# Diagnosing Violations of Time-based Properties Captured in iCFTL

Cristina Stratan
University of Luxembourg
Luxembourg, Luxembourg
cristina.stratan@uni.lu

Joshua Heneage Dawes
University of Luxembourg
Luxembourg, Luxembourg
joshua.dawes@uni.lu

Domenico Bianculli
University of Luxembourg
Luxembourg, Luxembourg
domenico.bianculli@uni.lu

## ABSTRACT

Runtime Verification (RV) dynamically analyses the sequence of events recorded during system execution, typically stored in traces, and provides a verdict on system behavior. RV has tended to use Boolean, or sometimes quantitative, verdicts to express whether an execution satisfied some specification. However, engineers often want to know the reason for the verdict, which can be found by carrying out *diagnostics*.

In this paper, we develop a diagnostics approach for a time-based fragment of iCFTL, a specification language designed for capturing properties concerning inter-procedural, source code-level behaviour of programs. We begin by developing an instrumentation scheme that builds on iCFTL's original scheme, enabling the construction of more informative traces. These traces are then used to determine a *point of no return*, which is an event past which a specification can never be satisfied. Our diagnostics approach then highlights a section of the trace in question that leads to the point of no return. We conclude the paper by presenting an evaluation of a prototype tool. Across 21 diverse programs, we observe that our approach is effective, efficient, and induces low time and space overhead.

## KEYWORDS

Runtime Verification, Diagnostics, Static Analysis, Temporal Logic

**ACM Reference Format:**
Cristina Stratan, Joshua Heneage Dawes, and Domenico Bianculli. 2024. Diagnosing Violations of Time-based Properties Captured in iCFTL. In *Formal Methods in Software Engineering (FormaliSE) (FormaliSE '24), April 14–15, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3644033.3644375

## 1 INTRODUCTION

Runtime Verification (RV) [3] is the process of deciding whether an execution of a computational system (often represented by a *trace*) holds some *property*. The concern of such a property, expressed by a *requirement specification* using a *specification language* (such as MTL [20], STL [21], CFTL [12], iCFTL [9], SB-TemPsy-DSL [7]), varies depending on the system being considered. For example, if one is checking an execution of a program, then the property in question may concern the data held in the program's variables, or the amount of time taken by key operations performed by the program. The type of computational system, as well as the type of properties to be captured, dictate the specification language.

RV in itself is a challenging problem, requiring the development of an algorithm that can decide whether a given trace satisfies a given specification, ultimately yielding a *verdict*. Hence, such algorithms tend to focus on generating as verdict either a Boolean value, indicating whether the specification was satisfied by the trace, or a quantitative value [13], indicating *to what extent* the specification was satisfied. However, in practice, a more detailed explanation of why a certain verdict was obtained would be useful to engineers. Specifically, engineers using RV as a software development tool often want to know *why* the RV approach generated a verdict.

To see this problem concretely, consider the source code-level property that "*all calls of the function* `query` *should take less than 1 second*", along with a program execution that does not hold that property (hence, violates the specification that captures it). A conventional RV approach may simply tell an engineer that the program's execution did not satisfy the specification in question. At best, the engineer may find out *how far away* their program was from satisfying the specification (maybe the offending function call only took 1 ms too long, or maybe it took 3 s too long). However, in practice, engineers will probably want to know what happened during the execution of the offending call of `query` to lead to the violation. Determining *what happened* is usually referred to as *diagnostics*, and is specific to the specification language being used.

Diagnostics approaches have been introduced for a host of specification languages, including MTL [16], STL [16], CFTL [11], and SB-TemPsy-DSL [8]. In this paper, we consider the diagnostics problem for iCFTL, which is a specification language designed for capturing properties concerning the inter-procedural, source code-level behaviour of programs. We focus on iCFTL, rather than any other language, because iCFTL enables engineers to capture source code-level properties with minimal effort [9].

More specifically, we consider the diagnostics problem for a fragment of iCFTL, which allows only *time-based* properties to be captured. By considering this fragment of iCFTL, we highlight that our diagnostics approach will enable engineers to find performance issues in code. While iCFTL also allows engineers to capture properties concerning the values of program variables, we highlight that analysing the performance of code at runtime is a critical part of the software development process, and so a diagnostics approach restricted to time-based properties will still yield a useful tool.

Such properties capture requirements like "*all calls of the function* `commit` *during the function* `write` *should take less than 1 second*" or "*once the program variable* `query_string` *is assigned a value, the call of the function* `commit` *should return within 2 seconds*".

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
*FormaliSE '24, April 14–15, 2024, Lisbon, Portugal*
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0589-2/24/04
https://doi.org/10.1145/3644033.3644375

Our approach begins by enriching traces generated by program executions with additional information (including the timestamps at which function executions began and ended) that can be used by the diagnostics process. To do this, we extend the instrumentation scheme already developed for iCFTL. This results in more detailed traces that allow one to determine sections of traces that contain events that caused a violation. We then refine this idea by determining a unique event in a trace called a *point of no return*, past which the specification in question becomes impossible to satisfy. This approach is the central contribution of this paper.

We have implemented our approach in a prototype tool called `iCFTL-Diagnostics`. We have evaluated the effectiveness, efficiency, and overhead induced by our tool by diagnosing specification violations on 21 diverse projects. The experimental results show that our tool could correctly identify the injected fault in 100% of cases, taking at most 38.04 ms to compute a diagnosis for a given violation; moreover, the additional instrumentation performed by our tool induces a reasonable amount of overhead over a project instrumented with iCFTL's original instrumentation scheme (on average ≈1 % of time overhead, and ≈7 % of memory overhead).

The rest of the paper is structured as follows. Section 2 introduces the necessary iCFTL background to support the rest of the paper. Section 3 describes our approach. Section 4 describes our prototype tool. Section 5 reports on the experimental evaluation. Section 6 positions our contribution in the literature. Section 7 offers concluding remarks and a roadmap for future work.

## 2 BACKGROUND

In this section, we introduce a statically-computable representation of a program, along with a notion of trace. We then present the fragment of iCFTL considered in this paper and its associated semantics. All material introduced is based on iCFTL [9].

### 2.1 Systems of Multiple Procedures

We begin our summary of iCFTL introducing a statically-computable representation of a program, which we call a *symbolic control-flow graph* (SCFG). Intuitively, this is a directed graph that uses a vertex to represent the *symbolic* state reached by executing a statement in code. Such a *symbolic state*, denoted by $\sigma$, indicates that a program variable's value may have changed, or a function may have been called, but encodes 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.

Since an SCFG is a directed graph, an edge from a symbolic state $\sigma$ to another symbolic state $\sigma'$ indicates that some statement would be executed at runtime causing the program's execution to pass into another 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 $\mathsf{SCFG}(p)$, is a triple $\langle V, E, v_s \rangle$ for $V$ a set of *symbolic states*, $E \subset V \times V$, and $v_s$ the symbolic state in which nothing has happened.

To deal with programs that contain multiple procedures, we collect SCFGs together using a map that we call a *system of multiple procedures*. Such a map, denoted by $\mathcal{S}$, sends the name of each procedure to its corresponding symbolic control-flow graph.

In practice, we focus on Python programs, which admit SCFGs because of their imperative nature. While we do not have practical experience of working with languages other than Python, we remark that SCFGs are likely to be computable for programs in a range of imperative languages; we plan to explore this aspect as part of future work.

### 2.2 Inter-procedural Dynamic Runs

We represent program executions as *traces*, which we build by considering paths through SCFGs. Specifically, since SCFGs encode reachability, one can represent a program's execution by taking the vertices from an SCFG and pairing them with timestamps. We use these pairs as the basis of our notion of *concrete states*, which are intuitively symbolic states augmented with information obtained at runtime (such as timestamps and actual program variable values).

Formally, a concrete state is a triple $\langle t, \sigma, m \rangle$ for $t$ a real-numbered timestamp, $\sigma$ a symbolic state, and $m$ a map from program variables to concrete values observed at runtime. We write $\mathsf{time}(\langle t, \sigma, m \rangle)$ to refer to the timestamp $t$.

We then represent executions of individual procedures by collecting together concrete states in sequences, 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 $\sigma_i$ to $\sigma_{i+1}$, for $1 \leq i < n$, in the relevant SCFG. Given a dynamic run, often denoted by $\mathcal{D}$, we denote by $\mathsf{states}(\mathcal{D})$ the set of all concrete states contained in $\mathcal{D}$. Once concrete states have been collected into sequences, we refer to a pair of consecutive concrete states in a sequence as a *transition*. We define the *duration* of a transition $\langle t, \sigma, m \rangle, \langle t', \sigma', m' \rangle$ by $t' - t$.

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.

We lift the notion of a dynamic run to model an execution of a *system of multiple procedures* 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 a system of multiple procedures $\mathcal{S}$ 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 $\mathcal{L} : \{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_n\} \to \mathcal{P}$ is a map that labels each dynamic run $\mathcal{D}_i$ with a procedure name from $\mathcal{P}$. Finally, for brevity we will often refer to inter-procedural dynamic runs simply as $\iota$–traces.

### 2.3 iCFTL Syntax and Semantics

iCFTL specifications consist of *quantifiers* and Boolean combinations of *atomic contraints*. Quantifiers enable one to capture concrete states or transitions from $\iota$–traces, and bind them to variables. Atomic constraints enable one to place constraints over values extracted from concrete states and transitions. Specifically, atomic constraints are formed of *expressions*, which enable one to select the concrete states or transitions to be used in the constraint. Further, iCFTL specifications are in prenex normal form, and we assume that there are no free variables. For example, one can capture the property that "*every time the program variable* x *is changed during the procedure* h*, the next change of* y *during* g *should take place within less than 1 s*" by writing

$$\forall s \in \mathsf{changes}(\mathsf{x}).\mathsf{during}(\mathsf{h}) : \\ \mathsf{timeBetween}(s, s.\mathsf{next}(\mathsf{changes}(\mathsf{y}).\mathsf{during}(\mathsf{g}))) < 1. \tag{1}$$

This example specification serves as a template for the fragment of iCFTL that we consider in this work: specifications with a single quantifier that use the timeBetween operator. We now describe the key parts of this specification and describe how the semantics would be used to compute a truth value, given an $\iota$−trace.

We begin with the quantifier, $\forall s \in \mathsf{changes}(\mathsf{x}).\mathsf{during}(\mathsf{h})$. We refer to the term $\mathsf{changes}(\mathsf{x}).\mathsf{during}(\mathsf{h})$ as a *predicate*; it is used by the semantics to identify concrete states in the $\iota$−trace that will be relevant to the specification. In this case, the semantics looks for all concrete states representing a change of the program variable x, during the function h. Formally, this involves 1) computing the symbolic control-flow graph of the procedure h, and 2) inspecting the symbolic state $\sigma$ of each concrete state $\langle t, \sigma, m\rangle$ to check for a change of x. For each such concrete state $s_i$ containing such a symbolic state, the semantics constructs a *binding* $\beta_i$, which maps the variable $s$ (from the specification) to the concrete state $s_i$.

For each binding $\beta_i$, the semantics then determines whether the atomic constraint $\mathsf{timeBetween}(s, s.\mathsf{next}(\mathsf{changes}(\mathsf{y}).\mathsf{during}(\mathsf{g}))) < 1$ holds. The first step in doing this is to determine the unique concrete states to which the expressions $s$ and $s.\mathsf{next}(\mathsf{changes}(\mathsf{y}).\mathsf{during}(\mathsf{g}))$ correspond. For this, the eval function is used, which takes the $\iota$−trace $\mathcal{I}$, the binding $\beta_i$, and an expression, and extracts the relevant concrete states from the $\iota$−trace. We denote the result of applying the eval function by $\mathsf{eval}(\mathcal{I}, \beta_i, expr)$. Once eval has been applied, the semantics computes the time elapsed between the two concrete states, and checks the final constraint.

## 3 APPROACH

Our goal is to identify the *regions* of a given $\iota$−trace that can help to explain why an iCFTL fragment specification was violated. Further, we aim to do this using a single $\iota$−trace. Assuming access to multiple $\iota$−traces would imply that the system under scrutiny could be executed multiple times; for larger systems, this may not be feasible or practical. Hence, by assuming a single $\iota$−trace, we make our approach more widely applicable.

Our approach begins with inspection of the atomic constraints in the specification. For each atomic constraint, we then inspect the $\iota$−trace with the aim of identifying a *slice* that contains information relevant to the atomic constraint.

Key challenges in developing such an approach are concerned with identifying the slice of the $\iota$−trace. Such a slice must contain information that can help a software engineer to understand why a given atomic constraint was violated. However, a slice cannot be too detailed; in practical terms, every piece of information held in an $\iota$−trace corresponds to additional instrumentation of the program. Hence, we must provide software engineers with *conservative*, but *informative* slices. Ultimately, the development of our approach can be broken down into key subproblems:

```
def f():       def h():       def g():
    h()            x = 10         k()
    g()            m()            y = 20
```

**Figure 1: An example Python program**

$$\mathcal{D}'_1 = \langle 0, [\,], [\,]\rangle, \langle 0.7, [h \mapsto called], [\,]\rangle \langle 1.4, [g \mapsto called], [\,]\rangle$$
$$\mathcal{D}'_2 = \langle 0.1, [\,], [\,]\rangle, \langle 0.15, [x \mapsto changed], [x \mapsto 10]\rangle, \langle 0.6, [m \mapsto called], [x \mapsto 10]\rangle$$
$$\mathcal{D}'_3 = \langle 0.75, [\,], [\,]\rangle, \langle 1.2, [k \mapsto called], [\,]\rangle, \langle 1.3, [y \mapsto changed], [y \mapsto 20]\rangle$$

**Figure 2: $\iota$−trace $\mathcal{I}'$**

(1) *Computing Slices*: we must be able to extract *slices* from an $\iota$−trace based on the atomic constraints found in a specification. We address this problem in § 3.1.
(2) *Instrumentation*: since our diagnostics approach works with specific concrete states, we must develop an instrumentation approach that enables us to determine which concrete states in an $\iota$−trace are required by our diagnostics approach. We address this problem in § 3.2.
(3) *Refining Slices*: given the slices computed in § 3.1, along with the instrumentation approach described in § 3.2, in § 3.3 we describe how one can refine slices to contain only the information needed by our diagnostics approach.
(4) *Diagnostics*: we combine the slices computed for each binding/atomic constraint pair into a map, defined in § 3.4.

### 3.1 Computing Slices

The first step in diagnosing a violation of a timeBetween constraint is to determine the concrete states in an $\iota$−trace that are relevant to that constraint. For example, given the constraint $\mathsf{timeBetween}(expr_1, expr_2) < n$, it will be useful to 1) determine the concrete states to which $expr_1$ and $expr_2$ correspond (under a given binding), and then 2) extract all intermediate concrete states from the $\iota$−trace. Ultimately, this will allow us to discover what happened at runtime between the two key concrete states indicated by our atomic constraint.

More formally, given an atomic constraint $\mathsf{timeBetween}(expr_1, expr_2) < n$ and a binding $\beta$, we aim to construct a slice of an $\iota$−trace $\mathcal{I}$ that contains information relevant to the atomic constraint.

We begin the description of our approach in Section 3.1.2, where we formally define what it means to construct a *sub-trace* of another $\iota$−trace. Next, in Section 3.1.3, we lift this definition to use predicates. That is, we define what it means to *filter* an $\iota$−trace with respect to a predicate. Finally, in Section 3.1.4, we use sub-traces and predicates to construct *slices*. All of these definitions are supported by references to a running example, introduced in Section 3.1.1.

*3.1.1 Running Example.* Consider the $\iota$−trace $\mathcal{I}'$ in Figure 2 obtained from running the program in Figure 1. $\mathcal{I}'$ includes the dynamic runs: $\mathcal{D}'_1$ for the procedure $f$, $\mathcal{D}'_2$ for procedure $h$ and $\mathcal{D}'_3$ for procedure $g$. The dynamic runs for $m$ and $k$ are not relevant, so are omitted.

*3.1.2 Sub-traces.* We now introduce the notion of an $\iota$−trace $\mathcal{I}$ being a *sub-trace* of another $\iota$−trace $\mathcal{I}'$. Intuitively, $\mathcal{I}$ being a subtrace of $\mathcal{I}'$ means that, if we take $\mathcal{I}'$ and discard some concrete states (and potentially even some dynamic runs), we get $\mathcal{I}$.

More formally, consider two $\iota$−traces $\mathcal{I} = \langle \mathcal{P}, \{\mathcal{D}_1, \dots, \mathcal{D}_k\}, \mathcal{L}\rangle$ and $\mathcal{I}' = \langle \mathcal{P}', \{\mathcal{D}'_1, \dots, \mathcal{D}'_n\}, \mathcal{L}'\rangle$. We say $\mathcal{I}$ is a *sub-trace* of $\mathcal{I}'$ if 1) $\mathcal{P} \subset \mathcal{P}'$, 2) $0 \le k \le n$, 3) there exists an injective function $\gamma : \{\mathcal{D}_1, \dots, \mathcal{D}_k\} \to \{\mathcal{D}'_1, \dots, \mathcal{D}'_n\}$ such that for each $\mathcal{D}_i \in \{\mathcal{D}_1, \dots, \mathcal{D}_k\}$ we have $\text{states}(\mathcal{D}_i) \subset \text{states}(\gamma(\mathcal{D}_i))$, where $\text{states}(\mathcal{D})$ represents the set of concrete states in the dynamic run $\mathcal{D}$, and 4) $\mathcal{L}$ is a map from $\{\mathcal{D}_1, \dots, \mathcal{D}_k\}$ to $\mathcal{P}$, such that for each $\mathcal{D}_i \in \{\mathcal{D}_1, \dots, \mathcal{D}_k\}$, $\mathcal{L}(\mathcal{D}_i) \in \mathcal{P}'$.

Intuitively, $\gamma$ tells us, for a given dynamic run $\mathcal{D}'$, which dynamic run $\mathcal{D}$ had concrete states removed to yield $\mathcal{D}'$. We require that $\gamma$ be injective, so distinct dynamic runs from $\mathcal{I}$ cannot have been generated by removing concrete states from the same dynamic run in $\mathcal{I}'$. Further, we do not require that $\gamma$ be surjective, since some dynamic runs in $\mathcal{I}$ may vanish as a result of having all of their concrete states removed.

Considering the running example in Figure 2, we say that $\mathcal{D}_2 = \langle 0.6, [m \mapsto \textit{called}], [x \mapsto 10]\rangle$ is a sub-trace of $\mathcal{I}'$, as we can construct the function $\gamma$ with $\gamma(\mathcal{D}_2) = \mathcal{D}'_2$ because the concrete state in $\mathcal{D}_2$ represents a subset of the original concrete states from $\mathcal{D}'_2$. Specifically, concrete state $\langle 0.6, [m \mapsto \textit{called}], [x \mapsto 10]\rangle$ is contained in the dynamic run $\mathcal{D}'_2 = \langle 0.1, [], []\rangle, \langle 0.15, [x \mapsto \textit{changed}], [x \mapsto 10]\rangle, \langle 0.6, [m \mapsto \textit{called}], [x \mapsto 10]\rangle$.

#### 3.1.3 Sub-traces with predicates.

The *sub-trace* can be refined further to include an additional constraint on the concrete states that are actually removed from the original $\iota$−trace. Specifically, the sub-trace relation places no constraints on which concrete states we remove; only that it is possible to see which (if any) concrete states were removed. Now, we refine the sub-trace relation to require that any concrete states removed must satisfy some predicate. In particular, we say that $\mathcal{I}$ is a *sub-trace under P* of $\mathcal{I}'$, where $P$ is a predicate on concrete states, if 1) $\mathcal{I}$ is a sub-trace of $\mathcal{I}'$, and 2) any concrete state $c$ found in both $\mathcal{I}$ and $\mathcal{I}'$ must have $P(c) = \textit{true}$.

For instance, given the $\iota$−trace $\mathcal{I}'$ in Figure 2, we define the predicate $P_1(c) = \textit{true}$ if $\text{time}(c) > 1$ else *false*. We say that the $\iota$−trace $\mathcal{I}$ with the dynamic runs $\mathcal{D}_3 = \langle 1.2, [k \mapsto \textit{called}], []\rangle, \langle 1.3, [y \mapsto \textit{changed}], [y \mapsto 20]\rangle; \mathcal{D}_1 = \langle 1.4, [g \mapsto \textit{called}], []\rangle$ is a sub-trace under $P_1$ of $\mathcal{I}'$, because all the concrete states in the sub-trace satisfy the predicate $P_1$ (i.e., are attained at time greater than 1 s).

#### 3.1.4 Slices.

We now use the *sub-trace under P* relation to define *slices*. In particular, we say that an $\iota$−trace $\mathcal{I}$ is a *slice* of another $\iota$−trace $\mathcal{I}'$ if and only if $\mathcal{I}$ is a sub-trace under $P_{c_1,c_2}$ of $\mathcal{I}'$, where

$$P_{c_1,c_2}(c) = \begin{cases} \textit{true} & \text{time}(c_1) \le \text{time}(c) \le \text{time}(c_2) \\ \textit{false} & \text{otherwise.} \end{cases} \tag{2}$$

Hence, an $\iota$−trace $\mathcal{I}$ is a slice of another $\iota$−trace $\mathcal{I}'$ if and only if 1) we can construct a $\gamma$ between the two sets of dynamic runs, and 2) the concrete states remaining in $\mathcal{I}$ satisfy the predicate $P_{c_1,c_2}$. Here $c_1$ and $c_2$ are concrete states.

With this definition in place, we now give a procedure that takes an $\iota$−trace $\mathcal{I}'$, and computes another $\iota$−trace $\mathcal{I}$ that is a slice of $\mathcal{I}'$ with respect to concrete states $c_1$ and $c_2$.

We use Algorithm 1 to derive the set $\{\mathcal{D}_1, \dots, \mathcal{D}_k\}$ from the set $\{\mathcal{D}'_1, \dots, \mathcal{D}'_n\}$. This algorithm works by processing each concrete state and checking for its satisfaction of some predicate $P$ (lines 6,7). Further, the algorithm makes use of the concatSeq function, which

**Algorithm 1:** filterTrace

```
1  Input: A set $\{\mathcal{D}'_1, \dots, \mathcal{D}'_n\}$ of dynamic runs, and a predicate $P$
   Result: A new set $\{\mathcal{D}_1, \dots, \mathcal{D}_k\}$ of dynamic runs, and a function $\gamma$
2  newDynamicRuns : set ← {};
3  $\gamma$ : function ← [];
4  for dynamic run $\mathcal{D}'_i$ in $\{\mathcal{D}'_1, \dots, \mathcal{D}'_n\}$ do
5      filteredDynamicRun : sequence ← ⟨⟩;
6      for concrete state $c'$ in $\mathcal{D}'_i$ do
7          if $P(c) = true$ then
8              filteredDynamicRun ←
                   concatSeq(filteredDynamicRun, ⟨$c$⟩);
9      if filteredDynamicRun is not empty then
10         newDynamicRuns ←
               newDynamicRuns ∪ {filteredDynamicRun};
11         $\gamma$ ← updateFunc($\gamma$, filteredDynamicRun, $\mathcal{D}'_i$);
12 return newDynamicRuns, $\gamma$;
```

concatenates two sequences (line 8). The algorithm also constructs $\gamma$ (line 13) using updateFunc function, which takes a function $f$, along with a new key $k$ and value $v$, and sets $f(k) = v$.

Having used Algorithm 1 to compute $\gamma$ and the new set of dynamic runs (line 12), we must address the remaining components of an $\iota$−trace: the set $\mathcal{P}$ of procedure names, and the map $\mathcal{L}$ that labels dynamic runs with procedure names.

Let $\mathcal{I}' = \langle \mathcal{P}', \{\mathcal{D}'_1, \dots, \mathcal{D}'_n\}, \mathcal{L}'\rangle$ be an $\iota$−trace,$\{\mathcal{D}_1, \dots, \mathcal{D}_k\}$ a filtered set of dynamic runs, and $\gamma$ a function, we have the following:

(1) The set $\mathcal{P}$ can be computed by evaluating $\{p : \text{there is a } \mathcal{D} \in \text{dom}(\gamma) : \mathcal{L}'(\gamma(\mathcal{D})) = p\}$. Intuitively, we first find all dynamic runs in $\mathcal{I}'$ that are also present in the filtered set $\{\mathcal{D}_1, \dots, \mathcal{D}_k\}$ of dynamic runs. For each such dynamic run, we determine the original dynamic run in $\mathcal{I}'$ by evaluating $\gamma(\mathcal{D})$. The result is then a dynamic run in the domain of the map $\mathcal{L}'$. Hence, evaluating $\mathcal{L}'(\gamma(\mathcal{D}))$ gives us the procedure with which the original dynamic run is labelled in $\mathcal{I}'$.
(2) The map $\mathcal{L}$ can be computed based on the rule $\mathcal{L}(\mathcal{D}) = p \iff \mathcal{L}'(\gamma(\mathcal{D})) = p$. Intuitively, $\mathcal{L}$ labels a dynamic run $\mathcal{D}$ with a procedure $p$ if and only if the original dynamic run, obtained by evaluating $\gamma(\mathcal{D})$, is also labeled with $p$.

Ultimately, we denote by $\mathcal{I}_{c_1,c_2}$ the slice of $\mathcal{I}$, using the predicate $P_{c_1,c_2}$. Intuitively, this is the $\iota$−trace obtained by removing any concrete state from $\mathcal{I}$ whose timestamp is outside the interval $[\text{time}(c_1), \text{time}(c_2)]$.

To see this applied in the context of an example, consider the $\iota$−trace $\mathcal{I}'$ in Figure 2. Suppose that we would like to obtain a slice that would contain all the concrete states between $\langle 0.15, [x \mapsto \textit{changed}], [x \mapsto 10]\rangle$ and $\langle 1.2, [k \mapsto \textit{called}], []\rangle$. To do this we use the predicate $P_{c_1,c_2}$ defined in Equation 2 with $c_1 = \langle 0.15, [x \mapsto \textit{changed}], [x \mapsto 10]\rangle$ and $c_2 = \langle 1.2, [k \mapsto \textit{called}], []\rangle$.

The $\iota$−trace $\mathcal{I}$ with the following dynamic runs $\mathcal{D}_1 = \langle 0.7, [h \mapsto \textit{called}], []\rangle; \mathcal{D}_2 = \langle 0.1, [], []\rangle, \langle 0.15, [x \mapsto \textit{changed}], [x \mapsto 10]\rangle, \langle 0.6, [m \mapsto \textit{called}], [x \mapsto 10]\rangle; \mathcal{D}_3 = \langle 0.75, [], []\rangle, \langle 1.2, [k \mapsto \textit{called}]$ is a slice of $\mathcal{I}'$ as it satisfies the two conditions we defined: 1) we can construct $\gamma$ between $\mathcal{D}_2$ and $\mathcal{D}'_2$, $\mathcal{D}_3$ and $\mathcal{D}'_3$, and $\mathcal{D}_1$ and $\mathcal{D}'_1$; 2) all the concrete states in $\mathcal{I}$ satisfy the predicate $P_{c_1,c_2}$.

## 3.2 Instrumentation

With a procedure introduced for computing slices of $\iota$−traces, our ultimate goal is to filter these slices to contain only the concrete states required for our diagnostics approach. We do this by performing *instrumentation* which, in this work, refers to the process of deciding which concrete states need to be kept in an $\iota$−trace for trace checking and diagnostics to be performed successfully.

An instrumentation approach has already been introduced for iCFTL [9]. This approach takes an iCFTL specification, along with a system of multiple procedures $\mathcal{S}$, and determines a set of *instrumentation points* (i.e., symbolic states). These instrumentation points are then used to remove any concrete states from an $\iota$−trace that are not needed to decide whether that $\iota$−trace satisfies the given iCFTL specification. In the remainder of this paper, we will refer to this initial instrumentation scheme as *vanilla instrumentation.*

In this work, we extend this *vanilla instrumentation* scheme by developing a *diagnostics instrumentation* scheme. This new scheme identifies more symbolic states in a system of multiple procedures, thus resulting in the construction of more informative $\iota$−traces (that can be exploited by our diagnostics approach).

For an illustration of this idea, consider the example specification presented in Equation 1. While the vanilla instrumentation scheme for iCFTL would allow us to determine the symbolic states for the change of x, and the subsequent change of y, we may also be interested in knowing what happened between these two events. This would be useful from a diagnostics perspective; if the time taken to move between two events was greater than expected, what happened along the way? In this work, we opt to focus on a select few events that took place: the beginnings and endings of procedures' executions. Applying the specification in Equation 1 to the program in Figure 1, our $\iota$−trace would contain a concrete state for the change of x at line 6, and then a concrete state for the change of y at line 11. Our augmented instrumentation procedure would then result in the inclusion of concrete states representing the start and end of the execution of procedures h, g and f. With the inclusion of these concrete states, one would then be able to determine whether the time defined by the timeBetween constraint was used between 1) the change of x and the end of h, 2) the end of h and the start of g, or 3) the start of g and the change of y. Hence, one can obtain an indication of which code led to the time constraint being violated without adding many additional instrumentation points.

We present the diagnostics instrumentation scheme by first recalling the definition of call graphs, along with some additional definitions (§ 3.2.1). We follow this by explaining how we determine additional instrumentation points (§ 3.2.2). This section develops the necessary definitions, and finishes by giving an algorithm for our diagnostics instrumentation procedure.

*3.2.1 Call Graphs.* Our first step towards computing paths between two symbolic states in a given system involves computing the *call graph* [24] of a system of multiple procedures $\mathcal{S}$.

In the scope of this work, for a system of multiple procedures $\mathcal{S}$, a *call graph* $\mathsf{callGraph}(\mathcal{S})$ is a tuple $\langle \mathcal{P}, C \rangle$, where $\mathcal{P}$ is the set of names of procedures taken from $\mathsf{dom}(\mathcal{S})$, and $C \subset \mathcal{P} \times \mathcal{P}$ a set of edges. Each pair $\langle p, p' \rangle \in C$ indicates that *p calls p′*.

We say that a *path* $\pi$ through a call graph $\mathsf{callGraph}(\mathcal{S})$ is a sequence $p_1, p_2, \ldots, p_n$ of names of procedures, such that for each $p_i, p_{i+1}$ in the sequence we have $\langle p_i, p_{i+1} \rangle \in C$. We denote by $\mathsf{procs}(\pi)$ the set $\{p_1, p_2, \ldots, p_n\}$ of names of procedures from $\pi$.

Despite a path between two nodes in a call graph being straightforward to compute, the structures of certain programs pose a problem. Consider the code listing in Figure 1 along with the specification in Equation 1. In this case, the change of x of interest to the specification occurs in the call of h, and the change of y occurs in the call of g. In the call graph constructed from this code listing, the procedure g is not reachable from the procedure h; there are only edges from f to h, and f to g. Hence, we cannot simply compute a path between the procedures of interest in this case.

Our solution is to identify the *lowest common ancestor* (LCA) [6] of the two procedures of interest to the specification. We denote this node by $p_{\mathtt{lca}}$, and then compute the shortest path from $p_{\mathsf{lca}}$ to h, which we denote by $\pi_{\mathsf{h}}$ and from $p_{\mathsf{lca}}$ to g, which we denote by $\pi_{\mathsf{g}}$. Using these two paths, we can then compute $\mathsf{lcaProcs} = \mathsf{procs}(\pi_{\mathsf{h}}) \cup \mathsf{procs}(\pi_{\mathsf{g}})$, which is intuitively the set of all procedures found on the paths from the LCA to g, and the LCA to h.

*3.2.2 Determining Additional Instrumentation Points.* We introduce our instrumentation approach in Algorithm 2 which takes as input the set lcaProcs (defined in § 3.2.1) and gives a set of additional symbolic states to instrument, additionalSymStates.

Firstly, with the set lcaProcs of all procedures (found on paths from the LCA to g and h) computed, we seek to expand this set to include more procedures. Consider the code listing in Figure 1. When this code is executed, between the change of x and the change of y that are relevant to the specification, the functions m and k would also be called. If the specification were violated, it may be the case that m or k took longer to execute than expected. Therefore, we explore procedures called during each of the procedures in lcaProcs.

To accomplish this, we iterate over the procedures in lcaProcs (line 4) and, for each one, compute the procedure's SCFG; determine the procedures called by that SCFG; add the procedures to allProcs; and recurse on each procedure.

Secondly, our goal is to identify starting symbolic states, along with final symbolic states, of the SCFG of each procedure in allProcs. To do this, we follow Algorithm 2 where we first compute $\mathcal{S}(\mathsf{proc})$ (line 8) for each procedure with name proc. Since the resulting SCFGs are tuples of the form $\langle V, E, v_s \rangle$ (where $V$ is a set of symbolic states, $E$ is a set of directed edges, and $v_s$ is the *starting symbolic state* of the SCFG), for each SCFG $\mathcal{S}(\mathsf{proc})$, we extract $v_s$ (lines 9–11), along with the SCFG's *final* states (i.e., any state that has no successor) (lines 12–14). We denote a final state by $v_e$.

Finally, on line 16 in Algorithm 2 we compute the set of additional instrumentation points additionalSymStates.

As a further remark, in practice we refine the set allProcs before computing additionalSymState. We demonstrate this refinement by considering the modified procedure `def h(): m(); x = 10` in which the order of the statements has been swapped (compared to procedure h in Figure 1). In this case, the instrumentation procedure described so far would still identify the procedure m as being of interest. At runtime, concrete states would then be generated by the executions of these procedures. This is the case even though m is called before the change of x. We address this situation by eliminating procedures from allProcs that are not reachable from the instrumentation points identified based solely on the specification. Specifically, we omit m (along with any procedures that it

**Algorithm 2:** getAdditionalSymStates

```
1  Input: A set lcaProcs
   Result: A new set additionalSymStates of instrumentation points
2  allProcs ← lcaProcs
3  additionalSymStates ← {}
4  for procedure lcaProc in lcaProcs do
5      allProcs ← allProcs ∪ getProcedures(lcaProc)
6  for procedure proc in allProcs do
7      for symbolic state $\sigma$ in $\mathcal{S}$(proc) do
8          if $\sigma$ is starting symbolic state then
9              $v_s \leftarrow \sigma$
10         else if $\sigma$ is final state then
11             $v_e \leftarrow \sigma$
12     additionalSymStates ← additionalSymStates ∪ $\{v_s, v_e\}$
13 return additionalSymStates
```

**Algorithm 3:** getPNR

```
1  Input: An atomic constraint timeBetween($expr_1$, $expr_2$) $< n$, concrete
     states $c_1$, $c_2$, and a slice $\mathcal{I}_{c_1,c_2}$
   Result: A concrete state from $\mathcal{I}_{c_1,c_2}$
2  concreteStateSeq : sequence ← getConcreteStateSequence($\mathcal{I}_{c_1,c_2}$);
3  reversedStates : sequence ← reversed concreteStateSeq;
4  PNR : concrete state ← final concrete state in reversedStates;
5  for concrete state c in reversedStates do
6      if time($c$) < PNR and time($c$) − time($c_1$) ≥ $n$ then
7          PNR ← $c$
8  return PNR;
```

calls, directly or indirectly), as long as it is not called at some point between the changes of x and y.

## 3.3 Refining Slices

With procedures defined for computing slices of $\iota$−traces, and computing instrumentation points, we now combine these to refine slices. This refinement takes place in two steps: 1) we use the results of our instrumentation scheme (Section 3.2) to throw away concrete states, and 2) we identify a unique concrete state (which we call the *point of no return - PNR*) at which the specification in question has been violated, and discard all concrete states after this.

We highlight that the second step depends on the results of the first. In particular, the instrumentation scheme employed affects the concrete state that is identified.

*3.3.1 Using instrumentation.* Suppose that, for a system of multiple procedures $\mathcal{S}$, an $\iota$−trace $\mathcal{I}$, a binding $\beta$, and an atomic constraint $\alpha$, we have computed a slice $\mathcal{I}_{c_1,c_2}$ (using the procedure described in Section 3.1). We now use the notion of predicates, introduced in Section 3.1.3, to refine this slice according to the predicate $P$, that takes a concrete state $\langle t, \sigma, m\rangle$ and gives *true* if $\sigma \in$ additionalSymStates$(\mathcal{S}, \alpha)$, and *false* otherwise.

*3.3.2 The point of no return.* If an atomic constraint of the form timeBetween$(expr_1, expr_2) < n$ is not satisfied, then there must be a *unique* concrete state $c_3$ in the slice $\mathcal{I}_{c_1,c_2}$ at which the time $n$ is exceeded for the first time. Here $c_1$ and $c_2$ are the concrete states that $expr_1$ and $expr_2$ correspond to.

More formally, there must be a concrete state $c_3$ with time$(c_3)$ − time$(c_1) \geq n$ such that any other concrete state $c'$ with time$(c')$ < time$(c_3)$ is such that time$(c')$ − time$(c_1) < n$. We call $c_3$ the *PNR*.

**Algorithm 4:** computeMap

```
1  Input: An iCFTL specification $\varphi$ and an $\iota$−trace $\mathcal{I}$
   Result: A map from binding/atomic constraint pairs to slices of $\mathcal{I}$
2  finalMap ← [];
3  bindings ← $\{\beta : [\mathcal{I}, \varphi]_S(\beta) = \text{false}\}$;
4  for binding $\beta$ ∈ bindings do
5      falsifyingAtomicConstraints ←
         getFalsifyingAtomicConstraints($\mathcal{I}, \beta, \varphi$);
6      for constraint $\alpha$ in falsifyingAtomicConstraints do
7          slice ← computeSlice($\mathcal{I}, \beta, \alpha$);
8          finalMap ← updateMap(finalMap, $\langle\beta, \alpha\rangle$, slice);
9  return finalMap;
```

Intuitively, this concrete state is a *checkpoint* past which the atomic constraint in question cannot be satisfied.

Algorithm 3 computes this *PNR* (lines 5–9), given a slice constructed with respect to concrete states $c_1$ and $c_2$. The algorithm makes use of the function getConcreteStateSequence (line 2), which takes an $\iota$−trace and combines its dynamic runs into a single sequence of concrete states, ordered by timestamp ascending. Once Algorithm 3 has been applied, we compute a final, refined slice, denoted by $\mathcal{I}_{c_1,\text{PNR}}$. We now give an example of this process.

Given the program in Figure 1, the specification in Equation 1, and the $\iota$−trace $\mathcal{I}'$ of the program in Figure 2, we first identify the concrete states for the expressions in the timeBetween atomic constraint. The concrete state for the change of variable $x$ in procedure h is $\langle 0.15, [x \mapsto \textit{changed}], [x \mapsto 10]\rangle$ and the concrete state for the change of $y$ in procedure g is $\langle 1.3, [y \mapsto \textit{changed}], [y \mapsto 20]\rangle$. Now we can construct the slice $\mathcal{I}'_{c_1,c_2}$ using the predicate $P_{c_1,c_2}$, defined in Equation 2, with concrete states $c_1 = \langle 0.15, [x \mapsto \textit{changed}], [x \mapsto 10]\rangle$ and $c_2 = \langle 1.3, [y \mapsto \textit{changed}], [y \mapsto 20]\rangle$.

Finally, given the slice $\mathcal{I}'_{c_1,c_2}$ with the dynamic runs $\mathcal{D}_1 = \langle 0.7, [h \mapsto \textit{called}]$; $\mathcal{D}_2 = \langle 0.1, [], []\rangle, \langle 0.15, [x \mapsto \textit{changed}], [x \mapsto 10]\rangle, \langle 0.6, [m \mapsto \textit{called}], [x \mapsto 10]\rangle$; $\mathcal{D}_3 = \langle 0.75, [], []\rangle, \langle 1.2, [k \mapsto \textit{called}], []\rangle, \langle 1.3, [y \mapsto \textit{changed}], [y \mapsto 20]\rangle$, we determine the *PNR*. The concrete state $\langle 1.2, [k \mapsto \textit{called}], []\rangle$ is the first concrete state to exceed $time(c_1)+1$ and is therefore our *PNR*. As such, our final refined slice would be $\mathcal{I}'_{c_1,\text{PNR}}$ with $\mathcal{D}''_1 = \langle 0.7, [h \mapsto \textit{called}]$; $\mathcal{D}''_2 = \langle 0.1, [], []\rangle, \langle 0.15, [x \mapsto \textit{changed}], [x \mapsto 10]\rangle, \langle 0.6, [m \mapsto \textit{called}], [x \mapsto 10]\rangle$; $\mathcal{D}''_3 = \langle 0.75, [], []\rangle, \langle 1.2, [k \mapsto \textit{called}], []\rangle$.

## 3.4 The final diagnostics procedure

We now complete our approach by computing a map from pairs, consisting of bindings and atomic constraints, to slices.

Algorithm 4 computes the map that we need using multiple auxiliary functions. In particular, getFalsifyingAtomicConstraints (line 5) takes an $\iota$−trace $\mathcal{I}$, a binding $\beta$ and a specification $\varphi$ and gives a set of all atomic constraints from the specification whose truth value resulted in $[\mathcal{I}, \varphi]_S(\beta)$ being false. Further, computeSlice (line 7) takes an $\iota$−trace $\mathcal{I}$, a binding $\beta$ and an atomic constraint $\alpha$ and applies the following procedure:

(1) As described in Section 3.1, an initial slice $\mathcal{I}_{c_1,c_2}$ is computed. Specifically, the predicate $P$ takes a concrete state $c$ and gives *true* if time(eval$(\mathcal{I}, \beta, expr_1)$) ≤ time$(c)$ ≤ time(eval$(\mathcal{I}, \beta, expr_2)$), and *false* otherwise. We assume the atomic constraint timeBetween( $expr_1, expr_2$) $< n$.

(2) As described in Section 3.3, the initial slice $\mathcal{I}_{c_1,c_2}$ is then refined using information from instrumentation (Section 3.2), as well as the point of no return.

Finally, updateMap (line 8) takes a map $m$, along with a key $k$ and a value $v$ and updates the map $m$ to have $m(k) = v$.

As an example of this procedure being applied, consider the program in Figure 1 and the specification in Equation 1, given the atomic constraint $\alpha = \mathsf{timeBetween}(s, s.\mathsf{next}(\mathsf{changes}(\mathsf{y}).\mathsf{during}(\mathsf{g}))) < 1$ and the single binding $\beta_1$ (in the program we have only one change of $x$), we construct the map $(\beta_1, \alpha) \mapsto \mathcal{I}'_{c_1,\mathrm{PNR}}$, where $\mathcal{I}'_{c_1,\mathrm{PNR}}$ was obtained at the end of § 3.3.2.

## 4 IMPLEMENTATION

We have implemented the approach described above in a prototype tool, called `iCFTL-Diagnostics`. It is based on an existing framework for instrumentation and trace checking with respect to iCFTL [9], which we have extended. Our extension includes modifications to the instrumentation component and to the trace checking algorithm to enable diagnostics. Below we give further details of each addition that we made to the iCFTL machinery.

*Additional instrumentation.* We modified the existing vanilla instrumentation by developing diagnostics instrumentation (§ 3.2).

*Diagnostics.* The existing iCFTL machinery provided no diagnostics capabilities. Hence, our goal was to extend the existing Boolean verdict with the map discussed in §3.4. Specifically, we opted to include diagnostics as an additional part of trace checking : once trace checking is completed, our `iCFTL-Diagnostics` tool then applies our diagnostics algorithm. Hence, while trace checking generates a Boolean verdict, the tool (inline with the approach described in §3.1) computes a slice that ends with a point of no return. The tool computes such a diagnosis for each falsifying atomic constraint associated with a binding that violates the specification. However, while the concrete states in a slice consist of timestamps, symbolic states and maps from program variables to values, the slice generated by `iCFTL-Diagnostics` includes only timestamps and line numbers (derived from symbolic states).

## 5 EVALUATION

In this section, we report on the experimental evaluation of our `iCFTL-Diagnostics` tool. The evaluation focuses on the effectiveness of our approach in providing diagnostics information, as well as on its efficiency, in terms of execution time and memory consumption, and the overhead introduced by the corresponding instrumentation. Specifically, we define the following research questions:

**RQ1:** How effective is our approach in determining the cause of a specification violation?

**RQ2:** How much time and memory does the diagnostics approach consume?

**RQ3:** What is the time and memory consumption of a diagnostics instrumented program vs. a non-instrumented program vs. a vanilla instrumented program?

### 5.1 Dataset and settings

*Evaluation subjects.* To evaluate our approach, we used an existing dataset of small Python projects (available at https://github.com/Python-World/python-mini-projects). We chose this dataset because the small Python projects it contains simplify the process of defining the ground truth used to answer RQ1. Since iCFTL specifications require code to be wrapped in a function, we chose Python projects from this dataset that contained at least one function, alongside being executable (rather than just being usable as an importable module). This would allow us to write iCFTL specifications, paving the way for our diagnostics approach to be evaluated. Based on this criterion, we identified 37 projects out of the 134 projects included in the dataset. These projects perform a variety of operations, including launching GUI-based games, and manipulating files. We discarded 16 projects that we were unable to execute (e.g., one project had a database whose schema we were unable to retrieve). The remaining 21 projects were actually functional and could be used in our evaluation. The final filtered set of projects is shown in Table 1. For each project, we define an identifier, and give the number of functions (column "NF"), and the number of lines of code in the file used (column "SLOC"). Notice that some projects have multiple files, but we worked with specifications written over the code in a single file.

*Specifications.* For each of the 21 projects, we defined an iCFTL specification using the timeBetween operator, which would enable our diagnostics approach to be applied. In particular, we used the template specification in Equation 1 (on page 3) where `x`, `y`, `h`, and `g` were to be replaced with names appropriate to each project for which we wrote a specification and the constant 1 substituted with a real number. In projects where `x` and `y` are defined in the same function, `h` and `g` would be one function. Focusing on such specifications allows us to perform our initial evaluation of `iCFTL-Diagnostics` by finding performance issues in the small Python projects.

*Fault Injection.* In addition to writing a specification for each project, evaluating our diagnostics approach also required that we modify the project source code to ensure that our specification is violated (providing an opportunity for our diagnostics approach to be applied). To ensure this, we injected a single fault into each project at a strategic position in code. Since we are working with time-based specifications, violation would be caused by a delay, so we inserted a call of our own function called `sleep_function`. This function wraps the `time.sleep` function (provided by Python's standard library) and ensures that this function is called with a delay that will violate the iCFTL specification in question.

*Settings.* The results reported in this section have been obtained using a MacBook Pro, running macOS Sonoma 14.1.1, with an Apple M1 Pro chip, and 32 GB of memory. Our code ran with Python 3.9.

### 5.2 RQ1

*Methodology.* Measuring the effectiveness of our tool required first that we decide on the answer that the tool should give for each project and specification. Recall that we ensure that time-based specifications are violated by inserting a delay (specifically, a call of `sleep_function`). Hence, we expect the PNR identified to be the event generated by the end of the execution of `sleep_function`.

Now that we have defined what PNR the tool is expected to output, we can describe the steps we performed in order to reach this output. First, we define the line number where the `sleep_function`

**Table 1: Programs used in the evaluation**

| ID | Project name | NF | SLOC |
|---|---|---|---|
| P1 | Ascii-art | 2 | 54 |
| P2 | Baidu-POI-crawl | 1 | 27 |
| P3 | cat-command | 5 | 79 |
| P4 | Check-website-connectivity | 1 | 25 |
| P5 | Cli-todo | 4 | 68 |
| P6 | Compute-IoU | 1 | 33 |
| P7 | Convert-numbers-to-word | 1 | 83 |
| P8 | Duplicate files remover | 1 | 40 |
| P9 | Instagram-profile | 1 | 53 |
| P10 | Merge-pdfs | 2 | 28 |
| P11 | Movie Information Scraper | 1 | 100 |
| P12 | Scraping Medium Articles | 4 | 71 |
| P13 | Send-email-from-csv | 3 | 52 |
| P14 | Speed-Game | 10 | 272 |
| P15 | Split-File | 2 | 55 |
| P16 | steganography | 4 | 85 |
| P17 | Store-emails-in-csv | 4 | 118 |
| P18 | Terminal-Based-Hangman-Game | 4 | 183 |
| P19 | Time-to-load-website | 1 | 31 |
| P20 | Write-script-to-compress-folder-and-files | 3 | 52 |
| P21 | Zip-Bruter | 5 | 94 |

should be injected. Then, during instrumentation, we injected both the call of `sleep_function` on the mentioned line, as well as the definition of `sleep_function`.

Finally, our tool offers a diagnosis, that contains the line number that generated the PNR that it found and we can check whether this line number matches the last line of the body of `sleep_function`.

We now recall from Section §3.3.2 that the PNR is computed for a specific binding/atomic constraint combination. Hence, we evaluated our tool by computing the PNR for each binding/atomic constraint combination, for each project.

*Results. The answer to RQ1 is that for each project in the dataset, the correct PNR was identified by* `iCFTL-Diagnostics` *in 100% of the cases (specifically, the PNR was the concrete state with line number matching the end of the injected* `sleep_function`*).*

## 5.3 RQ2

*Methodology.* We now describe the procedure that we carried out to determine the time taken and memory consumed by our tool when applied to each project in our dataset.

When measuring the time taken (respectively, the memory consumed) by our tool, we had to choose a method that would allow us to measure the time taken (respectively, the memory consumed) by a particular part of code. This was the case because we extended the existing trace checking tool for iCFTL, meaning that trace checking and diagnostics would take place within the same process.

To measure the time, using the `time` command provided by Linux would not be appropriate. Instead, we opted to use the `time` module[1] from the Python standard library, which provides the `perf_counter` function for measuring wall clock time. To measure the memory, we used the `tracemalloc` module[2] from the Python standard library, which provides tracing functionality that can be turned on and off at strategic points in code. Once tracing has taken place, the current size of memory blocks, along with the peak size, can be obtained by calling `get_traced_memory`. In our case, we recorded the peak size.

[1]https://docs.python.org/3/library/time.html
[2]https://docs.python.org/3/library/tracemalloc.html

**Table 2: Average (over 5 runs) time and memory consumed by the diagnostics approach**

| ID | T (ms) | M (kB) | # $\beta$ | Avg TL |
|---|---|---|---|---|
| P1 | 0.29 | 3.44 | 1 | 3.0 |
| P2 | 0.28 | 3.41 | 1 | 3.0 |
| P3 | 0.35 | 3.49 | 1 | 4.0 |
| P4 | 1.05 | 4.62 | 3 | 6.0 |
| P5 | 0.3 | 3.43 | 1 | 3.0 |
| P6 | 0.29 | 3.48 | 1 | 3.0 |
| P7 | 0.29 | 3.41 | 1 | 3.0 |
| P8 | 6.48 | 8.91 | 10 | 16.5 |
| P9 | 0.28 | 3.46 | 1 | 3.0 |
| P10 | 0.31 | 3.45 | 1 | 3.0 |
| P11 | 0.28 | 3.51 | 1 | 3.0 |
| P12 | 0.29 | 3.45 | 1 | 3.0 |
| P13 | 0.33 | 3.54 | 1 | 6.0 |
| P14 | 0.3 | 3.46 | 1 | 3.0 |
| P15 | 0.28 | 3.55 | 1 | 3.0 |
| P16 | 38.04 | 7.01 | 2 | 389.5 |
| P17 | 0.34 | 3.42 | 1 | 3.0 |
| P18 | 0.31 | 3.6 | 1 | 4.0 |
| P19 | 0.28 | 3.58 | 1 | 3.0 |
| P20 | 0.29 | 3.46 | 1 | 3.0 |
| P21 | 1.09 | 4.62 | 3 | 10.0 |
| **Mean ±SD** | **2.46 ±0.05** | **4.01 ±0.01** | | |

(continues on the right)

For both time taken and memory consumed, for each project we ran our tool 5 times and computed the mean and the standard deviation of both time taken and memory consumed.

*Results.* Table 2 reports the average time taken in column $T$(ms) and memory consumed in column $M$(kB) by `iCFTL-Diagnostics` for each project. Further, we define two columns: $\#\beta$ shows the number of bindings per project, and *Avg TL* shows the average length of the $\iota$−trace (this is needed as the $\iota$−trace is filtered to contain events specific to a binding).

We start by observing the memory usage. Across all projects, the mean memory consumed by our tool was 4.01 kB with maximum memory used being 8.91 kB for the P8 project and minimum being 3.41 kB for the P7 and P2 projects. The mean time taken by our tool was 2.46 ms, with a maximum of 38.04 ms when applied to the P16 project, and a minimum of 0.28 ms in projects like P19 and P15.

We observe that `iCFTL-Diagnostics` takes more time (and consumes more memory) to obtain a diagnosis when applied to projects P8 and P16. This is due to observing large $\iota$−traces or encountering multiple bindings. Specifically, project P16 includes a loop that generates more events (we observed an average $\iota$−trace length of 389.5 events), and project P8 includes a loop that generates events that are identified by quantifiers, thus generating more bindings (we observed 10 bindings).

For projects such as P2, P4, and P6 (that generate shorter $\iota$−traces, often generate only one binding, and contain on average 49 lines of code), we see that our tool takes on average 1.15 ms and consumes on average of 4.2 kB of memory. For other projects, such as P16 and P8, where longer $\iota$−traces are generated, and more bindings are extracted, our tool takes longer (P8 takes 6.48 ms and 8.91 kB, while P16 takes 38.04 ms and 7.01 kB).

*The answer to RQ2 is that, as the length of the $\iota$−trace under consideration increases, and the number of bindings extracted based on the specification increases,* `iCFTL-Diagnostics` *takes more time and consumes more memory. Ultimately, the time taken and the memory used by our tool indicate that it is suitable for practical applications.*

## 5.4 RQ3

*Methodology.* While previous research questions required us to execute our tool, this research question focuses on the project that is instrumented by our tool. Specifically, we investigate the overhead introduced to the project by our new instrumentation approach, compared to the project under both no instrumentation, and vanilla instrumentation (as defined in Section 4).

Similarly to RQ2, to measure the time taken and the memory consumed, we opted to use the `time` module and the `tracemalloc` library, respectively, from the Python standard library.

We remark that the usage of the `tracemalloc` library required a small modification to each project: we had to first import the module, and then add statements to 1) start/stop memory tracing, and 2) collect the tracing results. We applied this modification to all types of instrumentation schemes; as such we ensure that each version of the project executed was subjected to memory tracing. This means that the baseline of the non-instrumented program used in the comparison was also subjected to memory tracing.

To measure the time taken and memory consumed, we executed each project (under each of the three instrumentation schemes) 5 times and computed the mean time taken and memory consumed.

*Results.* The results, in terms of average time taken and memory consumed, are shown in Table 3 for each of the instrumentation schemes (non-instrumented program, vanilla instrumented program and diagnostics instrumented program). For simplicity, we report our results by marking N for a non-instrumented program, V for a vanilla instrumented program and D for a diagnostics instrumented program. Columns $\Delta^t_{D/V}$ and $\Delta^m_{D/V}$ indicate the approximate percentage overheads of diagnostics instrumentation over the vanilla instrumentation, for time and memory respectively.

The average memory consumption over all projects is as follows for N: 952 kB, V: 938 kB, and D: 941 kB. We observe that P6 has the least memory consumption with N: 1.23 kB, V: 1.87 kB and D: 3.66 kB. Further, P14 is the project with maximum memory consumption, with N: 9369 kB, V: 9128 kB and D: 9128 kB.

Regarding time taken, P14 is again in the top two projects according to time taken, with the most time taken being the P8 project, with N: 31 548 ms, V: 35 286 ms and D: 35 469 ms. The minimum time taken is found in P6, with N: 3505 ms, V: 3746 ms and D: 3893 ms.

As expected, memory consumed and time taken by the projects increases as the project code is subjected to heavier instrumentation. Specifically, executing a project with no instrumentation consumes less time and memory than a project with vanilla instrumentation. A project with vanilla instrumentation then, in turn, consumes less time and memory than a project with diagnostics instrumentation. While this holds in the majority of cases, there are exceptions, such as projects P4, P5, P9, P12, P14.

These exceptions are happening due to the fact that projects P4, P9, and P12 require website connectivity so the time and memory consumed are directly affected by the network. While projects P5 and P14 represent command line interface games, which require the user to interact with them. This means we may see a slight difference in the time and memory consumed based on 1) how fast the human interacted with the system, and 2) how fast the system responded to different actions performed by the user.

*The answer to RQ3 is that, on average, the diagnostics instrumented program takes ≈ 1% more time to execute and consumes ≈ 7% more memory than the vanilla instrumented program, indicating the suitability of our tool for practical applications.*

## 5.5 Threats to validity

Our initial observation is that we have assumed, throughout our experiments, that our prototype tool is correct. Despite our best efforts to test our tool thoroughly, we highlight that there may still be corner cases that we did not consider when implementing our tool (and that were not exposed during our evaluation).

Further, the dataset that we opted to use in our evaluation was composed of multiple small Python projects. While these projects allowed us to perform an initial evaluation of our prototype, we acknowledge that they represent a small subset of real-world systems. This means that, by applying our tool to other projects (and working with real engineers), we may find opportunities for improvements of both our approach and our prototype.

In addition, the fragment of iCFTL that we are considering covers only timeBetween specifications, where the only possible reason for violation would be a time delay. While our experiments only involved injecting this delay with the help of the `time.sleep` function, in reality the violation of such a specification could be caused by a more complex construct such as a loop.

Finally, while the projects that we used in our evaluation generated $\iota$−traces of various lengths, we acknowledge that we have not evaluated the performance of our tool on much larger $\iota$−traces (containing millions of events).

**Table 3: Mean of time and memory consumed by different instrumentation schemes**

| | Time (ms) | | | | Memory (kB) | | | |
|---|---|---|---|---|---|---|---|---|
| ID | N | V | D | $\Delta^t_{D/V}$ (%) | N | V | D | $\Delta^m_{D/V}$ (%) |
| P1 | 3514 | 3967 | 3999 | ≈ 1 | 419 | 419 | 420 | <1 |
| P2 | 4106 | 4654 | 4786 | ≈ 3 | 223 | 225 | 226 | <1 |
| P3 | 3513 | 3882 | 3925 | ≈ 1 | 41 | 42 | 43 | ≈ 4 |
| P4 | 11 816 | 13 169 | 13 940 | ≈ 6 | 1852 | 1860 | 1704 | <1 |
| P5 | 4317 | 5232 | 4767 | <1 | 67 | 73 | 71 | <1 |
| P6 | 3505 | 3746 | 3893 | ≈ 4 | 1.23 | 1.87 | 3.66 | ≈ 96 |
| P7 | 6622 | 7314 | 8142 | ≈ 11 | 9 | 10 | 12 | ≈ 18 |
| P8 | 31 548 | 35 286 | 35 469 | <1 | 134 | 140 | 142 | ≈ 2 |
| P9 | 3927 | 4579 | 4540 | <1 | 1661 | 1664 | 1664 | <1 |
| P10 | 3555 | 3965 | 4045 | ≈ 2 | 659 | 480 | 480 | <1 |
| P11 | 6140 | 6077 | 6151 | ≈ 1 | 271 | 273 | 274 | <1 |
| P12 | 4129 | 4805 | 4745 | <1 | 2590 | 2590 | 2590 | <1 |
| P13 | 8206 | 8343 | 8946 | ≈ 7 | 682 | 686 | 687 | <1 |
| P14 | 23 159 | 26 376 | 22 083 | <1 | 9369 | 9128 | 9128 | <1 |
| P15 | 3532 | 4070 | 4208 | ≈ 3 | 461 | 464 | 465 | <1 |
| P16 | 3540 | 4256 | 4265 | <1 | 412 | 479 | 669 | ≈ 39 |
| P17 | 4722 | 5204 | 5307 | ≈ 2 | 475 | 478 | 482 | <1 |
| P18 | 12 992 | 13 279 | 13 517 | ≈ 2 | 123 | 125 | 125 | <1 |
| P19 | 3672 | 4146 | 4153 | <1 | 279 | 277 | 277 | <1 |
| P20 | 3516 | 3810 | 3966 | ≈ 4 | 36 | 55 | 56 | ≈ 2 |
| P21 | 3567 | 3807 | 3968 | ≈ 4 | 238 | 244 | 249 | ≈ 2 |
| **Mean ±SD** | **7314 ±577** | **8093 ±572** | **8038 ±708** | **≈1** | **952 ±0** | **938 ±1** | **941 ±16** | **≈7** |

## 5.6 Data Availability

We make the data obtained during our experiments available on https://figshare.com/articles/dataset/iCFTL-Diagnostics_dataset/24835338, and our source code publicly available at https://github.com/SNTSVV/icftl-diagnostics.

## 6 RELATED WORK

Our work is mainly related to two research areas: trace diagnostics and performance analysis. However, we can also see similarities between our diagnostics approach and causality analysis.

*Causality analysis.* Causality analysis in the context of system diagnostics involves studying the cause-and-effect relationships among different components or subsystems within the system. For example, Zibaei et al. [26] propose a diagnostics approach that automatically finds the causal chain of events that led to a failure of a Cyber-Physical System. While the approach employed by Zibaei et al. processes logs in order to find a chain of events leading to the incident, `iCFTL-Diagnostics` checks the system behaviour during the execution with the focus of finding the *cause* for specification violation in the source-code components of a system.

*Trace diagnostics.* Trace diagnostics is part of RV and analyses a trace in order to give more information to the engineer. A diagnostics approach has been proposed for CFTL [11] (the predecessor of iCFTL) which, similarly to ours, tries to reveal a segment of code that could explain a specification violation. However, while the CFTL approach uses path profiling to identify problematic segments of source code, our approach involves profiling function calls (by analysing call graphs).

TD-SB-TemPsy [8] is a trace-diagnostic approach for signal-based temporal properties that includes a catalogue of 34 violation causes, each associated with one diagnosis, for properties expressed in SB-TemPsy-DSL. Similarly, TemPsy-Report [15] proposes a model-driven approach for trace diagnostics, based on retrieving the diagnostics information associated with a violation (from a list of violations that can occur with each type of TemPsy property) from a trace that violated a TemPsy property. Bartocci et al. [4] propose the CPSDebug tool, which explains failures in Simulink/Stateflow models by returning a sequence of snapshots that provide a step-by-step illustration of the failure with explicit indication of the faulty behaviours. Ferrère et al. [16] define the diagnostics problem as the search for a small fragment of the input signal that implies the violation of a specification. The tool AMT 2.0, proposed by Ničković et al. [22], incorporates two trace diagnostics procedures, with one yielding a minimal explanation that implements the trace diagnostics algorithm proposed by Ferrère et al. [16] and the other returning an extended explanation obtained by implementing epoch diagnostics (which computes a subset of the trace that contains all minimal subsets accounting for the violation). The main difference with our work lies in the nature of the artifact for which the diagnostic is computed. `iCFTL-Diagnostics` gives a diagnosis for violating specifications over the system source code, while Bartocci et al. [4] explains failures in Simulink/Stateflow models, and Boufaied et al. [8], Dou et al. [15], Ferrère et al. [16] and Ničković et al. [22] diagnose violations of signal-based properties.

*Performance Analysis.* Performance analysis is related to our work due to the fact that we check the system execution against a time-based specification and `iCFTL-Diagnostics` offers a diagnosis in case the specification is violated. Trubiani et al. [25] focus on automatically detecting Java antipatterns (as defined in a set of 7 antipatterns) in order to improve system performance. Holmqvist and Memeti [19] propose a profiling tool named Embedded Domain-specific Language for Performance Monitoring (EDPM), in which users have to annotate regions of code that require instrumentation in C and C++ programs. Further, users need to specify which performance counters (e.g., counters related to CPU: cycles, and instructions) to collect for each region. While EDPM allows the annotation of the regions around function calls in order to capture performance metrics, it burdens engineers with the extra work of manually introducing code related to collecting performance counters. Our work instead automatically performs the necessary steps in order to obtain a diagnosis, and just requires the engineer to write a specification.

Our approach combines analysis of call graphs with further static and dynamic analysis in the RV process to obtain a diagnosis. The use of call graph profilers goes back to Hall [17], who proposed an approach that analyses the call paths in a call graph in order to detect performance bottlenecks and provide performance information about nested function call sequences. This approach relies on multiple executions of the system. In contrast, our work requires only one run of the system, making it suitable not only for smaller systems but also for bigger systems (that may be constrained by the cost associated with multiple test runs).

## 7 CONCLUSION AND FUTURE WORK

In this paper, we proposed `iCFTL-Diagnostics`, a tool that diagnoses violations of time-based properties captured in iCFTL. We have established a comprehensive mathematical framework that offers a diagnosis for each atomic constraint that led to the violation of an iCFTL fragment specification. Furthermore, we have successfully translated this framework into a practical diagnostic tool that gives a conservative, but informative diagnosis. We evaluated `iCFTL-Diagnostics` by measuring the effectiveness and efficiency of the diagnostics approach, along with the efficiency of the program instrumented under three instrumentation schemes.

Future work includes expanding `iCFTL-Diagnostics` to cover both source code and signal-based behaviour; a language that offers such capabilities is SCSL [10]. Furthermore, in this work we considered diagnosing only time-based specifications but engineers can benefit from diagnostics over both state-based (LTL [23], ppDATE [1], QEA [2]) and time-based (TLTL [5], MTL [20], STL [21]) specifications. We also plan to offer diagnostics capabilities for online monitoring [13, 14, 18], as there are systems that require maintenance over continuous execution. Finally, since our approach currently works with only a single program execution, as part of future work we will consider using information from multiple program executions to refine the diagnosis.

## ACKNOWLEDGMENTS

This research was funded in whole, or in part, by the Luxembourg National Research Fund (FNR) grant reference 17065054. For the purpose of open access, and in fulfilment of the obligations arising from the grant agreement, the author has applied a Creative Commons Attribution 4.0 International (CC BY 4.0) license to any Author Accepted Manuscript version arising from this submission.

## REFERENCES

[1] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J Pace, and Gerardo Schneider. 2015. A specification language for static and runtime verification of data and control properties. In *FM 2015: Formal Methods*. Springer International Publishing, Cham, 108–125.

[2] Howard Barringer, Ylies Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. 2012. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM 2012: Formal Methods*. Springer Berlin Heidelberg, Berlin, Heidelberg, 68–84.

[3] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. *Introduction to Runtime Verification*. Springer International Publishing, Cham, 1–33.

[4] Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, and Dejan Ničković. 2019. Automatic failure explanation in CPS models. In *International Conference on Software Engineering and Formal Methods*. Springer, Cham, 69–86.

[5] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20, 4 (2011), 1–64.

[6] Michael A Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. 2005. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms* 57, 2 (2005), 75–94.

[7] Chaima Boufaied, Claudio Menghi, Domenico Bianculli, Lionel Briand, and Yago Isasi Parache. 2020. Trace-Checking Signal-based Temporal Properties: A Model-Driven Approach. In *International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, New York, NY, USA, 1004–1015.

[8] Chaima Boufaied, Claudio Menghi, Domenico Bianculli, and Lionel C. Briand. 2023. Trace Diagnostics for Signal-Based Temporal Properties. *IEEE Transactions on Software Engineering* 49, 5 (2023), 3131–3154. https://doi.org/10.1109/TSE.2023.3242588

[9] Joshua Heneage Dawes and Domenico Bianculli. 2021. Specifying Properties over Inter-procedural, Source Code Level Behaviour of Programs. In *International Conference on Runtime Verification*. Springer International Publishing, Cham, 23–41.

[10] Joshua Heneage Dawes and Domenico Bianculli. 2022. Specifying Source Code and Signal-based Behaviour of Cyber-Physical System Components. In *Formal Aspects of Component Software*. Springer International Publishing, Cham, 20–38.

[11] Joshua Heneage Dawes and Giles Reger. 2019. Explaining violations of properties in control-flow temporal logic. In *Runtime Verification*. Springer International Publishing, Cham, 202–220.

[12] Joshua Heneage Dawes and Giles Reger. 2019. Specification of temporal properties of functions for runtime verification. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing* (Limassol, Cyprus) *(SAC '19)*. Association for Computing Machinery, New York, NY, USA, 2206–2214.

[13] Jyotirmoy V Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A Seshia. 2017. Robust online monitoring of signal temporal logic. *Formal Methods in System Design* 51, 1 (2017), 5–30.

[14] Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos. 2015. Metric interval temporal logic specification elicitation and debugging. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, Austin, TX, USA, 70–79.

[15] Wei Dou, Domenico Bianculli, and Lionel Briand. 2018. Model-driven trace diagnostics for pattern-based temporal specifications. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. Association for Computing Machinery, New York, NY, USA, 278–288.

[16] Thomas Ferrère, Oded Maler, and Dejan Ničković. 2015. Trace diagnostics using temporal implicants. In *International Symposium on Automated Technology for Verification and Analysis*. Springer International Publishing, Cham, 241–258.

[17] Robert J. Hall. 1995. Call path refinement profiles. *IEEE Transactions on Software Engineering* 21, 6 (1995), 481–496.

[18] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. 2014. Online monitoring of metric temporal logic. In *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5*. Springer International Publishing, Cham, 178–192.

[19] David Weisskopf Holmqvist and Suejb Memeti. 2023. Enhancing Performance Monitoring in C/C++ Programs with EDPM: A Domain-Specific Language for Performance Monitoring.

[20] Ron Koymans. 1990. Specifying real-time properties with metric temporal logic. *Real-time systems* 2, 4 (1990), 255–299.

[21] Oded Maler and Dejan Nickovic. 2004. Monitoring temporal properties of continuous signals. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 152–166.

[22] Dejan Ničković, Olivier Lebeltel, Oded Maler, Thomas Ferrère, and Dogan Ulus. 2020. AMT 2.0: qualitative and quantitative trace analysis with extended signal temporal logic. *International Journal on Software Tools for Technology Transfer* 22, 6 (2020), 741–758.

[23] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE, Providence, RI, USA, 46–57.

[24] Barbara G Ryder. 1979. Constructing the call graph of a program. *IEEE Transactions on Software Engineering* SE-5, 3 (1979), 216–226.

[25] Catia Trubiani, Riccardo Pinciroli, Andrea Biaggi, and Francesca Arcelli Fontana. 2023. Automated Detection of Software Performance Antipatterns in Java-Based Applications. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2873–2891.

[26] Ehsan Zibaei, Sebastian Banescu, and Alexander Pretschner. 2018. Diagnosis of Safety Incidents for Cyber-Physical Systems: A UAV Example. In *2018 3rd International Conference on System Reliability and Safety (ICSRS)*. IEEE, Barcelona, Spain, 120–129. https://doi.org/10.1109/ICSRS.2018.8688886