states σ_i, σ_{i+1} there is an edge $\langle \sigma_i, \sigma_{i+1}\rangle$ in E. A path is *complete* if $\sigma_1 = v_s$ and σ_k is final. Our previous work [10] gave a construction to generate $\mathsf{SCFG}(P)$ for a given program P.

Dynamic Runs. A *dynamic run* is an abstraction of a run of a program P and is associated with a path through $\mathsf{SCFG}(P)$. Let Val be the finite set of possible values to which the elements of Sym can be mapped at runtime. Then a *concrete state* is a triple $\langle t, \sigma, \tau\rangle$ for timestamp $t \in \mathbb{R}^{\geq}$, symbolic state σ and a map $\tau : \mathsf{Sym} \rightharpoonup \mathsf{Val}$ from program symbols to values. A dynamic run $\mathcal{D}$ is a finite sequence of such concrete states.

Definition 2. *A* dynamic run *over SCFG(P) $= \langle V, E, v_s\rangle$ is a finite sequence $\mathcal{D} = \langle t_1, \sigma_1, \tau_1\rangle, \ldots, \langle t_n, \sigma_n, \tau_n\rangle$ such that timestamps t_i are strictly increasing, $\sigma_1 = v_s$, σ_n is final, and there is a path in SCFG(P) between every pair of symbolic states σ_i and σ_{i+1} i.e. $\sigma_1, \ldots, \sigma_n$ can be extended to a complete path.*

A *transition* is a pair of concrete states. A transition $\langle\langle t, \sigma, \tau\rangle, \langle t', \sigma', \tau'\rangle\rangle$ is *atomic* if $\langle \sigma, \sigma'\rangle$ is an edge in $\mathsf{SCFG}(P)$. A dynamic run $\mathcal{D}$ is *most-general* if every transition is atomic. Later we will restrict the condition on σ_n to get *partial dynamic runs*, therefore we sometimes refer to these as *total*. Given some concrete state $q = \langle t, \sigma, \tau\rangle$, the *symbolic support* $\mathsf{support}(q)$ of q is symbolic state σ. Similarly for a transition $tr = \langle\langle t, \sigma, \tau\rangle, \langle t', \sigma', \tau'\rangle\rangle$, the symbolic support $\mathsf{support}(tr)$ is the edge $\langle \sigma, \sigma'\rangle$.

[3] Practically, in our implementation we target a subset of Python.

Evaluation of CFTL formulas. The semantics of a CFTL formula φ with respect to a dynamic run $\mathcal{D}$ is defined by iterating over the quantifiers of φ (by the well-formedness criteria [10] there must be at least one) to generate sets of *points of interest*, and then evaluating the inner part of the formula at each of these points. This evaluation relies on an eval function that gives the next state or transition satisfying a state or transition constraint. More formally, given a dynamic run $\mathcal{D}$, a point of interest θ, and an expression Exp, $\mathsf{eval}(\mathcal{D}, \theta, \mathsf{Exp})$ gives the unique concrete state or transition to which Exp refers based on $\mathcal{D}$ and θ. In the total semantics (for total dynamic runs) this is guaranteed to exist. A full definition is omitted here but can be found in previous work [10].

2.3 Instrumentation and Observations

Given $\mathsf{SCFG}(P) = \langle V, E, v_s \rangle$, instrumentation is the process of choosing a subset $\mathsf{Inst} \subset V$ of *instrumentation points* such that φ can be evaluated given concrete states and transitions whose symbolic supports are the elements of Inst. One could of course take $\mathsf{Inst} = V$, but the intention of instrumentation is to throw away concrete states from a most-general dynamic run which are not needed to monitor φ, thus reducing work for a monitoring algorithm.

To compute Inst, we first inspect the quantification sequence of φ to determine which symbolic states could generate *points of interest*. We call such symbolic states *candidate points of interest*. For example, a formula with quantification sequence $\forall q \in \mathsf{calls}(f)$ would lead us to identify all pairs of symbolic states that may correspond to calls to the function f in a dynamic run. Once these points of interest are obtained, we then inspect the quantifier-free part of φ to determine the actual instrumentation points.

We then use Inst to filter $\mathcal{D}$ to give a second dynamic run $\mathcal{D}'$ whose concrete states are only those with symbolic supports in Inst. This filtered dynamic run has the property that $\mathcal{D}, [] \models \varphi$ if and only if $\mathcal{D}', [] \models \varphi$ (see [10]). Given a computed set Inst and a (not necessarily most-general) dynamic run $\mathcal{D}$, we call a concrete state or transition an *observation* if its symbolic support is in Inst. We call any concrete state/transition that is not an observation *redundant with respect to* φ.

2.4 What Matters for Explanation?

If we wanted to replace CFTL with another specification language we would need to ensure that there is a direct correspondence between the assertions/predicates in that language and points in the SCFG. Properly defined, this should be compatible with the above notion of redundancy. To take advantage of the concept of *verdict severity* the language should contain real-time constraints.

3 Identifying Failing Observations

Later we will explain violations using paths through the associated SCFG to the observation causing the failure. To do this we assume that the property is a safety property (CFTL only captures safety properties), e.g. all violations are witnessed by finite prefixes, and the semantics for incomplete runs is *impartial* [17] e.g. the verdict cannot

change from true to false or vice versa with more information. In the following we outline a partial semantics for CFTL that holds the impartiality property. We also give a quantitative extension of this semantics called *verdict severity*, which will be helpful later when determining the extent to which a violation has occurred.

3.1 Partial Semantics for CFTL

Our aim is to identify the observation in a most-general dynamic run $\mathcal{D}$ after which a CFTL formula φ can no longer be satisfied. We call such an observation a *falsifying observation*. If $\mathcal{D}$ violates φ there is exactly one such observation. Unfortunately, CFTL semantics is defined over total dynamic runs corresponding to *complete* paths through SCFGs, hence we have no way to talk about such an observation. We now describe a partial semantics over dynamic runs that do not finish at final symbolic states in SCFGs.

We define a *partial dynamic run* as a dynamic run where the last symbolic state is not final in the SCFG. This ensures that the semantics is well defined: the total semantics (with truth domain $\{\mathsf{true}, \mathsf{false}\}$) should be used for total dynamic runs.

The first change required is to update the evaluation function (which returns the unique concrete state or transition corresponding to an expression) so that it is partial and returns null when the expression cannot be evaluated. For example, for the property

$$\forall q \in \mathsf{changes}(\texttt{var}) : q(\texttt{var}) = \texttt{True} \implies \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(\texttt{func}))) \in [0, 1].$$

when evaluating $\mathsf{next}(q, \mathsf{calls}(\texttt{func}))$ we would get null if there is no next transition satisfying the condition. Note that this extended evaluation function coincides with the original function for total dynamic runs.

Once the evaluation function has been updated it is then necessary to update the way that points of interest are defined so that they can also be partial. For example, if the quantification were $\forall q \in \mathsf{changes}(\texttt{var}) : \forall t \in \mathsf{future}(q, \mathsf{calls}(f))$ we would include a partial quantification for all satisfying states q even if no satisfying transitions t exist.

Finally, we get a partial semantics with a truth domain $\{\mathsf{false}, \mathsf{notSure}, \mathsf{trueSoFar}\}$. Expressions are evaluated as false if their value is known and they are false, otherwise (if their value is known and it is true) we get trueSoFar. If the expression cannot be fully evaluated then the result is notSure. For the above example, if $\mathsf{next}(q, \mathsf{calls}(\texttt{func}))$ evaluates to null then the atom $\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(\texttt{func}))) \in [0, 1]$ would evaluate to notSure. The truth domain has the ordering $\mathsf{false} < \mathsf{notSure} < \mathsf{trueSoFar}$ with $\neg\mathsf{false} \equiv \mathsf{trueSoFar}$, and $\neg\mathsf{notSure} \equiv \mathsf{notSure}$. We take $\sqcap$ to be the greatest lower bound with respect to this order, and $\sqcup$ to be the least upper bound. This can be used to interpret the boolean operators in the language as expected e.g. using $\sqcup$ for $\vee$.

As soon as a dynamic run is extended to be total, its satisfaction of some CFTL formula φ is subject to the normal semantics with the truth domain $\{\mathsf{true}, \mathsf{false}\}$. To ensure well-definedness on which semantics to use, we consider dynamic runs before filtering by instrumentation (this could remove the last concrete state corresponding to a final symbolic state in the SCFG).

The resulting partial semantics holds the *verdict impartiality* property [17]: true can never be declared for a dynamic run identifying with a path that is not complete because extensions to a complete path can introduce new points of interest that cause

violations. In addition, the partial semantics will be able to give a false verdict since CFTL formulas are universally quantified, hence a single falsifying observation means false for every possible extension of the dynamic run (they are safety properties).

With a partial semantics defined for partial dynamic runs, we can now isolate the observation in a total dynamic run that prevented satisfaction of a property φ. We do this by taking a total dynamic run and extracting a partial dynamic run whose final state causes the partial semantics to switch to the false verdict.

Given a total dynamic run $\mathcal{D}$ let $\mathcal{D}_p(q)$ be the partial dynamic run that is the prefix of $\mathcal{D}$ ending with concrete state q. For a CFTL formula φ such that $\mathcal{D}, [] \not\models \varphi$, the *falsifying observation* is $q \in \mathcal{D}$ such that:

1. $[\mathcal{D}_p(q), [] \models \varphi] = \mathsf{false}$.
2. Given the previous state q' in $\mathcal{D}$ (if it exists), $[\mathcal{D}_p(q'), [] \models \varphi] = \mathsf{trueSoFar}$ or $[\mathcal{D}_p(q'), [] \models \varphi] = \mathsf{notSure}$.

It suffices to consider the previous state due to the impartiality of the partial semantics.

3.2 Verdict Severity

Later we will divide a set of runs into *good* and *bad*. This can be trivially done based on whether the runs satisfy the given property or not. However, for timing properties things are not so clear-cut; perhaps small deviations are acceptable, or more likely, the more problematic violations are grouped with less problematic ones. To handle this situation we introduce a quantitative extension to our semantics that uses a notion of *severity* of violation such that a negative severity means violation and a positive one means success, with the magnitude indicating the level to which this verdict is reached. In essence, this gives a metric of *by how much* some function call was or was not a falsifying observation. As mentioned later, this can also be used to decide whether a path is only a borderline satisfaction/violation i.e. whether it could be included in the paths that generated the opposite verdict.

Given an observation c it is always possible to identify the atom α that is evaluated for c (indeed, our monitoring algorithm makes this explicit). We define the verdict severity of c with respect to this atom α.

Definition 3. *Given an observation c evaluated at atom α, the* verdict severity $\mathsf{Sev}(\alpha, c)$ *is 1 if the atom is satisfied and -1 otherwise, with the exception of the case where $\alpha = (\mathsf{duration}(t) \in I) \in A_\varphi$ for some finite, bounded $I \subset \mathbb{R}_{>0}$, in which case*

$$\mathsf{Sev}(\alpha, c) = \inf\{|\mathsf{duration}(c) - n| : n \in I\} \cdot \mathcal{X}(\alpha, c)$$

such that $\mathcal{X}(\alpha, c) = 1$ if $\mathsf{duration}(c) \in I$, -1 otherwise.

The term $\mathcal{X}(\alpha, c)$ allows us to differentiate between satisfaction and violation of the constraint, and the infimum captures by how much.

As an example, consider again the property

$$\forall q \in \mathsf{changes}(\texttt{var}) : q(\texttt{var}) = \texttt{True} \implies \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(\texttt{func}))) \in [0, 1].$$

which can only be violated by breaking the duration constraint. If the duration of the failing transition were 2 then the severity would be -1, whereas if it were 1.3, then the severity would be -0.3.

4 Path Reconstruction

As discussed previously, the first step in generating explanations is to reconstruct the path through the monitored program that leads to the observation we want to explain (often the falsifying observation, which we showed how to identify for CFTL in the previous section). Note that later we may also want to reconstruct paths for satisfying runs, so this section talks about reconstructing paths for an observation in general.

Given an observation and a symbolic control-flow graph $\mathsf{SCFG}(P)$, we consider the task of deciding precisely which path was taken through $\mathsf{SCFG}(P)$ to reach the observation. When the dynamic run given is most-general, this is straightforward, but if it has been filtered by instrumentation, it is not necessarily possible.

Let $\mathsf{SCFG}(P) = \langle V, E, v_s \rangle$ be the symbolic control-flow graph of P and q be an observation in a most-general dynamic run $\mathcal{D}$ over $\mathsf{SCFG}(P)$. We consider the dynamic run $\mathcal{D}'$ obtained by removing concrete states that are redundant with respect to a formula φ, and what can be done to determine the path taken through $\mathsf{SCFG}(P)$ by $\mathcal{D}'$. Given that $\mathcal{D}'$ contains only the concrete states required to check φ, it is clear that, for consecutive concrete states $\langle t, \sigma, \tau \rangle, \langle t', \sigma', \tau' \rangle$, there may be multiple paths between σ and σ'. This means there may not be enough information in $\mathcal{D}'$ to decide the exact path taken. We therefore introduce the notion of a *branch-aware dynamic run*, which is a dynamic run whose concrete states allow the exact path taken to be reconstructed.

Definition 4. *A dynamic run $\mathcal{D}_b$ is a* branch-aware dynamic run *if between any two consecutive concrete states $\langle t, \sigma, \tau \rangle$ and $\langle t', \sigma', \tau' \rangle$, there is a single path from σ to σ'.*

Let us denote the set of concrete states added to some $\mathcal{D}$ to make it branch-aware by $\mathsf{branching}(\mathcal{D}_b)$. If there is no branching in P, possibly $\mathsf{branching}(\mathcal{D}_b) = \emptyset$. The concrete states in $\mathsf{branching}(\mathcal{D}_b)$ are considered redundant with respect to φ since they are added after instrumentation allowed removal of states.

The most obvious example of a branch-aware dynamic run is a most-general dynamic run, since all transitions correspond to single edges. However, given that making a dynamic run branch-aware makes additional instrumentation necessary, a most-general dynamic run is not economical.

A dynamic run $\mathcal{D}_b$ is *minimally branch-aware* if it is branch-aware and there is no concrete state which is redundant with respect to φ and whose removal would not stop $\mathcal{D}_b$ being branch-aware. This definition captures the intuition that a minimally branch-aware dynamic run should have additional concrete states placed in strategic places.

4.1 Instrumentation for Branch-Aware Dynamic Runs

To make $\mathsf{branching}(\mathcal{D}_b)$ minimal we determine the minimal set of symbolic states $\mathsf{SCFG}(P)$ which will be the symbolic supports of the elements of $\mathsf{branching}(\mathcal{D}_b)$. We now present a strategy for determining such a set of symbolic states.

There are multiple structures that result in multiple possible directions for a path to take. For conditionals the branch taken can be determined if the first vertex on that branch is known, hence we instrument the first vertex on each branch. Further, it is necessary to instrument the first vertex after the branches have converged, distinguishing

Algorithm 1 Path reconstruction algorithm given a minimally branch-aware dynamic run $\mathcal{D}_b$ and a symbolic control-flow graph $\mathsf{SCFG}(P) = \langle V, E, v_s \rangle$.

```
1: π ← ⟨⟩                                                    ▷ Initialise an empty path.
2: branchingIndex ← 0 ▷ Initialise the index of the element of branching(D_b) to be used next.
3: curr ← v_s
4: while branchingIndex < |branching(D_b)| do
5:    if ∃⟨curr, σ⟩ ∈ outgoing(curr) : σ = support(L(branchingIndex)) then
6:       curr ← the σ from ⟨curr, σ⟩
7:       branchingIndex += 1
8:    else
9:       curr ← the σ such that there is ⟨curr, σ⟩ ∈ E
10:   π += ⟨curr, σ⟩
```

observations from inside and after the body. For loops we instrument the first vertex of the loop body (to capture the number of iterations) and the post-loop vertex. For try-catch blocks we insert instruments at the beginning of each block but, so far, we have no efficient way to capture the jump from an arbitrary statement inside the try block to the catch block. However, it would be possible to use an error trace to determine the statement at which the exception was thrown, and use this in path reconstruction.

Applying this method for instrumentation to the entire SCFG gives a minimally branch-aware dynamic run, e.g. one whose path we can reconstruct, using a small and conservative set of new instrumentation points.

4.2 Computing Reconstructed Paths

In order to finally determine the path taken to reach some observation we step through the (minimally) branch-aware dynamic run collecting the relevant symbolic states. Algorithm 1 takes a minimally branch-aware dynamic run $\mathcal{D}_b$ with n concrete states and a symbolic control-flow graph $\mathsf{SCFG}(P)$, and reconstructs the path taken by $\mathcal{D}_b$ as a sequence of edges. It makes use of:

- A labelling $L(i)$ on $\mathsf{branching}(\mathcal{D}_b)$ giving the i^{th} concrete state with respect to timestamps and $L(0)$ being the first concrete state.
- A function $\mathsf{outgoing}(\sigma)$ which gives $\{\langle \sigma, \sigma' \rangle \in E\}$.

The intuition is that we follow edges in the symbolic control-flow graph until we arrive at a symbolic state at which branching occurs. At this point, we use $\mathsf{branching}(\mathcal{D}_p)$ to decide on which direction to take.

4.3 What Matters for Explanation?

If we wanted to replace CFTL with another specification language we would need to solve the path reconstruction problem separately. The main challenge would be to ensure that enough information is captured in the recorded dynamic run for path reconstruction. This is made easier in CFTL as the semantics is defined in terms of the SCFG.

5 Explaining Verdicts with Paths

We now consider the following problem: given a set of dynamic runs (for a CFTL formula φ over a single symbolic control-flow graph $\mathsf{SCFG}(P)$) containing a violating dynamic run $\mathcal{D}^v$ how can we explain what makes this run a *bad* run?

Our first step is identify the other dynamic runs in our original set that follow the same edge in $\mathsf{SCFG}(P)$. Let $c^\perp$ be the falsifying observation for $\mathcal{D}^v$ and consider $\mathsf{support}(c^\perp)$, the edge in $\mathsf{SCFG}(P)$ corresponding to $c^\perp$. To save space, we consider only the case where $c^\perp$ is the only transition with this symbolic support (ie, it is not inside a loop). Let $\mathcal{F} = \mathcal{D}_1, \ldots, \mathcal{D}_n$ be the set of dynamic runs containing an observation c_i such that $\mathsf{support}(c_i) = \mathsf{support}(c^\perp)$. To explain the violating run $\mathcal{D}^v$ (which must appear in $\mathcal{F}$), we will take the reconstructed paths up to each observation c_i and compare them. Note that for satisfying runs we will only examine the behaviour up until the corresponding observation.

We use the notion of verdict severity (or if there are no timing constraints, just the satisfaction relation) to separate these paths. Let $C_\top$ be the set of pairs $\langle c_i, \mathsf{Sev}(\alpha, c_i)\rangle$ such that $\mathsf{Sev}(\alpha, c_i) \geq 0$ for each c_i. We define $C_\perp$ similarly, but with $\mathsf{Sev}(\alpha, c_i) < 0$. Using these sets, we will reconstruct paths up to each observation and associate each path with the verdict severity to which it leads. The differences between these two sets will then be used to construct the explanation.

5.1 Reconstructed Paths as Parse Trees

We now consider what all the paths in $C_\top$ and $C_\perp$ have in common. If there are common characteristics, we may conclude that such characteristics could affect the verdict. This requires us to represent reconstructed paths (computed by Algorithm 1) in a concise way that makes it easy to isolate divergent behaviours. Our solution is to derive a context-free grammar from the SCFG and use this to parse the paths and then compare the parse trees. This representation will allow comparison of path characteristics such as branches taken and number of iterations completed by loops. Our approach is similar to the standard approach to deriving context free grammars from finite state automata, with the major difference being that we recognise that there are commonly found structures in symbolic control-flow graphs, such as conditionals and loops. Such structures are used to generate grammars that yield parse trees which make it easy to compare path characteristics.

Figure 1 gives a detailed schema for deriving these grammars. For each *component* of a SCFG we give the corresponding generated grammar. The grammar of an entire SCFG can be constructed by recursively applying this schema. An application of this is illustrated in Figure 2, which shows a SCFG on the left with a grammar derived on the right. Non-terminal symbols (symbolic states) are written in bold. The grammar on the right works by mapping symbolic states in the SCFG to sequences of edges and other symbolic states. Symbolic states are always non-terminal, so any path generated by such a grammar is a sequence of edges. The difference between our approach to deriving a grammar vs the traditional approach is reflected in the right hand side of rule $\boldsymbol{\sigma_1}$. Using this fact that all complete paths through the SCFG must pass through the

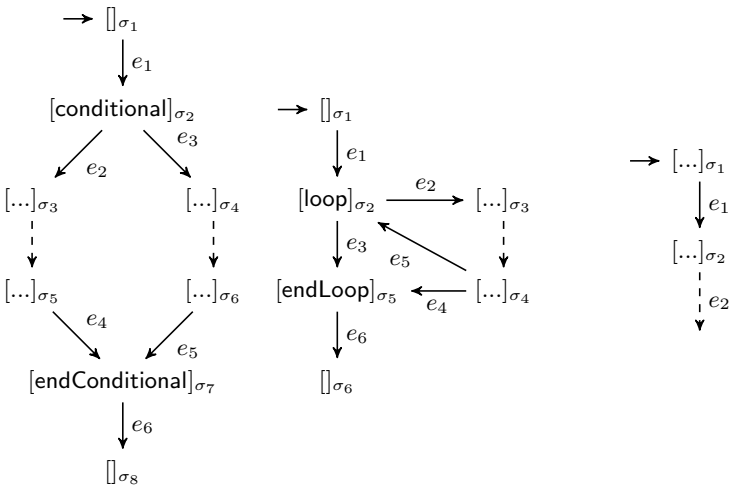

Fig. 1. A subset of the full schema for deriving a context free grammar from a SCFG.

edge e_7 to exit the loop, we encode loops in grammars by using one rule for the loop body, and another for the post-loop control-flow.

Once we have constructed these parse trees we want to find commanilities between them. Figure 3 shows how we can take parse trees from multiple paths and form the *intersection*. We define the intersection of two parse trees $T(\pi_1)$ and $T(\pi_2)$, written $T(\pi_1) \cap T(\pi_2)$, by the parse tree which contains a subtree if and only if that subtree is found in both $T(\pi_1)$ and $T(\pi_2)$ at the same path. Intersection is given by the recursive definition in Figure 4. In this definition, a subtree is a pair $\langle r, \{h_1, \ldots, h_n\}\rangle$ for root r and child vertices $h_1, \ldots, h_n$, and the empty tree is denoted by null. The base case of recursion is for leaves l and l'.

We abuse notation and write $\pi_1 \cap \pi_2$ for the path obtained by reading the leaves from left to right from the intersection of the parse trees $T(\pi_1)$ and $T(\pi_2)$. If such a path contains symbolic states, we call it a *parametric path* and call a symbolic state contained by such a path a *path parameter*. In particular, the vertex to which this symbolic state corresponds in the intersection parse tree is given different subtrees by at least two parse trees in the intersection. The values given to those parameters by each path in

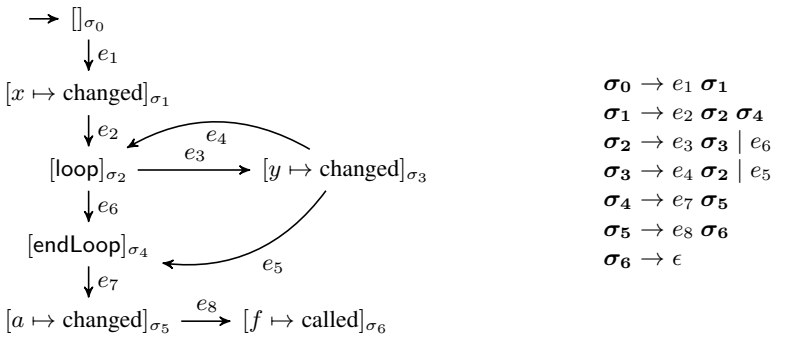

Fig. 2. A symbolic control-flow graph and its context free grammar.

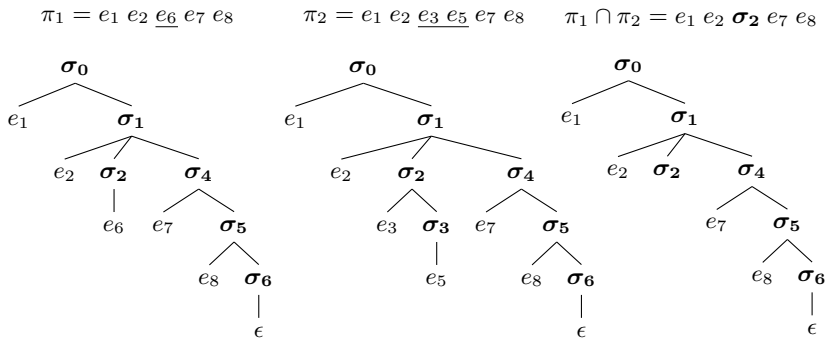

Fig. 3. The parse trees of two paths, and their intersection.

the intersection can be determined by following the path to the path parameter's vertex through each parse tree.

5.2 Representing Paths up to Observations

So far we have seen how one can represent *complete* paths through symbolic control-flow graphs, however paths up to symbolic states or edges that are symbolic supports of observations are rarely complete. We choose to represent paths that are not complete as *partially evaluated parse trees*, that is, parse trees which still have leaves which are non-terminal symbols. Further, we denote the path up to an observation q by $\pi(q)$.

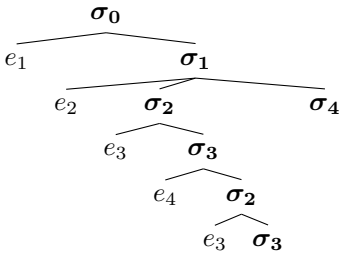

Fig. 5. A partial parse tree.

As an example, consider the path $e_1\ e_2\ e_3\ e_4\ e_3$ through the symbolic control-flow graph in Figure 2. This path is not complete, since it does not end with the edge e_8,

$$\mathsf{intersect}(l, l') = \begin{cases} \mathsf{null} & \text{if } l \neq l' \\ l & \text{otherwise} \end{cases}$$

$$\mathsf{intersect}(\langle r, \{h_1, \ldots, h_n\}\rangle, \langle r', \{h'_1, \ldots, h'_m\}\rangle) = \begin{cases} \langle r, \{\ldots \mathsf{intersect}(h_i, h'_i) \ldots\}\rangle & \text{if } r = r' \wedge n = m \\ r & \text{if } r = r' \wedge n \neq m \\ \mathsf{null} & \text{if } r \neq r' \end{cases}$$

Fig. 4. A recursive definition of parse tree intersection.

hence its parse tree with respect to the context free grammar must contain non-terminal symbols. Figure 5 shows its partial parse tree.

5.3 Producing Explanations

We show how we can use intersections of the paths up to observations c_i of pairs $\langle c_i, \mathsf{Sev}(\alpha, c_i)\rangle$ in our *good*, $C_\top$, and *bad*, $C_\bot$, sets to determine whether certain parts of code may be responsible for violations or not. This intersection is our *explanation*. In the case that the path taken is not likely to be responsible, we give an alternative method that is a sensible approach for our work at the CMS Experiment at CERN.

For each $\langle c_i, \mathsf{Sev}(\alpha, c_i)\rangle \in C_\top$, we compute $\pi(c_i)$, the path taken by $\mathcal{D}_i$ to reach $\mathsf{support}(c_i)$. We then form the intersection of the parse trees $\bigcap_{c_i \in C_\top} T(\pi(c_i))$. Any path parameter in the resulting intersection tells us that this part of the path is unlikely to contribute to the verdict reached by the observations c_i which are not falsifying. We draw a similar conclusion when we compute the parse trees for $c_i \in C_\bot$; any parts of the paths that disagree across dynamic runs in which an observation was always falsifying are unlikely to affect the verdict.

Verdict severity is useful if there are multiple paths $\pi(c_i)$ which disagree with all others on a specific path parameter, but which have verdict severity $\mathsf{Sev}(\alpha, c_i)$ close to 0, ie, borderline. In these cases, we could move the associated runs to the other set and redo the analysis e.g. reclassify a run that should not contribute to a particular class.

In the case of disagreement of values of path parameters for the same verdict, we capture input parameters of the relevant function calls. The space of maps from input parameters to their values will then give us an indication of whether state, rather than control flow, contributed to a verdict. For example, all violations may occur when an input variable is negative. If neither factor shows to affect the verdict, in the cases we have dealt with so far at the CMS Experiment at CERN, it is reasonable to conclude that external calls (e.g. network operations) are a contributing factor.

6 Implementation in VyPR2

We have implemented our explanation technique as an extension of the VyPR2 framework [11] (http://cern.ch/vypr). The code used to perform the analysis in this section is found at http://github.com/pyvypr/. The necessary modifications to VyPR2 included

```
1  def f(l):
2    for item in l:
3      time.sleep(0.1)
4
5  @app.route(
6    '/test/<int:n>/',
7    methods=["GET", "POST"]
8  )
9  def test(n):
10   a = 10
11   if n > 10:
12     l = []
13     for i in range(n):
14       l.append(i**2)
15     print(''test'')
16   else:
17     l = []
18   f(l)
19   return "..."
```

```
1  "app.routes" : {
2    "test" : [
3      Forall(
4        s = changes('a')
5      ).\
6      Check(
7        lambda s : (
8          If(
9            s('a').equals(10)
10         ).then(
11           s.next_call('f').\
12             duration()._in([0, 1])
13         )
14       )
15     )
16   ]
17 }
```

Fig. 6. The program (left) and PyCFTL specification (right) we use to demonstrate our path comparison approach.

1) additional instrumentation, to make the dynamic run derived by VYPR2 *branch-aware*; 2) changes to the relational verdict schema to allow detailed querying of the data now being obtained by VYPR2 during monitoring; and 3) path reconstruction and comparison tools to allow construction of explanations.

We now demonstrate how our prototype implementation can be used on a representative program to conclude that one branch is more likely to cause violation than another when an observation generated after the branches converge is checked. Work is currently underway at the CMS Experiment at CERN to build an analysis library, since everything in the remainder of this section required custom scripts.

A representative program and PyCFTL specification are given in Figure 6. Since the current implementation of VYPR2 works with Python-based web services that are based on the Flask [1] framework (this being a commonly used framework at the CMS Experiment), the code in Figure 6 is typical of the code seen in a Flask-based web service. In this case, the result of the function `test` is returned when the URL `test/n/`, where `n` is a natural number, is accessed.

6.1 Performing an Analysis

Path Reconstruction This step generates sequences of edges in the SCFG of the function being monitored using Algorithm 1. The minimal amount of information is stored in the verdict database to perform such reconstruction by storing observations and *mapping* them to the previous branching condition that was satisfied to reach them. This way, the complexity of the specification has no effect on the efficiency of path reconstruction. The results of this step are not visible to the user.

Path Comparison Our initial implementation of path comparison processes all recorded dynamic runs, and then focuses on observations generated by the call to f at line 18 in Figure 6. These observations are grouped into two sets; those generating the false

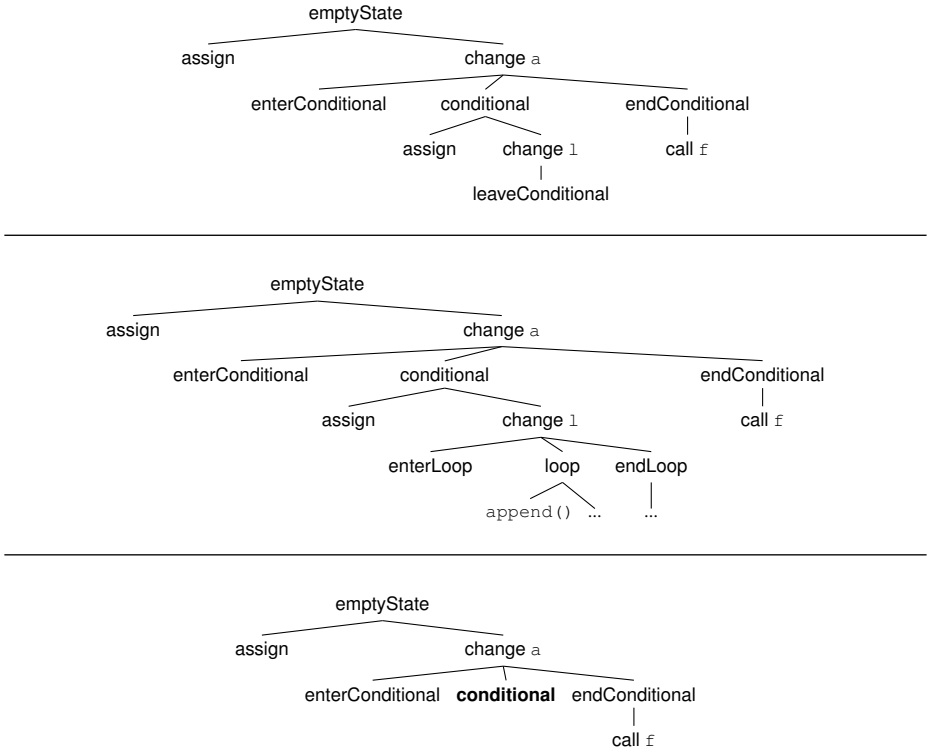

Fig. 7. The intersection (bottom) of the good (top) and bad (middle) path representatives, with the single path parameter highlighted in bold.

verdict, and those generating trueSoFar. Figure 7 (top) shows the intersection of all parse trees derived from observations that generated trueSoFar, and Figure 7 (middle) those that generated false. We call these parse trees *representatives*.

Finally, in the implementation used for these experiments, Figure 7 (bottom) is the result of our analysis. It shows the intersection of the representative parse trees from trueSoFar and false verdicts with the single *path parameter* highlighted in bold. It is clear from the path parameter that variation in paths is found at an if-statement, and the branch taken here may affect the verdict, since the representatives show that good and bad paths follow single paths.

Our notion of verdict severity would come in useful if there were observations on the borderline of a verdict that were stopping us from obtaining a single path parameter value for all paths giving a single verdict. We could redo our analysis, counting the anomalous observation(s) as contributing to different verdicts.

6.2 Performance

The overhead induced by additional instrumentation for path reconstruction has been shown to be minimal. When the program in Figure 6 traverses the branch starting at line

17, the overhead is large (approximately 43%), but we observe, exactly as in [10], that this is due to Python's Global Interpreter Lock preventing truly asynchronous monitoring. When the branch starting at line 12 is traversed, the overhead becomes negligible because `time.sleep` is called, which is thread-local.

Our implementation of offline path comparison (including path reconstruction, context free grammar construction, parse tree derivation and intersection) has shown to scale approximately quadratically with the number of paths. Reconstruction and comparison of 400 paths (200 good, 200 bad) took approximately 3.6 seconds, while 1000 paths (500 good, 500 bad) took approximately 16.6 seconds. These measurements were taken by turning off many of the writes to disk used to store SCFGs and parse trees, since these are performed by the `graphviz` [2] library. We observe that it is possible to use previously computed parts of an intersection since intersection is commutative and that deriving the parse trees of multiple paths with respect to a context free grammar is parallelisable, with a possibly small increase in memory usage.

7 Related Work

An alternative approach to explaining violations measures the *distance* between a violating trace and the language of the specification [22, 3, 19]. Other work considers error traces and what changes could be made to prune violating extensions [8]. The idea is that, if a fix can be found that prunes all possible erroneous extensions, the code to be fixed could be regarded as a fault. This work lies inside the general field of Fault Localisation [26], where much work has been done, including Spectra-based [25] and Model-based [21] approaches. Our work differs from the existing work in that we consider *faults* to be potentially problematic control flow with respect to CFTL formulas.

Reconstructing paths is also not a new idea [4], where some approaches have compared paths [23]. Our work differs in its context free grammar-based comparison, and the subsequent use to construct explanations of violations of CFTL specifications.

Much work has been done on explanation in the Model Checking community. For example, finding the closest satisfying traces to the set of counterexamples and describing the differences [14] or localising the parts of an input signal violating the property [13, 7]. There has also been work quantifying the degree of severity of a violation/satisfaction [12]. Although the setting is different (in RV we deal with concrete runs), there are similarities with our approach, which will be explored in the future.

8 Conclusion

We introduced a new partial semantics for CFTL that allows isolation of the observation that causes a CFTL formula to evaluate to false. Following that, we extended the notion of dynamic runs to define *branch-aware dynamic runs* which allow reconstruction of the execution path of a program as a path through its symbolic control-flow graph. Finally, we gave our approach for comparing paths using context free grammars. Implementation of this approach in VyPR2 allows construction of explanations based on comparison of the paths taken with respect to verdicts generated.

Our next step is already underway: we are developing analysis tools for VyPR2, with services used at the CMS Experiment at CERN serving as use cases.

References

1. Flask for Python. http://flask.pocoo.org
2. Graphviz for Python. https://graphviz.readthedocs.io/en/stable/
3. Babenko, A., Mariani, L., Pastore, F.: Ava: Automated interpretation of dynamically detected anomalies. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. pp. 237–248. ISSTA '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1572272.1572300, http://doi.acm.org/10.1145/1572272.1572300
4. Ball, T., Larus, J.R.: Efficient path profiling. In: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture. pp. 46–57. MICRO 29, IEEE Computer Society, Washington, DC, USA (1996), http://dl.acm.org/citation.cfm?id=243846.243857
5. Bartocci, E., Falcone, Y., Francalanza, A., Leucker, M., Reger, G.: An introduction to runtime verification. In: Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457, pp. 1–23 (2018)
6. Basin, D.A., Krstic, S., Traytel, D.: Almost event-rate independent monitoring of metric dynamic logic. In: Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings. pp. 85–102 (2017). https://doi.org/10.1007/978-3-319-67531-2_6, https://doi.org/10.1007/978-3-319-67531-2_6
7. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining counterexamples using causality. Formal Methods in System Design **40**(1), 20–40 (Feb 2012). https://doi.org/10.1007/s10703-011-0132-2, https://doi.org/10.1007/s10703-011-0132-2
8. Christakis, M., Heizmann, M., Mansur, M.N., Schilling, C., Wüstholz, V.: Semantic fault localization and suspiciousness ranking. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 226–243. Springer International Publishing, Cham (2019)
9. Dawes, J.H., Reger, G.: Specification of State and Time Constraints for Runtime Verification of Functions (2018), arXiv:1806.02621
10. Dawes, J.H., Reger, G.: Specification of temporal properties of functions for runtime verification. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. pp. 2206–2214. SAC '19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3297280.3297497, http://doi.acm.org/10.1145/3297280.3297497
11. Dawes, J.H., Reger, G., Franzoni, G., Pfeiffer, A., Govi, G.: Vypr2: A framework for runtime verification of python web services. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 98–114. Springer International Publishing, Cham (2019)
12. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) Formal Modeling and Analysis of Timed Systems. pp. 92–106. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
13. Ferrère, T., Maler, O., Ničković, D.: Trace diagnostics using temporal implicants. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) Automated Technology for Verification and Analysis. pp. 241–258. Springer International Publishing, Cham (2015)
14. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. International Journal on Software Tools for Technology Transfer **8**(3), 229–247 (Jun 2006). https://doi.org/10.1007/s10009-005-0202-0, https://doi.org/10.1007/s10009-005-0202-0
15. Havelund, K., Reger, G.: Specification of parametric monitors - quantified event automata versus rule systems. In: Formal Modeling and Verification of Cyber-Physical Systems (2015)
16. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A runtime assurance approach for java programs. Form. Methods Syst. Des. **24**(2), 129–155 (Mar 2004). https://doi.org/10.1023/B:FORM.0000017719.43755.7c, https://doi.org/10.1023/B:FORM.0000017719.43755.7c

17. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming **78**(5), 293 – 303 (2009). https://doi.org/https://doi.org/10.1016/j.jlap.2008.08.004, http://www.sciencedirect.com/science/article/pii/S1567832608000775, the 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS07)
18. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. STTT **14**(3), 249–289 (2012). https://doi.org/10.1007/s10009-011-0198-6
19. Reger, G.: Suggesting edits to explain failing traces. In: Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings. pp. 287–293 (2015)
20. Reger, G., Cruz, H.C., Rydeheard, D.: MarQ: monitoring at runtime with QEA. In: TACAS'15 (2015)
21. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence **32**(1), 57 – 95 (1987). https://doi.org/https://doi.org/10.1016/0004-3702(87)90062-2, http://www.sciencedirect.com/science/article/pii/0004370287900622
22. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering. pp. 30–39. ASE'03, IEEE Press, Piscataway, NJ, USA (2003). https://doi.org/10.1109/ASE.2003.1240292, https://doi.org/10.1109/ASE.2003.1240292
23. Reps, T., Ball, T., Das, M., Larus, J.: The use of program profiling for software maintenance with applications to the year 2000 problem. In: Jazayeri, M., Schauer, H. (eds.) Software Engineering — ESEC/FSE'97. pp. 432–449. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
24. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language, version 1.5-4 (Mar 2014), frama-c.com/download/e-acsl/e-acsl.pdf
25. de Souza, H.A., Chaim, M.L., Kon, F.: Spectrum-based software fault localization: A survey of techniques, advances, and challenges. CoRR **abs/1607.04347** (2016), http://arxiv.org/abs/1607.04347
26. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Trans. Softw. Eng. **42**(8), 707–740 (Aug 2016). https://doi.org/10.1109/TSE.2016.2521368, https://doi.org/10.1109/TSE.2016.2521368