

Diagnosing Violations of State-based Specifications in iCFTL

Cristina Stratan, Claudio Mandrioli, Domenico Bianculli

Abstract—As modern software systems grow in complexity and operate in dynamic environments, the need for runtime analysis techniques becomes a more critical part of the verification and validation process. Runtime verification monitors the runtime system behaviour by checking whether an execution trace—a sequence of recorded events—satisfies a given specification, yielding a Boolean or quantitative verdict. However, when a specification is violated, such a verdict is often insufficient to understand why the violation happened. To fill this gap, *diagnostics* approaches aim to produce more informative verdicts. In this paper, we address the problem of generating informative verdicts for violated Inter-procedural Control-Flow Temporal Logic (iCFTL) specifications that express constraints over program variable values. We propose a diagnostic approach based on backward data-flow analysis to statically determine the relevant statements contributing to the specification violation. Using this analysis, we instrument the program to produce enriched execution traces. Using the enriched execution traces, we perform the runtime analysis and identify the statements whose execution led to the specification violation. We implemented our approach in a prototype tool, *iCFTLdiagnostics*, and evaluated it on 112 specifications across 10 software projects. Our tool achieves 90% precision in identifying relevant statements for 100 of the 112 specifications. It reduces the number of lines that have to be inspected for diagnosing a violation by at least 90%. In terms of computational cost, our experiments show that *iCFTLdiagnostics* generates a diagnosis within 7 min, and requires no more than 25 MB of memory. The instrumentation required to support diagnostics incurs an execution time overhead of less than 30% and a memory overhead below 20%.

I. INTRODUCTION

MANY software systems operate in environments that are partially unknown or unpredictable at design time [1], making static verification techniques, like model checking, insufficient. *Runtime Verification (RV)* [2] has emerged as a lightweight formal method that monitors the behaviour of a system at runtime to determine whether it satisfies a given specification. These specifications, often expressed in specification languages (such as MTL [3], STL [4], CFTL [5], iCFTL [6], SB-TemPsy-DSL [7]), define properties over execution traces. The monitoring process yields a *verdict*—typically Boolean or quantitative [8]—indicating whether the trace conforms or not to the given specification. While indicating whether the system runtime behaviour satisfies the specification is useful, once a violation is detected, engineers require a deeper understanding of the underlying cause. For example, consider a source-level specification such as: “the value of variable *temp* in procedure *proc* should always be less than 10.” If a program execution violates this property,

a conventional RV tool will report a Boolean verdict indicating failure. However, in practice, engineers are interested in understanding what contributed to the violation—such as which operations or inputs influenced the value of *temp*. This process of determining what led to the violation is known as *diagnostics*, and it is tailored to the semantics of the specification language in use. Diagnostics aim to explain *why* a verdict was reached by identifying the violated subformulas and the relevant parts of the trace that contributed to the violation. For example, Ferrère et al. [9] proposed a method for extracting minimal subtraces that explain violations in MTL and STL specifications. Dawes and Reger [10] presented a framework for identifying problematic segments of source-code that could explain a CFTL specification violation. Similarly, Boufaied et al. [11] introduced a diagnostics approach for SB-TemPsy-DSL that annotates trace segments and state transitions to highlight the source of violations.

We address the diagnostics problem for *Inter-procedural Control-Flow Temporal Logic* (iCFTL), a specification language designed to express inter-procedural, source code-level properties of programs [6]. iCFTL enables engineers to easily capture source code properties, making it suitable for software development workflows. There are two types of iCFTL specifications: (i) time-based: temporal constraints over program points, and (ii) state-based: constraints over program variables. Our previous work [12], tackled the diagnostics problem for time-based properties by determining the unique event in the trace, called *point of no return*, after which it is impossible to satisfy the specification. Using this event, we identify the execution trace slice that contains the events that can explain the specification violation. However, this approach does not generalise to *state-based* properties since their violation is not tied to a specific moment in time but rather to how a variable’s value was computed.

In this work, we fill this gap by introducing a diagnostics approach tailored for iCFTL *state-based* properties, which focuses on data-flow analysis to trace how variables obtain their values across procedures and statements. Such properties capture requirements like “after the change of variable *temp* in procedure *proc*, variable *humidity* in procedure *proc* should have a value less than 20” or “the values of variable *humidity* in procedure *proc* should always be less than the variable *temp* in procedure *proc*.” When encountering a violation of these specifications, we are interested in diagnosing how the variables *humidity* and *temp* obtained their values. In this way, we enrich iCFTL with full diagnostics capabilities for both time and state based properties. To diagnose violations of state-based properties, we consider a single (inter-procedural) execution trace, denoted as an *ι -trace*—a sequence of events

recorded during program execution, each tied to a specific program point within a procedure. Diagnosing a single trace reflects practical constraints in real-world systems, where re-execution (to obtain multiple traces) may be costly or infeasible, especially in large-scale or production environments. The goal of state-based diagnostics is to determine which events in the ι -trace contributed to the violation of a specification. We define an event as *relevant* if it contributed to the computation of the value of any variable used in the specification. As an example, consider the property “the value of variable *temp* in procedure *proc* should always be less than 10,” and two consecutive statements $x = 6$ and $temp = x + 5$. Although the violation occurs at $temp = x + 5$, the assignment $x = 6$ is also relevant, as it influences the value of *temp*. We propose an automated approach to identify such relevant events using backward data-flow analysis, traversing the procedure’s control flow graph, starting from the point of violation (i.e. the event that violates the specification). During traversal, we collect all the nodes connected via backwards data-flow dependency to the point of violation, allowing us to trace how the procedure execution computed the variable’s value that led to the violation. Leveraging the inter-procedural capabilities of iCFTL, we extend this analysis beyond individual procedures and capture data-flow dependencies across procedures boundaries. Beyond identifying relevant events, we introduce a parameter called *multiplicity*, which quantifies an event’s importance by counting the number of distinct data flow paths—such as those involving conditionals, loops, or control transfers (e.g., break, continue)—linking it to the specification violation. Intuitively, an event with higher multiplicity plays a more central role in the violation, as it influences the variable’s value through multiple execution paths.

We implemented our approach in a prototype tool, iCFTLdiagnostics, by extending the original iCFTL instrumentation approach to produce an enriched trace that includes events relevant to the specification violation. This enriched trace forms the basis for computing the diagnosis, i.e., the set of relevant events and associated statements that explain the specification violation. We empirically evaluated iCFTLdiagnostics by diagnosing 112 specification violations for different procedures across 10 projects. We address four distinct research questions (RQs), assessing iCFTLdiagnostics effectiveness in identifying relevant statements (RQ1), its ability to reduce the complexity of understanding specification violations (RQ2), the time and memory required for diagnosis (RQ3), and the overhead introduced by its instrumentation approach (RQ4). Our experimental results demonstrate that iCFTLdiagnostics accurately identifies the events relevant to specification violations with at least 90% precision for 100 out of 112 specifications (RQ1). In terms of complexity reduction (RQ2), iCFTLdiagnostics significantly narrows down the portions of code that need to be inspected—by at least 90%—thereby facilitating the understanding of specification violations. In terms of computational cost (RQ3), the diagnosis process completes within 7 min and requires no more than 25 MB of peak memory for the specifications we evaluated. Furthermore, the additional instrumentation introduced by our tool incurs a reasonable

overhead—less than 30% in execution time and under 20% in peak memory usage (RQ4). To summarise, the main contributions of the paper are:

- An approach based on backward data-flow analysis that identifies relevant events in an ι -trace and computes a *diagnosis* for the violated specification.
- iCFTLdiagnostics, a prototype implementation of our approach.
- An empirical evaluation assessing the effectiveness and efficiency of iCFTLdiagnostics on 10 projects and 112 specifications.

The rest of the paper is structured as follows. Section II introduces the necessary iCFTL background. Section III discusses the modification we applied to how programs are represented in iCFTL. Section IV describes our approach. Section V describes our prototype tool. Section VI reports on the experimental evaluation. Section VII positions our contribution with respect to the literature. Section VIII offers concluding remarks and a roadmap for future work.

II. BACKGROUND: INTER-PROCEDURAL CONTROL-FLOW TEMPORAL LOGIC

In this section, we introduce a statically-computable representation of a program, followed by the notion of trace, i.e., the representation of a program’s execution. We then present the fragment of iCFTL considered in this paper and its associated semantics. All material introduced in this section is based on the original iCFTL paper [6] and our previous work [12].

A. Statically-Computable Representation of a Program

1) *Systems of Multiple Procedures*: We define a program as a *system of multiple procedures* S , which is a pair $\langle \mathcal{P}, \text{prog} \rangle$ where $\mathcal{P}$ is a finite set of procedure names (i.e., the names of callable subroutines), and *prog* is a map that sends each procedure name to the portion of code that defines it.

2) *Symbolic Control Flow Graphs*: In iCFTL, for each procedure in S , we compute a statically-computable representation, which we call *symbolic control-flow graph* (SCFG). Intuitively, this is a directed graph that uses vertexes to represent the *symbolic* states reached by executing a statement in code. Such *symbolic states*, denoted by σ , indicate that a program variable’s value may have changed, or a function may have been called, but encode no concrete values (since these are often only known at runtime). For example, an assignment statement $x = a$ would result in two symbolic states: one to represent the symbolic state of the program before x has been assigned a new value, and another to represent the symbolic state of the program in which x has just been assigned a value. Formally, a symbolic state σ is a pair containing the program point of a statement $\rho(\text{stmt})$ (i.e., its location in the code), and a map from program variables to one of the statuses $\{\text{changed}, \text{unchanged}, \text{undefined}, \text{called}\}$. Since an SCFG is a directed graph, an edge from a symbolic state σ to another symbolic state σ' indicates that a statement is executed at runtime, transitioning the program to a new state. Further, branching (caused by a conditional or loop) is represented by vertices having multiple successors. Formally,

for a procedure p , a symbolic control-flow graph, denoted by $\text{SCFG}(p)$, is a triple $\langle V, E, v_s \rangle$ with V being a set of *symbolic states*, $E \subset V \times V$ being a set of edges, and v_s being the initial symbolic state. The initial symbolic state v_s represents the entry point of the procedure—before any computation has occurred—and encodes the assignment of the procedure’s parameters. A path π through an SCFG is defined as a sequence of edges $e_1, e_2, \dots, e_n$, where each edge $e_i \in E$ (with $1 \leq i \leq n$), and for every consecutive pair e_i and e_{i+1} , we have $e_i = \langle \sigma, \sigma' \rangle$ and $e_{i+1} = \langle \sigma', \sigma'' \rangle$; that is, the edges must be adjacent.

B. Dynamic Runs

We represent program executions as *traces*, which we build by considering paths through SCFGs. Each trace consists of a sequence of SCFG vertices annotated with timestamps. We use these annotated vertices as the basis of our notion of *concrete states*, which are symbolic states augmented with information obtained at runtime (such as timestamps and program variables values). Formally, a concrete state is a triple $\langle t, \sigma, m \rangle$ for t a real-numbered timestamp, σ a symbolic state, and m a map from program variables to values observed at runtime. We then represent executions of individual procedures as sequences of concrete states, which we call *dynamic runs*. Hence, a dynamic run $\mathcal{D}$ is a sequence $\langle t_1, \sigma_1, m_1 \rangle, \dots, \langle t_n, \sigma_n, m_n \rangle$ of concrete states such that there must be a path from each σ_i to σ_{i+1} , for $1 \leq i < n$, in the relevant SCFG. Once concrete states have been collected into sequences, we refer to a pair of consecutive concrete states in a sequence as a *transition*. Intuitively, since concrete states correspond to symbolic states, we use pairs of concrete states (i.e., transitions) to model the computation that takes place at runtime to reach one concrete state from another. Concretely, we can use transitions to model operations like program variable value changes and function calls.

Inter-procedural Dynamic Runs: We lift the notion of a dynamic run to model an execution of a system of multiple procedures $\mathcal{S}$ by labelling each dynamic run in a set with the name of the procedure to which it corresponds. Formally, an *inter-procedural dynamic run* $\mathcal{I}$ of $\mathcal{S}$ is a tuple $\langle \mathcal{P}, \{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_n\}, \mathcal{L} \rangle$. Here, $\mathcal{P}$ is the same $\mathcal{P}$ (set of procedure names) used to define $\mathcal{S}$, $\{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_n\}$ is a set of dynamic runs, and $\mathcal{L} : \{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_n\} \rightarrow \mathcal{P}$ is a map that labels each dynamic run $\mathcal{D}_i$ with a procedure name from $\mathcal{P}$. For brevity, we will often refer to inter-procedural dynamic runs as ι -traces.

To reason about parts of such ι -traces, we introduce the notions of *sub-trace* and *sub-trace under a predicate* as defined in our previous work [12]. Intuitively, $\mathcal{I}'$ is a sub-trace of $\mathcal{I}$ if, when we take $\mathcal{I}$ and remove some concrete states, or dynamic runs, we obtain $\mathcal{I}'$. More formally, we can define an injective function γ that from a dynamic run in $\mathcal{I}'$ returns which dynamic run from $\mathcal{I}$ had concrete states removed to obtain it. For example, given $\mathcal{I} = \langle \mathcal{P}, \{\mathcal{D}_1, \mathcal{D}_2\}, \mathcal{L} \rangle$, with $\mathcal{D}_1 = \langle t_1, \sigma_1, m_1 \rangle, \langle t_2, \sigma_2, m_2 \rangle, \langle t_3, \sigma_3, m_3 \rangle$, and $\mathcal{I}' = \langle \mathcal{P}', \{\mathcal{D}'_1\}, \mathcal{L}' \rangle$, with $\mathcal{D}'_1 = \langle t_1, \sigma_1, m_1 \rangle, \langle t_2, \sigma_2, m_2 \rangle$, then $\mathcal{I}'$ is a sub-trace of $\mathcal{I}$ as $\mathcal{D}'_1$ contains a subset of concrete states of $\mathcal{D}_1$ and $\mathcal{I}'$ does not contain $\mathcal{D}_2$. Then, we can construct γ

as $\gamma(\mathcal{D}'_1) = \mathcal{D}_1$. As the sub-trace relation just states that we remove concrete states from dynamic runs but not which ones, we extend the sub-trace relation to require that any concrete state removed from the ι -trace satisfies a given predicate. We say that $\mathcal{I}'$ is a *sub-trace under a predicate* P of a trace $\mathcal{I}$, if (i) $\mathcal{I}'$ is a sub-trace of $\mathcal{I}$, and (ii) any concrete state c found in the dynamic runs of $\mathcal{I}'$ must satisfy $P(c) = \text{true}$. For example, a predicate P could retain only concrete states that occurred within the first 3s of system execution, i.e., $P(c) = \text{true}$ iff $t_c < 3$ with $c = \langle t_c, \sigma_c, m_c \rangle$.

C. iCFTL Specifications

Here, we briefly introduce how iCFTL specifications are composed and illustrate their semantics. For a detailed definition of iCFTL syntax and semantics we refer to [6]. iCFTL Specifications consist of *quantifiers* and Boolean combinations of *atomic constraints*. Quantifiers enable one to capture concrete states or transitions from ι -traces, and bind them to variables. Quantifiers in iCFTL are always universal ($\forall$); existential quantifiers are not permitted, and all specifications must be written in *prenex normal form*. They use predicates such as `changes(x).during(p)` or `future(q; changes(x).during(p))` to select relevant states or transitions from traces.

Atomic constraints predicate over values extracted from concrete states and transitions, and can be combined using standard Boolean operators ($\wedge, \vee, \Rightarrow, \neg$). Specifically, they predicate on *expressions*, which enable one to select the concrete states or transitions to be used in the constraint. Common forms include `q(x)` (value of variable x in the state bound to q) and `q.next(changes(y).during(p))` (value after the next change of y during procedure p).

In this work, we focus on *state-based* specifications, which are specifications with one or more quantifiers that capture concrete states, i.e., variable value changes. Here, we provide three examples of iCFTL state-based specifications that also serve as a template for our empirical evaluation in § VI.

- The property “every time the program variable y is changed during the procedure g , the value of y during g should be less than 4 and the value of the next change of variable x after y should be less than 10” can be expressed as:

$$\forall q \in \text{changes}(y).\text{during}(g) : q(y) < 4 \wedge q.\text{next}(\text{changes}(x).\text{during}(g)) < 10 \quad (1)$$

This specification has one quantifier $\forall q \in \text{changes}(y).\text{during}(g)$ and two atomic constraints, `q(y) < 4` and `q.next(changes(x).during(g)) < 10`, connected by logical conjunction. Each atomic constraint contains an expression, respectively, `q(y)` (which extracts the value of variable y at q) and `q.next(changes(x).during(g))` (which is the updated value of variable x after q).

- The property “for each change of x during *proc*, and for each change of y during *proc*, the value of x should be less

than 5 or the value of y should equal the value of x ” can be expressed as:

$$\begin{aligned} & \forall q \in \text{changes}(x).\text{during}(\text{proc}) : \\ & \forall r \in \text{changes}(y).\text{during}(\text{proc}) : \\ & \quad q(x) < 5 \vee r(y) = q(x) \end{aligned}$$

This specification uses two quantifiers ($\forall q$ and $\forall r$) and two atomic constraints connected by logical disjunction, which contain the expressions $r(y)$ and $q(x)$.

- The property “for each change of a , b , and c in procedures *procA*, *procB*, and *procC*, respectively, the value of a should be greater than b , and the value of c should be greater than a ” can be expressed as:

$$\begin{aligned} & \forall q_1 \in \text{changes}(a).\text{during}(\text{procA}) : \\ & \forall q_2 \in \text{changes}(b).\text{during}(\text{procB}) : \\ & \forall q_3 \in \text{changes}(c).\text{during}(\text{procC}) : \\ & \quad q_1(a) > q_2(b) \wedge q_3(c) > q_1(a) \end{aligned}$$

This specification uses three quantifiers and two atomic constraints connected by logical conjunction. The expressions are $q_1(a)$, $q_2(b)$, and $q_3(c)$.

We now illustrate how the iCFTL semantics computes a truth value over an arbitrary ι -trace, using the specification in Equation 1 as an example. We begin with the quantifier, $\forall q \in \text{changes}(y).\text{during}(g)$, which looks for all concrete states representing a change of the program variable y , during the execution of function g . Formally, this involves (i) computing the SCFG of the procedure g ; (ii) inspecting the symbolic state σ of each concrete state $c_i = \langle t, \sigma, m \rangle$ to check for a change of y . For each concrete state c_i , the semantics constructs a *binding* β_i , which maps q (from the quantifier) to the concrete state c_i . For each binding β_i , the semantics determines whether the atomic constraints $q(y) < 4$ and $q.\text{next}(\text{changes}(x).\text{during}(g)) < 10$ hold. We first identify the unique concrete states to which the expressions $q(y)$ and $q.\text{next}(\text{changes}(x).\text{during}(g))$ correspond. For this, the $\text{eval}(I, \beta_i, \text{expr})$ function is used, which takes an ι -trace I , a binding β_i , and an expression expr , and extracts the concrete states relevant to that expression. Once the eval function returns the concrete state or transition that satisfies the expression, we can finally resolve the constraint. First, we separately check each constraint for satisfaction, and then we compute the verdict for the whole specification. The atomic constraint $q(y) < 4$ is true if the value mapped to y in the concrete state returned by eval is less than 4. Similarly, the atomic constraint $q.\text{next}(\text{changes}(x).\text{during}(g)) < 10$ is true if the constraint (in the example, the value being less than 10) is satisfied. Then, the satisfaction of the specification for this binding is given by the conjunction of the values of the two atomic constraints. Finally, the specification verdict is computed as a map, given by $[I, \varphi]_S$ that maps each binding to its truth value.

III. AUGMENTING THE SCFG WITH DATAFLOW INFORMATION

In § II-A2, we introduced the concept of SCFG, used in iCFTL to statically represent procedures. While effective for

<pre> 1 def k(): 2 a = 8 3 for y in [0, 1]: 4 b = a + 1 5 endFor 6 a = a + b 7 m(a) 8 def m(a): 9 c = 9 10 g(c, a) </pre>	<pre> 11 def g(b, y): 12 l = y + 3 13 if l < 8: 14 k = l + 1 15 else: 16 k = b + l 17 endIf 18 y = k + 4 </pre>
--	--

Fig. 1: Example of program with three procedures k, m, g . The execution of the program starts with procedure k , which calls procedure m , which eventually calls procedure g .

capturing control flow, the original definition of SCFG lacks dataflow information about how variable values are computed, particularly in the context of state-based specifications. This limitation prevents the detection of the program points that we want to include in the diagnosis of violated specifications. For example, consider the program in Figure 1, and the specification “for each change of variable a its value should always be less than 10,” which can be expressed in iCFTL with the formula $\forall p \in \text{changes}(a).\text{during}(k) : p(a) < 10$. For diagnosing the specification, we want to understand how the variable a obtained its value at line 6 (i.e., when the value of a changes during the execution of k). At line 6, the statement $a = a + b$ shows that a depends on both a and b . However, in the SCFG, this dataflow dependency is not reflected, as the symbolic state for statement at line 6 holds only information about the program point and the map from the variable a to one of the statuses: $\langle \rho_5(\text{stmt}), [a \mapsto \text{changed}] \rangle$.

To address this limitation, we refine the SCFG by enriching symbolic states with additional dataflow information. In the remainder of this paper, to distinguish between the two representations, we refer to the original SCFG as SCFG^0 , and to the modified version—enriched with dataflow information—as simply SCFG. In the modified SCFG, we incorporate dataflow information by tracing which variables are *written*, and which variables are *read* at each program point. Furthermore, to enable the tracking of the dataflow across different procedures (i.e., inter-procedural analysis), we want to trace which functions are *called* at each program point.¹ To formalise this modification, we redefine the symbolic state to include information about which variables are read, written, and which functions are called. Specifically, instead of recording only the status of the program variables with the map m , we use a triple $\langle \text{written}, \text{read}, \text{called} \rangle$, where: *written* is the set of all the written variables at that program point $\rho(\text{stmt})$; *read* is the set of variables whose values are read at that program point $\rho(\text{stmt})$; *called* is the set of procedures that are called at that program point $\rho(\text{stmt})$. For example, in the program of Figure 1, the symbolic state corresponding to line 6 has the triple $\langle \text{written} = \{a\}, \text{read} = \{a, b\}, \text{called} = \{\} \rangle$. We showcase

¹We chose the names *written* and *read*—rather than the naming *defined* and *used*—to be closer to the terminology of programming languages and reflect the concrete operations. This choice aligns with the iCFTL goal of having a program representation—the SCFG—intuitive and directly mappable to the program’s syntax and execution behaviour.

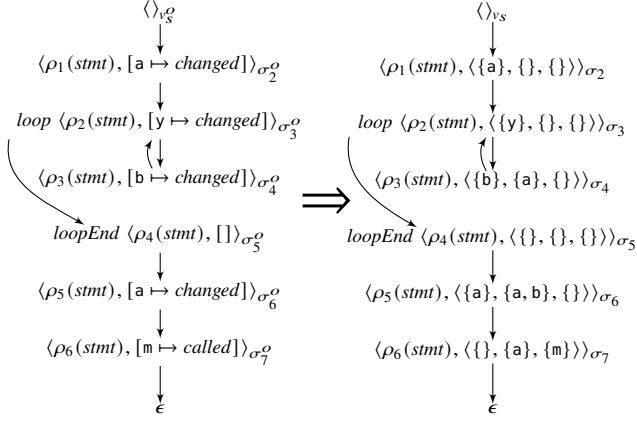


Fig. 2: SCFG of procedure k: original (left) and new (right).

the difference between SCFG^o and our modified SCFG using procedure k from Figure 1; both representations are shown in Figure 2, with symbolic states indexed by the line number they correspond to. To distinguish between the two representations, symbolic states in the SCFG^o are annotated with a superscript “o” (e.g., σ_2^o), while in the modified SCFG—and throughout the rest of the paper—we refer to the modified symbolic states simply as σ . The SCFG^o captures only variable status changes (e.g., a is changed in σ_6^o), while on the right, the modified SCFG explicitly records which variables are written, read, and which functions are called at each program point. By extending symbolic states with this dataflow information, we can perform analysis to identify program points that influence a variable’s value.

For example, symbolic state σ_4 (where b is in *written*) can be now linked to the computation of a in σ_6^o . This was not possible in the SCFG^o , since σ_6^o carried no information of b being used to compute a .

Importantly, this model also allows for capturing the dataflow across procedures when they call each other. This is enabled by the *called* set, which contains the functions called in each state and can be used to navigate the system of multiple procedures $\mathcal{S}$. In this context, to reconstruct the dataflow across procedures, the *written* set of the calling symbolic state and *read* set of the first symbolic state of each SCFG can be used. Specifically, the starting symbolic state of each SCFG (like v_s in Figure 1) represents the procedure’s entry point. In this state, the procedure parameters are assigned, and thus included in the *written* set. Complementarily, in the calling node (such as a state where procedure k from Figure 1 is found in the *called* set), the variables passed to the function call are accessed, and thus included in the *read* set. By matching the variables in the *read* set in the calling node and the variables in the *written* set from the starting symbolic state of the procedure, we can reconstruct the dataflow across procedures.

IV. APPROACH

Our goal is to identify the concrete states in an ι -trace that are useful to understanding how state-based atomic constraints were falsified in a specification and led to its overall violation. Specifically, for each expression in the atomic constraint, we

want to provide the set of concrete states that are relevant to the constraint falsification, i.e., the concrete states that contributed to the computation of the expression value whose assignment led to the constraint violation. Figure 4 illustrates the overall workflow of our approach. The approach takes as input a Python Program and an iCFTL Specification and produces the Diagnosis as output. The key steps of the approach are:

- (i) Processing the input, where we generate the SCFGs and ι -trace;
- (ii) Instrumentation, where we detect the instrumentation points, i.e., the symbolic states potentially relevant to the specification violation;
- (iii) Filter ι -trace, where we use the instrumentation points to filter the ι -trace;
- (iv) Monitoring, where we obtain the verdict on the specification; and
- (v) Diagnostics, where, in case of state-based specification violation, we retrieve the concrete states relevant to understanding the violated specification.

Our contribution lies in the Instrumentation, Filter ι -trace and Diagnostics steps (highlighted in red dashed boxes in Figure 4). In the Instrumentation step (described in § IV-A), we traverse the SCFG and identify the symbolic states needed for explaining each of the expressions in the atomic constraints. We refer to such symbolic states as “instrumentation points”. Using these instrumentation points, we filter the ι -trace so that it includes only concrete states relevant to each expression (Filter ι -trace, described in § IV-B).

In the theoretical development of our approach, we assume that the entire execution trace is available, i.e., that all of the concrete states associated to each visited symbolic state during execution. However, in practice, storing all concrete states is unnecessary and impractical, as it will increase the program execution overhead to store not-needed symbolic states and generate very long traces. To avoid this, we follow the same approach as previous work on iCFTL [6], and, while in this theoretical part we present the Instrumentation as an independent step from the program execution, in the actual implementation we first instrument the program and then execute it to directly obtain a filtered ι -trace. We run the iCFTL monitoring algorithm on the filtered ι -trace to obtain the specification Verdict (Monitoring). Finally, based on the verdict and ι -trace, we construct the diagnosis by first finding all the falsified atomic constraints, and then constructing a diagnosis for each expression (Construct Diagnosis, described in § IV-C).

A. Instrumentation

The goal of the *Instrumentation* step is to identify the set of symbolic states in an ι -trace such that the concrete states connected to these symbolic states are sufficient for both trace checking and diagnostics. An existing instrumentation approach for iCFTL [6], referred to as *vanilla instrumentation*, determines the minimal set of instrumentation points (i.e., symbolic states) sufficient to decide if an ι -trace meets a given specification. Building on the vanilla instrumentation, we extend the approach for diagnosing state-based specifications.

$\mathcal{I}_{ex} = (\{k, m, g\}, \{\mathcal{D}_k, \mathcal{D}_m, \mathcal{D}_g\}, \{k \mapsto \mathcal{D}_k, m \mapsto \mathcal{D}_m, g \mapsto \mathcal{D}_g\})$
 $\mathcal{D}_k = \langle 0, [], [] \rangle, \langle 0.2, \sigma_2, [a \mapsto 8] \rangle, \langle 0.4, \sigma_3, [y \mapsto 0] \rangle, \langle 0.6, \sigma_4, [b \mapsto 9] \rangle, \langle 0.8, \sigma_3, [y \mapsto 1] \rangle, \langle 1.0, \sigma_4, [b \mapsto 9] \rangle, \langle 1.1, \sigma_5, [] \rangle, \langle 1.2, \sigma_6, [a \mapsto 17] \rangle, \langle 3.1, \sigma_7, [] \rangle$
 $\mathcal{D}_m = \langle 1.4, [], [] \rangle, \langle 1.6, \sigma_9, [c \mapsto 9] \rangle, \langle 2.9, \sigma_{10}, [] \rangle$
 $\mathcal{D}_g = \langle 1.7, [], [] \rangle, \langle 1.9, \sigma_{12}, [l \mapsto 20] \rangle, \langle 2.0, \sigma_{13}, [] \rangle, \langle 2.2, \sigma_{15}, [] \rangle, \langle 2.4, \sigma_{16}, [k \mapsto 29] \rangle, \langle 2.5, \sigma_{17}, [] \rangle, \langle 2.7, \sigma_{18}, [y \mapsto 23] \rangle$

Fig. 3: Example of ι -trace $\mathcal{I}_{ex}$ from the execution of the program in Figure 1. The trace is a triple of procedures, dynamic runs and a map from procedures to dynamic runs. The three dynamic runs $\mathcal{D}_k, \mathcal{D}_m$, and $\mathcal{D}_g$ each represent the execution of one procedure.

Starting from the symbolic states used for providing the verdict, we select the additional symbolic states needed to backtrack (diagnose) how the program execution led to the values of the variables relevant to the verdict. Specifically, given a program and an expression, we aim to collect all the symbolic states (both the one determined by the vanilla instrumentation and the additional ones required for diagnostics) that contain information relevant, in terms of dataflow, to this expression. We refer to this extended instrumentation that includes both vanilla and additional instrumentation points as *diagnostics instrumentation*.

Running Example: We showcase the difference in the instrumentation approaches with an example based on the program in Figure 1 and the following iCFTL specification:

$$\forall q \in \text{changes}(y). \text{during}(g) : q(y) < 4. \quad (2)$$

This specification constrains the value of the variable y during the execution of procedure g . The iCFTL vanilla instrumentation scheme focuses on determining symbolic states where y changes within SCFG(g), hence it will identify as instrumentation point only the symbolic state σ_{18} , associated with line 18. However, diagnostics instrumentation requires understanding how y obtained its value—specifically, which variables contributed to its computation. In this case, we track k , which is directly used to compute the value of y . Such a dataflow dependency is formally captured by the fact that, in the symbolic state σ_{18} , where y belongs to the *written* set, k belongs to the *read* set. Reasoning recursively, we also want to understand how k obtained its values. Thus, the diagnostics instrumentation includes also lines 16 and 14, as they assign a value to k —indeed, their associated symbolic states k belongs to the *written* set. Iterating further, we also track the values of l and b , to be able to diagnose how k obtained its value. Accordingly, the instrumentation should include also lines 12, 9, 6, 4 and 2. This iteration continues until all the symbolic states that are connected by dataflow to the assignment of y at line 18 are identified. By statically instrumenting these symbolic states, one will be able to determine the chain of variable changes that led to the value of variable y that (potentially) violates the specification. To rigorously identify these relevant states, we propose Algorithm 1, which we call *system of procedures traversal*, shortened to “S-traversal.”

Algorithm 1: S-traversal

```

1 Input:  $\sigma_e = \langle \rho_e(stmt), \langle \text{written}_e, \text{read}_e, \text{called}_e \rangle \rangle, \text{proc}, S$ 
   Result: List  $\mathcal{E}$  of symbolic states relevant to  $\sigma_e$ 
2  $\mathcal{E} : \text{list} \leftarrow \{\sigma_e\};$ 
3  $\mathcal{U} : \text{set} \leftarrow \{\text{read}_e\};$ 
4 S-traversal( $\sigma_e, \mathcal{U}, \mathcal{E}, \text{proc}, S$ );
5 def S-traversal( $\sigma_n$ : symbolic state,  $\mathcal{U}$ : read variables,  $\mathcal{E}$ : additional instrumentation points,  $\text{proc}$ : procedure,  $S$ : system of multiple procedures):
6   incomingStar = getIncomingStar( $\sigma_n, \text{proc}$ );
7   for  $\langle \rho_x(stmt), \langle \text{written}_x, \text{read}_x, \text{called}_x \rangle \rangle \in \text{incomingStar}$  do
8      $\mathcal{U}' = \text{copy of } \mathcal{U};$ 
9     // Get starting symbolic state of this procedure
10     $v_s = \text{getEntryPoint}(\text{proc});$ 
11    // Check if  $\sigma_n$  is a procedure entry point
12    if  $\langle \rho_n(stmt), \langle \text{written}_n, \text{read}_n, \text{called}_n \rangle \rangle == v_s$  then
13       $\mathcal{U}' = \text{recurseOnCallers}(\text{proc}, S, \mathcal{E}, \mathcal{U}', v_s)$ 
14      // Check if  $\sigma_x$  is not relevant
15      if  $\mathcal{U}' \cap \text{written}_x == \emptyset$  then
16        // If not relevant, recurse on  $\sigma_x$ 
17        S-traversal( $\langle \rho_x(stmt), \langle \text{written}_x, \text{read}_x, \text{called}_x \rangle \rangle, \mathcal{U}', \mathcal{E}, \text{proc}, S$ );
18        continue;
19      // If we got here, we found a relevant symbolic state
20       $\mathcal{E}.add(\langle \rho_x(stmt), \langle \text{written}_x, \text{read}_x, \text{called}_x \rangle \rangle);$ 
21       $\mathcal{U}' = \mathcal{U}' \cup \text{written}_x;$ 
22      // Check if we solved all data dependencies
23      if  $\mathcal{U}' \neq \emptyset \vee \text{read}_x \neq \emptyset$  then
24        // If not, recurse on incoming star
25         $\mathcal{U}' = \mathcal{U}' \cup \text{read}_x;$ 
26        S-traversal( $\langle \rho_x(stmt), \langle \text{written}_x, \text{read}_x, \text{called}_x \rangle \rangle, \mathcal{U}', \mathcal{E}, \text{proc}, S$ );
27      return  $\mathcal{E};$ 
28 def recurseOnCallers( $\text{proc}, S, \mathcal{E}, \mathcal{U}', v_s$ ):
29   // Iterate over the procedure parameters ( $\text{written}_s$  of  $v_s$ )
30   for  $w_s \in \text{written}_s$  do
31     // Check if the parameter is relevant
32     if  $w_s \in \mathcal{U}'$  then
33       // If relevant, find all the calls to  $\text{proc}$ 
34        $\text{callers} = \text{getCallers}(\text{proc}, S);$ 
35        $\text{parameterIndex} = \text{getParameterIndex}(v_s, w_s);$ 
36       // Iterate over the calls to  $\text{proc}$ 
37       for  $c \in \text{callers}$  do
38         // Get calling procedure and symbolic state
39          $c\_proc = c[0];$ 
40          $c\_state = c[1];$ 
41         // Get parameter's name in caller's scope
42          $p = \text{getRenamedParameter}(SCFG(c\_proc), c\_state, \text{parameterIndex});$ 
43         // Recurse on the calling symbolic state
44         S-traversal( $c\_state, \{p\}, \mathcal{E}, c\_proc, S$ );
45        $\mathcal{U}' = \mathcal{U}' \cup \{w_s\};$ 
46   return  $\mathcal{U}';$ 

```

1) **S-traversal:** We apply S-traversal to each symbolic state σ_e that satisfies an expression of an atomic constraint in the specification—i.e., the symbolic states instrumented by the vanilla instrumentation. Algorithm 1 shows the pseudocode of S-traversal, relying on auxiliary functions that we define later in this section.² It takes as input a symbolic state $\sigma_e = \langle \rho_e(stmt), \langle \text{written}_e, \text{read}_e, \text{called}_e \rangle \rangle$, a procedure proc , and a system of multiple procedures S ; it returns the list $\mathcal{E}$ of additional instrumentation points that are relevant to diagnosing the expression satisfied by the symbolic state σ_e . Importantly, Algorithm 1 shows the first call of S-traversal on the symbolic state σ_e that satisfies the specification-violating expression at line 4, and the initialised $\mathcal{U}$ (line 3) and $\mathcal{E}$ (line 2). Then, from line 5 it shows the actual algorithm

²The auxiliary functions are: `getIncomingStar`, which retrieves the incoming symbolic states for a given state, `getEntryPoint` which retrieves the starting symbolic state of a procedure, `getCallers` which finds all procedures where a given procedure is called, `getParameterIndex` which determines the index of a parameter in a procedure's definition, and `getRenamedParameter` which retrieves the name of a procedure's parameter in another function that calls it.

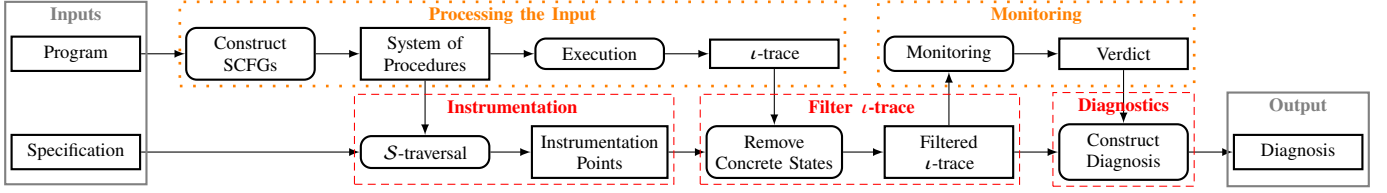


Fig. 4: Overview of the diagnostic approach workflow. Rounded-corner and sharp-corner rectangles represent execution steps and artefacts, respectively. The red-dashed boxes highlight our contributions; the orange-dotted boxes indicate components from the original iCFTL implementation.

definition for the generic symbolic state σ_n , which changes over the recursive calls.

This list of additional instrumentation points may have duplicates, since the same symbolic state may be relevant through different dataflow paths. We track such duplicates to highlight symbolic states that have higher relevance for a given expression, as they can influence the verdict in multiple ways.

The algorithm performs a backward dataflow analysis to collect the list of *relevant* symbolic states that can affect the value of the variable in the expression. This is done by recursively traversing the nodes in the SCFG starting from the symbolic state that satisfies the atomic constraint's expression. At each step, we identify symbolic states that influence σ_e either directly (if they assign a value to variable used directly in σ_e) or indirectly (if they assign a value to a variable further back in the dataflow dependency of σ_e). More formally, given a procedure $proc$, its $SCFG(proc) = \langle V, E, v_s \rangle$, and two symbolic states $\sigma_y = \langle \rho_y(stmt), \{written_y, read_y, called_y\} \rangle$ and $\sigma_x = \langle \rho_x(stmt), \{written_x, read_x, called_x\} \rangle$ in V ,

Definition IV.1. We say that σ_y *influences* σ_x , denoted by $\sigma_y \rightsquigarrow \sigma_x$, if a variable from $written_y$ exists in $read_x$ and there exists a path π in $SCFG(proc)$ between σ_y and σ_x :

$$\sigma_y \rightsquigarrow \sigma_x \iff (\exists var : var \in written_y \wedge var \in read_x) \wedge (\exists \pi = (e_1, \dots, e_n) : e_1, \dots, e_n \in E : e_1 = \langle \sigma_y, * \rangle \wedge e_n = \langle *, \sigma_x \rangle)$$

where $*$ can be any symbolic state, and $n \geq 1$.

Then, the *relevant* symbolic states in $SCFG(proc)$ can be defined as follows:

Definition IV.2. We say that σ_y is *relevant* to σ_x if there exists a sequence of symbolic states $\sigma_1, \dots, \sigma_n$ such that they sequentially influence each other, with $\sigma_1 = \sigma_y$ and $\sigma_n = \sigma_x$, formally $\exists \sigma_1, \dots, \sigma_n : \sigma_i \rightsquigarrow \sigma_{i+1} \wedge \sigma_1 = \sigma_y \wedge \sigma_n = \sigma_x$ where $1 \leq i \leq n$.

To apply this definition and identify the additional instrumentation points, while traversing the SCFGs in the program, we track the *read* variables of relevant symbolic states in the set $\mathcal{U}$, and the list of relevant symbolic states themselves in the list $\mathcal{E}$ (i.e., the instrumentation points). We initialise list $\mathcal{E}$ with the symbolic state $[\sigma_e]$ that satisfies the expression (line 2) and $\mathcal{U}$ with its read variables (line 3). At each symbolic state that we visit, we check if any of the variables in $\mathcal{U}$ is in its *written* set —i.e., we check if it satisfies Definition IV.2. If so,

the symbolic state is relevant to diagnosing the expression and should be instrumented. Accordingly, we (i) add the symbolic state to list $\mathcal{E}$ (line 16), (ii) remove from $\mathcal{U}$ the variables for which we found the definition of their value (line 17), and (iii) add the *read* variables of that symbolic state as they are relevant to the specification violation (lines 19). Notably, in certain statements, a variable might depend on itself, for example in an increment like $a=a+1$. Following the aforementioned steps, if a is in $\mathcal{U}$, the statement is relevant and thus added to $\mathcal{E}$. However, since the value of a before the statement also affects the dataflow, the variable is kept in $\mathcal{U}$: more precisely, it is first removed (as part of step (ii) above) and then re-added (part of step (iii) above). Similarly, when we have circular dependencies, like in $b=a+1$ followed by $a=b+1$, assuming again that a is in $\mathcal{U}$, the second statement is added to $\mathcal{E}$ and a is initially removed from $\mathcal{U}$. However, also in this case, variable a is then re-added when $b=a+1$ is visited, as it is found in the *read* set of this state.

To traverse backward the nodes of the program's SCFGs, we recursively explore them starting from the symbolic states that satisfy an expression of the atomic constraint. For each visited node we inspect its parents in the SCFG, which we refer to as its *incoming star*. When computing the incoming star of a node (function *getIncomingStar* at line 6), we have three main cases depending on the control flow of the program.

- If there is no control flow, the incoming star will contain only one incoming statement. *Example:* In the program in Figure 1, the incoming star of the symbolic state σ_{13} (line 13) is σ_{12} (line 12).
- In case the incoming symbolic state is the end of an *if* statement, we want to separately iterate over both paths of the conditional as we do not know which path will be executed at runtime. Thus, we will have two symbolic states in the incoming star, one for each branch of the conditional. *Example:* Considering again the program in Figure 1, the incoming star of the symbolic state σ_{18} (line 18) contains both σ_{14} and σ_{16} (lines 14 and 16).
- In case the incoming star of the symbolic state contains a loop statement, we want to avoid infinite iterations over the loop body. Accordingly, we traverse the loop body only once and then, if needed, continue to the symbolic states before the loop. Thus, in the incoming star of the symbolic state right after the loop we include only the symbolic state corresponding to the last statement in the loop body (and not the symbolic state before the loop). Conversely, in the

Algorithm 2: Function to find callers of a procedure in a program

```

1 Input:  $proc, \mathcal{S}$ 
   Result: A set of symbolic states and corresponding procedures that called  $proc$ 
2 def  $getCallers(proc: procedure, \mathcal{S}: system\ of\ multiple\ procedures):$ 
3    $callers : set \leftarrow \{\};$ 
4   for  $p \in \mathcal{S}$  do
5     for  $\langle \rho_x(stmt), \langle written_x, read_x, called_x \rangle \rangle \in SCFG(p)$  do
6       if  $proc \in called_x$  then
7          $name = p;$ 
8          $symbolicState =$ 
9            $\langle \rho_x(stmt), \langle written_x, read_x, called_x \rangle \rangle;$ 
10         $callers = callers \cup \{name, symbolicState\};$ 
11   return  $callers;$ 

```

incoming star of the first symbolic state in the loop body we include only the symbolic state before the loop (and not the symbolic state at the end of the loop body). *Example:* In program in Figure 1, the incoming star of σ_6 (line 6) is only σ_4 (line 4), and the incoming star of σ_3 (line 3) is only σ_2 (line 2).

We then iterate over the symbolic states in the incoming star (line 7). Since each symbolic state represents a different dataflow path, we use different copies of $\mathcal{U}$ to avoid interference (line 8). If the symbolic state is not relevant—formally, when the intersection $\mathcal{U}' \cap written_x$ is empty—we perform a recursive call to continue exploring the program (lines 12–15). Otherwise, we have found a relevant symbolic state: we update $\mathcal{E}$ by adding the current symbolic state and refine $\mathcal{U}$ by removing the variables that have now been explained (lines 16–17).

Finally, we are left with handling the inter-procedural aspect of the program. This appears when a symbolic state of the incoming star is the starting symbolic state of a procedure’s SCFG. We check this in lines 9 that retrieves the starting symbolic state of the procedure v_s using the function `getEntryPoint`, and in line 10 where we compare the symbolic state from the incoming star to v_s . To explore the calls to $proc$, we introduce the auxiliary function `recurseOnCallers`, defined in lines 23 to 35. This function takes as input the procedure $proc$, the system of multiple procedures $\mathcal{S}$, the set of used variables $\mathcal{U}'$, the list of additional instrumentation points $\mathcal{E}$, and the starting symbolic state v_s . It is responsible for the recursive calls to $\mathcal{S}$ -traversal in the symbolic states where $proc$ was called, and it finally returns the updated set of used variables $\mathcal{U}'$ —i.e., the set after removing the read variables that have been handled by the recursive call.

We now delve into the details of the `recurseOnCallers` function. In the entry point symbolic state $v_s = \langle \rho_s(stmt), \langle written_s, read_s, called_s \rangle \rangle$ of $proc$, the $written_s$ set contains the procedure parameters. Thus, we iterate over the procedure parameters (line 24) and check if $\mathcal{U}'$ contains any of them (line 25). If this is verified, it means that the dataflow continues outside of the procedure that we are currently exploring, and therefore we continue exploring the program where the procedure was called. To do so, we need to (i) find where the current procedure $proc$ was called (line 26), and (ii) update the name of the variable in $\mathcal{U}'$ to the one used in each of the calling procedures (lines 27–34).

To find the calls to $proc$ in the system of procedures $\mathcal{S}$, we

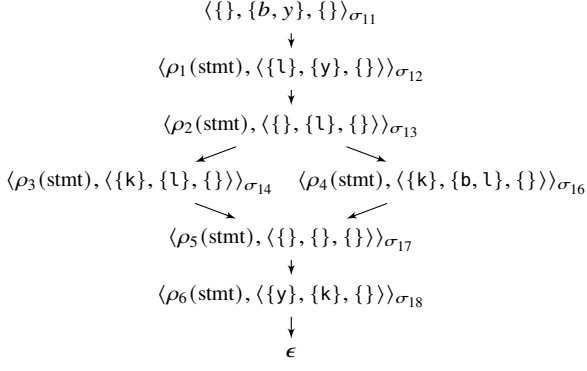
use Algorithm 2. This algorithm takes as input a procedure $proc$ and system of procedures $\mathcal{S}$; it returns the set $callers$ of procedures that contain a call to $proc$ together with the symbolic state where the call happens. The algorithm iterates over the procedures in $\mathcal{S}$ (line 4); for each symbolic state in the SCFG of that procedure (line 5) it checks if procedure $proc$ is in the set of called procedures (line 6). When such symbolic states are found, we add the procedure name and symbolic state to $callers$ (line 9).

Back to function `recurseOnCallers` in Algorithm 1, after computing the $callers$ at line 26, we proceed to find the name used for the variable in $\mathcal{U}'$ in them. First, in the calling function, we identify which parameter passed to the called function it corresponds to. To do so, the `getParameterIndex` function (line 27) returns the index of the positional arguments of the function call that corresponds to the variable in $\mathcal{U}'$. This index will be used to match the parameter passed when the function is called. Then, we iterate over the $callers$ (line 28). For each call to $proc$, we find the name of the variable in $\mathcal{U}'$ based on the parameter’s position (lines 28–32), and recursively call $\mathcal{S}$ -traversal over the symbolic state of the function call ($symbolicState$), the caller procedure ($name$) and with $\mathcal{U}'$ re-initialised to the new variable, while keeping the same $\mathcal{E}$ (line 33). Finally, we remove w_s from the set of used variables $\mathcal{U}'$ (line 34). The recursion of $\mathcal{S}$ -traversal stops when either $\mathcal{U}'$ is empty (i.e., we are able to fully diagnose the specification), or the incoming star is empty (i.e., there are no more parts of the program to explore). Then, we return the list of instrumentation points $\mathcal{E}$ (line 22).

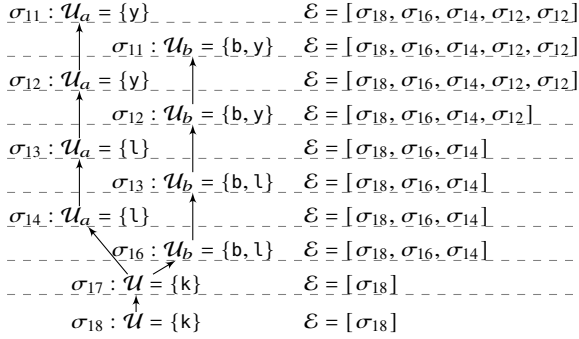
Application to the running example: We illustrate the working of $\mathcal{S}$ -traversal using the example program in Figure 1 alongside the specification in Equation 2. $\mathcal{S}$ -traversal is called over the symbolic state σ_{18} of procedure g . We first illustrate the intra-procedural part of $\mathcal{S}$ -traversal execution over the SCFG of g . Then, we showcase the inter-procedural part by describing the identification and traversal of the callers of g .

To illustrate the intra-procedural aspect, we use Figure 5, where Subfigure 5a shows $SCFG(g)$ and Subfigure 5b shows the recursive calls to the $\mathcal{S}$ -traversal algorithm. In the latter, each node represents a call to Algorithm 1. The graph is to be read bottom-up (as shown by the arrows) to reflect the backward traversal of the SCFG. For each call (i.e., for each node of the graph), we show the updated set $\mathcal{U}$ of used variables. Differently, for the updated list $\mathcal{E}$, since it is shared across all algorithm calls, we report it on the right as it gets updated by each algorithm call.

The traversal begins at node σ_{18} (as mentioned above, this is the symbolic state that satisfies the expression in Equation 2), where $\mathcal{U} = \{k\}$ is initialised with the variables read at σ_{18} , and $\mathcal{E}$ is initialised as $[\sigma_{18}]$. The incoming star of σ_{18} contains the symbolic state σ_{17} , which represents the statement marking the end of the conditional (i.e., `endIf`). The analysis continues with a recursive call on σ_{17} . The incoming star of σ_{17} , being after a conditional branching, contains two symbolic states: σ_{14} and σ_{16} . Thus, the algorithm is (recursively) called twice to independently explore the two paths, as represented by the branching of the graph in Figure 5b. Importantly, the used-variable set is copied for each of the two calls: $\mathcal{U}_a$ for the if



(a) SCFG of procedure g from the program in Figure 1. Each node represents a symbolic state, denoted by σ_i with an index i indicating the corresponding line number. The entry state of the procedure is σ_{10} and the exit state is ϵ .



(b) Changes of $\mathcal{U}$ and $\mathcal{E}$ when applying Algorithm 1 to SCFG(g). We showcase the branching of the paths and how $\mathcal{U}$ and $\mathcal{E}$ are computed in that case. In each node, the left side shows the symbolic state being processed and the updated $\mathcal{U}$; for each path we have a different copy of $\mathcal{U}$. On the right side we show the updated $\mathcal{E}$ after processing that symbolic state.

Fig. 5: Example of applying Algorithm 1 to procedure g .

branch (σ_{14}) and $\mathcal{U}_b$ for the else branch. This is needed as, after the two calls, in the if branch, only l is used to compute the value of k , resulting in an updated $\mathcal{U}_a = \{l\}$, while in the else branch both b and l are used, resulting in an updated $\mathcal{U}_b = \{b, l\}$. The two calls then recurse both over σ_{13} , where there is no assignment and thus both continue to σ_{12} . Here, both calls add σ_{12} to $\mathcal{E}$ (note the duplicate in the list), as it is needed for the explanation of l , found both in $\mathcal{U}_a$ and $\mathcal{U}_b$. Ultimately, both calls reach σ_{11} , one with $\mathcal{U}_a = \{y\}$ and the other with $\mathcal{U}_b = \{b, y\}$.

At this point, the $\mathcal{S}$ -traversal algorithm transitions to the inter-procedural analysis and identifies the call site of g within procedure m . The algorithm performs inter-procedural analysis twice—once for each branch—by separately exploring the definitions of variables in $\mathcal{U}_a$ and $\mathcal{U}_b$. Before performing the recursive call on procedure m and the symbolic state where g is called, the sets are updated as follows: $\mathcal{U}_a$ is updated to $\{a\}$, as the variable y is named a in the caller; analogously, $\mathcal{U}_b$ is updated to $\{c, a\}$. When traversing procedure m , in the symbolic state σ_9 there is an assignment to the variable c , found in $\mathcal{U}_b$. Thus, this symbolic state is added to the list of instrumentation points, which is now $\mathcal{E} =$

$[\sigma_{18}, \sigma_{16}, \sigma_{14}, \sigma_{12}, \sigma_{12}, \sigma_9]$, and c is removed from the set $\mathcal{U}_a$ of *read* variables. Finally, to resolve the definition of variable a , the two calls recurse on σ_9 in function k , where m is called. Ultimately the $\mathcal{S}$ -traversal algorithm concludes by returning the list $\mathcal{E} = [\sigma_{18}, \sigma_{16}, \sigma_{14}, \sigma_{12}, \sigma_{12}, \sigma_9, \sigma_6, \sigma_6, \sigma_4, \sigma_4, \sigma_2, \sigma_2]$. In the final list, we highlight the duplicates of σ_6 , σ_4 , and σ_2 , generated by the two separate recursive calls caused by the branching at line 13.

2) *Multiplicity*: As mentioned above, the list of symbolic states $\mathcal{E}$ obtained using Algorithm 1 might contain duplicates, in the case that a symbolic state is relevant to a specification through multiple dataflow paths. Accordingly, we want to compute the multiplicity of each symbolic state as a measure of its relevance for the specification. To compute the multiplicity, we compute from $\mathcal{E}$ a multiset, i.e., a pair $(\mathcal{M}_{\mathcal{E}}, \mu_{\mathcal{E}})$, where the set $\mathcal{M}_{\mathcal{E}}$ contains the unique elements of $\mathcal{E}$ and $\mu_{\mathcal{E}} : \mathcal{M}_{\mathcal{E}} \rightarrow \mathbb{N}$ is a map representing the multiplicity of elements in $\mathcal{M}_{\mathcal{E}}$. For each element $\sigma \in \mathcal{M}_{\mathcal{E}}$, its multiplicity is computed as: $\mu_{\mathcal{E}}(\sigma) = \sum_{x=\sigma_1}^{\sigma_n} [\sigma = x]$ where $[y = z]$ denotes the Iverson bracket, which evaluates to 1 if the predicate $y = z$ is true and 0 otherwise, and $\sum_{x=\sigma_1}^{\sigma_n}$ is the sum over the elements $\sigma_1, \dots, \sigma_n$ of the list $\mathcal{E}$. For example, given the list $\mathcal{E} = [\sigma_{18}, \sigma_{16}, \sigma_{14}, \sigma_{12}, \sigma_{12}, \sigma_9, \sigma_6, \sigma_6, \sigma_4, \sigma_4, \sigma_2, \sigma_2]$, the set $\mathcal{M}_{\mathcal{E}}$ is $\{\sigma_{18}, \sigma_{16}, \sigma_{14}, \sigma_{12}, \sigma_9, \sigma_6, \sigma_4, \sigma_2\}$, and the multiplicity function μ is defined as $\mu(\sigma_{18}) = 1$, $\mu(\sigma_{16}) = 1$, $\mu(\sigma_{14}) = 1$, $\mu(\sigma_{12}) = 2$, $\mu(\sigma_9) = 1$, $\mu(\sigma_6) = 2$, $\mu(\sigma_4) = 2$, $\mu(\sigma_2) = 2$. Thus, the multiset is $(\{\sigma_{18}, \sigma_{16}, \sigma_{14}, \sigma_{12}, \sigma_9, \sigma_6, \sigma_4, \sigma_2\}, \mu)$ with the multiplicity function μ providing the count of each element in $\mathcal{E}$.

B. Filter ι -trace

In this step, we use the set of instrumentation points $\mathcal{M}_{\mathcal{E}}$ to filter an ι -trace I , and obtain a filtered ι -trace that contains only the concrete states necessary for trace checking and diagnostics. We use the notion of sub-traces under a predicate to define a *filtered slice* of an ι -trace I , denoted by f-slice.

Definition IV.3. We say that I' is an f-slice of I if, I' is a sub-trace of I under the predicate

$$P(\langle t, \sigma, m \rangle) = \begin{cases} \text{true} & \text{if } \sigma \in \mathcal{M}_{\mathcal{E}} \\ \text{false} & \text{if } \sigma \notin \mathcal{M}_{\mathcal{E}} \end{cases}, \quad (3)$$

where $\mathcal{M}_{\mathcal{E}}$ is a set of instrumentation points.

We now introduce how we take a ι -trace I and compute a f-slice. Given an ι -trace $I = \langle \mathcal{P}, \{\mathcal{D}_1, \dots, \mathcal{D}_n\}, \mathcal{L} \rangle$ and the predicate P over concrete states (from Equation 3), we compute the f-slice $I' = \langle \mathcal{P}', \{\mathcal{D}'_1, \dots, \mathcal{D}'_k\}, \mathcal{L}' \rangle$ as follows:

- (i) We compute the filtered set of dynamic runs by traversing each original dynamic run $\mathcal{D}_i \in \{\mathcal{D}_1, \dots, \mathcal{D}_n\}$, and retaining only the concrete states c for which $P(c) = \text{true}$. Each non-empty original dynamic run becomes a new dynamic run $\mathcal{D}'_j \in \{\mathcal{D}'_1, \dots, \mathcal{D}'_k\}$. Further, we record the mapping $\gamma(\mathcal{D}'_j) = \mathcal{D}_i$ to link each filtered dynamic run from the f-slice to its original in the ι -trace.
- (ii) We compute the set of procedures $\mathcal{P}'$ in the f-slice as $\mathcal{P}' = \{p : \exists \mathcal{D}' \in \text{dom}(\gamma) : \mathcal{L}(\gamma(\mathcal{D}')) = p\}$; that is, for each filtered dynamic run $\mathcal{D}' \in \{\mathcal{D}'_1, \dots, \mathcal{D}'_k\}$, we

Algorithm 3: computeMap

```

1 Input: An iCFTL specification  $\varphi$  and an  $\iota$ -trace  $\mathcal{I}$ 
   Result: A map from binding/expression pairs to f-slices of  $\mathcal{I}$ 
2  $\text{diagnosticMap} \leftarrow []$ ;
3  $\text{bindings} \leftarrow \{\beta : [\mathcal{I}, \varphi]_S(\beta) = \text{false}\}$ ;
4 for  $\text{binding } \beta \in \text{bindings}$  do
5    $\text{falsifyingAtomicConstraints} \leftarrow$ 
      $\text{getFalsifyingAtomicConstraints}(\mathcal{I}, \beta, \varphi)$ ;
6   for  $\text{constraint } \alpha$  in  $\text{falsifyingAtomicConstraints}$  do
7     for  $\text{expression } e$  in  $\text{constraint } \alpha$  do
8        $\text{f-slice } \langle \mathcal{P}', \{\mathcal{D}'_1, \dots, \mathcal{D}'_n\}, \mathcal{L} \rangle \leftarrow \text{getF-Slice}(\mathcal{I}, \beta, e)$ ;
9        $\text{annotatedFSlice} \leftarrow []$ ;
10      for  $\text{dynamic run } \mathcal{D}'$  in  $\{\mathcal{D}'_1, \dots, \mathcal{D}'_n\}$  do
11        for  $\text{concrete state } \langle t, \sigma, m \rangle$  in  $\mathcal{D}'$  do
12           $\text{annotatedFSlice} \leftarrow$ 
             $\text{annotatedFSlice} \cup \langle \langle t, \sigma, m \rangle, \mu(\sigma) \rangle$ ;
13       $\text{diagnosticMap} \leftarrow$ 
         $\text{updateMap}(\text{diagnosticMap}, \langle \beta, e \rangle, \text{annotatedFSlice})$ ;
14 return  $\text{diagnosticMap}$ ;

```

identify its dynamic run from $\mathcal{I}$ via $\gamma(\mathcal{D}')$ and deduce the corresponding procedure by using the original mapping $\mathcal{L}$ from $\mathcal{I}$.

- (iii) We define the map $\mathcal{L}'$ such that $\mathcal{L}'(\mathcal{D}') = p$ if and only if the original dynamic run $\gamma(\mathcal{D}')$ was mapped to procedure p in $\mathcal{L}$.

Consequently, the f-slice retains only the concrete states that satisfy predicate P in Equation 3, and builds the procedure set and map according to the remaining dynamic runs. We compute the f-slice for each expression in the atomic constraints of the specification, using the set of instrumentation points $\mathcal{M}_E$ of each expression (computed during the instrumentation step, described in § IV-A).

Application to the running example: Continuing our example from § IV-A2, we use the set $\mathcal{M}_E = \{\sigma_{18}, \sigma_{16}, \sigma_{14}, \sigma_{12}, \sigma_9, \sigma_6, \sigma_4, \sigma_2\}$ to filter the ι -trace $\mathcal{I}_{ex}$ in Figure 3. In the ι -trace, we identify all the concrete states across the dynamic runs that are also in $\mathcal{M}_E$, and discard the rest. The resulting f-slice $\mathcal{I}'_{ex}$ consist of the filtered dynamic runs $\mathcal{D}'_k = \langle 0.2, \sigma_2, [a \mapsto 8] \rangle, \langle 0.6, \sigma_4, [b \mapsto 9] \rangle, \langle 1.0, \sigma_4, [b \mapsto 9] \rangle, \langle 1.2, \sigma_6, [a \mapsto 17] \rangle$, $\mathcal{D}'_m = \langle 1.6, \sigma_9, [c \mapsto 9] \rangle$, $\mathcal{D}'_g = \langle 1.9, \sigma_{12}, [l \mapsto 20] \rangle, \langle 2.4, \sigma_{16}, [k \mapsto 29] \rangle, \langle 2.7, \sigma_{18}, [y \mapsto 23] \rangle$. Notably, there is no concrete state corresponding to σ_{14} , as it belongs to a conditional branch that was not taken during execution.

C. Diagnostics

Using the concept of f-slice $\mathcal{I}'$ from the previous section (Definition IV.3), in this step we construct the diagnosis for a falsified specification. More precisely, we construct a diagnosis:

- (i) for each falsified binding of the specification (i.e., each time the specification is assessed over the trace), and
- (ii) for each expression that led to the negative value of the binding.

Accordingly, a diagnosis is a map from pairs of falsified binding and expression, to the f-slice containing the concrete states relevant to that expression.

To compute this diagnostics map from pairs of falsified bindings and expressions to their corresponding f-slice, we apply Algorithm 3 to the iCFTL specification and the ι -trace. This algorithm iterates over the falsified bindings

(line 4). For each binding, it retrieves the set of atomic constraints that falsify it (line 5). Specifically, function `getFalsifyingAtomicConstraints` takes an ι -trace $\mathcal{I}$, a binding β and a specification φ , and returns the set of atomic constraints whose truth value resulted in $[\mathcal{I}, \varphi]_S(\beta)$ being false. Then, for each falsifying constraint (line 6) and for each expression in it (line 7), we retrieve the f-slice containing only the concrete states relevant for the diagnosis (line 8). More precisely, function `getF-Slice` takes the ι -trace $\mathcal{I}$, the binding β and the expression e in the atomic constraint α and retrieves the corresponding f-slice (see § IV-B). To include the multiplicity of each of the concrete states in the f-slice, we expand the f-slice to an *annotatedFSlice* holding such information (line 12). Finally, `updateMap` (line 13) takes the map *diagnosticMap*, along with the key $\langle \beta, e \rangle$ and the value *annotatedFSlice* and updates the map to have $\text{diagnosticMap}(\langle \beta, e \rangle) = \text{annotatedFSlice}$.

Application to the running example: Let us consider again the program in Figure 1 and the specification in Equation 2. In the program's execution trace in Figure 3, the quantifier in the specification has a single binding β from q to $\langle 2.7, \sigma_{18}, [y \mapsto 23] \rangle$ (the only change of variable y during the execution of g). We retrieve the f-slice $\mathcal{I}'_{ex}$ from § IV-B example and annotate it with multiplicity information. We compute the annotated f-slice $\mathcal{I}'^A_{ex}$, which consists of the following dynamic runs:

$$\begin{aligned}
 \mathcal{D}'_k &= \langle \langle 0.2, \sigma_2, [a \mapsto 8] \rangle, 2 \rangle, \langle \langle 0.6, \sigma_4, [b \mapsto 9] \rangle, 2 \rangle, \\
 &\quad \langle \langle 1.0, \sigma_4, [b \mapsto 9] \rangle, 2 \rangle, \langle \langle 1.2, \sigma_6, [a \mapsto 17] \rangle, 2 \rangle, \\
 \mathcal{D}'_m &= \langle \langle 1.6, \sigma_9, [c \mapsto 9] \rangle, 1 \rangle, \\
 \mathcal{D}'_g &= \langle \langle 1.9, \sigma_{12}, [l \mapsto 20] \rangle, 2 \rangle, \langle \langle 2.4, \sigma_{16}, [k \mapsto 29] \rangle, 1 \rangle, \\
 &\quad \langle \langle 2.7, \sigma_{18}, [y \mapsto 23] \rangle, 1 \rangle.
 \end{aligned}$$

Finally, we construct the diagnostic map $(\beta, q(y)) \mapsto \mathcal{I}'^A_{ex}$.

V. IMPLEMENTATION

We have implemented our diagnostics approach in a prototype tool named `iCFTLdiagnostics`. Our tool builds upon a prior extension for diagnosing time-based specifications [12], which in turn is based on the existing framework for instrumentation and trace checking with respect to iCFTL [6]. To support state-based specifications diagnostics, we further extended this framework by modifying both the instrumentation component and the ι -trace filtering algorithm. As mentioned in § IV, in practice, we perform the Instrumentation and Execution steps sequentially: we first instrument the program, and then we execute it. In this way, the execution directly records only the relevant concrete states, generating a smaller ι -trace and reducing the computational overhead of the instrumentation. Below, we describe the state-based diagnostics enhancements made to the extended iCFTL infrastructure and list the limitations of our tool implementation.

Diagnostics instrumentation: We further modified the existing instrumentation, by extending the diagnostics instrumentation to include program points relevant to state-based specifications (as described in § IV-A).

Diagnostics: We further extended the diagnostics capabilities of iCFTLdiagnostics to diagnose state-based specifications, by extending the verdict to the map discussed in § IV-C. We integrated the diagnostics component after the Monitoring process. Given the Verdict and Filtered ι -trace, we apply the diagnostics approach and compute a diagnosis for each expression in the violated specification. Each concrete state included in the diagnosis is annotated with its multiplicity, indicating through how many distinct execution paths it contributed to the specification violation.

Limitations: The implementation of our approach inherited the known limitations of the original implementation of iCFTL [6] that it builds upon:

- **Global variables:** The implementation of the tool does not instrument global variables, which may lead to missed concrete states when the analysis involves concrete states defined or modified outside the programs’ procedures.
- **Class Instantiations and Class-level Variables:** The implementation of the tool does not instrument program points within class-level constructs, such as object instantiations, and class member variables.
- **Array/List Element Definitions:** Arrays and list are treated as a “whole,” and definitions or updates to individual elements within arrays or lists are not tracked.
- **Complex Function Parameters:** The implementation of the tool does not fully resolve nested or transformed parameters within function calls. For example, in the function call `move(a, int(b))`, the implementation will fail to recognize that `int(b)` references the variable `b`, potentially omitting concrete states from the ι -trace relevant to the computation of its value.

We will discuss the impact of these limitations in the next section, when analysing the experimental results.

VI. EVALUATION

In this section, we report on the experimental evaluation of our iCFTLdiagnostics tool. We focus on the effectiveness of the tool in identifying the causes of state-based specification violations, as well as on its efficiency. Regarding the *effectiveness*, we assess the tool ability to identify statements that are relevant to a given specification (i.e., the statements that can affect the value of the variables used in a specification), as well as the reduction—in terms of statements to inspect—of the effort for identifying the cause of specification violations when the tool output is used. As for the *efficiency*, we assess the time taken and the memory used by our diagnostics approach, as well as the overhead introduced by the corresponding instrumentation. We summarise these objectives with the following research questions:

- RQ1:** How effective is iCFTLdiagnostics in identifying the statements relevant to a specification violation?
- RQ2:** To what extent does iCFTLdiagnostics reduce the complexity of understanding specification violations?
- RQ3:** How much time and memory does the iCFTLdiagnostics approach require?
- RQ4:** What is the overhead in terms of time and memory consumption introduced by the iCFTLdiagnostics instrumentation?

TABLE I: List of Selected Project from the DyByBench benchmark.

Project Name	Domain	# Specs	SLOC			# Functions		
			Min	Max	Avg	Min	Max	Avg
PythonRobotics	Robotics	92	53	454	193.85	4	33	12.66
flask_api	RESTful API	2	52	97	74.50	4	7	5.50
seaborn	Data visual.	7	410	2497	1944	26	74	58.71
pyfilesystem2	Files	4	70	677	410.75	6	61	36.50
pypdf	Format proc.	2	317	481	399	20	32	26
pdoc	Documentation	1	341	341	341	21	21	21
pyjwt	Authentic.	1	109	109	109	12	12	12
arrow	Date & time	1	868	868	868	71	71	71
pudb	Debugging	1	443	443	443	18	18	18
elasticsearch_dsl	Search	1	45	45	45	6	6	6

A. Datasets and Settings

1) *Evaluation Subjects:* We evaluated iCFTLdiagnostics using subjects from different domains. As the tool supports Python programs, we looked for a benchmark written in this language and including test cases (so that we could assess our approach on realistic executions of the programs). Under these constraints, we selected the DyByBench benchmark [13], which encompasses 50 Python open-source projects from various application domains, with multiple test cases for each project. Notably, the benchmark includes projects for which the test cases provide high code coverage. Such a high coverage is important to assess our tool, as it allows us to assess our tool ability to trace relevant statements through different parts of the program.

The assessment of iCFTLdiagnostics requires to define state-based specifications, i.e., specifications that constrain the values assigned to numerical variables, like integers and floats. Thus, among the projects in the benchmark, we selected those containing programs over which we could write state-based specifications (leaving us with 26 projects). However, due to the limitation of the underlying iCFTL implementation, we could not write specification over elements of arrays and lists. Consequently, we excluded 16 projects where numerical variables appeared only in arrays and lists. After filtering out these projects, we were left with a total of 10 projects, listed in Table I together with their application domain and the number of specifications (# Specs) written for the programs in each of the projects. Additionally, for the programs within these projects over which the specifications were written, we provide the minimum, maximum, and average values of Source Lines of Code (SLOC) and the number of functions (# Functions); we computed the SLOC using the Radon tool [14] to count non-comment, non-blank lines, and used the Python ast module [15] to traverse the abstract syntax tree and count function definitions.

2) *Specifications:* To operate our tool on diverse portions of the programs, for each of the 10 selected projects, we wrote each specification for a different procedure in them; this ensures different instrumentation points for the different specifications. When writing the specifications, we targeted variables in procedures executed by the test cases. Furthermore, to write specifications relevant to the intended purpose of the procedure under test, when possible, we based them on the oracles available with the test cases. For example, if a test case contained an assertion statement over a variable, such as `assert(len(x)==3)`, we included this assertion in the specification. Finally, to ensure that the trace diagnostics be triggered,

we wrote specifications that would be violated (i.e., would provide a negative verdict) by the test execution. The specifications contain either a single atom or a Boolean combination of atoms. For example, a single atom specification looks like $\forall q \in \text{changes}(x).\text{during}(g) : q(x) < 10$, while a specification with multiple atoms looks like $\forall q \in \text{changes}(x).\text{during}(g) : q(x) < 0 \wedge q.\text{next}(\text{changes}(y).\text{during}(g) > 10)$. To give some figures, 64 specifications have one quantifier and one atom, 7 specifications have one quantifier with two atoms, 35 specifications have two quantifiers with one atom, 3 specifications have two quantifiers with two atoms, and 3 specifications have three quantifiers with two atoms. Importantly, the specification examples that we provided Section II-C served as templates for the specifications that we used in our evaluation. In total, we wrote 112 specifications for the different functions in the 10 selected projects from the benchmark.

3) *Settings*: All experiments were conducted on the Aion High-Performance Computing system at the University of Luxembourg.³

We did not consider any baseline because, to the best of our knowledge, no existing tool provides diagnostics for *state-based* specifications in the context of iCFTL or similar specification languages.

B. RQ1: Effectiveness of iCFTLdiagnostics

To assess the effectiveness of our tool in identifying statements relevant to a specification violation, we compared the instrumentation points generated by iCFTLdiagnostics for the evaluation subjects (and their respective specifications) with a manually defined ground truth (GT). We created this GT by manually inspecting the code and computing an explanation set for each specification. Since the GT is computed statically, we do not have access to the execution traces. Therefore, we evaluate effectiveness with respect to the statically determined instrumentation points produced by the iCFTLdiagnostics approach.

1) *Ground Truth Definition*: The GT is the set of program points that can affect the outcome verdict of a given specification (i.e., the set of relevant program points). We created this GT by manually inspecting the code and computing the instrumentation points for each specification. The GT was first created by the first author, then validated independently by the second author, and finally discussed by the two until convergence.

While all the identified program points can affect the specification outcome, their impact on the specification outcome can vary. In particular, program points that are closer—in terms of backward dataflow dependency—to where the specification is violated are expected to have a more direct influence on the outcome. Intuitively, missing a program point that directly assigns a value to a variable used in the specification is more detrimental than missing one that assigns a value to a variable not directly used in the specification. More rigorously, while

all the program points in the GT are connected by a dataflow dependency with at least one of the program points that can violate the specification, each of them can be located in a different procedure of the program. Accordingly, missing a program point close (in terms of dependency) to where the specification is violated is more detrimental than missing one that is further away in the dependency chain.

To capture this distinction, we assign a value, which we call *proximity*, to each program point in the GT. We define the proximity value based on the function that the program point belongs to: it measures the distance from the function where the specification was violated to the function the program point is in. Each program point receives a proximity value equal to 0 if it is in the same function where the specification was violated; 1 if it is in a function that calls the function at proximity 0; 2 if it is in a function that calls a function at proximity 1; and so on. While we define proximity for any inter-procedural distance, we also define proximity for relevant program points outside of any procedure in the program (e.g., global and class variables) for which we set the proximity value to ∞ . In the special case that a program point is connected by backward dependency with multiple points where the specification might be violated and thus it can have multiple proximity values, we select the lowest one. We exemplify the proximity values using the program points in the diagnosis of the specification in Equation 2 for the program in Figure 1. As mentioned above, the quantifier in the specification is satisfied by the symbolic state σ_{18} . Among the symbolic states identified in our diagnosis (see § IV-C), proximity 0 includes σ_{16} , σ_{14} , and σ_{12} , as they are within the same procedure as σ_{18} . Proximity 1 includes σ_9 , as it is located in procedure *m*, which directly calls procedure *g*. The remaining program points within procedure *k* are at proximity 2, since *k* calls *m*, which in turn calls *g*. Finally, in this example, there are no program points with assigned proximity of ∞ , i.e., global or class variables.

2) *Methodology*: For each specification, we measured the effectiveness of our tool by comparing the GT with the instrumentation points generated by the tool itself. We performed this comparison for different proximities, grouping program points cumulatively based on their proximity level: for a given proximity *n*, we included program points having a proximity value ranging from 0 to *n*. When comparing at proximity ∞ , we compared the instrumentation points with the GT set including all program points. If, for a given specification and proximity, the two subsets coincided, i.e., they contained the same program points, then we concluded that our tool identified *all* the relevant statements. The more the two subsets differed, the least effective we deemed our tool.

As such, we compare the GT set with the iCFTLdiagnostics instrumentation points set $\mathcal{M}_E$ for a given proximity. To compare the sets, we considered as a True Positive (TP) a program point in $\mathcal{M}_E$ that is part of the GT, a False Positive (FP) a program point in $\mathcal{M}_E$ that is not part of the GT, and a False Negative (FN) a program point in the GT that is not present in $\mathcal{M}_E$. We then computed *precision* $p = \frac{TP}{TP+FP}$ and *recall* $r = \frac{TP}{TP+FN}$. Precision measures the percentage of the identified program points that are actually

³Experiments were conducted on a node equipped with an AMD EPYC ROME 7H12 processor (64 cores, 2.6 GHz, 280 W) and 128 GB of RAM. Further system specifications are available at: <https://hpc-docs.uni.lu/systems/aion/>.

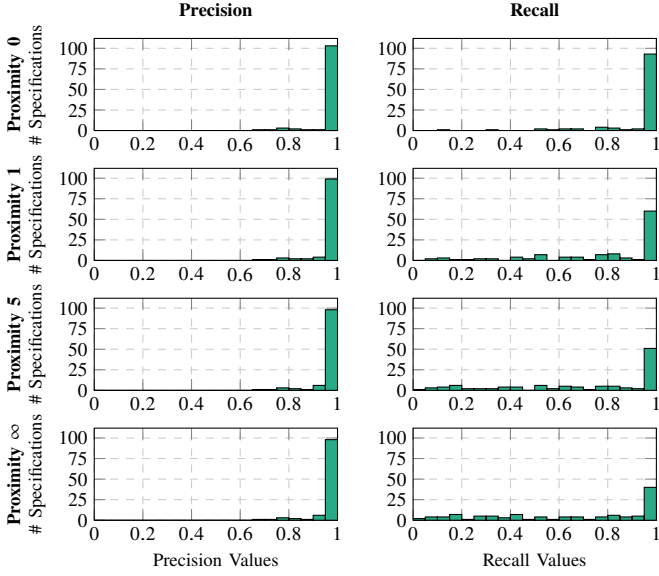


Fig. 6: Histograms of precision (left) and recall (right) values obtained when comparing the GT and the $\mathcal{M}_E$ sets. In each row, we report the results for increasing proximity values. High precision and recall values indicate overlap between the two sets, and therefore good performance of our tool.

relevant to the specification violation. Recall instead measures the percentage of relevant statements that our tool was able to identify. Using this definition, to answer our RQ, we executed our tool on the 112 specifications, obtained the corresponding $\mathcal{M}_E$ set and computed precision and recall at different proximities.

3) *Results*: We report the results of our experiments in Figure 6 as histograms of the precision and recall values over the 112 specifications. In the figure, the histograms in the left-hand column show the precision values, and the ones in the right-hand column show the recall values. The different rows instead show the results for proximity values 0, 1, 5 and ∞ . In terms of precision, our results show high values across all proximity values, with at least 103 out of 112 specifications having a precision above 90%. This gives us confidence that the vast majority of program points detected by iCFTLdiagnostics are indeed relevant.

In terms of recall, we observe that for proximity 0, 95 out of 112 specifications have a recall higher than 90%, indicating that iCFTLdiagnostics detects the vast majority of relevant program points within the function where the specification violation happens. When we consider higher proximity values, the recall values decrease for some of the specifications. We have 61 specifications with recall higher than 90% for proximity 1, 53 specifications for proximity 5, and 45 specifications for proximity ∞ . Upon inspection, we observed that the missing statements leading to a reduction of the recall value are due to the known limitations of iCFTLdiagnostics. Notably, iCFTLdiagnostics does not instrument global variables, class instantiations and class-level variables, array/list elements, and complex function parameters. For instance, global variables assignments can be

part of the dataflow path leading to a specification violation and therefore are included in the GT. However, since these statements are not part of any procedure, they are not detected by iCFTLdiagnostics, resulting in missing statements, as well as any other statements relevant to them. This limitation appears at higher proximity levels, contributing to the observed decrease in recall for proximity ∞ . As an example, consider the template specification in Equation 2 that predicates on the change of variable y , and an example statement $y = \text{self.h} + n[j]$ where y is changed. In this statement, self.h refers to a class-level variable and $n[j]$ is an array element. Although these statements are included in the GT as they are part of the dataflow, they are not instrumented by iCFTLdiagnostics, since resolving the values of self and j in an automated fashion during instrumentation is a difficult task. Consequently, statements that affect those variables will not be identified as relevant, leading to a decrease in recall. Since higher proximity values increase the probability of finding such constructs, the recall decreases for higher proximity values.

The complete list of such limitations is reported in § V.

RQ1 answer. iCFTLdiagnostics identifies the program points relevant to the specification violation with high precision across different proximity values. Furthermore, our experiments also show high recall at proximities 0 and 1, indicating that the tool is effective in detecting relevant program points in the function where the specification is violated and in the functions that call it. While the recall decreases for higher proximities (>1), due to the known limitations of the tool implementation, this is less critical than missing program points closer to the specification violation.

C. RQ2: Reducing Complexity in Understanding Violations

To answer RQ2, we measured the complexity of understanding a specification violation in terms of the code that needs to be inspected after observing a specification violation. The idea is that, compared to using no tool, iCFTLdiagnostics leads to the inspection of a *smaller* portion of the program for understanding the violation.

1) *Methodology*: We evaluate the reduction of complexity achieved by iCFTLdiagnostics by comparing the code that needs to be inspected with and without our tool. We consider two separate debugging scenarios, distinguishing between two developer profiles. In the first scenario, we consider an “unfamiliar developer” that has no prior knowledge of the codebase. For this developer, without tool support, the baseline is to inspect all program lines. In the second scenario, we consider a “familiar developer” that has prior knowledge of the codebase—e.g., having contributed via Git commits [16, 17]. Specifically, we consider that the familiar developer already knows which functions contain relevant statements. For this developer, without iCFTLdiagnostics, the baseline is to inspect the (lines of code in the) functions that contain at least one statement in our GT.

We compare the code that needs to be inspected in terms of the Halstead Effort metric [18], which measures

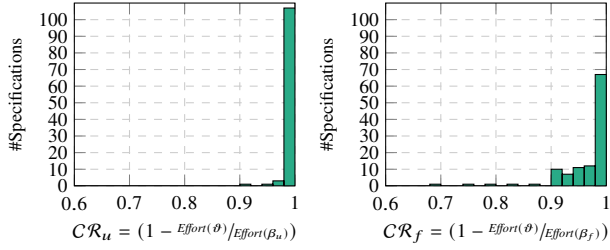


Fig. 7: Reduction of complexity across the specifications for the two developer profiles: unfamiliar (CR_u , left) and familiar (CR_f , right). The reduction of complexity is computed as percentage variation of the Halstead Effort.

the work required to comprehend a piece of code based on its size and the diversity of its operators and operands. Importantly, recent work [19] showed that this metric exhibits the strongest correlation with cognitive load measured with electroencephalography (EEG) among state of the art code-complexity metrics. The Halstead Effort is computed based on the *total* number of operators N_1 and operands N_2 , and the number of *distinct* operators n_1 and operands n_2 . The metric is then defined as $Effort = Volume \times Difficulty$ where $Volume = (N_1 + N_2) \times \log_2(n_1 + n_2)$ and $Difficulty = \frac{n_1}{2} \times \frac{N_2}{n_2}$. The *Volume* represents the information content of the code, while *Difficulty* reflects the complexity of the relationships among operators and operands.

For each developer profile, we express the reduction of complexity as the percentage decrease in Halstead Effort compared to the baseline. In practice, we compute the Halstead Effort for the set ϑ of program points identified by `icFTLdiagnostics`, and for the two baselines, i.e., the entire program β_u for the unfamiliar developer and the code of the functions that contain at least one GT statement β_f for the familiar developer. For each of our 112 specifications, the reduction of complexity for the unfamiliar developer is

$$CR_u = \left(1 - \frac{Effort(\vartheta)}{Effort(\beta_u)}\right)$$

where $Effort(\cdot)$ denotes the Halstead Effort computed on the given portion of the program. Similarly, the reduction of complexity for the familiar developer is

$$CR_f = \left(1 - \frac{Effort(\vartheta)}{Effort(\beta_f)}\right).$$

To compute the Halstead Effort, we use the multimetric Python library⁴, which calculates software metrics over a given file.

2) *Results*: In Figure 7, we report the CR_u and CR_f values obtained for our specifications. The histograms show the distribution of the reduction of complexity values across the 112 specifications. For the unfamiliar developer (left-hand side of the figure), the histogram shows that for all specification our tool achieves a reduction of complexity greater than 90%. Notably, for 107 of 112 specifications, the reduction of complexity is greater than 98%. For the familiar

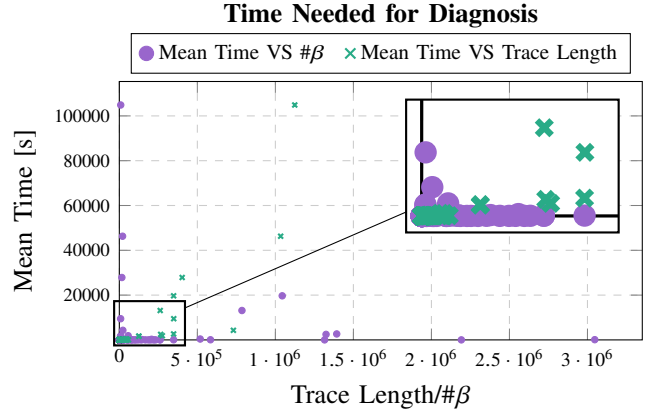


Fig. 8: This figure shows the computational time required by our diagnosis step for our evaluation subjects. For each dot of the scatter plot, the vertical coordinate is the time required for performing the diagnosis, the horizontal coordinate is the number of bindings for the purple dots, and trace length for the green crosses.

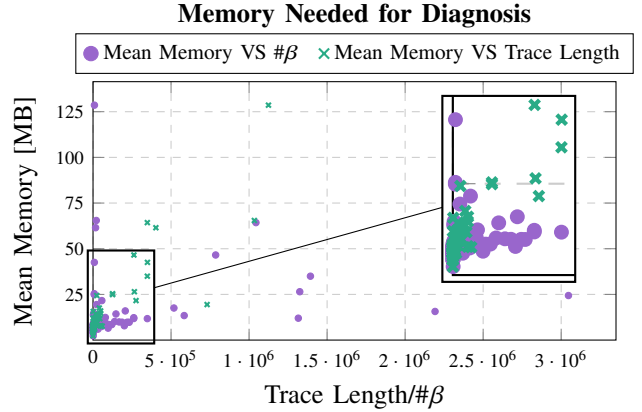


Fig. 9: This figure shows the peak memory required by our diagnosis step for our evaluation subjects. For each dot of the scatter plot, the vertical coordinate is the peak memory required for performing the diagnosis, the horizontal coordinate is the number of bindings for the purple dots, and trace length for the green crosses.

developer (right-hand side of the figure), the reduction of complexity is greater than 90% for 107 of 112 specifications, with 67 specifications showing a reduction larger than 98%. These high percentages of reduction of complexity confirm that `icFTLdiagnostics` substantially minimises the analysis effort, also for developers already familiar with the codebase.

RQ2 answer. Our experiments show that `icFTLdiagnostics` can reduce the effort needed to understand a specification violation (quantified in terms of the Halstead Effort) by more than 90% for the vast majority of our considered specifications. Notably, the reduction of complexity is high for both the developer profiles considered, unfamiliar and familiar. By narrowing the focus to a small relevant subset of

⁴<https://pypi.org/project/multimetric/>

the codebase, our tool effectively reduces cognitive load compared to the baseline.

D. RQ3: Efficiency of iCFTLdiagnostics

To answer this RQ, we measured the time and memory taken by the iCFTLdiagnostics tool to analyse the traces generated by our evaluation subjects for each specification.

1) *Methodology*: To measure the time cost, we used the time module from the Python standard library to obtain the wall clock time. We then computed the difference between the clock value before and after running the diagnosis approach algorithm to obtain its time cost. To measure the memory cost, we used the tracemalloc module, which allows one to trace memory allocations. We enabled and disabled the tracing at the start and end of the diagnostics execution, and measured the *peak size* of allocated memory. By measuring the peak memory allocation, we obtain an upper bound of the memory required by the iCFTLdiagnostics tool. To account for the time and memory consumption variability across different executions, we executed the diagnosis five times and calculated the average time and memory. Furthermore, to avoid interferences between the time and memory measurements, we used separate runs for the two quantities.

For each program and specification pair, the diagnostics is triggered by the occurrence of a falsified binding. Furthermore, the diagnostics cost is influenced by the number of events to be processed, which depends on the length of the trace. Since the number of bindings and trace length quantities can vary over different program and specification pair, we also studied the time and memory cost in relation to them. Our 112 specifications show a wide variety of trace length values and number of bindings. Trace lengths vary from 1 to 1 124 250, with an average of 54 930.71, a median of 1031, and a standard deviation of 173 105.47. The number of bindings ranges from 3 to 3 046 663, with an average of 140 088.16, a median of 1874, and a standard deviation of 427 137.97. The difference between the average and the median shows that the distributions are more skewed toward the lower values. However, the high value for standard deviation shows that there is high variety of trace lengths and number of bindings in our dataset. This diversity allows us to assess our diagnostics approach on diverse application contexts.

2) *Results*: We report our experiments results for the time and memory cost in Figures 8 and 9, respectively. In each figure, we show the computational cost of each specification in relation to the trace length (TL) with green crosses, and in relation to the number of bindings ($\#\beta$) with purple dots.

In Figure 8, we observe that most of the dots/crosses are grouped in the bottom part of the plot, desirably showing short diagnosis times. Specifically, we report that 100 out of the 112 specifications require less than 400 s, or approximately 7 min. Over the five repeated runs, for specifications requiring less than 400 s, the average standard deviation is approximately 4 s. In contrast, when considering all specifications (thus also the more computationally-intensive ones), the average standard deviation rises to about 10 min. This is expected, as longer processes are exposed to more computational variability.

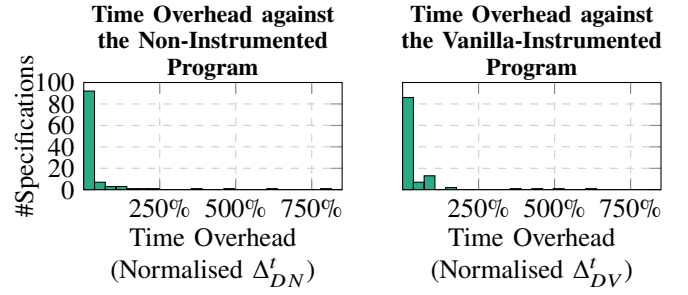


Fig. 10: The program execution time overhead generated by iCFTLdiagnostics in our evaluation subjects.

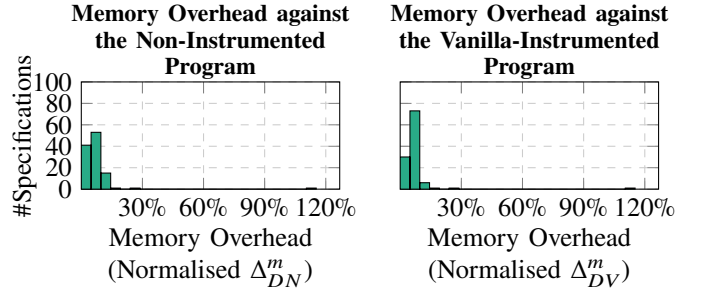


Fig. 11: The program execution peak-memory overhead generated by iCFTLdiagnostics in our evaluation subjects.

To quantitatively study the correlation between time cost, and trace length and number of bindings, we used the *Spearman's rank correlation*, usually denoted with the letter ρ , which studies monotonic relationships between variables and does not require assumptions on the distribution of the data. A Spearman's correlation value close to 1 corresponds to a strong monotonic positive relation between the two variables, a value close to -1 corresponds to a strong monotonic negative relation, and a value close to 0 corresponds to no monotonic correlation. Between time t and trace length TL, we observe $\rho_{TL}^t = 0.93$, which is classified as a *very strong* positive monotonic relationship. This means that as TL grows, the time required for diagnosis increases. Between time t and number of bindings ($\#\beta$), we observe $\rho_{\#\beta}^t = 0.87$, also indicating a strong monotonic relationship. This value of the Spearman's correlation coefficient indicates that, across the dataset, specifications with higher binding counts consistently require more time, even if the increase is modest for most cases, as the purple dots are mostly concentrated in the bottom part of the plot.

In Figure 9, we observe that 103 out of 112 specifications desirably require less than 25 MB of peak memory. For these 103 specifications, the average standard deviation is approximately 3 MB. When considering all specifications, the average standard deviation is 3.45 MB, with the 9 specifications that exceed the 25 MB threshold, showing a variability of 9 MB. We observe that the memory cost increases as both TL and $\#\beta$ increase. Between memory usage m and TL, we observe $\rho_{TL}^m = 0.72$, indicating a strong monotonic relationship. Between memory usage m and $\#\beta$, the coefficient is similar,

with $\rho_{\#\beta}^m = 0.72$. Spearman’s correlation confirms that specifications with higher binding counts consistently require more memory.

RQ3 answer. The computational cost of iCFTLdiagnostics is low, requiring less than 7 min of time for the diagnosis and a peak memory usage of less than 25 MB for processing our specifications. This indicates that iCFTLdiagnostics is usable in practice, also for specifications that yield trace length (TL) and number of bindings ($\#\beta$) up to 500 000. While the use of the High-Performance Computing system may have influenced the observed performance positively when compared to using a smaller machine, the diagnostics can be run as part of a continuous integration pipeline, making our results adherent to a practical use case. Furthermore, Spearman’s correlation confirms that specifications with larger trace lengths and binding number increase both time and memory costs.

E. RQ4: iCFTLdiagnostics Instrumentation Overhead

We answer this RQ by measuring the execution time and memory overhead that our new instrumentation approach introduces on program execution. As our work builds on top of the iCFTL [6] framework, we study the program-execution overhead generated by iCFTLdiagnostics both in absolute terms, comparing with the non-instrumented program, and relative terms, comparing with the basic iCFTL instrumentation without any diagnostics functionality (hereafter referred to as vanilla instrumentation).

1) *Methodology*: Similar to RQ3, to measure the peak memory cost, we used the `tracemalloc` module. To measure the execution time of each test case, we used the `Unix time` command. We computed the two overheads as the time and memory delta between the diagnostics instrumentation and both the vanilla instrumentation and the non-instrumented program. To ensure comparability across different specifications, we normalised these differences, specifically:

- The overhead relative to the vanilla instrumentation was divided by the vanilla time and memory costs.
- The overhead relative to the non-instrumented program was divided by the corresponding non-instrumented time and memory costs.

For each specification, we run the associated test case triggering the execution of the program under the different instrumentation schemes. Like RQ3, we run separate experiments for time and memory. To account for the variability of the individual executions, we executed the test cases 5 times each and compute the mean of both quantities.

2) *Results*: We report the overhead introduced by our instrumentation on program execution in Figure 10 for time and in Figure 11 for memory. In both figures, the histograms on the left show the diagnostics instrumentation (D) overhead with respect to the non-instrumented program execution (N), which we denote as Δ_{DN}^t for time and Δ_{DN}^m for memory. The histograms on the right show the diagnostics instrumentation (D) overhead with respect the vanilla instrumented program

execution (V), which we denote as Δ_{DV}^t and Δ_{DV}^m . As discussed above, the x-axis shows the normalised delta in percentage computed as $\Delta_{DV} = \frac{D-V}{V} * 100$ and $\Delta_{DN} = \frac{D-N}{N} * 100$. The x-axis counts the number of specifications showing a given normalised overhead value.

In Figure 10, we observe that the time overhead Δ_{DN}^t is under 30% for 91 out of 112 specifications. Similarly, the time overhead Δ_{DV}^t is under 30% for 81 specifications. Similar values of Δ_{DN}^t and Δ_{DV}^t for most of our specifications indicate that the execution times for non-instrumented and vanilla-instrumented programs are very close. Thus, we deduce that most of the overhead comes from the instrumentation for the diagnosis rather than the instrumentation for the sole monitoring of the specification (i.e., the vanilla instrumentation points). This is coherent with the fact that the diagnosis instrumentation points include both the vanilla instrumentation points and all those program points that can affect them. Finally, by inspecting the outliers where D takes more than 200% compared to N and V , we note that they are associated with traces with at least half million events. For these specifications, the diagnostics requires instrumenting program points within loops executed numerous times that therefore generate a large number of events. In Figure 11, we observe for peak memory overhead similar trends as for time. We see that both for Δ_{DN}^m and for Δ_{DV}^m almost all of our specifications (110 out of 112) have under 20% increase in memory consumption. Only two specifications have an overhead over 20% and only one of those has an overhead over 100%.

RQ4 answer. In 90 out of 112 of our experiments, iCFTLdiagnostics shows a limited overhead of less than 30% in time. In terms of peak memory, 109 out of 112 specifications show less than 15% overhead. While not negligible, this overhead is still limited and, depending on the specific application, it can be outweighed by the gained diagnosis information.

F. Threats to validity

To answer RQ1, we relied on a manually constructed ground truth of relevant program points for each specification, due to the absence of established diagnosis benchmarks in the literature. The manual construction of the GT introduces potential threats to construct validity, as the human factor lead to oversight, under-approximation (missing program points that belong to the GT), or over-approximation (including extra program points that are not actually relevant). To the best of our ability, we mitigated these risks by adopting practices recommended in empirical software engineering research: (i) we defined clear inclusion and exclusion criteria prior to annotation to reduce ambiguity (i.e., Definition IV.2), (ii) two authors *independently* annotated the ground truth and cross-validated their results, and (iii) discrepancies were resolved through joint code review and consensus. Thus, any missing or extra program point should have been independently and erroneously missed or included by both authors. These measures minimise individual bias, ensuring that the ground truth provides a reliable basis for our evaluation.

Another potential threat comes from the manual definition of the specifications, which may limit the generalisability of our experiments to real-world applications. Since we, rather than the developers of the projects, wrote the specifications, they might not fully capture meaningful or representative behaviours of the systems under study. However, we based our specifications on available test cases and documentation, either by formalising constraints checked in assertions, or by covering code exercised during test execution. This approach increases our confidence that the specifications are reasonably aligned with what a project developer might have written.

The measurement of memory usage and execution time—both for the diagnostic process (RQ3) and instrumentation overhead (RQ4)—is subject to variability, particularly in a language like Python, which lacks strict performance guarantees. To mitigate this threat, we used repeated experiments to average out the variability. Furthermore, the results show consistently a low demand of time and computational resources, with few exceptions. This gives confidence on the limited computational demands of iCFTLdiagnostics and thus on our conclusions about its practical applicability.

G. Data availability

We make the data obtained during our experiments and the source code of our implementation available on figshare at <https://doi.org/10.6084/m9.figshare.29666057>.

VII. RELATED WORK

Our work focuses on identifying the program points that contribute to how a variable obtains its value. This goal intersects with several areas of software engineering, which include trace diagnostics (TD), fault localisation (FL), causal-analysis (CA), and vulnerability analysis (VA).

Table II compares our approach with the related fields of TD, FL, CA, and VA, considering the following facets:

- **Representative Techniques:** Examples of well-known tools or frameworks in each area, illustrating typical approaches.
- **Problem Statement:** The primary goal each area addresses, such as explaining specification violations, locating faulty statements, reconstructing cause-effect chains, or identifying security weaknesses.
- **Input Artefact:** The primary artefacts consumed by the analysis, including dynamic execution data (e.g., traces, state transitions), static program representations (e.g., source code, control-flow graphs, code property graphs), and formal specifications.
- **Output:** The artefacts generated by the approach, ranging from minimal trace segments to ranked statements, cause-effect chains, or vulnerability reports.
- **Granularity:** The level of detail at which the analysis operates (e.g., specification-level, statement-level, state-level, slice-level). It determines how fine-grained or coarse-grained the reasoning is, which in turn affects the precision and interpretability of the output.
- **Use of Runtime Info:** Whether the proposed approach relies on dynamic execution traces or runtime monitoring.

- **Use of Static Analysis:** Whether the proposed approach relies on static program analysis techniques.
- **Application Domain:** The broader areas the techniques are typically applied, such as debugging, verification, or security auditing.

From the comparison table, we observe that for the input artefacts, TD, FL, and CA depend heavily on execution traces or available test cases. Differently, our approach operates mostly on the specification and program source code, and requires only one execution trace. In terms of generated output, all approaches generate either trace segments, ranked statements, or vulnerability reports. Our iCFTLdiagnostics provides an f-slice of the execution linked by data-flow to the specification violation. In terms of granularity, unlike other techniques that typically work at statement-, state-, or specification-level, iCFTLdiagnostics offers expression-level granularity, where the diagnosis of the violated specification showcases the exact expression in the specification that contributes to the violation and its relevant program points. The comparison on the use of runtime information and static analysis highlights the main distinguishing point of iCFTLdiagnostics, being it the only approach that combines a strong use of both static and dynamic analysis for diagnosis generation.

In the following, we discuss in more detail, for each field, the commonalities and distinctions with our approach.

Trace diagnostics: Our approach belongs to the field of TD, aiming to explain why a specification is violated by analysing execution traces. A number of TD techniques have been proposed for both source code-level and signal-level specifications. Notably, Dawes and Reger [10] proposed a diagnostic method for CFTL (the predecessor of iCFTL), which identified problematic code segments using path profiling. In contrast, our approach focuses on identifying the program points that contribute to how a variable obtains its value, rather than focusing on path profiling. Other approaches focus on signal-based properties. Boufaied et al. [11] proposed the TD-SB-TemPsy approach for trace diagnostics of SB-TemPsy-DSL properties, including a catalogue of 34 violation causes, each linked to a corresponding diagnosis. Similarly, Dou et al. [32] proposed TemPsy-Report, a model-driven approach that retrieved diagnostic information from a predefined list of violation types. Ferrère et al. [9] formulated the diagnostics problem as identifying a minimal portion of the input signal that is sufficient to imply the violation of a specification. Ničković et al. [21] implemented in the AMT 2.0 tool two diagnostic procedures: one that computed minimal explanations based on the method of Ferrère et al. [9], and another that returned extended explanations using epoch diagnostics. Bartocci et al. [20] proposed the CPSDebug tool, which explains failures in Simulink/Stateflow models by generating a sequence of snapshots that highlight faulty behaviours. Basnet and Abbas [33] presented a signal-level diagnostic technique that transforms time-domain signals into the frequency domain to identify frequency components relevant to a temporal logic formula, thereby enabling monitoring-safe compression. Clauss et al. [34] modelled program behaviour by segmenting execution traces into phases and fitting low-degree polynomials with pe-

TABLE II: Comparison of iCFTLdiagnostics with related approaches in the fields of trace diagnostics, fault localisation, causality analysis, and vulnerability analysis.

Field Facet	Trace Diagnostics (TD)	Fault Localisation (FL)	Causality Analysis (CA)	Vulnerability Analysis (VA)	Our approach
Representative Techniques	TD-SB-TemPsy [11], CPSDebug [20], AMT 2.0 [21].	Ochiai, Tarantula (Spectrum based FL) [22, 23], GenProg, Fix-Con (Automated repair) [24, 25], Failure proximity [26].	Cause-Effect Chains [27], Cause Transitions [28], ROCAS (Root Cause Analysis for ADS) [29].	Security slices for injection vulnerabilities [30], Joern (graph-based static analysis) [31].	iCFTLdiagnostics (diagnosing state-based specifications).
Problem statement	Explain why a specification is violated by analysing execution traces often focusing on identifying trace segments or signal fragments contributing to violations [10, 11, 9, 21, 32, 20, 33, 34].	Identify faulty code statements causing test failures [22, 23, 26], Isolate cause-effect chains [27, 28], Enable automated repair [24, 25].	Identify chains of events leading to failures [27, 29], Locate critical transitions where responsibility for failure shifts [28].	Identify and evaluate security weaknesses in systems, networks, and applications [30, 31].	Generate informative verdicts for violated iCFTL state-based specifications.
Input Artefact	Execution traces: Signal-level [11, 9, 21, 33], Model-level [20, 32, 34], Source-level [10].	Passed and failed test execution traces [22, 23, 26], Execution traces and state transitions for cause-effect isolation [27, 28], Program code for automated repair [24, 25].	Execution traces and state transitions [27, 28, 29].	Source code and code property graphs [31], Security slices for auditing [30].	State-based specification and program.
Output	Minimal trace segments or signal fragments [9, 21, 33], Structured diagnostic reports or catalogues of violation causes [11, 32], Visual or model-based representations of faulty behaviours [20, 34], Identification of problematic code segments [10].	Suspicion scores or ranked faulty statements [22, 23], patches for repair [24, 25], cause-effect chains [27, 28], Failure proximity-based ranking [26].	Cause-effect chains [27], Cause transitions [28], Root cause reports for accidents in ADS [29].	List of vulnerable code segments or security slices [30], Graph-based vulnerability reports [31].	f-slice for a falsified specification.
Granularity	Specification-level [11, 32, 9, 21, 33, 10], model-level [20, 34].	Statement-level [22, 23], transition-level [23].	State-level or variable-level [27, 28, 29].	Statement-level or slice-level [30, 31].	Violated expression in specification.
Use of Runtime Info	Yes (execution traces) [10, 11, 9, 21, 32, 20, 33, 34].	Yes (execution traces) [22, 23].	Yes (execution traces and state changes) [27, 28, 29].	Rarely (focus on static analysis) [31].	Yes (enriched execution traces).
Use of Static Analysis	Rarely (focus on dynamic traces [10]).	Rarely (e.g. slicing [23]).	Rarely (focus on dynamic behaviour [27, 28]).	Yes (graph-based analysis and slicing) [30, 31].	Yes (backward data-flow analysis over inter-procedural SCFGs).
Application Domain	Specification violation explanations (signal, source-code or model-based) [10, 11, 9, 21, 32, 20, 33, 34].	Debugging and automated repair [24, 25], Debugging Simulink/Stateflow CPS models [23], Failure proximity for fault ranking [26].	Failure analysis and root cause identification in software and ADS [27, 28, 29].	Cybersecurity and vulnerability detection in software systems [30, 31].	Diagnostics for source-code level specification.

riodic coefficients, enabling the capture of recurring patterns. While our work shares the goal of explaining specification violations, it differs in the nature of the artefact being diagnosed. iCFTLdiagnostics targets violations over the system’s source code, whereas the aforementioned approaches focus on signal traces or model-level behaviours.

Fault localisation: Our approach is related to FL, as both aim to identify program points that are relevant to explaining unexpected program behaviour. FL typically focuses on locating the source of a failure in a program and ranking code statements based on their likelihood of being faulty. For example, Bartocci et al. [23] proposed a fault-localisation approach for Simulink/Stateflow models using Signal Temporal Logic (STL) specifications, combining monitoring, trace diagnostics, slicing, and spectrum-based analysis to identify likely states or transitions that are the most likely to explain the fault. Many FL techniques focus on ranking code statements by a suspicion score using passed and failed tests [22]. These techniques, such as the ones surveyed by Wong et al. [35],

aim to identify the most likely faulty code segments by analysing the execution traces of successful and failed test cases. Other approaches go further and also aim to create patches for program repair. For instance, Pearson et al. [24] evaluated various automated program repair techniques that generate patches to fix faults. Similarly, Louloudakis et al. [25] proposed Fix-Con, an automated approach for FL and repair in deep learning models. From the failed tests, some techniques automatically isolate the cause-effect chain to pinpoint the root cause of failures. Some notable examples include, Zeller [27]’s work on isolating cause-effect chains and Cleve and Zeller [28]’s work on locating causes of program failures. Liu and Han [26] introduced the concept of failure proximity to identify similar fault locations, enhancing the precision of FL. While our work shares the goal of identifying relevant program points, it extends beyond the typical objectives of FL, constructing a diagnosis that explains how a variable obtains its value, contributing to program comprehension, verification, and reasoning about correct behaviour.

Causality analysis: Our approach shares with CA the goal of identifying chains of events that lead to specific behaviours in the analysed system. In the context of software, CA focuses on determining how specific inputs, states, or events lead to particular outcomes, especially failures. Zeller [27]’s Cause-Effect Chains looked at the sequence of state changes leading to a failure, which helped developers understand the progression of the fault through the program’s execution. Cleve and Zeller [28]’s Cause Transitions, instead, focused on identifying critical moments where the responsibility for the failure shifts from one variable to another, offering precise defect localisation. Feng et al. [29] contributed to CA by providing a structured approach to identify the root causes of accidents in Autonomous Driving Systems (ADS). While our approach also inspects system behaviour during execution, CA typically centres on identifying state changes that directly lead to failures or bugs. iCFTLdiagnostics focuses on identifying relevant program points that explain expressions in violated atoms, where an expression may not directly represent a fault or failure. Instead, it may reflect a combination of events that together lead to a specification violation.

Vulnerability analysis: Our approach shares with VA the goal of identifying critical program points that influence undesired outcomes. Vulnerability analysis is an area in the domain of cybersecurity that aims to identify, evaluate, and address security weaknesses in systems, networks, and applications. The goal is to discover vulnerabilities before they can be exploited by malicious actors. Thomé et al. [30] proposed an approach that assists security auditors in identifying and fixing common injection vulnerabilities in web applications. The main commonality with our diagnostics approach lies in the proposed “security slices”, which are reduced versions of program slices that contain only the essential information needed for security auditing. However, our approach is not limited to security contexts. Joern [31], a static analysis platform that generates code property graphs, also supports vulnerability discovery by enabling expressive graph-based queries over code structure and semantics. While our approach also uses graph-based representations to identify instrumentation points, it differs by combining static analysis with runtime monitoring. iCFTLdiagnostics captures concrete states that correspond to these instrumentation points to construct a diagnosis, going beyond vulnerability detection to explain how specific values evolve during execution.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we have presented a novel approach for diagnosing state-based iCFTL specifications. Our approach relies on the diagnostics instrumentation that statically identifies all the instrumentation points relevant to a given expression in a specification. After program execution, we introduced a new ι -trace filtering algorithm that takes these instrumentation points and identifies all corresponding concrete states within the ι -trace. In this way, for each expression within a falsified atom of a specification, we provide the set of concrete states—i.e., statements executions—that have contributed to the computation of the expression value that falsified the specification. We

complement this analysis with the multiplicity metric, which measures the significance of each event as the number of distinct data flow paths linking it to the expression that caused the specification violation. We have evaluated our approach on 10 different projects and 112 specifications, in terms of effectiveness, ability to reduce the complexity of understanding specifications, computational cost, and instrumentation overhead. Empirically, iCFTLdiagnostics identified relevant events with 90% precision and reduced code inspection by over 90%, while keeping diagnosis time under 7 min and instrumentation overhead below 30%.

Future research directions include extending our diagnostic framework to handle specifications that combine both time-based and state-based atoms. Such an integration would yield more comprehensive and informative diagnosis compared to getting a diagnosis for each atom in isolation. Additionally, we plan to extend iCFTLdiagnostics to support both source code and signal-based behaviours; a suitable language for expressing such hybrid specifications is SCSL [36]. This direction presents the challenge of designing diagnostic techniques capable of seamlessly handling specifications that integrate time-based, state-based, and signal-based constraints. Furthermore, we highlight that iCFTL currently does not support concurrent or parallel programs, as it was originally designed for sequential executions. Extending its semantics and instrumentation to handle interleavings and synchronization would enable diagnostics in concurrent settings, which are known to be difficult to debug.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Joshua Dawes for his guidance at the early stage of this research. This research was funded by the Luxembourg National Research Fund (FNR), grant reference 17065054, and by the European Union’s Horizon Europe program under Grant Agreement No. 101148870 (ConTestCPS). The experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg [37] (see <https://hpc.uni.lu>).

REFERENCES

- [1] C. Sánchez, G. Schneider, W. Ahrendt, E. Bartocci, D. Bianculli, C. Colombo, Y. Falcone, A. Francalanza, S. Krstić, J. M. Lourenço *et al.*, “A survey of challenges for runtime verification from advanced application domains (beyond software),” *Formal Methods Syst. Des.*, vol. 54, no. 3, pp. 279–335, 2019.
- [2] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, “Introduction to runtime verification,” in *Lectures on Runtime Verification: Introductory and Advanced Topics*. Springer, 2018, pp. 1–33.
- [3] R. Koymans, “Specifying real-time properties with metric temporal logic,” *Real-Time Systems*, vol. 2, no. 4, pp. 255–299, 1990.
- [4] O. Maler and D. Nickovic, “Monitoring temporal properties of continuous signals,” in *Proc. FTRTFT’04*. Springer, 2004, pp. 152–166.

- [5] J. H. Dawes and G. Reger, “Specification of temporal properties of functions for runtime verification,” in *Proc. SAC’19*. ACM, 2019, pp. 2206–2214.
- [6] J. Dawes and D. Bianculli, “Specifying properties over inter-procedural, source code level behaviour of programs,” in *Proc RV’21*. Springer, 2021, pp. 23–41.
- [7] C. Boufaied, C. Menghi, D. Bianculli, L. Briand, and Y. I. Parache, “Trace-checking signal-based temporal properties: A model-driven approach,” in *Proc. ASE’20*. ACM, 2020, pp. 1004–1015.
- [8] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia, “Robust online monitoring of signal temporal logic,” *Formal Methods Syst. Des.*, vol. 51, no. 1, pp. 5–30, 2017.
- [9] T. Ferrère, O. Maler, and D. Ničković, “Trace diagnostics using temporal implicants,” in *Proc. ATVA’15*. Springer, 2015, pp. 241–258.
- [10] J. H. Dawes and G. Reger, “Explaining violations of properties in control-flow temporal logic,” in *Proc. RV’19*. Springer, 2019, pp. 202–220.
- [11] C. Boufaied, C. Menghi, D. Bianculli, and L. C. Briand, “Trace diagnostics for signal-based temporal properties,” *IEEE Trans. Softw. Eng.*, vol. 49, no. 5, pp. 3131–3154, 2023.
- [12] C. Stratan, J. H. Dawes, and D. Bianculli, “Diagnosing violations of time-based properties captured in icft,” in *Proc. FormaliSE’24*. ACM, 2024, pp. 33–43.
- [13] I. Bouzenia, B. P. Krishan, and M. Pradel, “Dypybench: A benchmark of executable python software,” *Proc. ACM’24*, vol. 1, no. FSE, pp. 1–21, Jul. 2024.
- [14] M. Lacchia. (2025) Radon api documentation. Accessed: August 21, 2025. [Online]. Available: <https://radon.readthedocs.io/en/latest/api.html>
- [15] Python Software Foundation. (2025) ast — abstract syntax trees. Accessed: August 21, 2025. [Online]. Available: <https://docs.python.org/3/library/ast.html>
- [16] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse, “How developers drive software evolution,” in *Eighth International Workshop on Principles of Software Evolution (IWPSE’05)*, 2005, pp. 113–122.
- [17] D. W. McDonald and M. S. Ackerman, “Expertise recommender: a flexible recommendation system and architecture,” in *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, ser. CSCW ’00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 231–240.
- [18] M. H. Halstead, *Elements of Software Science*, ser. Operating and Programming Systems Series. New York, NY, USA: Elsevier Science Inc., 1977.
- [19] G. Hao, H. Hijazi, J. Medeiros, J. Durães, C. T. Lam, P. De Carvalho, and H. Madeira, “Complementarity in software code complexity metrics,” *JSS*, vol. 232, p. 112679, 2026.
- [20] E. Bartocci, N. Manjunath, L. Mariani, C. Mateis, and D. Ničković, “Automatic failure explanation in cps models,” in *Proc. SEFM’19*. Springer, 2019, pp. 69–86.
- [21] D. Ničković, O. Lebeltel, O. Maler, T. Ferrère, and D. Ulus, “Amt 2.0: Qualitative and quantitative trace analysis with extended signal temporal logic,” *Int. J. Softw. Tools Technol. Transf.*, vol. 22, no. 6, pp. 741–758, 2020.
- [22] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, “Effective fault localization using code coverage,” in *Proc. COMP-SAC’07*, vol. 1, 2007, pp. 449–456.
- [23] E. Bartocci, T. Ferrère, N. Manjunath, and D. Ničković, “Localizing faults in simulink/stateflow models with stl,” in *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week)*, 2018, pp. 197–206.
- [24] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, “Evaluating and improving fault localization,” in *Proc. ICSE’17*, 2017, pp. 609–620.
- [25] N. Louloudakis, P. Gibson, J. Cano, and A. Rajan, “Fixcon: Automatic fault localization and repair of deep learning model conversions between frameworks,” 2025.
- [26] C. Liu and J. Han, “Failure proximity: A fault localization-based approach,” in *Proc. SIGSOFT ’06*. ACM, 2006, pp. 46–56.
- [27] A. Zeller, “Isolating cause-effect chains from computer programs,” *SIGSOFT Softw. Eng. Notes*, vol. 27, no. 6, pp. 1–10, 11 2002.
- [28] H. Cleve and A. Zeller, “Locating causes of program failures,” in *Proc. ICSE’05*. ACM, 2005, pp. 342–351.
- [29] S. Feng, Y. Ye, Q. Shi, Z. Cheng, X. Xu, S. Cheng, H. Choi, and X. Zhang, “Rocas: Root cause analysis of autonomous driving accidents via cyber-physical commutation,” in *Proc. ASE’24*, 2024, pp. 1620–1632.
- [30] J. Thomé, L. K. Shar, D. Bianculli, and L. Briand, “Security slicing for auditing common injection vulnerabilities,” *Journal of Systems and Software*, vol. 137, pp. 766–783, 2018.
- [31] J. T. B. H. Workbench. (2025) Joern: Static code analysis for vulnerability research. Accessed: August 21, 2025. [Online]. Available: <https://github.com/joernio/joern>
- [32] W. Dou, D. Bianculli, and L. Briand, “Model-driven trace diagnostics for pattern-based temporal specifications,” in *Proc. MODELS’18*. ACM, 2018, pp. 278–288.
- [33] N. Basnet and H. Abbas, “Logical signal processing: A fourier analysis of temporal logic,” in *Proc. RV’20*. Springer, 2020, pp. 359–382.
- [34] P. Clauss, B. Kenmei, and J. C. Beyler, “The periodic-linear model of program behavior capture,” in *Proc. Euro-Par’05*. Springer, 2005, pp. 325–335.
- [35] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, 2016.
- [36] J. H. Dawes and D. Bianculli, “Specifying source code and signal-based behaviour of cyber-physical system components,” in *Proc. FACS’22*. Springer, 2022, pp. 20–38.
- [37] S. Varrette, H. Cartiaux, S. Peter, E. Kieffer, T. Valette, and A. Olloh, “Management of an academic hpc & research computing facility: The ulhpc experience 2.0,” in *Proc. HPCCT’22*. ACM, Jul. 2022.