# Specification of Temporal Properties of Functions for Runtime Verification

Joshua Heneage Dawes
University of Manchester
Manchester, UK
CERN
Geneva, Switzerland
joshua.dawes@cern.ch

Giles Reger
University of Manchester
Manchester, UK

## ABSTRACT

Runtime verification (RV) is the process of checking whether a run of a computer system satisfies a specification. RV techniques often utilise specification languages that are (i) reasonably expressive, and (ii) relatively abstract (i.e. they operate on a level of abstraction separating them from the monitored system). Inspired by the problem of monitoring systems involved in processing data generated by the high energy physics experiments at CERN, we propose a specification language, Control-Flow Temporal Logic (CFTL), whose distinguishing characteristic is its tight coupling with the control-flow of the programs for which it is used to write specifications. The coupling admits an efficient monitoring algorithm and optimised instrumentation techniques based on static analysis.

**ACM Reference Format:**
Joshua Heneage Dawes and Giles Reger. 2019. Specification of Temporal Properties of Functions for Runtime Verification. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3297280.3297497

## 1 INTRODUCTION

Runtime verification [5, 14, 15] is often presented as the problem of deciding whether an *abstracted* run of a computational system (often called a trace) satisfies a specification (often given in temporal logic) either *whilst* the system is running or after the fact. This has led to the development of many specification languages and runtime verification tools (e.g. [6, 17, 19, 22–24]) which often focus on *expressing* complex specifications and checking them *efficiently*. In our work applying RV techniques to monitor properties of web services being used by the CMS Experiment [8] on the LHC [13] at CERN we have found that existing approaches do not satisfy our requirements for two reasons.

Our first observation is that most (not all) previous work assumes that the computational system being observed is abstracted to produce events of interest i.e. specification and instrumentation are separated. We call this the *separation* issue. As an example of this issue consider the program, MTL formula [20], and instrumentation mapping in Figure 1. The advantage of such an approach is that the formula could be reused with different instrumentation mappings and can make use of concise event names (preserving readability). Conversely, it is necessary to maintain separate instrumentation information, which must be consulted to understand how the formula applies to the specific program. More subtly, this separation also implies that the structure of the monitored program and how it interacts with the specification is not considered when developing monitoring algorithms. This work introduces a specification logic called Control-Flow Temporal Logic (CFTL), that is tightly coupled with the control-flow of a program; propositions range over actions taken in the program. The utility of this tight coupling is twofold. Firstly, the author of specifications does not consider specification and instrumentation as separate activities and the control-flow information can be used to optimise the monitoring process. Secondly, the low level of abstraction means that specifications can be written directly in terms of the events that happen at runtime, rather than propositions linked to the events by an instrumentation mapping. We argue that this way of writing low level specifications is easier for engineers.

```
def process(value,quick):
  if not quick:
    rebalance()
  if newValue(value):
    balanceIns(value)
  result = search(value)
  logging.log(result)
  update(value,result)
  return result
```

$$\Box\left(\begin{array}{l}\mathsf{quick} \rightarrow \\ \left(\begin{array}{l}(\neg \mathsf{proc}\ \mathcal{U}_{[0,5]}\ \mathsf{out}) \\ \wedge\ \Diamond_{10}\ \mathsf{fin}\end{array}\right)\end{array}\right)$$

| | | |
|---|---|---|
| quick | ↔ | **call** `process` *with* `quick = 1` |
| proc | ↔ | (**call** `rebalance`) ∨ (**call** `balanceIns`) |
| out | ↔ | **call** `logging.log` |
| fin | ↔ | **return** `process` |

**Figure 1: A small example where specification and instrumentation are separated.**

Our second observation is that most (not all) previous work prioritises expressiveness and flexibility over the ability to specify low-level constraints. A typical example of this is the focus on interface or API properties where the usage of the interface may be spread across a codebase (e.g. for Java collections) and instrumentation is limited to the level of method-calls. For example, a typical specification (see [18]) might be

$$\Lambda i.(\mathsf{hasNextTrue}(i).\mathsf{hasNextTrue}(i)^*.\mathsf{next}(i))^*$$


*SAC '19, April 8–12, 2019, Limassol, Cyprus*

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297497

specifying that calls to `next` on iterators are guarded by `hasNext` calls. This leads to generic and expressive specifications but the techniques struggle to specify the local behaviour of functions. We call this the *locality* issue. We argue that for many cases it is sufficient to be able to describe the intended relationship between behaviour at a small number of particular program points and supporting more than this can reduce the utility and efficiency of an approach. The specification language developed in this paper considers the *local* behaviour of a function (and is complementary to approaches that target interface properties). One can consider this as an extension of typical functional specifications seen in, e.g., JML [21]. As such, it supports the instrumentation of assignments and local function calls, but considers each run of the function in separation.

The contributions of this paper are as follows:

- We define *symbolic control-flow graphs* (Section 2) as a program abstraction that preserves information about variable updates and function calls
- We define *Control-Flow Temporal Logic* (Section 3) in which formulas specify state and time constraints over dynamic runs
- We develop a monitoring algorithm (Section 5) that can efficiently check whether a run of a program satisfies a CFTL specification
- We develop an instrumentation approach (Section 6) that identifies the minimal set of points in a symbolic control-flow graph required to monitor a CFTL specification and utilises these to optimise the monitoring approach.

Further details, including examples and experiments with verification of more complex programs, can be found in a technical report [10].

## 2 A PROGRAM MODEL

This section introduces an abstraction of monitored functions. To keep our approach general we define a simple imperative programming language in which to define our functions; in the remainder of the paper we will refer to the monitoring of such programs. The implementation discussed later uses a subset of Python. Advanced features such as concurrency are not covered and the heap is abstracted by splitting program variables into *primitive* and *reference*. We only cover single functions; extensions would be required for inter-procedural analysis. The programs we consider are of the form

$$\begin{array}{lll} \mathit{Program} & := & x = \mathit{expr} \mid \mathit{Program}; \mathit{Program} \mid \\ & & \mathsf{if}\ \mathit{expr}\ \mathsf{then}\ \mathit{Program}\ (\mathsf{else}\ \mathit{Program}) \mid \\ & & \mathsf{while}\ \mathit{expr}\ \mathsf{do}\ \mathit{Program} \mid \mathsf{for}\ \mathit{expr}\ \mathsf{in}\ \mathit{Program} \\ \mathit{expr} & := & x \mid f(\mathit{expr}_1, \ldots, \mathit{expr}_n) \mid \mathit{arithExpr} \mid \mathit{boolExpr} \end{array}$$

for variables $x$ and function symbols $f$. We omit any information about types (we assume programs are well-typed) and ignore the structure of arithmetic and boolean expressions but note that they may include *expr* as subexpressions. The only information we are interested in for expressions is whether they contain function calls. Let $\mathsf{fn}(\mathit{expr})$ be the set of function symbols used in *expr*.

Given a program $P$ we can associate a unique *program point* with each node in the abstract syntax tree of the program. This idea is illustrated in Figure 2. We will use these program points to label the states of the control-flow graph. Given a subprogram $P'$ of $P$ let $p(P')$ be the program point of the next statement in $P'$ e.g. in this example $p(m = a(i); x = i) = 6$.

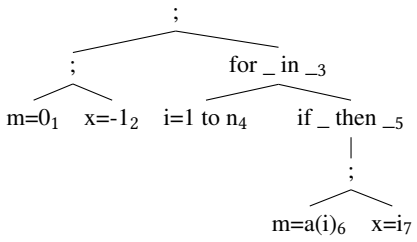


**Figure 2: Illustrating program points.**

### 2.1 Symbolic Control-Flow Graphs

Next we introduce a control-flow graph representation of programs that captures additional information about the usage of variable and function symbols. In traditional control-flow graph (CFG) representations nodes represent *basic blocks* (sequences of statements without any branching). In symbolic control-flow graphs (SCFG) nodes take the form of so-called *symbolic states* capturing information about the usage of variable and function symbols at the current point in the program i.e. the information that can be extracted *statically*.

Let Sym be a set of symbols representing variables and functions. A *symbolic state* $\sigma$ is a pair $\langle p, m \rangle$ where $p$ is a program point and $m$ is a map (partial function with finite domain) from symbols to the set of statuses {changed, unchanged, called, undefined} i.e. symbolic states record whether a program variable has just been changed or whether a function has just been called. For a symbolic state $\sigma$ we write $\sigma(x)$ for the value to which $\sigma$ maps $x$ and $\mathsf{dom}(\sigma)$ for its domain. We abuse notation and lift symbolic states to total functions on symbols such that $\sigma(x) = \text{undefined}$ if $x \notin \mathsf{dom}(\sigma)$. Symbolic control-flow graphs are directed graphs with symbolic states as nodes.

*Definition 2.1.* A symbolic control-flow graph (SCFG) is a directed graph $\langle V, E, v_s \rangle$ with a finite set of symbolic states $V$, a finite set of edges $E \subseteq V \times V$, and an initial symbolic state $v_s \in V$.

A symbolic state $\sigma$ is *final* if it has no successors e.g. there is no edge $\langle \sigma, \sigma' \rangle$ in $E$. A *path* $\pi$ through a SCFG $\langle V, E, v_s \rangle$ is a finite sequence of symbolic states $\sigma_1, \ldots, \sigma_k$ such that for every pair of adjacent symbolic states $\sigma_i, \sigma_{i+1}$ there is an edge $\langle \sigma_i, \sigma_{i+1} \rangle$ in $E$. A path is *complete* if $\sigma_1 = v_s$ and $\sigma_k$ is final.

### 2.2 Construction of SCFGs

Next we show how to construct a SCFG for a program $P$. Let $\mathsf{VarP}, \mathsf{VarR}, \mathsf{Fun} \subseteq \mathsf{Sym}$ be the sets of primitive variable symbols, reference variable symbols and function symbols used in $P$. We begin by defining the set of edges induced by $P$. These are given by a translation function T recursively defined on the structure of a $P$ given in Figure 3. This takes the symbolic state $\sigma$ reached by the previous part of the program and the remaining program $P'$ and produces a set of edges from $\sigma$ captured by $P'$. We omit certain cases for space reasons. In general we separate the cases where function calls are present and where they are not - in the former case we must assume that reference variables may be updated in nested calls. For assignments the translation adds a single edge. For if-then-else statements we introduce edges to the first program point of each sub-program and translate each sub-program extended with

$\mathsf{T}(\sigma, x = expr; P) =$
$\quad \{\langle \sigma, \langle p(P), [x \mapsto \text{changed}]\rangle\rangle\} \cup \mathsf{T}(\langle p(P), [x \mapsto \text{changed}]\rangle, P)$
if $\mathsf{fn}(expr) = \emptyset$, and
$\quad \{\langle \sigma, \langle p(P), [x_i \mapsto \text{changed}, f_i \mapsto \text{called}]\rangle\rangle\}$
$\qquad \cup \mathsf{T}(\langle p(P), [x_i \mapsto \text{changed}, f_i \mapsto \text{called}]\rangle, P)$
for $x_i \in \mathsf{VarR}$ and $f_i \in \mathsf{fn}(expr)$ otherwise

$\mathsf{T}(\sigma, \text{if } b \text{ then } P_1 \text{ else } P_2; P_3) =$
$\quad \{\langle \sigma, \langle p(P_1), []\rangle\rangle, \langle \sigma, \langle p(P_2), []\rangle\rangle\}$
$\qquad \cup \mathsf{T}(\langle p(P_1), []\rangle, P_1; P_3) \cup \mathsf{T}(\langle p(P_2), []\rangle, P_2; P_3)$
if $\mathsf{fn}(b) = \emptyset$, and otherwise
$\quad \{\langle \sigma, \langle p(P_1), m\rangle\rangle, \langle \sigma, \langle p(P_2), m\rangle\rangle\} \quad \cup \mathsf{T}(\langle p(P_1), m\rangle, P_1; P_3)$
$\qquad \cup \mathsf{T}(\langle p(P_2), m\rangle, P_2; P_3)$
where $m = [x_i \mapsto \text{changed}, f_i \mapsto \text{called}]$ for $x_i \in \mathsf{VarR}$
and $f_i \in \mathsf{fn}(b)$

$\mathsf{T}(\sigma, (\text{while } b \text{ do } P_1); P_2) =$
$\quad \{\langle \sigma, \langle p(P_1), []\rangle\rangle, \langle \sigma, \langle p(P_2), []\rangle\rangle\} \cup \mathsf{T}(\langle p(P_2), []\rangle, P_2)$
$\quad \cup \mathsf{T}(\langle p(P_1), []\rangle, P_1) \cup \{\langle \sigma', \langle p(P_1), []\rangle\rangle, \langle \sigma', \langle p(P_2), []\rangle\rangle\}$
$\quad$ where $\sigma'$ is the final state of $\mathsf{T}(\langle p(P_1), []\rangle, P_1)$
if $\mathsf{fn}(b) = \emptyset$, and otherwise the $\mathsf{fn}(b) \neq \emptyset$ case extended
in a similar way to above for if-then-else

**Figure 3: The translation of a program into a set of edges.**

the remaining program. A similar strategy is taken with while loops; at the end of the loop body there are two edges, one to the end of the loop and one back to the start of the loop body.

Let the symbolic control-flow graph SCFG($P$) of a program $P$ be

$$\langle \{\sigma_1, \sigma_2 \mid \langle \sigma_1, \sigma_2\rangle \in \mathsf{T}(\langle p(P), []\rangle, P)\}, \mathsf{T}(\langle p(P), []\rangle, P), \langle p(P), []\rangle, \rangle$$

## 2.3 Runs based on SCFGs

A symbolic control-flow graph for a program $P$ captures information about its behaviour that can be constructed statically. We also want to model a run of $P$ over its symbolic control-flow graph.

Let Val be the finite set of all values possible at runtime. A *concrete state* is a triple $\langle t, \sigma, \tau\rangle$ where $t \in \mathbb{R}^+$ is a real-valued timestamp, $\sigma$ is a symbolic state and $\tau : \mathsf{Sym} \rightharpoonup \mathsf{Val}$ is a map from variables to values. Let $\mathsf{t}(\langle t, \sigma, \tau\rangle) = t$. A *dynamic run* (modelling a single run of $P$) of SCFG($P$) is a finite sequence of concrete states where the induced sequence of symbolic states forms a sub-path of the symbolic states on some complete path in SCFG($P$).

*Definition 2.2.* A *dynamic run* of SCFG($P$) = $\langle V, E, v_s\rangle$ is a finite sequence $\mathcal{D} = \langle t_1, \sigma_1, \tau_1\rangle, \ldots, \langle t_n, \sigma_n, \tau_n\rangle$ such that timestamps $t_i$ are strictly increasing, $\sigma_1 = v_s$, $\sigma_n$ is final, and there is a path in SCFG($P$) between every pair of symbolic states $\sigma_i$ and $\sigma_{i+1}$ i.e. $\sigma_1, \ldots, \sigma_n$ can be extended to a complete path of SCFG($P$).

Next we introduce the notion of a *transition* of a dynamic run. A transition of $\mathcal{D}$ is a pair of adjacent concrete states in $\mathcal{D}$. A transition is *atomic* if the only (acyclic) path between $\sigma_i$ and $\sigma_{i+1}$ is of length 1 i.e. the transition corresponds to a single edge in the symbolic control-flow graph. A dynamic run $\mathcal{D}$ is *most general* if every transition is atomic. Let $\mathsf{paths}(tr)$ be the finite set of acyclic paths in SCFG($P$) for the transition $tr$. For a transition

$$\begin{array}{lcl}
\phi & := & \forall^S q \in \Gamma_S : \phi \mid \forall^T t \in \Gamma_T : \phi \mid \phi \vee \phi \mid \neg\phi \mid \\
 & & \phi_S \mid \phi_T \mid \mathit{true} \\
\phi_S & := & S(x) = v \mid S(x) = S(x) \mid S(x) \in (n, m) \mid S(x) \in [n, m] \\
\phi_T & := & \mathsf{duration}(T) \in (n, m) \mid \mathsf{duration}(T) \in [n, m] \\
\Gamma_S & := & \mathsf{changes}(x) \mid \mathsf{future}_S(q, \mathsf{changes}(x)) \mid \\
 & & \mathsf{future}_S(t, \mathsf{changes}(x)) \\
\Gamma_T & := & \mathsf{calls}(f) \mid \mathsf{future}_T(q, \mathsf{calls}(f)) \mid \mathsf{future}_T(t, \mathsf{calls}(f)) \\
S & := & q \mid \mathsf{source}(T) \mid \mathsf{dest}(T) \mid \mathsf{next}_S(S, \mathsf{changes}(x)) \mid \\
 & & \mathsf{next}_S(T, \mathsf{changes}(x)) \\
T & := & t \mid \mathsf{incident}(S) \mid \mathsf{next}_T(S, \mathsf{calls}(f)) \mid \mathsf{next}_T(T, \mathsf{calls}(f))
\end{array}$$

**Figure 4: Syntax of CFTL.**

$tr \in \mathcal{D}$ between concrete states $\langle t_i, \sigma_i, \tau_i\rangle$ and $\langle t_{i+1}, \sigma_{i+1}, \tau_{i+1}\rangle$ we define $\mathsf{duration}(tr) = t_{i+1} - t_i$, $\mathsf{source}(tr) = \langle t_i, \sigma_i, \tau_i\rangle$, $\mathsf{dest}(tr) = \langle t_{i+1}, \sigma_{i+1}, \tau_{i+1}\rangle$, $\mathsf{t}(tr) = t_i$ (i.e. the time of the transition is that of the source state), and $\mathsf{incident}(\mathcal{D}, \langle t_{i+1}, \sigma_{i+1}, \tau_{i+1}\rangle) = tr$.

# 3 CONTROL-FLOW TEMPORAL LOGIC

This section defines a logic satisfying our aims given in Section 1 i.e. it describes constraints over state and time (including the duration of function calls), is efficient to check at runtime (see later), and avoids the separation problem. The last point is achieved by working at a level of abstraction that does not require atoms in formulas to be explicitly related to runtime events.

## 3.1 Syntax of CFTL

We begin by recursively defining the syntax of CFTL formulas, see Figure 4, where $q$ refers to state variables, $t$ refers to transition variables, $x$ refers to program variables, $f$ refers to program functions, $v$ refers to values and $n, m$ refer to numeric expressions. One can express implication using the identity $\phi_1 \implies \phi_2 \equiv \neg\phi_1 \vee \phi_2$ and mixed intervals for $m > n$ such as $S(x) \in [n, m)$ with, for example, $S(x) \in [n, (m+n)/2] \vee S(x) \in ((m+n)/2, m)$.

Intuitively, $\forall^S$ quantifies over states and $\forall^T$ quantifies over transitions. The expressions $\Gamma_S$ and $\Gamma_T$ give state and transition conditions respectively. These place restrictions on sets of states and transitions of interest. The formulas one can write using $\phi_S$ and $\phi_T$ state properties of individual states and transitions respectively. Finally, $S$ and $T$ define state and transition *terms* i.e. expressions evaluated to either states or transitions. The only temporal operators available in the logic refer to the next state or transition satisfying some condition. However, the quantifiers range over points in the dynamic run (at different points in time) and therefore also capture temporal behaviour.

We only consider a subset of well-formed formulas defined by the above grammar. A formula is well-formed if:

- It is well-sorted e.g. state variables and transition variables are used in correct places (e.g., it does not refer to the value to which a transition maps $x$)
- All variables are bound exactly once, and
- It is in prenex-form, e.g. all quantification is at the top level, and all nested quantification is dependent on the previously quantified variable. This constraint is illustrated below and motivated later.

We give some examples of formulas with their intuitive meaning.

(1) "The calls to function $f$ take less than 5 time units" can be expressed by

$$\forall^T t \in \mathsf{calls}(f) : \mathsf{duration}(t) \in (0, 5).$$

(2) "All calls to function $f$ leave the value of $x$ unchanged" can be expressed by

$$\forall^T t \in \mathsf{calls}(f) : \mathsf{source}(t)(x) = \mathsf{dest}(t)(x).$$

(3) "Whenever $x$ is changed it is not zero" can be expressed by

$$\forall^S q \in \mathsf{changes}(x) : \neg(q(x) = 0).$$

(4) "Whenever $x$ changes, its value remains unchanged until the next call of $f$", i.e. $f$ always sees every change to $x$, can be expressed by

$$\forall^S q \in \mathsf{changes}(x) : q(x) = \mathsf{source}(\mathsf{next}_T(q, \mathsf{calls}(f)))(x).$$

(5) Finally, "whenever $x$ changes, if its value is in $[0, 5)$, then all future calls to $f$ should take units of time in $(0, 10)$" can be expressed (using $[0, 5) = (0, 5) \cup [0, 1]$) by

$$\forall^S q \in \mathsf{changes}(x) :$$
$$\forall^T t \in \mathsf{future}_T(q, \mathsf{calls}(f)) :$$
$$(q(x) \in (0, 5) \vee q(x) \in [0, 1]) \implies \mathsf{duration}(t) \in (0, 10).$$

## 3.2 Semantics of CFTL

The semantics of CFTL formulas is defined in two parts. Firstly, the quantification is inspected to generate a set of *points of interest* (states or transitions in the dynamic run). Secondly, the formula is evaluated over these points. Before we give the formal definitions, let us give a brief example.

*Example 3.1.* Let us assume we have a program containing a function symbol $g$ and we want to check the property that every call to $g$ takes less than 5 units of time. As seen above, this can be written

$$\forall^T t \in \mathsf{calls}(g) : \mathsf{duration}(t) \in (0, 5).$$

Now let us consider the following dynamic run where, for conciseness, we omit information about program points and ignore the runtime values as they do not matter for this property.

$$\langle 0, [], \tau_1\rangle, \langle 4, [g \mapsto \text{called}], \tau_1\rangle, \langle 10, [], \tau_2\rangle), \langle 16, [g \mapsto \text{called}], \tau_3\rangle$$

There are two transitions of interest; between the first two labelled concrete states, and between the last two. These are both selected by the quantification. In this case the first duration has an allowed duration but the second does not and the property is not satisfied by the dynamic run.

*Extracting Points of Interest.* The sets over which quantification occurs (the *quantification domains*) can be seen as sets of the *points of interest* mentioned previously: a formula in CFTL reasons primarily over these points. Further, for each point in the set over which a formula is quantified, the formula also reasons over the set of points derived from that point, based on the future time operators (e.g, next) present in the formula. For example, the property

$$\forall^S q \in \mathsf{changes}(x) : \mathsf{duration}(\mathsf{next}_T(q, \mathsf{calls}(f))) \in (0, 10),$$

$$\begin{array}{lll}
\mathcal{D}, \langle t, \sigma, \tau\rangle & \vdash & \mathsf{changes}(x) \quad \text{iff} \quad \sigma(x) = \text{changed} \\
\mathcal{D}, q & \vdash & \mathsf{future}_S(s, \mathsf{changes}(x)) \quad \text{iff} \\
 & & \quad \mathsf{t}(q) > \mathsf{t}(s) \text{ and } \mathcal{D}, q \vdash \mathsf{changes}(x) \\
\mathcal{D}, tr & \vdash & \mathsf{calls}(f) \quad \text{iff} \\
 & & \quad \text{for every path } \pi \in \mathsf{paths}(tr) \text{ there is:} \\
 & & \quad \text{some } \langle \sigma_1, \sigma_2\rangle \in \pi \\
 & & \quad \text{such that } \sigma_2(f) = \text{called} \\
\mathcal{D}, tr & \vdash & \mathsf{future}_T(s, \mathsf{calls}(f)) \quad \text{iff} \\
 & & \quad \mathsf{t}(tr) > \mathsf{t}(s) \text{ and } \mathcal{D}, tr \vdash \mathsf{calls}(f)
\end{array}$$

$$\begin{array}{lcl}
\mathsf{eval}(\mathcal{D}, \beta, q) & = & \beta(q) \\
\mathsf{eval}(\mathcal{D}, \beta, tr) & = & \beta(tr) \\
\mathsf{eval}(\mathcal{D}, \beta, \mathsf{source}(T)) & = & \mathsf{source}(\mathsf{eval}(\mathcal{D}, \beta, T)) \\
\mathsf{eval}(\mathcal{D}, \beta, \mathsf{dest}(T)) & = & \mathsf{dest}(\mathsf{eval}(\mathcal{D}, \beta, T)) \\
\mathsf{eval}(\mathcal{D}, \beta, \mathsf{incident}(S)) & = & \mathsf{incident}(\mathcal{D}, \mathsf{eval}(\mathcal{D}, \beta, S)) \\
\mathsf{eval}\left(\begin{array}{l}\mathcal{D}, \beta, \\ \mathsf{next}_S(X, \mathsf{changes}(x))\end{array}\right) & = & q \text{ such that:} \\
\multicolumn{3}{c}{\mathsf{t}(q) > \mathsf{t}(\mathsf{eval}(\mathcal{D}, \beta, X)) \text{ and } \mathcal{D}, q \vdash \mathsf{changes}(x) \text{ and there is no}} \\
\multicolumn{3}{c}{q' \text{ with } \mathsf{t}(\mathsf{eval}(\mathcal{D}, \beta, X)) < \mathsf{t}(q') < \mathsf{t}(q) \text{ and } \mathcal{D}, q' \vdash \mathsf{changes}(x)} \\
\mathsf{eval}\left(\begin{array}{l}\mathcal{D}, \beta, \\ \mathsf{next}_T(X, \mathsf{calls}(f))\end{array}\right) & = & tr \text{ such that:} \\
\multicolumn{3}{c}{\mathsf{t}(tr) > \mathsf{t}(\mathsf{eval}(\mathcal{D}, \beta, X)) \text{ and } \mathcal{D}, tr \vdash \mathsf{calls}(f) \text{ and there is no}} \\
\multicolumn{3}{c}{tr' \text{ with } \mathsf{t}(\mathsf{eval}(\mathcal{D}, \beta, X)) < \mathsf{t}(tr') < \mathsf{t}(tr) \text{ and } \mathcal{D}, tr' \vdash \mathsf{calls}(f)}
\end{array}$$

$$\begin{array}{lll}
\mathcal{D}, \beta & \models & \forall^S q \in \Gamma_S : \phi \text{ iff} \\
 & & \quad \text{for all } c \in \Gamma_S \text{ we have } \mathcal{D}, \beta[q \mapsto c] \models \phi \\
\mathcal{D}, \beta & \models & \forall^T tr \in \Gamma_T : \phi \text{ iff} \\
 & & \quad \text{for all } c \in \Gamma_T \text{ we have } \mathcal{D}, \beta[tr \mapsto c] \models \phi \\
\mathcal{D}, \beta & \models & \mathit{true} \\
\mathcal{D}, \beta & \models & \phi_1 \vee \phi_2 \text{ iff } \mathcal{D}, \beta \models \phi_1 \text{ or } \mathcal{D}, \beta \models \phi_2 \\
\mathcal{D}, \beta & \models & \neg\phi \text{ iff not } \mathcal{D}, \beta \models \phi \\
\mathcal{D}, \beta & \models & S(x) = v \text{ iff } \mathsf{eval}(\mathcal{D}, \beta, S)(x) = v \\
\mathcal{D}, \beta & \models & S_1(x_1) = S_2(x_2) \text{ iff} \\
 & & \quad \mathsf{eval}(\mathcal{D}, \beta, S_1)(x_1) = \mathsf{eval}(\mathcal{D}, \beta, S_2)(x_2) \\
\mathcal{D}, \beta & \models & S(x) \in [n, m] \text{ iff } \mathsf{eval}(\mathcal{D}, \beta, S)(x) \in [n, m] \\
\mathcal{D}, \beta & \models & S(x) \in (n, m) \text{ iff } \mathsf{eval}(\mathcal{D}, \beta, S)(x) \in (n, m) \\
\mathcal{D}, \beta & \models & \mathsf{duration}(T) \in (n, m) \text{ iff} \\
 & & \quad \mathsf{duration}(\mathsf{eval}(\mathcal{D}, \beta, T)) \in (n, m) \\
\mathcal{D}, \beta & \models & \mathsf{duration}(T) \in [n, m] \text{ iff} \\
 & & \quad \mathsf{duration}(\mathsf{eval}(\mathcal{D}, \beta, T)) \in [n, m]
\end{array}$$

**Figure 5: The semantic rules for CFTL.**

"every time the value of $x$ changes, the next call to $f$ should take time in $(0, 10)$", reasons first over concrete states that change $x$ and then, for each of those points, the next transition that is a call to $f$.

To define the quantification domains, we first define what it means for a state or transition condition to be satisfied. We introduce a models relation $\vdash$ which takes a dynamic run and either a state or transition *that exists within the run* and determines whether the condition is true. This is defined in Figure 5.

We overload notation to use $q \in \Gamma_S$ to iterate over all states $q$ appearing in $\mathcal{D}$ such that $\mathcal{D}, q \vdash \Gamma_S$, similarly for transitions. Therefore, a single quantification defines a set of either states or transitions

as points of interest which the formula should be evaluated over. In the case of multiple quantifications, this implicitly produces a cross-product of such sets. A member of this cross-product is a *binding*, a map from quantification variables to concrete states or transitions.

*Evaluating State and Transition Expressions.* Before we can define the truth values for CFTL formulas we need to be able to evaluate state and transition expressions. To do this we introduce an eval method that takes a dynamic run $\mathcal{D}$, a binding $\beta$, and such an expression and returns the relevant state or transition. This is defined in Figure 5 and assumes that next states or transitions always exist. This may not always be the case e.g. $\text{next}_S(q, \text{changes}(x))$ where $q$ is a final state of $\text{SCFG}_f$. There are various solutions to this issue however we take the approach to view formulas that lead to such cases as not *well-defined* and assume formulas are well-defined (which can be checked statically).

*Defining Truth for CFTL Formulas.* The satisfaction relation $\models$ captures whether a specific binding $\beta$ derived from a dynamic run $\mathcal{D}$ satisfies a CFTL formula. The recursive definition of $\models$ is given in Figure 5. A dynamic run $\mathcal{D}$ *satisfies* a (well-formed, well-defined) CFTL formula $\phi$ if $\mathcal{D}, [] \models \phi$, otherwise $\mathcal{D}$ *violates* $\phi$.

## 4 THE MONITORING AND INSTRUMENTATION PROBLEMS

So far we have introduced a static and dynamic abstraction of a program $P$ and a method for specifying properties over these abstractions. However, it is necessary to make the connection between $\mathcal{D}$ and $\phi$ explicit. In general, we assume that the function and variable symbols used in some property $\phi$ of SCFG($P$) are a subset of the symbols appearing in SCFG($P$). Given this setting we can introduce two problems (which are addressed in the following two sections).

The monitoring problem for CFTL is to check whether a dynamic run $\mathcal{D}$ produced from SCFG($P$) satisfies a given CFTL formula $\phi$. Clearly, it will often be the case that the full dynamic run is not required to check a given formula. The instrumentation problem is to determine a subset of symbolic states and edges in SCFG($P$) that must be instrumented to check a given formula. Implicitly this implies the existence of *redundant* states and edges which may be condensed into a single transition in $\mathcal{D}$.

## 5 A SIMPLE MONITORING ALGORITHM

We introduce a simple online monitoring algorithm that solves the monitoring problem. We refer to this as simple as we introduce optimisations in the next section made possible via information extracted during instrumentation. We first give a brief example of two monitoring scenarios and then introduce the general structure of the algorithm and details of its component parts.

*Example 5.1.* Consider the program in Figure 6. This is deterministic and therefore admits a single set of dynamic runs that only vary by the timestamps i.e. by running through the SCFG travelling through the loop exactly 10 times.

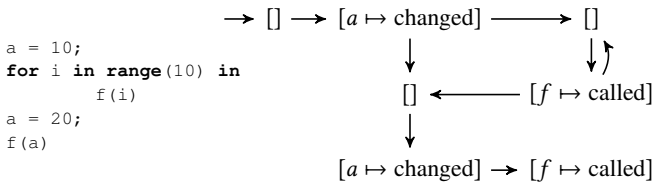


**Figure 6: A sample code snippet with its SCFG.**

Let us consider how we should monitor two different properties on this dynamic run. Firstly, the following formula

$$\forall^S q \in \text{changes}(a) : \\ q(a) \in [0, 20] \implies \\ \text{duration}(\text{next}_T(q, \text{calls}(f))) \in [0, 1]$$

which states that whenever $a$ changes if its value is between 0 and 20 then the duration of the next call to $f$ is within 1 time unit. When monitoring the dynamic run the second concrete state to be observed will be $\langle t_1, [a \mapsto \text{changed}], [a \mapsto 10]\rangle$, which is in $\text{changes}(a)$ and therefore a new binding is created. This binding is associated with a *formula tree*, which implements a mechanism for checking the current status of $\phi$ for a given binding. After processing this first state, the tree records the obligation that $\text{duration}(\text{next}_T(q, \text{calls}(f))) \in [0, 1]$. The next concrete state has no effect on this tree but the following concrete state of the form $\langle t_2, [f \mapsto \text{called}], [a \mapsto 10, i \mapsto 0]\rangle$ does. This concrete state does not change $a$ so it does not generate a new binding but it does update the tree associated with the binding $[q \mapsto \langle t_1, [a \mapsto \text{changed}], [a \mapsto 10]\rangle]$ to be true if $t_2 - t_1 < 1$ and false otherwise. Assuming that the result is true and monitoring continues, there are now no bindings and no more are introduced until $a$ changes again (as the intermediate calls to $f$ do not change $a$). This introduces a new binding with an obligation that is discharged by the next call to $f$ as before.

As the next example let us consider the following formula

$$\forall^S q \in \text{changes}(a) : \\ \forall^T t \in \text{future}_T(q, \text{calls}(f)) : \\ q(a) \in [0, 20] \implies \text{duration}(t) \in [0, 1]$$

which states that whenever $a$ is changed if its value is between 0 and 20 then the duration of *all* following calls to $f$ are within 1 time unit. This differs from the above as when the concrete state $\langle t_2, [f \mapsto \text{called}], [a \mapsto 10, i \mapsto 0]\rangle$ is observed the binding $[q \mapsto \langle t_1, [a \mapsto \text{changed}], [a \mapsto 10]\rangle]$ must be extended with the transition that has just been observed as this transition is a member of the set $\text{future}_T(q, \text{calls}(f))$. It is necessary to extend the existing binding and copy the formula tree so that the fact that $q(a) \in [0, 20]$ has been satisfied is preserved. Furthermore, each subsequent call to $f$ will also extend this binding (to create a new binding). In each case the obligation (represented by a formula tree) associated with each binding will be immediately discharged.

We now describe the simple monitoring algorithm and formula trees.

*The Simple Algorithm.* The simple monitoring algorithm is given in Algorithm 1. This iterates through the concrete states in a dynamic run and maintains a map M from bindings of quantified variables to

**Algorithm 1** Monitoring formula $\forall_1 s_1 \in \Gamma_1 : \ldots : \forall_n s_n \in \Gamma_n : \phi$ assuming that $\Gamma_1$ is of the form $\mathsf{changes}(x)$ or $\mathsf{calls}(f)$ and $\Gamma_i$ for $i > 1$ are of the form $\mathsf{future}(s_{i-1}, \Gamma)$ (i.e. the formula is well-formed).

1: $\mathrm{M} \leftarrow []$ ▷ empty map from bindings to formula trees
2: $\mathsf{prev} \leftarrow \langle 0, [], [] \rangle$ ▷ to store the previous state
3: **for** concrete state $\mathsf{curr} \in \mathcal{D}$ **do**
4: *%Handle the cases where a new binding should be generated*
5: *%New bindings are generated if the state/transition is in* $\Gamma_1$
6: **if** $\mathsf{curr} \in \Gamma_1$ **then**
7: $\mathrm{M} \mathrel{+}= ([s_1 \mapsto \mathsf{curr}] \mapsto \mathsf{update}(\mathsf{ftree}(\phi), [s_1 \mapsto \mathsf{curr}]))$
8: **if** $(\mathsf{prev}, \mathsf{curr}) \in \Gamma_1$ **then**
9: $\mathrm{M} \mathrel{+}= ([s_1 \mapsto (\mathsf{prev}, \mathsf{curr})] \mapsto$
10: $\mathsf{update}(\mathsf{ftree}(\phi), [s_1 \mapsto (\mathsf{prev}, \mathsf{curr})]))$
11: *%Bindings are extended if the state/transition is in* $\Gamma_i$ *for* $i > 1$
12: **for** $(\beta = [s_1 \mapsto v_1, \ldots s_k \mapsto v_k], T)$ in M where $k < n$ **do**
13: **if** $\mathsf{curr} \in \Gamma_{k+1}$ **then**
14: $\mathrm{M} \mathrel{+}= (\beta[s_{k+1} \mapsto \mathsf{curr}] \mapsto \mathsf{update}(T, \beta[s_{k+1} \mapsto \mathsf{curr}]))$
15: **if** $(\mathsf{prev}, \mathsf{curr}) \in \Gamma_{k+1}$ **then**
16: $\mathrm{M} \mathrel{+}= (\beta[s_{k+1} \mapsto (\mathsf{prev}, \mathsf{curr})] \mapsto$
17: $\mathsf{update}(T, \beta[s_{k+1} \mapsto (\mathsf{prev}, \mathsf{curr})]))$
18: *%Now update formula trees for existing bindings*
19: **for** $(\beta, T)$ in M **do**
20: $T' \leftarrow \mathsf{update}(T, \mathsf{curr})$
21: **if** $T' = \mathit{false}$ **then return** Fail
22: **if** $T' \neq \mathit{true}$ **then** $\mathrm{M} \leftarrow M \dagger (\beta \mapsto T')$
23: *%Finally save the current state as the last state*
24: $\mathsf{prev} \leftarrow \mathsf{curr}$
25: **return** Success ▷ M should be empty if input is well-defined

formula trees (discussed below). We use the notation $\mathrm{M} \dagger (\beta \mapsto T)$ to represent M updated with $\beta$ mapped to $T$. The bindings built up here are the *points of interest* described in Section 3.2. Lines 10-15 deal with extending existing bindings in the case where there are multiple quantifiers. This relies on the restriction that each nested quantified variable depends on the previous quantified variable. Checks such as $\mathsf{curr} \in \Gamma_1$ can be implemented by checking curr directly (in the case of future expressions we know we must be in the future set as the binding being extended exists).

*Formula Trees.* Formula trees are and-or trees whose leaves are (possibly negated) atomic CFTL formulas. These are updated with concrete states or transitions until they evaluate to *true* or *false*.

*Definition 5.2 (Formula Tree).* Given a quantifier-free CFTL formula $\phi$ in *negated normal form*, let $\mathsf{ftree}(\phi)$ be a directed graph $(N, A, O, n_s)$ where $N$ is a set of nodes corresponding to sub-formulas of $\phi$; $A$ and $O$ are sets of *and* and *or* edges respectively where $(\psi_1 \wedge \psi_2, \psi_1), (\psi_1 \wedge \psi_2, \psi_2) \in A$ if $\psi_1 \wedge \psi_2$ is a sub-formula of $\phi$, similarly for $O$; and $n_s$ is the root corresponding to $\phi$.

A formula tree can be *simplified* by applying the standard rules for *true* and *false* to collapse sub-trees. Given a formula tree $T$ let $\mathsf{update}(T, \beta)$ be the formula tree where all formulas have $s \in \mathsf{dom}(\beta)$ replaced by $\beta(s)$ and let $\mathsf{update}(T, \kappa)$ be the formula tree where all formulas are evaluated with respect to $\kappa$ (or the implicit transition it represents). For example, if a leaf stores $q(x) = \mathsf{next}_S(q, \mathsf{changes}(x))(x)$ then updating with $[q \mapsto \langle t, \sigma, [x \mapsto 5] \rangle]$ produces the leaf $5 = \mathsf{next}_S(q, \mathsf{changes}(x))(x)$ and updating this with the concrete state $\langle t, [x \mapsto \text{changed}], [x \mapsto 4] \rangle$ produces the leaf $5 = 4$, which evaluates to *false*.

*Correctness.* This algorithm determines whether a dynamic run satisfies a formula assuming that the dynamic run is most general i.e. contains as much information about the run as possible. We use instrumentation to relax this constraint next. Correctness follows from the facts that (i) all bindings are necessarily generated and (ii) the evaluation of formula trees corresponds directly to the evaluation of quantifier-free formulas.

*Complexity.* The algorithm iterates over M. The size of M is bounded by the maximum number of *live* bindings, which is itself bounded by the complexity of the program e.g. if the monitored program contains a nested loop giving quadratic behaviour then in the worse case a (constant number of) new binding(s) may be added and maintained for each iteration of the inner loop. Next we show how iterations over M can be removed using static information.

# 6 INSTRUMENTATION

In this section we discuss a solution to the instrumentation problem and the ways in which this can be used to optimise the simple monitoring algorithm.

## 6.1 Instrumentation Points

Given the symbolic control-flow graph $\langle V, E, v_s \rangle$ of a program $P$, our aim is to compute subsets $V_i \subseteq V$ of *instrumentation points* such that the dynamic run from those points is sufficient to check a given formula $\phi$. We only instrument states; to capture edges we instrument the state at either end. We call such states *instrumentation points*. A trivial solution is to pick $V_i = V$ but we want $V_i$ to be as small as possible to reduce the work of the monitoring algorithm.

We compute these sets in two steps. The first step is to identify the sets of symbolic states that can possibly generate a new binding (i.e, the concrete states they would generate would belong to quantification domains derived from some dynamic run). The second step is to use the quantifier-free part of $\phi$ to detect all subsequent states that *could* appear in a dynamic run and be relevant to $\phi$. Notice that, if a program has branching, not all parts of the SCFG will be part of a dynamic run. As a final comment, once the instrumentation points are determined, we can use the structure of the atoms found in $\phi$ to determine the nature of the instruments to be placed.

This approach produces a set of instrumentation points that are *minimal* with respect to reachability in the symbolic control-flow graph from binding-generating states i.e. we only consider those relevant states reachable from a state that generates a new binding. Note that this is not the strictest notion of minimality as (i) it is with respect to a fixed SCFG i.e. there could be a smaller set of instrumentation points given a semantically equivalent SCFG, and (ii) it does not capture any semantic notion of relevance i.e. we do not attempt to statically determine whether constraints on values might hold.

Our approach can be illustrated using the SCFG in Figure 6 and the formula

$$\forall^S q \in \mathsf{changes}(a) : \\ q(a) \in [0,20] \implies \mathsf{duration}(\mathsf{next}_T(q, \mathsf{calls}(f))) \in [0,1]$$

from Example 5.1. The two states where $a$ changes are those that can produce new bindings and the two states where $f$ has been called are implicitly of interest since they may be required to evaluate the formula that we consider (both are the next call to $f$ after a change to $a$). This is quite a simple case but illustrates how we can identify instrumentation points. Although we note that instrumenting the call to $f$ in the loop will currently lead to concrete states being generated on every execution of the loop. An optimisation we have not yet implemented is to unroll the loop once in cases where we only need the first state of the loop.

In order to make precise the idea of finding all points in the SCFG that could generate a new binding when lifted to a dynamic run, we introduce the idea of a *symbolic support*. The idea is to determine the symbolic states/edges in the SCFG that will generate concrete states/transitions that will be part of quantification domains.

*Symbolic support.* Given the formula $\varphi = \forall_1 s_1 \in \Gamma_1 : \ldots \forall_n s_n \in \Gamma_n : \phi$ we denote the symbolic support of a quantification domain $\Gamma_i$ by $\mathsf{support}(\Gamma_i)$ where

$$\begin{aligned} \sigma &\in \mathsf{support}(\mathsf{changes}(x)) && \text{if } \sigma(x) = \text{changed} \\ \sigma_1, \sigma_2 &\in \mathsf{support}(\mathsf{calls}(f)) && \text{if } \langle \sigma_1, \sigma_2 \rangle \in E \text{ and } \sigma_2(f) = \text{called} \end{aligned}$$

$$\begin{aligned} \sigma \in \mathsf{support}(\mathsf{future}_S(s_i, \mathsf{changes}(x))) &\text{ if} \\ \exists \sigma' \in \mathsf{support}(\Gamma_i) : \mathsf{reaches}(\sigma', \sigma) &\text{ and } \sigma(x) = \text{changed} \\ \sigma_1, \sigma_2 \in \mathsf{support}(\mathsf{future}_T(s_i, \mathsf{calls}(f))) &\text{ if} \\ \exists \sigma' \in \mathsf{support}(\Gamma_i) : & \\ \mathsf{reaches}(\sigma', \sigma_1) &\text{ and } \langle \sigma_1, \sigma_2 \rangle \in E \text{ and } \sigma_2(f) = \text{called} \end{aligned}$$

where $\mathsf{reaches}(\sigma_1, \sigma_2)$ is true if it is possible to reach $\sigma_2$ from $\sigma_1$ in the symbolic control-flow graph (reachability is a transitive closure of $E$). We call $\mathsf{support}(\Gamma_i)$ the *support* of $s_i$.

Note that to observe a call of $f$ we record the state before and after the call. This is technically only necessary if the formula $\phi$ refers to the source or duration of the transition at some point. Again, this is a possible optimisation that has not yet been implemented.

*Reachable states relevant to $\phi$.* Next we identify the additional symbolic states that should be instrumented. For example, if $\phi$ contains $\mathsf{next}(s_i, \mathsf{changes}(x))$ then we also need to search through the symbolic control-flow graph and instrument the next state that changes $x$ on all paths from every state in the support of $s_i$. Let $\mathsf{terms}(\phi)$ be set of state and transition terms in $\phi$. The support $\mathsf{support}(X)$ of a term $X \in \mathsf{terms}(\phi)$ is dependent on the support of the quantified variables in $X$ and is defined recursively as

$$\begin{aligned} \sigma &\in \mathsf{support}(s_i) && \text{if} && \sigma \in \mathsf{support}(\Gamma_i) \\ \sigma &\in \mathsf{support}(\mathsf{dest}(T)) && \text{if} && \sigma \in \mathsf{support}(T) \\ \sigma &\in \mathsf{support}(\mathsf{source}(T)) && \text{if} && \sigma \in \mathsf{support}(T) \\ \sigma &\in \mathsf{support}(\mathsf{incident}(S)) && \text{if} && \exists \sigma' \in \mathsf{support}(S) : \langle \sigma, \sigma' \rangle \in E \end{aligned}$$

$\sigma \in \mathsf{support}(\mathsf{next}_S(s_i, \mathsf{changes}(x)))$ if
$\sigma(x) = \text{changed}$ and $\exists \sigma_1 \in \mathsf{support}(\Gamma_i)$ such that there is a path $\pi$ from $\sigma_1$ to $\sigma$ and there is no $\sigma_2 \in \pi$ with $\sigma_2(x) = \text{changed}$

$\sigma_1, \sigma_2 \in \mathsf{support}(\mathsf{next}_T(s_i, \mathsf{calls}(f)))$ if
$\sigma_2(f) = \text{called}$ and $\exists \sigma' \in \mathsf{support}(\Gamma_i)$ such that there is a path $\pi$ from $\sigma'$ to $\sigma_1$ and $\langle \sigma_1, \sigma_2 \rangle \in E$ and there is no $\sigma'' \in \pi$ with $\sigma''(f) = \text{called}$

For source and dest we only need to take the support of $T$ as we include the start and end of each transition by default.

*Instrumentation points.* Finally, the instrumentation points in SCFG($P$) given $\phi$ are given by

$$\bigcup_{X \in \mathsf{terms}(\phi)} \mathsf{support}(X)$$

noting that $\mathsf{support}(\Gamma_i)$ is included in $\mathsf{support}(X)$ if $s_i \in X$.

*Correctness.* Finally, we state that restricting a dynamic run to the instrumentation points identified above preserves satisfiability as identified by the previous monitoring algorithm.

THEOREM 6.1. *For SCFG(P), if $\mathcal{D}$ satisfies $\phi$ then the dynamic run produced by removing all states from $\mathcal{D}$ (by collapsing transitions) not identified as instrumentation points also satisfies $\phi$.*

We give a sketch proof. Suppose $\mathcal{D}$ satisfies $\phi$, and that we derive a $\mathcal{D}'$ from $\mathcal{D}$ by removing a state $s = \langle t_i, \sigma_i, \tau_i \rangle$ from $\mathcal{D}$ where $s_i$ is not in the set of instrumentation points. Suppose, with the intention of deriving a contradiction, that $\mathcal{D}'$ does not satisfy $\phi$. Then $s$ cannot generate a binding, because removal of bindings does not result in violation of $\phi$, so $s$ must contribute to the status of a formula tree instantiated for some binding. Therefore, by construction of the set of instrumentation points based on the symbolic support, $\sigma_i$ of $s$ must be an instrumentation point, which leads to a contradiction.

## 6.2 Static Optimisations

The above instrumentation points are further used to optimise the monitoring algorithm in two ways.

*Generating points.* Above we statically determined the instrumentation points that could be used to generate or extend bindings – the points in the support of $s_1$ generate new bindings and points in the support of $s_i$ (for $i > 1$) extend bindings created at points in the support of $s_{i-1}$. We can include this static information in the monitoring algorithm to remove the need to check if every piece of data received is relevant to the monitoring process.

*Binding indexing.* Whilst generating the set of instrumentation points it is possible to identify which instrumentation points will generate the bindings that will be updated at another instrumentation point. This immediately gives a method for organising bindings in the monitoring algorithm that allows for immediate identification of the relevant bindings at each instrumentation point. To achieve this we do the following. Let a *static binding* be a map from quantified variables to instrumentation points. The set of static bindings is given by $\mathsf{support}(\Gamma_1) \times \ldots \times \mathsf{support}(\Gamma_n)$. We associate two sets of static bindings with each instrumentation point $\sigma$. The set $\mathsf{gen}(\sigma)$ stores all static bindings containing $\sigma$. The set $\mathsf{use}(\sigma)$ stores all static bindings of the form $[\ldots, s_i \mapsto \sigma_i, \ldots]$ where $\sigma_i$ is an instrumentation

point where a binding is created that will be used at $\sigma$. Then during monitoring, when a new binding is created at $\sigma$ we associate it with the static bindings in $\text{gen}(\sigma)$, and the set of bindings relevant to $\sigma$ are those associated with the static bindings in $\text{use}(\sigma)$.

Furthermore, it is possible to statically identify which parts of the formula tree associated with each binding may be updated at an instrumentation point. This information is used to further minimise the amount of search required to process each concrete state during monitoring.

## 7 IMPLEMENTATION

We have implemented the previous techniques in a tool, called VyPR, which is found at http://cern.ch/vypr/. The tool takes a Python program containing the program to monitor and a property specification file as input. Properties are specified using PyCFTL, our library for specification of CFTL properties in Python. The tool (i) builds the symbolic control-flow graph of the program, (ii) identifies relevant instrumentation points and adds instruments, (iii) runs the monitoring algorithm *asynchronously* alongside the monitored program, and (iv) outputs a verdict report (of violations and non-violations) once monitoring has finished.

We perform monitoring asynchronously as we currently do not use the outcome of the monitoring algorithm to adjust the trajectory of the program under scrutiny.

## 8 EXPERIMENTS

We now present an analysis of the VyPR tool when used to monitor a sample program, on a machine with a 2.8 GHz Intel Core i7 CPU and 16GB of RAM, with respect to two CFTL properties:

$$\forall^S q \in \text{changes}(a) : \quad q(a) \in [0, 80] \implies \text{duration}(\text{next}_T(q, \text{calls}(f))) \in [0, 1] \tag{1}$$

$$\forall^S q \in \text{changes}(a) : \quad \forall^T t \in \text{future}(q, \text{calls}(f)) : \quad q(a) \in [0, 80] \implies \text{duration}(t) \in [0, 1] \tag{2}$$

For both properties, we give an analysis of the mean time overhead over 5 runs induced by VyPR. Overall the time overhead was very low (order of ms) and the memory overhead was negligible. We discuss *percentage overhead*, which is defined as $100(t_m - t_w)/t_w$ for $t_m$ and $t_w$ the running times of the program with monitoring and without monitoring respectively. For the property in Equation 1, Figure 7a shows that the overhead induced holds approximately constant with the size of the quantification domain increasing linearly. The program includes a loop and the size of the quantification domain (i.e. number of bindings) is proportional to the number of iterations the loop goes through at runtime. For the property in Equation 2, Figure 7b shows the overhead. The size of the quantification domain, for $n$ iterations of this program's loop, is approximately $\sum_{k=0}^{n} k = n(n+1)/2 \sim n^2$ (a negligible number of bindings are generated outside the loop), and the overhead increases approximately linearly.

Finally, Figure 7c shows the percentage overhead induced by VyPR as the time between observations varies, for a fixed size quantification domain. As the time between observations decreases, the overhead induced increases. At this point, we highlight that Python's standard interpreter uses a *Global Interpreter Lock* (GIL) for shared resource control in multithreaded programs. This mechanism, in our case, results in the monitoring algorithm running in serial with the program when the delay between observations is not sufficiently large. We see that, towards a delay of 50 milliseconds between observations, the overhead approaches an asymptote: in this case, there is time between observations during which Python's GIL gives control to VyPR's monitoring algorithm.

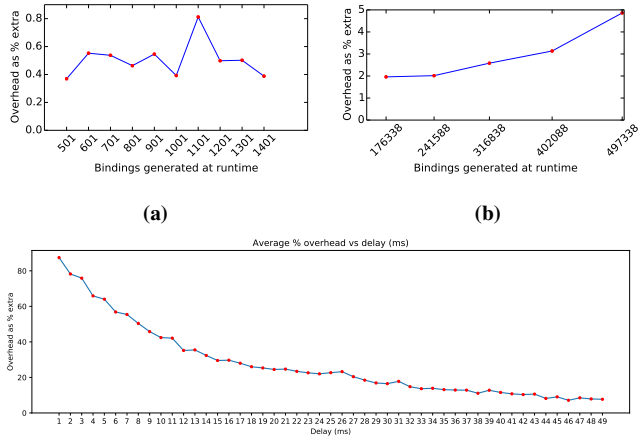


**Figure 7: Overhead Plots**

## 9 RELATED WORK

Runtime Verification has gained traction in the domain of specifying properties of time-critical systems, which has resulted in work on linear-time, temporal logics over timed state sequences. Such work includes logics, such as MTL [20, 25], TLTL [7] and MDL [6] (an extension of LDL [11] to timed state sequences), which are in general extensions of LTL such that each one of their semantics is defined over sequences of states paired with real-numbered timestamps. Existing work on CaRet [1] introduces a logic that can specify the non-regular properties involved in (nested) function calls. CaRet, however, does not allow specification of time constraints over functions. In contrast, CFTL allows specification of properties, including time constraints, over (non-nested) function calls and its formulas specify properties over transitions.

Returning to the *separation* issue, it is common, e.g. [1, 3, 4, 6, 7, 11, 16, 20, 25], to present a semantics and monitoring algorithm without addressing the details of how the data is to be taken from the monitored system. For example, to apply MDL to express a property over a program run, one has to do additional work to decide from which parts of the program run the propositional atoms in the MDL formula's alphabet are derived. This corresponds to the instrumentation mapping demonstrated in Figure 1.

Work does exist that describes solutions to the instrumentation problem. JavaMaC [19] presents a complete verification tool with a similar architecture to the one that we present: the program under

scrutiny is instrumented using information taken from the specification and the instruments pass data to a monitor. However, additional instrumentation information is still required. Some tools, such as JAVAMOP [22], include `AspectJ` instrumentation information in the specification but there is still a separation identifying the events to instrument and specifying the behaviour of those events. E-ACSL [24] works by rewriting C programs to include additional assertions and therefore derives instrumentation from the property to be checked but it does not have a separation between the monitored program and the monitoring algorithm. Further tools [2, 9, 12] include their own instrumentation languages and mapping to events.

## 10 CONCLUSION

We have introduced a new specification logic, Control-Flow Temporal Logic, for runtime verification along with a monitoring algorithm and optimisations based on information from static analysis. We have aimed to provide a framework where there is no *separation* between property and instrumentation and have focussed on *local* properties of functions, motivated by our work verifying web services at CERN. The techniques presented in this paper have been implemented in a prototype tool. There are two main potential criticisms of this work, which we address briefly here:

*Do we need another logic?* We could have taken the approach to extend an existing logic with our notions of separation and locality. However, with one of our aims being to produce a language that is easy for CERN engineers to use, experience so far has shown that the relationship between specifications and the system to which they relate must be as transparent as possible. We argue that complex temporal operators reduce transparency.

*Changes in the program model or program affect the specification.* A reliance on a particular program model is necessary to achieve the transparency mentioned above. We see the close relationship between program and specification as an advantage as changes in the program *should* require us to revisit low-level specifications.

Our next step is to carry out a detailed case study applying our tool to specify and verify properties of web services employed at CERN.

## REFERENCES


[1] Rajeev Alur, Kousha Etessami, and P Madhusudan. A Temporal Logic of Nested Calls and Returns. *Tacas*, 2988(Tacas):467–481, 2004.

[2] Shaun Azzopardi, Christian Colombo, Jean Paul Ebejer, Edward Mallia, and Gordon Pace. Runtime verification using VALOUR. In Giles Reger and Klaus Havelund, editors, *RV-CuBES 2017*, volume 3 of *Kalpa Publications in Computing*, pages 10–18. EasyChair, 2017.

[3] Howard Barringer and Klaus Havelund. Tracecontract: A scala DSL for trace analysis. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, pages 57–72, 2011.

[4] Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from eagle to ruler. *J. Log. Comput.*, 20(3):675–706, 2010.

[5] Ezio Bartocci, Ylies Falcone, Adrian Francalanza, Martin Leucker, and Giles Reger. An introduction to runtime verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 1–23. 2018.

[6] David A. Basin, Srdan Krstic, and Dmitriy Traytel. Almost event-rate independent monitoring of metric dynamic logic. In *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*, pages 85–102, 2017.

[7] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.

[8] The CMS Collaboration. The CMS experiment at the CERN LHC. *Journal of Instrumentation*, 3(08):S08004, 2008.

[9] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — safer monitoring of real-time java programs (tool paper). In Dang Van Hung and Padmanabhan Krishnan, editors, *SEFM 2009*, pages 33–37. IEEE Computer Society, 2009.

[10] Joshua Heneage Dawes and Giles Reger. Specification of State and Time Constraints for Runtime Verification of Functions. 2018. arXiv:1806.02621.

[11] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 854–860, 2013.

[12] Normann Decker, Martin Leucker, and Daniel Thoma. jUnit$^{RV}$-adding runtime verification to jUnit. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NFM 2013*, volume 7871 of *LNCS*, pages 459–464. Springer, 2013.

[13] Lyndon Evans and Philip Bryant. LHC machine. *Journal of Instrumentation*, 3(08):S08001, 2008.

[14] Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In *Engineering Dependable Software Systems*, pages 141–175. 2013.

[15] Ylies Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. In *Proceedings of the 18th International Conference on Runtime Verification*, 2018.

[16] Klaus Havelund. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer*, 17(2):143–170, Apr 2015.

[17] Klaus Havelund and Giles Reger. Specification of parametric monitors - quantified event automata versus rule systems. In *Formal Modeling and Verification of Cyber-Physical Systems*, 2015.

[18] Klaus Havelund and Giles Reger. Runtime verification logics - a language design perspective. In *Models, Algorithms, Logics and Tools*, pages 310–338, 2017.

[19] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-mac: A run-time assurance approach for java programs. *Form. Methods Syst. Des.*, 24(2):129–155, March 2004.

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

[21] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java modeling language. In *In Formal Underpinnings of Java Workshop at OOPSLA'98*, 1998.

[22] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012.

[23] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. Marq: monitoring at runtime with qea. In *TACAS'15*, 2015.

[24] Julien Signoles. *E-ACSL: Executable ANSI/ISO C Specification Language, version 1.5-4*, March 2014. frama-c.com/download/e-acsl/e-acsl.pdf.

[25] Prasanna Thati and Grigore Rosu. Monitoring algorithms for metric temporal logic specifications. *Electr. Notes Theor. Comput. Sci.*, 113:145–162, 2005.