Specifying Properties over Inter-Procedural, Source Code Level Behaviour of Programs

Joshua Heneage Dawes[1] [0000−0002−2289−1620] and Domenico Bianculli[0000−0002−4854−685X]

University of Luxembourg, Luxembourg
{joshua.dawes,domenico.bianculli}@uni.lu

Abstract. The problem of verifying a program at runtime with respect to some formal specification has led to the development of a rich collection of specification languages. These languages often have a high level of abstraction and provide sophisticated modal operators, giving a high level of expressiveness. In particular, this makes it possible to express properties concerning the source code level behaviour of programs. However, for many languages, the correspondence between events generated at the source code level and parts of the specification in question would have to be carefully defined.
To enable expressing — using a temporal logic — properties over source code level behaviour without the need for this correspondence, previous work introduced Control-Flow Temporal Logic (CFTL), a specification language with a low level of abstraction with respect to the source code of programs. However, this work focused solely on the intra-procedural setting. In this paper, we address this limitation by introducing Inter-procedural CFTL, a language for expressing source code level, inter-procedural properties of program runs. We evaluate the new language, iCFTL, via application to a real-world case study.

Keywords: Dynamic Analysis · Source Code · Inter-procedural

1 Introduction

Within the context of Runtime Verification [5], many languages have been introduced in order to allow the specification of properties that executions of programs should hold. These languages include temporal logics (such as Linear Temporal Logic [24] and Metric Temporal Logic [23]), stream equations [11], rule systems [4], automata [3,10], and others [21,18].

Specification languages typically achieve a high level of expressiveness. For example, temporal logics often combine a high level of abstraction with complex modal operators such as *next*, *until*, and *eventually* (along with timed extensions of these operators). This approach has clear benefits. For example, given different correspondences between the specification and the events generated at runtime, one specification language can be used to express properties concerning multiple levels of granularity of a system (for example, properties concerning both objects and individual lines of code). An example of a tool that provides support in constructing this correspondence is JAVAMAC [22]. However, the language then misses specific operators that would make expression of properties over specific types of runtime events easier. As an example, we

consider Metric Temporal Logic. The duration of a function call could be captured by referring to the time difference between the occurrence of the function return event, and the function call event. A language specialised for source code level properties could improve on this by 1) assuming a trace that contains the appropriate information and 2) introducing specific operators, such as *function call duration*.

In doing this, the expression of properties such as "the time taken by each call to the function `f` is no more than 0.001 times the length of the list `l` immediately before the call" would become more straightforward. Further, if one were to use a language specialised to the source code setting, there would be no need to define how events such as function calls and returns, or variable value changes, relate to parts of specifications.

Some approaches, such as the LARVA tool [10] (whose specification formalism is automata whose transitions trigger the execution of pieces of attached Java code), already allow properties over the source code level of programs to be expressed easily. Another example, which focuses less on the order of events, is Control-Flow Temporal Logic (CFTL) [17], which was introduced as a linear-time, temporal logic to be used specifically for expressing properties over the source code level behaviour of programs. Specifications written in CFTL do not require any additional information to have meaning with respect to a program.

CFTL has been shown to be a useful specification formalism (as seen in applications of VYPR [14,15], the framework built for analysing programs with respect to CFTL specifications). However, only properties concerning the *intra-procedural* behaviour of programs can be expressed (because these properties were sufficient for the case studies being considered in that work). This restriction means that one cannot express properties such as "if the variable `a` drops below some threshold in function[1] `func1`, then variable `flag` is set to `true` in function `func2`". Given that large programs are often divided into multiple procedures, many properties that software engineers could want to express would likely involve multiple procedures, like the property mentioned above.

In this paper, we introduce an extension of CFTL that enables one to express such properties. We call this new language *inter-procedural CFTL*, or *iCFTL*. iCFTL provides the same operators as CFTL (for example, to measure the duration of a function call and to obtain the value held by a variable at a given point in time), but allows the points at runtime referred to by properties to be taken from multiple procedures. This extension of the features offered by CFTL to the inter-procedural setting allows the expression of new classes of properties, and requires us to address challenges such as 1) constructing a new kind of trace that can represent the inter-procedural behaviour of a program; 2) extending the CFTL syntax to deal with the new kind of trace; and 3) performing instrumentation in a wider scope than that required for CFTL. With these challenges addressed, we demonstrate the utility of extending CFTL's features to the inter-procedural setting via a case study involving a real-world system used by the CMS Experiment at CERN [9], in which properties that cannot be expressed in CFTL are expressed in iCFTL.

[1] In the rest of the paper, we use the terms *function* and *procedure* interchangeably to denote a general, callable subroutine.

$$
\begin{array}{lll}
\mathit{Program} & := & x = \mathit{expr} \mid \mathit{func} \mid \mathit{Program}; \mathit{Program} \mid \\
& & \quad \mathsf{if}\ \mathit{expr}\ \mathsf{then}\ (\mathit{Program})\ \mathsf{else}\ (\mathit{Program}) \mid \\
& & \quad \mathsf{while}\ \mathit{expr}\ \mathsf{do}\ (\mathit{Program}) \mid \mathsf{for}\ x\ \mathsf{in}\ \mathit{iterator}\ \mathsf{do}\ (\mathit{Program}) \\
\mathit{expr} & := & x \mid \mathit{func} \mid \mathit{arithExpr} \mid \mathit{boolExpr} \\
\mathit{func} & := & f(\mathit{expr}_1, \ldots, \mathit{expr}_n) \\
\mathit{iterator} & := & \mathsf{range}(\mathit{expr}, \mathit{expr})
\end{array}
\tag{1}
$$

Fig. 1: A grammar for simple imperative programs.

Structure of the paper. In order to introduce iCFTL, the paper is structured as follows: In Section 2, we give background material on CFTL, since variations of much of the machinery are used by iCFTL. In Sections 3 and 4, we introduce iCFTL by giving its syntax and semantics. In Section 5, we introduce an instrumentation approach based on iCFTL specifications. In Section 6, we acknowledge that our initial monitoring algorithm is not efficient and describe how it can be optimised based on information from instrumentation. In Section 7, we report on our case study. In Section 8, we position our contribution in the literature (alongside giving a brief discussion of the expressive power of iCFTL) and in Section 9 we give concluding remarks.

2 Background: Control-Flow Temporal Logic

Control-Flow Temporal Logic (CFTL) [17] is a linear-time, temporal logic used to express properties over the source code level behaviour of programs. In this section, we will introduce CFTL by first defining the structures over which its semantics is defined, and then giving examples of specifications. The structures that we will introduce are the *symbolic control-flow graph* of a program and a *dynamic run*, our version of a trace.

2.1 Symbolic Control-Flow Graphs

We introduce a graph structure that can be used to encode the state change and reachability information found in a program. For simplicity of presentation, we will assume that P is a program generated by the grammar in Figure 1. We will also assume that each statement stmt in the program P can be associated with a unique *program point* taken from the abstract syntax tree of P. Such program points can be assigned simply by associating an integer with each node in the abstract syntax tree. We will denote the program point of a statement stmt by $\rho(\mathsf{stmt})$. Further, for a program P we denote by $\mathsf{Vars}(P)$ the set of program variables found in P. $\mathsf{Vars}(P)$ can be partitioned into $\mathsf{PVar}(P)$ (the primitive type variables) and $\mathsf{RVar}(P)$ (the reference type variables). Note that, in the CFTL case, we do not consider concurrency and concentrate on the intra-procedural setting.

Based on these assumptions about the structure of a program, we now define the components of a symbolic control-flow graph. First, a *symbolic state* σ associated with a statement stmt is a pair $\langle \rho(\mathsf{stmt}), m \rangle$, for a mapping m from $\mathsf{Vars}(P)$ to *statuses* in $\{\mathsf{changed}, \mathsf{unchanged}, \mathsf{undefined}, \mathsf{called}\}$. We abuse notation and write $\sigma(x)$ to mean

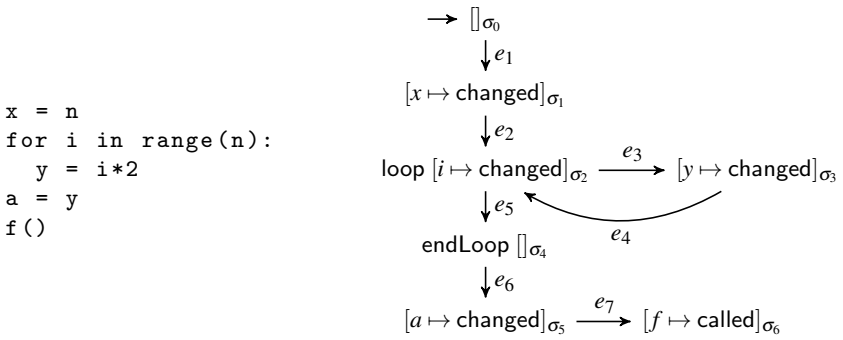

Fig. 2: A Python program with a for-loop with its symbolic control-flow graph.

$m(x)$, for m the map contained in σ. The symbolic control-flow graph $\mathsf{SCFG}(P)$ of a program is then a directed graph with symbolic states as vertices. Formally, $\mathsf{SCFG}(P) = \langle V, E, v_s \rangle$ where V is a set of symbolic states, $E \subset V \times V$ a set of edges, and $v_s \in V$ the *starting* symbolic state.

We say that a symbolic state σ is *final* in $\mathsf{SCFG}(P)$ if it has no successors, i.e., there is no edge $\langle \sigma, \sigma' \rangle \in E$ for some $\sigma' \in V$. Further, we say that a path π through $\mathsf{SCFG}(P)$ is a sequence of edges $e_1, e_2, \ldots, e_n$ such that each $e_i \in E$ and, for each e_i and e_{i+1}, $e_i = \langle \sigma, \sigma' \rangle$ and $e_{i+1} = \langle \sigma', \sigma'' \rangle$ (i.e., edges have to be adjacent).

We give an example of a program with its symbolic control-flow graph in Figure 2. One could construct the symbolic control-flow graph of a program in a language allowing more complex syntax than that described in Figure 1, provided that one can construct a scheme to translate programs to graphs.

2.2 Dynamic Runs

We now define the type of trace, which we call a *dynamic run*, over which the CFTL semantics is defined. Intuitively, a dynamic run follows a path through a symbolic control-flow graph and gives concrete timing and data values to each symbolic state encountered along the path.

More formally, a *dynamic run* of a program P is a sequence of triples $\langle t, \sigma, m \rangle$ with a timestamp $t \in \mathbb{R}_{\geq}$, a symbolic state σ, and a mapping m from program variables in $\mathsf{Vars}(P)$ to concrete values. Further, for each pair of consecutive triples $\langle t, \sigma, m \rangle$, $\langle t', \sigma', m' \rangle$, there is a path from σ to σ' in $\mathsf{SCFG}(P)$.

Each triple in a dynamic run is known as a *concrete state*. Given a concrete state $s = \langle t, \sigma, m \rangle$, we write $s(x)$ to refer to the value given to the program variable x by the map m. We denote by $\mathsf{t}(\langle t, \sigma, m \rangle)$ the timestamp t. We call a pair $\langle \langle t, \sigma, m \rangle, \langle t', \sigma', m' \rangle \rangle$ of consecutive concrete states a *transition*, which we usually denote by tr. We denote by $\mathsf{paths}(tr)$ the set of paths from σ to σ' in $\mathsf{SCFG}(P)$. A transition tr is atomic if the only acyclic path from σ to σ' in $\mathsf{SCFG}(P)$ is of length 1. We define $\mathsf{t}(.)$ for transitions by $\mathsf{t}(\langle \langle t, \sigma, m \rangle, \langle t', \sigma', m' \rangle \rangle) = t$, i.e., the time at which the transition started.

2.3 Examples of CFTL Specifications

With our notion of a trace introduced, we briefly describe the structure of CFTL specifications, and then give examples. CFTL specifications are always universally-quantified *at least once*, do not have existential quantifiers, and are in prenex normal form. The quantifiers use *predicates* in order to extract relevant concrete states or transitions from dynamic runs and bind them to *variables*. For example, $\mathsf{calls}(\mathtt{f})$ identifies all transitions whose second concrete state contains a symbolic state that maps `f` to called, and $\mathsf{changes}(\mathtt{x})$ identifies all concrete states whose symbolic states map `x` to changed. Further, there can be no free variables. Examples of CFTL specifications include the following:

- The property that "the next call to f after each change of the variable `var` should take less than 10 seconds" can be expressed by

$$\forall q \in \mathsf{changes}(\mathtt{var}) : \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 10).$$

  The predicate $\mathsf{calls}(f)$ captures all transitions that represent calls of the function f and next refers to the next transition (after q) in the dynamic run satisfying the predicate $\mathsf{calls}(f)$.
- The property "whenever the function f is called, its duration must be no more than 0.001 times the length of the list held in variable `x` immediately before the call" can be expressed by

$$\forall t \in \mathsf{calls}(f) : \mathsf{duration}(t) < \mathsf{length}(\mathsf{source}(t)(\mathtt{x})) \times 0.001.$$

  $\mathsf{source}(t)(\mathtt{x})$ gives the concrete state immediately before the transition t, and then gets the value of the program variable `x` in that concrete state.

3 iCFTL: Inter-procedural CFTL

We now present an extension of CFTL to the inter-procedural setting. This new language is called iCFTL.

3.1 Systems of Multiple Procedures

In the intra-procedural setting, we assume that traces being checked for satisfaction of some CFTL specification are generated by single procedures. In the inter-procedural setting, we will be checking a trace generated by some *system* consisting of multiple procedures, each of which is a program obtained from the grammar in Figure 1. This enables us to construct their symbolic control-flow graphs. We group the name and program associated with each procedure in a *system of multiple procedures*.

Definition 1. *A system of multiple procedures S is a pair $\langle \mathscr{P}, \mathsf{prog} \rangle$ for $\mathscr{P}$ a set of names of procedures and* prog *a map that sends each name in $\mathscr{P}$ to a program generated by the grammar in Figure 1.*

We will often refer to a *system of multiple procedures* simply as a *system*.

3.2 Inter-procedural Dynamic Runs

The dynamic run defined in Section 2.2, when considered in the scope of an entire system of multiple procedures, represents a single execution of some procedure. In order to define a language similar to CFTL, but with the ability to express properties concerning inter-procedural behaviour, we introduce a kind of trace that represents a run of a system of multiple procedures.

Our approach is to collect the dynamic runs generated by each procedure in a system, label each one with the name of the procedure that generated it, and assume that the timestamps of the concrete states in each dynamic run are synchronised[2]. We refer to a collection of dynamic runs generated by a run of a system S as an *inter-procedural dynamic run* over the system S.

Definition 2. *An inter-procedural dynamic run $\bar{\mathscr{D}}$ over the system S is a triple*

$$\langle \mathscr{P}, \{\mathscr{D}_1, \ldots, \mathscr{D}_n\}, \mathscr{L} \rangle,$$

where $\mathscr{P}$ is a set of names of procedures in the system S; $\{\mathscr{D}_i\}$ is a set of dynamic runs generated by the procedures in S; and $\mathscr{L}$ is a mapping that labels each dynamic run $\mathscr{D}_i$ with the name of the procedure in $\mathscr{P}$ that generated it.

Given a concrete state s in any dynamic run in an inter-procedural dynamic run, we denote by $\mathsf{dynamicRun}(s)$ the unique dynamic run to which s belongs (which exists because each s has a unique timestamp). We extend this to transitions $tr = \langle s, s' \rangle$ by $\mathsf{dynamicRun}(tr) = \mathsf{dynamicRun}(s)$. We then combine $\mathsf{dynamicRun}(.)$ with the map $\mathscr{L}$ to define a map proc by $\mathsf{proc}(s) = \mathscr{L}(\mathsf{dynamicRun}(s))$. Similarly for transitions $tr = \langle s, s' \rangle$, we set $\mathsf{proc}(tr) = \mathsf{proc}(s)$. Intuitively, $\mathsf{proc}(.)$ gives the name of the unique procedure that had control in the system when the concrete state/transition given was attained/taking place.

To generalise our approach to multiple types of systems, we assume that there is always more to observe. While it is possible to observe everything in some cases (e.g., programs that compute a single result and terminate), for other systems it is not. For example, Web services (such as the one used in our case study described in Section 7) constantly receive new requests that trigger repeated executions of procedures.

3.3 Syntax of iCFTL

We give a grammar for the iCFTL syntax in Figure 3. In the grammar, the non-terminal symbols used in rules are highlighted in blue. We also group the rules by *Quantifiers*, *Predicates*, and *Constraints*. We now describe the role of each group of rules, and give examples to illustrate how rules can be applied to construct certain specifications.

Quantifiers. The first rule to apply from the grammar to generate an iCFTL specification is ϕ. This rule can be applied repeatedly in order to generate multiple quantifiers. We will always assume specifications are in prenex normal form.

[2] This assumption is reasonable since either 1) everything will happen on the same machine, so the machine's clock can be used for synchronisation; or 2) if this is not the case, then protocols such as NTP can be used.

$$
\begin{array}{lcl}
 & & \underline{\text{Quantifiers}} \\
\phi & \rightarrow & \forall q \in \Gamma_{QS} : \phi \mid \forall t \in \Gamma_{QT} : \phi \mid \phi \vee \phi \mid \neg\phi \mid \phi_S \mid \phi_T \mid \mathit{true} \\
 & & \underline{\text{Predicates}} \\
\Gamma_{QS} & \rightarrow & \Gamma_S \mid \mathsf{future}(q, \Gamma_S) \mid \mathsf{future}(t, \Gamma_S) \\
\Gamma_{QT} & \rightarrow & \Gamma_T \mid \mathsf{future}(q, \Gamma_T) \mid \mathsf{future}(t, \Gamma_T) \\
\Gamma_S & \rightarrow & \mathsf{changes}(x).\mathsf{during}(p) \\
\Gamma_T & \rightarrow & \mathsf{calls}(f).\mathsf{during}(p) \\
 & & \underline{\text{Concrete State and Transition Selection}} \\
S & \rightarrow & q \mid \mathsf{before}(T) \mid \mathsf{after}(T) \mid S.\mathsf{next}(\Gamma_S) \mid T.\mathsf{next}(\Gamma_S) \\
T & \rightarrow & t \mid S.\mathsf{next}(\Gamma_T) \mid T.\mathsf{next}(\Gamma_T) \\
 & & \underline{\text{Constraints}} \\
\phi_S & \rightarrow & S(x) = v \mid S(x) = S(x) \mid S(x) \in (n,m) \mid S(x) \in [n,m] \\
\phi_T & \rightarrow & \mathsf{duration}(T) \in (n,m) \mid \mathsf{duration}(T) \in [n,m] \\
 & & \mid \mathsf{timeBetween}(S,S) \in (n,m) \mid \mathsf{timeBetween}(S,S) \in [n,m]
\end{array}
$$

Fig. 3: Syntax of iCFTL.

Predicates. Each quantifier requires a *predicate* in order to identify concrete states or transitions to which the variable used in the quantifier should be bound. These predicates are generated by the Γ_{QS} and Γ_{QT} rules. In these cases, there are two parts: one to select the relevant dynamic run (during) and one to select the relevant concrete state or transition from that dynamic run (see Section 2.3).

To give some examples, the predicate $\mathsf{changes}(x).\mathsf{during}(p)$ captures concrete states generated by the procedure p in which the program variable x has just been changed. Similarly, the predicate $\mathsf{calls}(f).\mathsf{during}(p)$ captures transitions representing a call of the procedure f during the procedure p. The future operators extends these predicates to identify all such concrete states or transitions in the future, rather than just the next occurrence.

Combining quantifiers and predicates. Quantifiers and predicates are combined to form *sequences of quantifiers*. An example is

$$\forall q \in \mathsf{changes}(x).\mathsf{during}(f) : \forall tr \in \mathsf{future}(q, \mathsf{calls}(g).\mathsf{during}(h)) : \ldots$$

This sequence of quantifiers would capture each concrete state representing a change of the program variable x (during calls of the procedure f) and, for each change, every call of the function g occurring in the future (during calls of the procedure h).

Concrete State and Transition Selection. Quantifiers allow us to select concrete states and transitions to be used in the inner-most, quantifier free part of a specification. Given these concrete states and transitions, we must be able to *navigate* the inter-procedural dynamic run in order to select others. Using the rules S and T, this can be done by 1) applying the simple operators before and after to transitions (which obtain the concrete state immediately before and immediately after the transition); or 2) using next in

conjunction with one of the predicates $\mathsf{changes}(x).\mathsf{during}(p)$ or $\mathsf{calls}(f).\mathsf{during}(p)$ to search forwards in time. Hence, given concrete states or transitions identified by quantifiers, one can either write constraints over those directly, or use the before, after, or next operators to *navigate* the inter-procedural dynamic run.

Constraints. For an iCFTL specification φ, we denote by $\mathsf{inner}(\varphi)$ the inner-most, quantifier-free part of the specification. Once the sequence of quantifiers has been generated, one can generate $\mathsf{inner}(\varphi)$, which is intuitively the constraint to check at each combination of concrete states/transitions identified by the quantifiers. The grammar allows for disjunction and negation, but we frequently use additional Boolean connectives such as conjunction and implication.

Within the Boolean combination of constraints, each part of the specification generated by an application of either ϕ_S or ϕ_T is called an *atom*. Atoms place constraints on quantities extracted from concrete states or transitions identified using the S and T rules.

Atoms generated by rules containing only one non-terminal symbol are called *normal* and atoms generated by rules containing two non-terminal symbols are called *mixed*. Atoms are the parts of the specification that place constraints on quantities extracted from dynamic runs. We refer to the parts of specifications generated by the rules S and T as *expressions*.

Building Constraints. Suppose we have the sequence of quantifiers

$$\forall q \in \mathsf{changes}(x).\mathsf{during}(f),$$

and would like to assert that each concrete state bound to the variable q maps the program variable x to a value that is strictly less than 10. Our first step would be to take the variable q (treating it as a concrete state) and determine the value to which it maps the variable x. Since there is no *navigation* of the inter-procedural dynamic run to be performed (we are placing a constraint over a quantity measurable directly from the concrete state held by q), we can go immediately to the ϕ_S rule and generate the constraint $q(x) \in (0, 10)$ (acknowledging that there would have to be a conjunction to include the possibility of $q(x)$ being equal to 0).

Formula Trees for iCFTL. Given $\mathsf{inner}(\varphi)$ of an iCFTL specification φ, we denote by $\mathsf{tree}(\mathsf{inner}(\varphi))$ the and-or formula tree of $\mathsf{inner}(\varphi)$. This formula tree is such that leaves correspond to either normal atoms or expressions in mixed atoms. We use this mechanism when defining our monitoring procedure for iCFTL in Section 5.

3.4 Examples

We now give some examples of properties that can be expressed using iCFTL:

– The property "when the variable `level` drops below 10 in the method `check`, the time until the next call of `adjust` in the method `control` should be no more than 1 second" can be expressed by

$$\forall q \in \mathsf{changes}(\texttt{level}).\mathsf{during}(\texttt{check}) : q(\texttt{level}) < 10 \implies \mathsf{timeBetween}(q, \mathsf{before}(q.\mathsf{next}(\mathsf{calls}(\texttt{adjust}).\mathsf{during}(\texttt{control})))) \in [0, 1].$$

$$
\begin{array}{l}
\bar{\mathscr{D}}, q \vdash \mathsf{changes}(\mathtt{x}).\mathsf{during}(\mathtt{func}) \text{ iff } \sigma(\mathtt{x}) = \mathsf{changed} \text{ and } \mathsf{proc}(q) = \mathtt{func} \\
\bar{\mathscr{D}}, q \vdash \mathsf{future}(s, \mathsf{changes}(\mathtt{x}).\mathsf{during}(\mathtt{func})) \text{ iff} \\
\qquad \mathsf{t}(q) > \mathsf{t}(s) \text{ and } \bar{\mathscr{D}}, q \vdash \mathsf{changes}(\mathtt{x}).\mathsf{during}(\mathtt{func}) \\
\bar{\mathscr{D}}, tr \vdash \mathsf{calls}(\mathtt{f}).\mathsf{during}(\mathtt{func}) \text{ iff} \\
\qquad \left(\begin{array}{l} \text{for every path } \pi \in \mathsf{paths}(tr) \text{ there is:} \\ \quad \text{some } \langle \sigma_1, \sigma_2 \rangle \in \pi \text{ such that } \sigma_2(f) = \mathsf{called} \end{array} \right) \text{ and } \mathsf{proc}(tr) = \mathtt{func} \\
\bar{\mathscr{D}}, tr \vdash \mathsf{future}(s, \mathsf{calls}(\mathtt{f}).\mathsf{during}(\mathtt{func})) \text{ iff } \mathsf{t}(tr) > \mathsf{t}(s) \text{ and } \bar{\mathscr{D}}, tr \vdash \mathsf{calls}(\mathtt{f}).\mathsf{during}(\mathtt{func})
\end{array}
$$

Fig. 4: The quantifier relation $\vdash$ for the iCFTL semantics.

– The property "for each change of the variable `user` during an execution of the procedure `login`, and for each future change of the variable `user` during executions of the procedure `getUser`, the value of variable `user` should remain the same", taking some liberties with syntax, can be expressed by:

$$
\begin{array}{l}
\forall q \in \mathsf{changes}(\mathtt{user}).\mathsf{during}(\mathtt{login}) : \\
\forall q' \in \mathsf{future}(q, \mathsf{changes}(\mathtt{user}).\mathsf{during}(\mathtt{getUser})) : q'(\mathtt{user}) = q(\mathtt{user})
\end{array} \tag{2}
$$

The key syntactic novelty in iCFTL is the during component of predicates, which allows one to refer to events across multiple procedures.

4 A Semantics for iCFTL

In order to align with our case study (see Section 7), which is a Web service whose traces must be assumed to be infinite, we define the semantics of iCFTL with respect to prefixes of infinite program traces. We take a similar approach to much existing work in the RV community: we define a semantics over prefixes of program traces [6] and give a *provisional* verdict. This semantics consists of two key steps. The first involves deriving a set of *bindings* by inspection of an inter-procedural dynamic run $\bar{\mathscr{D}}$ with respect to the quantifiers in an iCFTL specification φ. These bindings will collect together concrete states and transitions from $\bar{\mathscr{D}}$ and provide them to $\mathsf{inner}(\varphi)$. The second step involves evaluating $\mathsf{inner}(\varphi)$ with respect to each binding derived.

4.1 Finding Bindings

Given an inter-procedural dynamic run $\bar{\mathscr{D}}$ and an iCFTL specification φ, our goal is to inspect its quantifiers $\forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n$ in order to derive a set of maps, which we will refer to as *bindings*. These bindings will send the variable q_1 to a concrete state or transition satisfying Γ_1, the variable q_2 to a concrete state or transition satisfying Γ_2, and so on. We will then take each binding and decide on a truth value for $\mathsf{inner}(\varphi)$ based on the values given to each variable $q_1, \ldots, q_n$ by the binding.

We begin the construction of the set of bindings by defining the *quantifier* relation, denoted by $\vdash$, that indicates whether a given concrete state q or transition tr satisfies a predicate used in a quantifier. The definition is given in Figure 4.

$$\bar{\mathscr{D}} = \left\langle \begin{array}{l} \{\texttt{login}, \texttt{getUser}, \texttt{getUserData}\}, \{\mathscr{D}_1, \mathscr{D}_2, \mathscr{D}_3, \mathscr{D}_4\}, \\ [\mathscr{D}_1 \mapsto \texttt{login}, \mathscr{D}_2 \mapsto \texttt{getUser}, \mathscr{D}_3 \mapsto \texttt{getUserData}, \mathscr{D}_4 \mapsto \texttt{getUser}] \end{array} \right\rangle$$

$$\mathscr{D}_1 = \langle 0, [], [] \rangle, \langle 0.2, [\texttt{getUser} \mapsto \mathsf{called}, \texttt{user} \mapsto \mathsf{changed}], [\texttt{user} \mapsto 10] \rangle, \ldots$$

$$\mathscr{D}_2 = \langle 0.1, [], [] \rangle, \langle 0.15, [\texttt{user} \mapsto \mathsf{changed}, \ldots], [\texttt{user} \mapsto 10] \rangle, \ldots$$

$$\mathscr{D}_3 = \langle 0.3, [], [] \rangle, \langle 0.45, [\texttt{getUser} \mapsto \mathsf{called}, \texttt{user} \mapsto \mathsf{changed}], [\texttt{user} \mapsto 10] \rangle, \ldots$$

$$\mathscr{D}_4 = \langle 0.35, [], [] \rangle, \langle 0.4, [\texttt{user} \mapsto \mathsf{changed}, \ldots], [\texttt{user} \mapsto 10] \rangle, \ldots$$

Fig. 5: An example inter-procedural dynamic run.

The second and final step is to recurse on the quantifiers in order to progressively construct the set of bindings, which we denote by $\mathsf{bindings}_{\varphi}(\bar{\mathscr{D}}, \forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n)$. This is done by determining the concrete states or transitions that satisfy the predicate of the first quantifier and then, if there are multiple quantifiers, identifying the concrete states or transitions that satisfy the next predicates. The check for satisfaction at each step is based on the relation defined in Figure 4. We highlight that, if no concrete states or transitions are identified by a predicate, a binding is generated that does not include all variables from the specification. We refer to such a binding as *partial*.

An example. In order to illustrate the procedure for constructing bindings, we consider the iCFTL specification in Equation 2 along with an inter-procedural dynamic run given in Figure 5. In this inter-procedural dynamic run, there are three procedures, `login`, `getUser`, `getUserData`. We assume that both `login` and `getUserData` involve calls to `getUser`. One can see the caller-callee relationship between dynamic runs when all of the timestamps of concrete states of a callee dynamic run fall in between two timestamps of consecutive concrete states in the caller dynamic run.

Based on the inter-procedural dynamic run in Figure 5, binding construction would go as follows: The procedure would identify concrete states that satisfy the first quantifier, and then inspect the second quantifier. For the first quantifier

$$\mathsf{changes}(\texttt{user}).\mathsf{during}(\texttt{login}),$$

the concrete state $\langle 0.2, [\texttt{getUser} \mapsto \mathsf{called}, \texttt{user} \mapsto \mathsf{changed}], [\texttt{user} \mapsto 10] \rangle$ would be identified. Based on this initially identified concrete state, we look for further concrete states satisfying

$$\mathsf{future}(q, \mathsf{changes}(\texttt{user}).\mathsf{during}(\texttt{getUser})),$$

with respect to the concrete state $\langle 0.2, [\texttt{getUser} \mapsto \mathsf{called}, \texttt{user} \mapsto \mathsf{changed}], [\texttt{user} \mapsto 10] \rangle$ identified by the first quantifier. Hence, the concrete state $\langle 0.4, [\texttt{user} \mapsto \mathsf{changed}, \ldots], [\texttt{user} \mapsto 10] \rangle$ would be identified. Since all quantifiers have been inspected, we

conclude with the set of bindings:

$$\left\{ \begin{array}{l} \left[q \mapsto \langle 0.2, [\texttt{getUser} \mapsto \mathsf{called}, \texttt{user} \mapsto \mathsf{changed}], [\texttt{user} \mapsto 10] \rangle \right], \\ \left[\begin{array}{l} q \mapsto \langle 0.2, [\texttt{getUser} \mapsto \mathsf{called}, \texttt{user} \mapsto \mathsf{changed}], [\texttt{user} \mapsto 10] \rangle, \\ q' \mapsto \langle 0.4, [\texttt{user} \mapsto \mathsf{changed}, \ldots], [\texttt{user} \mapsto 10] \rangle \end{array} \right] \end{array} \right\}$$

Notice that we keep a binding that only sends q to a concrete state, and not q'. This is to capture the intuition that the binding with only q may be extended with a new value of q' given more observations from the monitored system.

4.2 Evaluation at a binding

The next step in developing the semantics is to evaluate the constraints defined by the specification at each of the bindings in the set $\mathsf{bindings}_{\varphi}(\bar{\mathscr{D}}, \forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n)$. For this, we introduce the $\mathsf{eval}(\bar{\mathscr{D}}, \beta, X)$ function.

This function takes an expression from the specification, along with a binding and an inter-procedural dynamic run, and gives the unique concrete state or transition that is required by that expression. If no such concrete state or transition exists, the function returns null.

Once we have obtained the concrete state or transition referred to by an expression, we can determine the truth values of atoms, and therefore the truth values of Boolean combinations of atoms. This process is encoded in the $[.]_{\beta}$ function, which is defined recursively in Figure 6. The function $[.]_{\beta}$ takes as input an inter-procedural dynamic run, a binding and either a Boolean combination of atoms, or a single atom, and gives a truth value from the set $\{\mathsf{true}, \mathsf{false}, \mathsf{inconclusive}\}$. This set has ordering $\mathsf{false} < \mathsf{inconclusive} < \mathsf{true}$ with $\neg\mathsf{inconclusive} = \mathsf{inconclusive}$.

If a single atom is given and the required concrete states and transitions are found, the truth value given is either true or false. If a single atom is given and no concrete state or transition is identified, the truth value is inconclusive. If a Boolean combination of atoms is given, the total order of the truth domain is used to determine the truth value, given the truth values of the subformulas[3].

4.3 The Semantics Function

We have now introduced the machinery for 1) extracting a set of bindings from an inter-procedural dynamic run based on a specification; and 2) determining the truth values of atoms in an iCFTL specification given a specific binding. The final step in defining the semantics for iCFTL is to combine all of these components in order to give a verdict.

While most existing work in RV concentrates on generating verdicts that are simple objects, such as true or false, taken from a truth domain, our approach differs. Instead, the verdict that we provide is a map from bindings extracted from the inter-procedural

[3] Of course, if the specification expresses a tautology or is unsatisfiable, this evaluation-by-composition approach is problematic. However, as seen in [13], satisfiability for CFTL (and therefore iCFTL) can only be decided once a sufficiently long trace has been observed, hence we do not consider it in the semantics.

$$[\bar{\mathscr{D}},\beta,\phi_1 \vee \phi_2]_\beta = [\bar{\mathscr{D}},\beta,\phi_1]_\beta \sqcup [\bar{\mathscr{D}},\beta,\phi_2]_\beta \qquad [\bar{\mathscr{D}},\beta,\neg\phi]_\beta = \neg[\bar{\mathscr{D}},\beta,\phi]_\beta$$

$$[\bar{\mathscr{D}},\beta,S(x)=n]_\beta = \begin{cases} \mathsf{true} & \mathsf{eval}(\bar{\mathscr{D}},\beta,S) \neq \mathsf{null} \wedge \mathsf{eval}(\bar{\mathscr{D}},\beta,S)(x) = n \\ \mathsf{false} & \mathsf{eval}(\bar{\mathscr{D}},\beta,S) \neq \mathsf{null} \wedge \mathsf{eval}(\bar{\mathscr{D}},\beta,S)(x) \neq n \\ \mathsf{inconclusive} & \text{otherwise} \end{cases}$$

$$[\bar{\mathscr{D}},\beta,S_1(x)=S_2(x)]_\beta = \begin{cases} \mathsf{true} & \mathsf{eval}(\bar{\mathscr{D}},\beta,S_1) \neq \mathsf{null} \wedge \mathsf{eval}(\bar{\mathscr{D}},\beta,S_2) \neq \mathsf{null} \\ & \wedge\, \mathsf{eval}(\bar{\mathscr{D}},\beta,S_1)(x) = \mathsf{eval}(\bar{\mathscr{D}},\beta,S_2)(x) \\ \mathsf{false} & \mathsf{eval}(\bar{\mathscr{D}},\beta,S_1) \neq \mathsf{null} \wedge \mathsf{eval}(\bar{\mathscr{D}},\beta,S_2) \neq \mathsf{null} \\ & \wedge\, \mathsf{eval}(\bar{\mathscr{D}},\beta,S_1)(x) \neq \mathsf{eval}(\bar{\mathscr{D}},\beta,S_2)(x) \\ \mathsf{inconclusive} & \text{otherwise} \end{cases}$$

Fig. 6: Part of the constraint function for iCFTL.

$$\mathsf{translate}(\bar{\mathscr{D}},\beta,\varphi) = \begin{cases} \mathsf{true} & [\bar{\mathscr{D}},\beta,\mathsf{inner}(\varphi)]_\beta = \mathsf{true} \wedge \beta \text{ is complete} \\ \mathsf{false} & [\bar{\mathscr{D}},\beta,\mathsf{inner}(\varphi)]_\beta = \mathsf{false} \wedge \beta \text{ is complete} \\ \mathsf{inconclusive} & [\bar{\mathscr{D}},\beta,\mathsf{inner}(\varphi)]_\beta = \mathsf{inconclusive} \wedge \beta \text{ is complete} \\ \mathsf{true}_p & [\bar{\mathscr{D}},\beta,\mathsf{inner}(\varphi)]_\beta = \mathsf{true} \wedge \beta \text{ is partial} \\ \mathsf{false}_p & [\bar{\mathscr{D}},\beta,\mathsf{inner}(\varphi)]_\beta = \mathsf{false} \wedge \beta \text{ is partial} \\ \mathsf{inconclusive}_p & [\bar{\mathscr{D}},\beta,\mathsf{inner}(\varphi)]_\beta = \mathsf{inconclusive} \wedge \beta \text{ is partial} \end{cases}$$

Fig. 7: The translation function.

dynamic run to truth values. In addition, we encode in these truth values whether or not the binding associated with the truth value is partial. We do this because, if a binding is partial, we cannot be sure that it will be extended to form a complete binding given further observations from the system.

In order to provide this distinction, we introduce the translate function. This function, defined in Figure 7, translates from the truth values $\{\mathsf{true},\mathsf{false},\mathsf{inconclusive}\}$ given by the function in Figure 6 to *complete* and *partial* versions of the same truth values:

$$\{\mathsf{true},\mathsf{true}_p,\mathsf{false},\mathsf{false}_p,\mathsf{inconclusive},\mathsf{inconclusive}_p\}.$$

Finally, we define the semantics function $[\bar{\mathscr{D}},\varphi]_S$, which simply computes the set of bindings for a given inter-procedural dynamic run with respect to an iCFTL specification and, for each one, gives the value of the translation function:

$$[\bar{\mathscr{D}},\varphi]_S = [\beta \mapsto \mathsf{translate}(\bar{\mathscr{D}},\beta,\varphi) : \beta \in \mathsf{bindings}_\varphi(\bar{\mathscr{D}},\forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n)].$$

Hence, for a given inter-procedural dynamic run and iCFTL specification, the *verdict* that we compute is the map given by $[\bar{\mathscr{D}},\varphi]_S$ indicating 1) the truth values given by the

constraints in the specification at each binding; and 2) the type of binding for which each truth value was obtained.

5 Monitoring

We now develop an initial algorithm that processes an inter-procedural dynamic run and an iCFTL specification in order to give a verdict. The algorithm is inspired by the work on CFTL and VYPR [17,14]. We will see that it does not scale well, and describe an instrumentation process in Section 6 to greatly improve the situation.

Given an inter-procedural dynamic run $\langle \mathscr{P}, \{\mathscr{D}_1, \ldots, \mathscr{D}_m\}, \mathscr{L} \rangle$, our "naive" monitoring approach for iCFTL, given in Algorithm 1, consists of iterating through the concrete states contained in all of the dynamic runs $\mathscr{D}_i$ in ascending order of timestamps. This sequence of concrete states is denoted by $\mathsf{flattened}(\bar{\mathscr{D}})$. For each concrete state curr, we see if 1) curr, or the transition leading into it, contributes to a binding and 2) curr, or the transition leading to it, contributes to the truth value of some atom. If curr or the transition leading to it contributes to the truth value of some atom, Algorithm 1 uses *formula trees* to determine the new truth value of $\mathsf{inner}(\varphi)$. We highlight that † is the *map update* operator, that is, $a \dagger [e \mapsto v]$ refers to the map that agrees with a on all elements of the domain of a, except for e which is mapped to v.

Formula trees. The monitoring algorithm often instantiates a new formula tree $\mathsf{tree}(\phi)$ (or uses an existing one) and then *updates* it with the update function. This function takes a formula tree with a concrete state or transition and replaces the relevant nodes in the formula tree accordingly.

If the concrete state/transition given is relevant to an expression, the node holding that expression is replaced by the value given to that expression by the concrete state/-transition. For example, if we have the concrete state $\langle 0.1, [\mathtt{x} \mapsto \mathsf{changed}], [\mathtt{x} \mapsto 1.5] \rangle$ and an expression q on a leaf of the formula tree, the latter can be replaced by 1.5 if the q is part of $q(x)$ in the specification. Alternatively, if the concrete state/transition is relevant to a normal atom, the node holding that atom is replaced with a truth value. For example, given the same concrete state and an atom $q(x) < 2$, the leaf could be replaced by true. Once this replacement has taken place, the formula tree is collapsed based on the conventional rules for propositional connectives.

Correctness. A correctness argument for Algorithm 1 involves showing that 1) the bindings generated by the algorithm and the semantics are the same; and 2) the procedure for obtaining truth values of $\mathsf{inner}(\varphi)$ in the algorithm has the same result as the semantics. It is similar to the one given in [13].

Complexity. The main loop of Algorithm 1 performs as many iterations as there are concrete states in $\bar{\mathscr{D}}$, which we will denote by $|\bar{\mathscr{D}}|$. For each of these iterations, we process the existing bindings, of which there are at most $|\bar{\mathscr{D}}|^m/m!$ [13]. Hence, the approximate complexity is $O(|\bar{\mathscr{D}}|^{m+1}/m!)$. We highlight that m is rarely greater than 2, hence the complexity can be seen as $O(|\bar{\mathscr{D}}|^{m+1})$, meaning that, even for specifications with only one quantifier, this initial monitoring algorithm scales quadratically in the length of the trace.

Algorithm 1 Monitoring for an iCFTL specification $\forall q_1 \in \Gamma_1 : \cdots : \forall q_m \in \Gamma_m : \phi$.

```
1:  M ← []                                          ▷ empty map from bindings to formula trees
2:  prev ← ⟨t_1, [], []⟩   ▷ to store the previous state, assuming t_1 is the first timestamp in the
    dynamic run
3:  for concrete state curr ∈ flattened(D̄) do
4:      ▷ Handle the cases where a new binding should be generated
5:      ▷ New bindings are generated if the state/transition is in Γ_1
6:      if curr ⊢ Γ_1 then
7:          M = M † ([q_1 ↦ curr] ↦ update(tree(φ), curr))
8:      if (prev, curr) ⊢ Γ_1 then
9:          M = M † ([q_1 ↦ ⟨prev, curr⟩] ↦ update(tree(φ), ⟨prev, curr⟩))
10:     ▷ Bindings are extended if the state/transition is in Γ_i for i > 1
11:     for (β = [q_1 ↦ v_1, ... q_k ↦ v_k], T) in M where k < m do
12:         if curr ⊢ Γ_{k+1} then
13:             M = M † ((β † [q_{k+1} ↦ curr]) ↦ update(T, curr))
14:         if (prev, curr) ⊢ Γ_{k+1} then
15:             M = M † ((β † [q_{k+1} ↦ ⟨prev, curr⟩]) ↦ update(T, ⟨prev, curr⟩))
16:     for (β, T) in M do                          ▷ Now update formula trees for existing bindings
17:         T' ← update(T, curr)
18:     prev ← curr                                 ▷ Finally save the current state as the last state
19: return M
```

6 Instrumentation

The inefficiency of Algorithm 1 has two principal causes: 1) the amount of unnecessary information contained in the inter-procedural dynamic run being processed; and 2) the lookup required to decide whether a concrete state/transition contributes to a binding. To improve the situation, we traverse the symbolic control-flow graphs of the relevant procedures in order to determine which program points are relevant to a specification. This traversal is part of the process of instrumentation, whose steps are described in the following sections.

6.1 Inspection of the Quantifiers

For an iCFTL specification with quantifiers $\forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n$, we recursively construct maps from the variables $q_1, \ldots, q_n$ to symbolic states from the relevant symbolic control-flow graphs. As an example, suppose that we are monitoring a system $S = \langle \{\mathtt{f}\}, \mathsf{prog} \rangle$. If a specification had the quantifier sequence $\forall q \in \mathsf{changes}(\mathtt{x}).\mathsf{during}(\mathtt{f})$, we would identify all symbolic states σ in the symbolic control-flow graph of the program $\mathsf{prog}(\mathtt{f})$ that had $\sigma(\mathtt{x}) = \mathsf{changed}$. A final step, which helps during monitoring, is the assignment of a unique integer (i.e., an index) to each map from the variables q_i to concrete states.

6.2 Inspection of the Atoms

Given the set of maps constructed from the inspection of quantifiers, we perform further traversal of the symbolic control-flow graphs of the system for each atom in $\mathsf{inner}(\varphi)$. For example, if we had the iCFTL specification

$$\forall q \in \mathsf{changes}(\mathtt{x}).\mathsf{during}(\mathtt{f}) : \mathsf{duration}(q.\mathsf{next}(\mathsf{calls}(\mathtt{g}).\mathsf{during}(\mathtt{h}))) < 1,$$

then we would traverse the symbolic control-flow graph of the procedure $\mathsf{prog}(\mathtt{h})$ in order to find symbolic states σ with $\sigma(\mathtt{g}) = \mathsf{called}$.

6.3 Filtering Dynamic Runs

After applying the procedure described in Sections 6.1 and 6.2, we have a set of symbolic states, which we call *instrumentation points*, taken from the various symbolic control-flow graphs in the system. This set of instrumentation points is such that we can remove any concrete state from an inter-procedural dynamic run that does not correspond to one of the symbolic states in the set. More formally, if a concrete state $\langle t, \sigma, m \rangle$ appears in an inter-procedural dynamic run and σ is not in the set of instrumentation points, the concrete state can be removed from the inter-procedural dynamic run. The safety of this approach is proved in [13]. While the proof there is for CFTL, the instrumentation approaches are sufficiently similar that it applies here.

6.4 Lookup during Monitoring

An important source of inefficiency in Algorithm 1 is the requirement to find the formula trees that must be updated, given a measurement from an inter-procedural dynamic run. In order to reduce the number of formula trees that must be checked, we group formula trees by the unique integer identifying each map constructed in Section 6.1. For each concrete state, we can then extract its symbolic state and determine which map (if any) it belongs to (this can be performed as a pre-processing step to improve lookup speeds further). With formula trees grouped with respect to these uniquely identifying integers, we can then determine the set of formula trees that must be updated.

6.5 Implications for Complexity

These optimisations mean that 1) there can be fewer concrete states to process, since we can filter them based on relevant symbolic states; and 2) lookup of the relevant formula trees is faster. A more in-depth discussion can be found in [17] and [13].

7 Case Study

To evaluate iCFTL, we have extended the existing VYPR framework [14] to include a new library for building iCFTL specifications, along with machinery for instrumentation and monitoring. The prototype implementation is available under the Apache 2.0

license at https://doi.org/10.5281/zenodo.5195959, with development occurring at https://github.com/SNTSVV/VyPR-iCFTL.

Using this implementation, we have performed some initial experiments on a Python-based case study [12] provided by the CMS Experiment at CERN [9], where the initial work on CFTL and VyPR was performed. This case study is a Web service, therefore consisting of a server and (to simplify this initial experimental setting) a single client. The client uploads data to the server over the course of multiple HTTP requests. We note that the restriction of CFTL to the intra-procedural setting means that properties that require measurements to be taken over multiple HTTP requests certainly cannot be expressed. With iCFTL this limitation is gone, since inter-procedural also means "inter-request".

In order to demonstrate this concretely, we present an iCFTL specification that we have been able to monitor, which could not be expressed in CFTL (hence demonstrating the usefulness of taking the features offered by CFTL and extending them to the inter-procedural setting). We also present initial measurements of the overhead induced by the extended implementation of VyPR.

Specification. Our principal specification, taking liberties with syntax, is:

$$\begin{aligned}&\forall c \in \mathsf{calls}(\texttt{find\_new}).\mathsf{during}(\texttt{app.routes.hashes}) : \\ &\quad \mathsf{timeBetween}(\mathsf{before}(c), \\ &\qquad \mathsf{after}(c.\mathsf{next}(\mathsf{calls}(\texttt{get\_usage}).\mathsf{during}(\texttt{app.routes.upload\_md}))) < 4\end{aligned}$$

This specification expresses the property that the time between two concrete states attained at runtime in two different procedures of the system under scrutiny must be less than four seconds. The first of these concrete states is the concrete state immediately before the transition representing a call of the function `find_new`, occurring during a dynamic run generated by the `hashes` procedure (found in the module `app.routes`). The second concrete state is the one immediately after the transition representing the next call of `get_usage` that occurs during a dynamic run generated by the `upload_md` procedure.

Overhead. In order to obtain initial measurements of the overhead induced by VyPR, we ran the same upload 100 times, with and without monitoring. When this was done with *no delay* between uploads, we obtained an overhead of approximately 3.22%. By introducing a delay between uploads, we could reduce the overhead to 1.69%. We highlight that this delay between requests allows any measurements not processed by the VyPR monitoring algorithm *during* requests to be processed between requests. A more detailed discussion of the overhead induced by VyPR is given in [13].

8 Related Work

We first discuss the relationship between iCFTL and CFTL [17]. Despite the fact that iCFTL can express properties that could not be expressed in CFTL, if we restrict the semantics of iCFTL to a single dynamic run, the newly introduced syntax does not allow any new kinds of properties to be expressed in this setting. That is, the notable

extensions of the syntax (the introduction of the during(p) component to various predicates) would not be useful in the intra-procedural setting. Hence, rather than referring to iCFTL as an improvement on the expressive power of CFTL, we refer to iCFTL as an extension of the features provided by CFTL into the inter-procedural setting.

iCFTL is a departure from the conventional approach seen in (or adapted to) the RV community, which often involves an extremely expressive specification formalism with a high level of abstraction [24,23,4,21]. iCFTL does not distinguish between the symbols that are used in a specification and the events that occur during the runtime of a program. This has the disadvantage that a change to the source code requires a change to the specification, however we argue that specifications should be actively maintained as code changes. Work with a similar approach includes LARVA [10], which provides specification formalisms with a low level of abstraction, but that focus on the order of events, which is not the focus of iCFTL. Further, CARET [25,2] allows references to (untimed) function calls and returns. In contrast, iCFTL enables one to talk about time and data, alongside function calls (which are not separated into call and return events).

The monitoring approach used for iCFTL varies from many used in RV in that many approaches use automata [7,20]. Given that iCFTL specifications are universally-quantified, and must be in prenex-normal form, there is some similarity between our monitoring approach and that used for the Quantified Event Automata formalism [3]. Here, bindings are generated and a central monitor structure (an automaton) is instantiated for each binding. If one replaces these automata with our formula trees, the approaches are similar.

Our instrumentation approach applies static analysis to determine the program points from which data should be taken at runtime. We also use instrumentation to optimise lookup. There are multiple bodies of work in RV that use instrumentation to optimise the monitoring process. The most notable include CLARA [8], which applies a series of static analyses in order to decide which statements do not need to be instrumented (and in order to generate a *residual property*, which is a property that has been partially proved during static analysis), and STARVOORS [1], which attempts to prove pre- and post-conditions of specifications written in a specification formalism that combines DATEs [10] and Dynamic Logic [19].

9 Conclusion

The central contribution of this paper is our new specification language, iCFTL (Inter-procedural Control-Flow Temporal Logic). This new logic enables the expression of properties concerning the inter-procedural, source code level behaviour of programs. Such properties cannot be expressed in CFTL, which focuses on the intra-procedural setting.

Our development of iCFTL has involved introduction of an initial monitoring algorithm, along with the acknowledgement that this does not scale well for larger traces. To remedy the situation, we have made reference to our previous work on instrumentation which can be applied with almost no modification. To demonstrate the expressiveness of iCFTL, we have extended the existing VYPR framework and applied it to a case study: a Web service used at the CMS Experiment at CERN. This case study has demon-

strated that more properties can be checked, leading to enhanced analysis capability of the VYPR framework.

Key directions for future work identified so far include 1) refining our instrumentation approach and 2) translating the explanation machinery presented recently [14,16] into the inter-procedural setting.

Acknowledgments. The research described has been carried out as part of the COSMOS Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 957254. The authors wish to thank Lionel Briand for his feedback on iCFTL, and the CMS Experiment at CERN for help with the case study.

References

1. Ahrendt, W., Pace, G.J., Schneider, G.: A Unified Approach for Static and Runtime Verification: Framework and Applications. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I. Lecture Notes in Computer Science, vol. 7609, pp. 312–326. Springer (2012). https://doi.org/10.1007/978-3-642-34026-0_24, https://doi.org/10.1007/978-3-642-34026-0_24
2. Alur, R., Etessami, K., Madhusudan, P.: A Temporal Logic of Nested Calls and Returns. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2988, pp. 467–481. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_35, https://doi.org/10.1007/978-3-540-24730-2_35
3. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7436, pp. 68–84. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_9, https://doi.org/10.1007/978-3-642-32759-9_9
4. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-Based Runtime Verification. In: Steffen, B., Levi, G. (eds.) Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2937, pp. 44–57. Springer (2004). https://doi.org/10.1007/978-3-540-24622-0_5, https://doi.org/10.1007/978-3-540-24622-0_5
5. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to Runtime Verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457, pp. 1–33. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_1, https://doi.org/10.1007/978-3-319-75632-5_1
6. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL Semantics for Runtime Verification. J. Log. Comput. **20**(3), 651–674 (2010). https://doi.org/10.1093/logcom/exn075, https://doi.org/10.1093/logcom/exn075

7. Bensalem, S., Bozga, M., Krichen, M., Tripakis, S.: Testing Conformance of Real-Time Applications by Automatic Generation of Observers. Electron. Notes Theor. Comput. Sci. **113**, 23–43 (2005). https://doi.org/10.1016/j.entcs.2004.01.036, https://doi.org/10.1016/j.entcs.2004.01.036
8. Bodden, E., Lam, P., Hendren, L.J.: Clara: A Framework for Partially Evaluating Finite-State Runtime Monitors Ahead of Time. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6418, pp. 183–197. Springer (2010). https://doi.org/10.1007/978-3-642-16612-9_15, https://doi.org/10.1007/978-3-642-16612-9_15
9. CERN: Compact Muon Solenoid experiment. https://home.cern/science/experiments/cms
10. Colombo, C., Pace, G.J., Schneider, G.: Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. In: Cofer, D.D., Fantechi, A. (eds.) Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers. Lecture Notes in Computer Science, vol. 5596, pp. 135–149. Springer (2008). https://doi.org/10.1007/978-3-642-03240-0_13, https://doi.org/10.1007/978-3-642-03240-0_13
11. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: Runtime Monitoring of Synchronous Systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA. pp. 166–174. IEEE Computer Society (2005). https://doi.org/10.1109/TIME.2005.26, https://doi.org/10.1109/TIME.2005.26
12. Dawes, J.H.: A Python object-oriented framework for the CMS alignment and calibration data. Journal of Physics: Conference Series **898**, 042059 (oct 2017). https://doi.org/10.1088/1742-6596/898/4/042059
13. Dawes, J.H.: Towards Automated Performance Analysis of Programs by Runtime Verification. Ph.D. thesis, University of Manchester (2021)
14. Dawes, J.H., Han, M., Javed, O., Reger, G., Franzoni, G., Pfeiffer, A.: Analysing the Performance of Python-Based Web Services with the VyPR Framework. In: Deshmukh, J., Nickovic, D. (eds.) Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12399, pp. 67–86. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_4, https://doi.org/10.1007/978-3-030-60508-7_4
15. Dawes, J.H., Han, M., Reger, G., Franzoni, G., Pfeiffer, A.: Analysis Tools for the VyPR Framework for Python. In: International Conference on Computing in High Energy and Nuclear Physics, Adelaide, Australia 2019 (2019)
16. Dawes, J.H., Reger, G.: Explaining Violations of Properties in Control-Flow Temporal Logic. In: Finkbeiner, B., Mariani, L. (eds.) Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11757, pp. 202–220. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_12, https://doi.org/10.1007/978-3-030-32079-9_12
17. Dawes, J.H., Reger, G.: Specification of temporal properties of functions for runtime verification. In: Hung, C., Papadopoulos, G.A. (eds.) Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019. pp. 2206–2214. ACM (2019). https://doi.org/10.1145/3297280.3297497, https://doi.org/10.1145/3297280.3297497

18. Dou, W., Bianculli, D., Briand, L.C.: A Model-Driven Approach to Trace Checking of Pattern-Based Temporal Properties. In: 20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017. pp. 323–333. IEEE Computer Society (2017). https://doi.org/10.1109/MODELS.2017.9, https://doi.org/10.1109/MODELS.2017.9
19. Fischer, M.J., Ladner, R.E.: Propositional Dynamic Logic of Regular Programs. J. Comput. Syst. Sci. **18**(2), 194–211 (1979). https://doi.org/10.1016/0022-0000(79)90046-1, https://doi.org/10.1016/0022-0000(79)90046-1
20. Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification. pp. 53–65. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
21. Hallé, S.: When RV Meets CEP. In: Falcone, Y., Sánchez, C. (eds.) Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10012, pp. 68–91. Springer (2016). https://doi.org/10.1007/978-3-319-46982-9_6, https://doi.org/10.1007/978-3-319-46982-9_6
22. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: A Run-Time Assurance Approach for Java Programs. Formal Methods Syst. Des. **24**(2), 129–155 (2004). https://doi.org/10.1023/B:FORM.0000017719.43755.7c, https://doi.org/10.1023/B:FORM.0000017719.43755.7c
23. Koymans, R.: Specifying Real-Time Properties with Metric Temporal Logic. Real Time Syst. **2**(4), 255–299 (1990). https://doi.org/10.1007/BF01995674, https://doi.org/10.1007/BF01995674
24. Pnueli, A.: The Temporal Logic of Programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32, https://doi.org/10.1109/SFCS.1977.32
25. Rosu, G., Chen, F., Ball, T.: Synthesizing Monitors for Safety Properties: This Time with Calls and Returns. In: Leucker, M. (ed.) Runtime Verification, 8th International Workshop, RV 2008, Budapest, Hungary, March 30, 2008. Selected Papers. Lecture Notes in Computer Science, vol. 5289, pp. 51–68. Springer (2008). https://doi.org/10.1007/978-3-540-89247-2_4, https://doi.org/10.1007/978-3-540-89247-2_4