Checking Complex Source Code-Level Constraints using Runtime Verification*

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

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

ABSTRACT

Runtime Verification (RV) is the process of taking a trace, representing an execution of some computational system, and checking it for satisfaction of some specification, written in a specification language. RV approaches are often aimed at being used as part of software development processes. In this case, engineers might maintain a set of specifications that capture properties concerning their source code's behaviour at runtime. To be used in such a setting, an RV approach must provide a specification language that is practical for engineers to use regularly, along with an efficient monitoring algorithm that enables program executions to be checked quickly.

This work develops an RV approach that has been adopted by two industry partners. In particular, we take a source code fragment of an existing specification language, *Source Code and Signal Logic*, which enables properties of interest to our partners to be captured easily, and develop 1) a new semantics for the fragment, 2) an instrumentation approach, and 3) a monitoring procedure for it. We show that our monitoring procedure scales to program execution traces containing up to one million events, and describe initial applications of our prototype framework (that implements our instrumentation and monitoring procedures) by the partners themselves.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; • **Theory of computation** → *Modal and temporal logics.*

KEYWORDS

Runtime Verification, Temporal Logic Semantics, Instrumentation, Trace Checking, Monitoring

ACM Reference Format:

Joshua Heneage Dawes and Domenico Bianculli. 2024. Checking Complex Source Code-Level Constraints using Runtime Verification. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3663529.3663845

*We thank Ricardo Alves and Hugo Reis from GMV, and Miguel Costa and Márcio Mateus from Unparallel Innovations, for their invaluable input. We also thank Alexander Vatov from the University of Luxembourg, for significant contributions to the prototype framework. This work was supported by the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 957254 (COSMOS).

FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663845

1 INTRODUCTION

Runtime Verification (RV) is the process of checking that a trace, which represents an execution of some computational system, is as described by some specification. RV approaches are often aimed at being used alongside conventional testing approaches during software development. In this case, engineers must be able to write and maintain specifications as their code evolves, while being able to determine whether executions of each new version of their code satisfy their specifications. To support this pattern of use, the specification language must be practical to use for engineers, and it must admit an efficient algorithm for checking satisfaction.

In this work, we aim to develop an RV approach that 1) can be integrated within software development processes, and 2) enables engineers to analyse the behaviour of program source code. The software development processes in question are those of our two industry partners, within the context of a large international research project. In particular, our partners develop software whose development places particular emphasis on properties concerning how long I/O operations take, the order in which certain operations occur, and the time taken to reach one operation from another.

An existing family of languages that focus on such properties includes *Control-Flow Temporal Logic* (CFTL) [17], *Inter-procedural Control-Flow Temporal Logic* (iCFTL) [13], and *Source Code and Signal Logic* (SCSL) [15]. CFTL and iCFTL focus solely on the source code domain, both having efficient monitoring procedures that have been shown to scale well when applied to real-world systems. SCSL extends iCFTL by 1) introducing a more flexible syntax that allows both universal and existential quantifiers, along with arbitrary nesting of quantifiers, and 2) providing syntax for describing constraints over signal values. However, SCSL does not yet have a monitoring procedure.

Both of our partners wanted to capture only source code-level properties, meaning that CFTL or iCFTL would be an ideal candidate for a specification language. However, certain properties were identified (of the form *whenever one event occurs, eventually another event should also occur*) that could not be captured in either of these languages. Since both CFTL and iCFTL require specifications to be in prenex form, capturing such a property would require existential quantifiers. These are not provided by CFTL or iCFTL, but are provided by SCSL. Hence, in this work, we consider a syntactic fragment of SCSL designed for capturing source code-level properties; this fragment is called *Source Code Logic* (SCSLc).

Using this syntactic fragment, our first contribution is a new semantics. While SCSL already has a semantics, this is defined over

traces to which additional information can be added (for example, by a system's continued execution). Our work with industry partners showed that such a situation need not be addressed, since the systems to which our work would be applied had short running times, so it would be safe to assume that traces would not be updated with additional information. With this in mind, we develop a semantics over such traces that has the truth domain $\mathbb{B}_2 = \{true, false\}$.

Our second contribution is a monitoring procedure for SCSLc that reflects the semantics that we develop. Specifically, this procedure takes a trace, along with an SCSLc specification, and gives a truth value in $\mathbb{B}_2$ that reflects the answer given by the semantics. We evaluate this monitoring procedure by investigating how well it scales (in terms of time taken and memory consumed) when checking traces containing up to one million events, generated by an open source Python-based project, for satisfaction of a representative set of SCSLc specifications. Our evaluation demonstrates that our monitoring procedure scales well, taking a maximum of ≈44.81 s and consuming a maximum of ≈1.39 GB of memory.

Finally, we also describe the application of our prototype framework in industry and discuss the lessons learned in this context. We show that 1) the framework can be used to check properties of interest to engineers in industry, on real systems; and 2) the framework can be integrated into existing automated testing pipelines, suggesting that it is practical to use. We arrive at these conclusions based on initial results reported by partners: to check traces containing at most 727 events for satisfaction of SCSLc specifications, our framework took a maximum of 0.558 s, and consumed a maximum of 286.52 kB of memory.

To present these contributions, the paper is organised as follows. We introduce SCSLc by recalling the notion of trace used by iCFTL, and then giving the fragment's syntax, in §2. We then develop a semantics for the syntactic fragment in §3. We develop a monitoring algorithm that uses information obtained during instrumentation in §4. The scalability of our monitoring algorithm on traces containing up to one million events is demonstrated in §5. After that, we report on the applicability in industrial contexts of our approach in §6. We then place our contribution in the literature in §7. Finally, we conclude and describe our ongoing activities in §8.

2 PRELIMINARIES: A FRAGMENT OF SCSL

The use cases provided by our industry partners consisted solely of source code, and the properties of interest concerned source code-level behaviour. In this case, one might conclude that iCFTL was an appropriate choice of specification language. However, one property identified by a partner was that "*if the program variable* x *is set to n, then eventually a call of the function* f *should take less than m seconds*", which would require an existential quantifier nested inside an implication. Such a property could not be captured in iCFTL because of its requirement that specifications be in prenex form, meaning that existential quantifiers could not be constructed. With this in mind, we instead considered SCSL.

SCSL does not require specifications to be in prenex form, enabling the construction of existential quantifiers. However, since

$$
\begin{aligned}
&\mathit{Program} \rightarrow x = \mathit{expr} \mid \mathit{func} \mid \mathit{Program}; \mathit{Program} \mid \\
&\qquad \text{if } \mathit{expr} \text{ then } (\mathit{Program}) \text{ else } (\mathit{Program}) \mid \\
&\qquad \text{while } \mathit{expr} \text{ do } (\mathit{Program}) \mid \text{for } x \text{ in } \mathit{iterator} \text{ do } (\mathit{Program}) \qquad (1)\\
&\mathit{expr} \rightarrow x \mid \mathit{func} \mid \mathit{arithExpr} \mid \mathit{boolExpr} \\
&\mathit{func} \rightarrow f(\mathit{expr}_1, \ldots, \mathit{expr}_n) \qquad \mathit{iterator} \rightarrow \text{range}(\mathit{expr}, \mathit{expr})
\end{aligned}
$$

Figure 1: A grammar for simple imperative programs.

that syntax also includes operators designed for capturing properties concerning signal-based behaviour, we now introduce a syntactic fragment, and name the language *Source Code Logic* (SCSLc). Considering a fragment allows us to develop a semantics, and monitoring algorithm, specifically for the source code setting, without considering signals (which would not be needed by our partners).

SCSLc specifications are defined over the same traces as iCFTL, so we begin by recalling their definition in §2.1, before giving the syntax of SCSLc in §2.2.

2.1 Traces

Systems of multiple procedures. The notion of trace used by SCSLc is the notion of a *system of multiple procedures*, S, used by iCFTL. We recall that a system of multiple procedures is a tuple $\langle P, \text{prog}\rangle$, where P is a set of procedure names and prog is a map from procedure names in P to programs (generated by the program grammar in Figure 1). We will often refer to a system of multiple procedures as simply a *system*.

Program points. Given a system $S = \langle P, \text{prog}\rangle$, one can compute the abstract syntax tree of $\text{prog}(p)$ for each $p \in P$ using the grammar in Figure 1. From there, one can then assign a unique identifier to each node in the abstract syntax tree of each $\text{prog}(p)$ (by labelling each node with an integer). Using this fact, given a statement stmt in a program p, we denote by pPoint(stmt) the *program point* of stmt (i.e., its unique identifier in p). We highlight that program points can be computed to be unique across all procedures in a system by taking disjoint sets of integers for each procedure in P when labelling the abstract syntax tree nodes.

Predicates. We now define predicates on program statements. To begin with, we denote by $\text{isCall}_{\mathsf{f}}$ a predicate that takes as input a statement stmt and is true if the statement stmt contains a call of the procedure f and false otherwise. We also denote by $\text{isChange}_{\mathsf{x}}$ the predicate that takes as input a statement stmt and is true if the statement stmt assigns a value to the program variable x, and false otherwise. Both of these predicates can be computed using a simple static analysis of the source code.

Executions of systems. We now develop our definition of an execution of a system S. We start by defining a *state*, often denoted by s, which is a tuple $\langle t, pp, m\rangle$ where t is a real-numbered timestamp, pp is a *program point* and m is a map from program variables to values that have been assigned to them during the execution of the system S. Intuitively, a state is an instantaneous checkpoint in the execution of a program. For a state s, we write $s(\mathsf{x})$ as shorthand for $m(\mathsf{x})$, for m the map in s. To talk about the time at which a state was attained, we denote by $\textsc{time}(s)$ the time t from the state $s = \langle t, pp, m\rangle$.

Specification
$QF \to Q : F \qquad Q \to \forall Var \in Pred \qquad Var \to tr \mid s$
$F \to QF \mid true \mid F \vee F \mid \neg F \mid Constr$
Constraints
$Cmp \to < \mid > \mid =$ $\qquad Constr \to Val\ Cmp\ Val$
Values
$Val \to Val_{Tr} \mid Val_{St} \mid Val_{multiple} \mid w \qquad Val_{St} \to St(\mathtt{x}) \mid f(Val_{St})$
$Val_{Tr} \to \textsc{duration}(Tr) \qquad Val_{multiple} \to \textsc{TimeBetween}(St, St)$
Predicates
$Pred \to Pred_{QT} \mid Pred_{QS} \qquad Pred_{QS} \to Pred_S \mid Pred_S.\mathtt{after}(Var)$
$Pred_S \to \mathtt{changes(x).during(proc)} \qquad Pred_{QT} \to Pred_T \mid Pred_T.\mathtt{after}(Var)$
$Pred_T \to \mathtt{calls(proc_1).during(proc_2)}$
States and Transitions
$St \to s \mid s.\textsc{next}(Pred_S) \mid tr.\textsc{next}(Pred_S) \mid \textsc{before}(Tr) \mid \textsc{after}(Tr)$
$Tr \to tr \mid s.\textsc{next}(Pred_T) \mid tr.\textsc{next}(Pred_T)$

Figure 2: The syntax of the *Source Code Logic*.

Since a state s contains a program point pp, we say that each state corresponds to the result of executing some statement stmt for which pPoint(stmt) = pp. Additionally, we denote by proc(pPoint(stmt)) the name of the procedure inside which the statement stmt is found that has program point pPoint(stmt).

We then say that an execution of a procedure $p \in P$, denoted by E, is a sequence of *states* $s_1, s_2, \ldots, s_n$ with $\textsc{time}(s_1) < \textsc{time}(s_2) < \cdots < \textsc{time}(s_n)$. We denote by exec($s$) the execution E that contains the state s. Further, a pair $\langle s_i, s_j \rangle$ of consecutive states (i.e., from the same execution) is called a *transition*. We usually denote a transition by tr. Given a transition $tr = \langle s_1, s_2 \rangle$, we denote by $\textsc{duration}(tr)$ the *duration* of tr, defined as $\textsc{time}(s_2) - \textsc{time}(s_1)$.

Collecting together multiple procedure executions, we say that a trace Π over a system $S = \langle P, \text{prog} \rangle$ is a tuple $\langle S, \{E_1, E_2, \ldots E_n\}, L \rangle$ where $\{E_1, E_2, \ldots E_n\}$ is a set of procedure executions, and $L : \{E_1, E_2, \ldots E_n\} \to P$ is a map that labels each procedure execution with the name of the relevant procedure. We assume that the timestamps of states across all E_i in a trace Π have a total order (i.e., we do not consider concurrency).

2.2 Syntax

We now give a syntax for SCSLc in Figure 2. This syntax makes use of a number of key terminal symbols, including: s, a variable representing a state; tr, a variable representing a transition; x, a program variable; proc, $\mathtt{proc}_1$, and $\mathtt{proc}_2$, names of procedures; w, a constant that is either a real number or a string; and f, a function symbol.

In addition, all non-terminal symbols are highlighted in blue, while all remaining symbols not highlighted are terminal. Alongside the restrictions on specifications that are encoded in the grammar, we assume that SCSLc specifications contain no free variables.

Each SCSLc specification is a *QF* term. This ensures that the specification starts with a quantifier. The inner part of the specification is then a Boolean combinations of subformulae, which may contain other quantifiers (in which case we have *QF* terms), or constraints (in which case we have *Constr* terms).

Each quantifier in a SCSLc specification consists of a *Var* term and a *Pred* term. A *Pred* term is used to identify states or transitions in a trace, which are then bound to the *Var* term (this mechanism is formally defined in §3).

$\Pi, \beta, \langle t, \text{pPoint(stmt)}, m \rangle \vdash \mathtt{changes(x).during(func)}$ iff
$\quad \text{isChange}_{\mathtt{x}}(\text{stmt}) = true$ and $\text{proc}(\text{pPoint(stmt)}) = \mathtt{func}$

$\Pi, \beta, \langle t, \text{pPoint(stmt)}, m \rangle \vdash \mathtt{changes(x).during(func).after}(Var)$ iff
$\quad \textsc{time}(q) > \textsc{time}(\beta(Var))$ and $\Pi, \beta, q \vdash \mathtt{changes(x).during(func)}$

$\Pi, \beta, \langle \langle t, \text{pPoint(stmt)}, m \rangle, \langle t', \text{pPoint(stmt}'), m' \rangle \rangle \vdash \mathtt{calls(f).during(func)}$ iff
$\quad \text{isCall}_{\mathtt{f}}(\text{stmt}') = true$ and $\text{proc}(\text{pPoint(stmt}')) = \mathtt{func}$

$\Pi, \beta, \langle \langle t, \text{pPoint(stmt)}, m \rangle, \langle t', \text{pPoint(stmt}'), m' \rangle \rangle \vdash$
$\quad \mathtt{calls(f).during(func).after}(Var)$ iff
$\quad\quad \textsc{time}(\langle \langle t, \text{pPoint(stmt)}, m \rangle, \langle t', \text{pPoint(stmt}'), m' \rangle \rangle) >$
$\quad\quad\quad \textsc{time}(\beta(Var))$ and $\Pi, \beta, tr \vdash \mathtt{calls(f).during(func)}$

Figure 3: The binding relation for SCSLc.

$\text{evalEvent}(\Pi, \beta, s) = \beta(s)$ if $s \in \text{dom}(\beta)$, null otherwise
$\text{evalEvent}(\Pi, \beta, tr) = \beta(tr)$ if $tr \in \text{dom}(\beta)$, null otherwise
$\text{evalEvent}(\Pi, \beta, \textsc{before}(Tr)) = s_1$ if $\text{evalEvent}(\Pi, \beta, Tr)) = \langle s_1, s_2 \rangle$, null otherw.
$\text{evalEvent}(\Pi, \beta, \textsc{after}(Tr)) = s_2$ if $\text{evalEvent}(\Pi, \beta, Tr)) = \langle s_1, s_2 \rangle$, null otherw.
$\text{evalEvent}(\Pi, \beta, s.\textsc{next}(\mathtt{calls(f).during(p)})) = tr$ if there is a tr such that:

$$\left(\begin{array}{c} \textsc{time}(tr) > \textsc{time}(\text{evalEvent}(\Pi, \beta, s)) \text{ and } \Pi, tr \vdash \mathtt{calls(f).during(p)} \\ \text{and there is no } tr' \text{ such that:} \\ \textsc{time}(\text{evalEvent}(\Pi, \beta, s)) < \textsc{time}(tr') < \textsc{time}(tr) \\ \text{and } \Pi, tr' \vdash \mathtt{calls(f).during(p)} \end{array} \right)$$

null otherwise

Figure 4: The evalEvent **function for SCSLc.**

Constr terms are used to express constraints on values extracted from traces, which are described by *Val* terms.

2.2.1 Examples. One can capture the property that "every call of the function commit by the procedure write should take less than 1 second" by writing $\forall tr \in \mathtt{calls(commit).during(write)} : \textsc{duration}(tr) < 1$. One can also capture the property that "whenever the program variable query is assigned during the procedure write, then the procedure commit should eventually be called by the procedure write" by writing $\forall s \in \mathtt{changes(query).during(write)} : \exists tr \in \mathtt{calls(commit).during(write).after}(s) : true$.

Finally, one can capture the property that "whenever the procedure execute is called by the procedure write, then the procedure commit should be called by the procedure write within 2 seconds" by writing $\forall tr \in \mathtt{calls(execute).during(write)} : \textsc{TimeBetween}(\textsc{after}(tr), \textsc{before}(tr.\textsc{next}(\mathtt{calls(commit).during(write)}))) < 2$.

3 SEMANTICS

We now present our first contribution: a semantics for SCSLc. This semantics is distinguished from the existing semantics developed for SCSL by the traces that it considers. In particular, while the SCSL semantics considers traces to which additional states may be added (and has a truth domain to reflect this), the semantics that we develop here considers traces to which no additional states may be added (hence, we consider the truth domain $\mathbb{B}_2 = \{true, false\}$).

This key distinction comes from requirements given by our industry partners. Specifically, both of our partners' use cases involved systems whose running times were short, and would generate short traces. Hence, there was no need for intermediate truth values to be reached by a monitoring algorithm; it was possible to wait for the entire trace to be generated.

The semantics that we develop takes a trace Π and a SCSLc specification φ, and outputs a truth value in $\mathbb{B}_2 = \{true, false\}$. If

$$\text{evalValue}(\Pi, \beta, w) = w$$
$$\text{evalValue}(\Pi, \beta, St(\mathtt{x})) = \begin{cases} \text{evalEvent}(\Pi, \beta, St)(\mathtt{x}) & \text{evalEvent}(\Pi, \beta, St) \neq \text{null} \\ \text{null} & \text{otherwise} \end{cases}$$
$$\text{evalValue}(\Pi, \beta, f(Val_{St})) = \begin{cases} f(\text{evalEvent}(\Pi, \beta, Val_{St})) & \text{evalEvent}(\Pi, \beta, Val_{St}) \neq \text{null} \\ \text{null} & \text{otherwise} \end{cases}$$
$$\text{evalValue}(\Pi, \beta, \textsc{duration}(Tr)) = \begin{cases} \textsc{duration}(\text{evalEvent}(\Pi, \beta, Tr)) & \text{evalEvent}(\Pi, \beta, Tr) \neq \text{null} \\ \text{null} & \text{otherwise} \end{cases}$$
$$\text{evalValue}(\Pi, \beta, \textsc{TimeBetween}(St_1, St_2) = \begin{cases} \textsc{time}(\text{evalState}(\Pi, \beta, St_2)) - \textsc{time}(\text{evalState}(\Pi, \beta, St_1)) & \text{evalState}(\Pi, \beta, St_2) \neq \text{null and evalState}(\Pi, \beta, St_1) \neq \text{null} \\ \text{null} & \text{otherwise} \end{cases}$$

Figure 5: The evalValue **function for SCSLc.**

$\Pi, \beta \models \forall c \in Pred : F$ iff
for every state or transition e in Π with $\Pi, \beta, e \vdash Pred, \Pi, \beta \dagger [c \mapsto e] \models F$
$\Pi, \beta \models F_1 \vee F_2$ iff $\Pi, \beta \models F_1$ or $\Pi, \beta \models F_2$ $\quad$ $\Pi, \beta \models \neg F$ iff $\Pi, \beta \not\models F$
$\Pi, \beta \models \neg true$ iff $false$
$\Pi, \beta \models Val_1\ Cmp\ Val_2$ iff
$\text{evalValue}(\Pi, \beta, Val_1) \neq \text{null}$ and $\text{evalValue}(\Pi, \beta, Val_2) \neq \text{null}$ and
$\text{evalValue}(\Pi, \beta, Val_1)\ \text{evalOp}(Cmp)\ \text{evalValue}(\Pi, \beta, Val_2)$

Figure 6: The semantics relation for SCSLc.

our semantics gives *true*, we will say that Π *satisfies* φ, whereas if it gives *false*, we will say that Π *violates* φ.

We highlight that, in choosing a two-valued truth domain, any part of a specification for which we do not find information in our trace will be resolved to a default truth value.

We develop our semantics by considering an example specification $\varphi = \forall tr \in \mathsf{calls}(\mathtt{execute}).\mathsf{during}(\mathtt{write}) : \textsc{TimeBetween}(\textsc{after}(tr), \textsc{before}(tr.\textsc{next}(\mathsf{calls}(\mathtt{commit}).\mathsf{during}(\mathtt{write})))) < 2$, along with a trace Π. Our first step in deciding whether Π satisfies φ is to find transitions in Π that can be *identified* by the *Pred* term $\mathsf{calls}(\mathtt{execute}).\mathsf{during}(\mathtt{write})$. In this case, we care about transitions $\langle s, s' \rangle$ such that $s' = \langle t, \mathsf{pPoint}(\mathsf{stmt}), m \rangle$ with $\mathsf{isCall}_{\mathtt{execute}}(\mathsf{stmt}) = true$. Intuitively, since a state represents the result of executing a given statement, this condition means that a transition begins at the state generated before the call of `execute`, and ends immediately after that call.

We define the notion of a state or transition being *identified* by a *Pred* term by using the *binding relation*, given in Figure 3. The binding relation enables us to construct *bindings*, which are maps from *Var* terms in our specification to states or transitions found in our trace. We then say that a state or transition e is *identified* by a *Pred* term, given a trace Π and a binding β, if and only if $\Pi, \beta, e \vdash Pred$. We can then construct a binding β by setting $\beta(Var) = e$, given a Q term with a *Var* term and *Pred* term, if and only if e is *identified* by *Pred*.

We now consider how we can take a *Val* term and assign it a value from our trace for each binding β that we construct. In our running example, we have a single *Val* term: $\textsc{TimeBetween}(\textsc{after}(tr), \textsc{before}(tr.\textsc{next}(\mathsf{calls}(\mathtt{commit}).\mathsf{during}(\mathtt{write}))))$. This *Val* term is composed of *St* terms ($\textsc{after}(tr)$ and $\textsc{before}(tr.\textsc{next}(\mathsf{calls}(\mathtt{commit}).\mathsf{during}(\mathtt{write}))))$ for which we must find unique states, for each binding β. To this end, we first define the evalEvent function, which takes a trace Π, a binding β, and either a *St* or a *Tr* term, and gives the unique state or transition from Π corresponding to the input. The evalEvent function is defined in Figure 4.

Once we have used evalEvent to determine the unique state or transition that corresponds to a given trace/binding/term input, we must then evaluate any *Val* terms. For this, we define the evalValue function, which is given in Figure 5. This function takes a trace Π, a binding β, and a *Val* term and gives the unique value, extracted from Π, corresponding to the input.

As an example of applying the evalEvent and evalValue functions, let us take a trace Π from which we have derived a binding $\beta = [s \mapsto \langle 0.1, pp, [\mathtt{x} \mapsto 10] \rangle]$. Then, consider the *Val* term $s(\mathtt{x})$. To compute a value for this term, we use the $\text{evalValue}(\Pi, \beta, St(\mathtt{x}))$ case to compute $\text{evalValue}(\Pi, \beta, s(\mathtt{x}))$. Hence, we must then compute $\text{evalEvent}(\Pi, \beta, St) = \text{evalEvent}(\Pi, \beta, s) = \langle 0.1, pp, [\mathtt{x} \mapsto 10] \rangle$. We then have $\text{evalValue}(\Pi, \beta, s(\mathtt{x})) = \text{evalEvent}(\Pi, \beta, s)(\mathtt{x}) = \langle 0.1, pp, [\mathtt{x} \mapsto 10] \rangle(\mathtt{x}) = [\mathtt{x} \mapsto 10](\mathtt{x}) = 10$.

Now that we can construct bindings and evaluate *Val* terms, we introduce the final semantics relation in Figure 6. The semantics relation is defined recursively on the structure of SCSLc specifications, and makes use of the evalOp function, which takes a *Cmp* term and interprets it as a binary relation. For example, $\text{evalOp}(<)$ is the binary relation $<$, so we can write $n\ \text{evalOp}(<)\ m$ for $n, m \in \mathbb{R}$.

We say that a trace Π and a binding β satisfy a SCSLc specification φ if and only if $\Pi, \beta \models \varphi$. If $\beta = [\,]$, then we write $\Pi \models \varphi$ as shorthand for $\Pi, [\,] \models \varphi$, and say that Π *satisfies* φ.

4 INSTRUMENTATION AND MONITORING

With SCSLc introduced in §2, we now turn our attention to the monitoring problem for SCSLc. Specifically, given a system S, a trace Π over S, and a SCSLc specification φ, our goal is to develop a procedure for deciding whether $\Pi \models \varphi$.

4.1 Instrumentation

In this work, instrumentation is the process of deciding which program points in a system S may generate states and transitions in a trace that are useful for individual parts of a specification. We now highlight the steps that one must take to accomplish this in the SCSLc setting.

4.1.1 Pred term lookup. Consider the problem of determining which states and transitions are identified by each *Pred* term in a specification. A naive approach to this could be iterating over every *Pred* term, for each state and transition in the trace. However, this would be inefficient for long traces and specifications with multiple *Pred* terms. There could also be a lot of wasted work, especially if *Pred* terms only identify a fraction of the states and transitions found in the trace. In this case, an opportunity for optimisation is to remove the lookup needed to find the *Pred* terms that identify a given state or transition.

4.1.2 St and Tr term lookup. Consider the problem of determining, for each state or transition in a trace, which *St* or *Tr* terms require it. A naive approach would involve iterating over every binding β derived from Π, and then every *St* and *Tr* term *Evt* in the specification, and computing $\text{evalEvent}(\Pi, \beta, Evt)$. If the state or transition being processed was equal to $\text{evalEvent}(\Pi, \beta, Evt)$, then we would know which term required the state or transition that

we were processing. In this case, an opportunity for optimisation is to remove the lookup needed to find the *St* or *Tr* terms that require a given state or transition.

Both opportunities for optimisation are addressed using static analysis of the system S before a trace is generated and subsequently processed. We describe our approach for optimising *Pred* term lookup in §4.1.3, and our approach for optimising *St* and *Tr* term lookup in §4.1.4.

4.1.3 Choosing Relevant Program Points for Quantifiers. We begin by defining a function for determining which states and transitions in a trace are identified by each *Pred* term in a specification. The function is called pPointsFromQuantifier, takes a Q term as input, and gives a set of program points as output. The goal of the function is to compute the set of program points whose statements could generate a state (or transition) in a trace that could be identified by a *Pred* term. For example, if we have a specification that contains the term calls(f).during(g), then we should look at the code for the procedure g for statements in code that involve a call of the procedure f. Based on this intuition, the function is defined as QTermPPoints(qTerm) = {pPoint(stmt) : stmt ∈ stmtsUsingSym(symbol, procedure) where ⟨symbol, procedure⟩ = getParams(qTerm)} While pPoint is the function defined in §2.2, stmtsUsingSym is a function that takes as input a symbol (such as a program variable or function name) and a procedure name and gives as output a set of program statements, from the relevant procedure, in which that symbol is used. Further, getParams is a function that takes as input a Q term, and gives as output a pair ⟨symbol, procedure⟩. For example, getParams(∀s ∈ changes(x). during(p)) = ⟨x, p⟩.

Once QTermPPoints is computed, we then compute its inverse, which we call qTerms. This function takes as input a program point pp and gives the set of all Q terms qTerm for which pp ∈ QTermPPoints(qTerm). More formally, we have qTerms(pp) = { qTerm : pp ∈ QTermPPoints(qTerm)}.

4.1.4 Choosing Relevant Program Points for States and Transitions. We now give a function for determining, for some *St* or *Tr* term, which program points could generate states or transitions required by that term. This function, evtTermPPoints, takes a *St* or *Tr* term as input, and gives a set of program points as output. The goal of the function is to compute the set of statements (again, via their program points), in the source code of the system S, that could generate a state in a trace that could be required by a *Val* or *Tr* term in our specification. For example, if we have a specification that contains the term s(x) = 10, then we would be interested in statements that give a value to the program variable x.

Based on this intuition, the function is defined as evtTermPPoints(evtTerm) = {pPoint(stmt) : stmt ∈ EvtTermStmts(evtTerm)}. Here, the EvtTermStmts function takes a *St* or *Tr* term and computes a set of program statements that would generate a state or transition relevant to that term.

Similarly to above, we also introduce an inverse of evtTermPPoints, which we call evtTerms. Similarly, this function takes as input a program point pp and gives the set of all *St* or *Tr* terms evtTerm such that pp ∈ evtTermPPoints(evtTerm). More formally, we have evtTerms(pp) = {evtTerm : pp ∈ evtTermPPoints(evtTerm)}.

4.2 Deciding whether $\Pi \models \varphi$

We now develop a procedure, for determining whether a given trace Π satisfies a given SCSLc specification φ, which reflects the SCSLc semantics given in §3. Before giving the complete procedure, we describe the individual steps that are required, before combining the steps into the final procedure at the end of the section.

The approach taken by our procedure is to process a given trace Π by iterating over its states using the total order induced by their timestamps (§2.1). During this iteration, we maintain a structure called a *monitoring tree* (§4.2.1), which is used to store bindings and values extracted from the trace. For each state that we process, our monitoring procedure then involves performing different transformations on the monitoring tree, depending on the state. In particular, for each state we ask two questions:

- Is s (or a transition containing it) identified by a *Pred* term in the specification?
  In this case, we update our monitoring tree to represent the potentially new or extended bindings that could result from processing this state. We describe this procedure in §4.2.2.
- Is s (or a transition containing it) required by a *St* or *Tr* term in the specification?
  In this case, we update our monitoring tree to encode the fact that some *St* or *Tr* term requires s. We describe this procedure in §4.2.3.

Once all states have been processed, our monitoring procedure then traverses the final monitoring tree in order to compute a final truth value. We describe our procedure for this in §4.2.4.

4.2.1 Monitoring trees. A *monitoring tree* $M(\Pi, \varphi)$ for a trace Π and SCSLc specification φ is a tree $\langle N, E\rangle$. Here, N is a set of nodes, which are triples $\langle \phi, \beta, v\rangle$ for ϕ a *QF*, *F*, *Constr*, *Val*, *St*, *Tr*, or *Var* term; β a binding; and v a value computing during the monitoring process. In addition, $E \subset N \times N$ is a set of edges, where there is an edge $\langle n, n'\rangle$ from a node $n = \langle \phi, \beta, v\rangle$ to a node $n' = \langle \phi', \beta', v'\rangle$ if and only if 1) ϕ' is a direct subformula of ϕ (for example, if ϕ is a *QF* term, then ϕ is a *F* term), and 2) dom(β) ⊂ dom(β') and for every $v \in$ dom(β), we have $\beta(v) = \beta'(v)$.

4.2.2 Constructing bindings. As the number of bindings derived from a trace increases, it becomes crucial to store bindings effectively. In particular, consider the specification ∀v ∈ changes(x). during(p) : v(x) < 10 → ∃v' ∈ calls(f).during(p).after(v) : DURATION(v') < 1, where $a \rightarrow b$ is shorthand for the Boolean combination $\neg a \vee b$.

Suppose further that we process a state $s = \langle t, pp, m\rangle$ such that $\text{isChange}_{\text{x}}(pp) = true$. Hence, this state would be identified by the *Pred* term changes(x).during(p). Then, suppose that we process a state $s' = \langle t', pp', m'\rangle$ such that $\text{isCall}_{\text{f}}(pp') = true$. The transition $tr = \langle s'', s'\rangle$, for s'' the state immediately before s' in the trace (in the same execution as s'), would be identified by the *Pred* term calls(f).during(p).after(v). After having processed both states, we would have constructed a binding $\beta = [v \mapsto s, v' \mapsto tr]$, and we would have enough information to compute both evalValue(Π, β, v(x)) and evalValue(Π, β, DURATION(v')).

Now, consider the case in which a second transition tr', occurring after the transition tr, is processed that is also identified

Algorithm 1: An algorithm for the function updateQs.

```
1  Input: A state s = ⟨t, pp, m⟩, a monitoring tree M, and a map
     execToPrevState
   Result: A modified monitoring tree M′
2  for qTerm ∈ qTerms(pp) do
3      if getPredicate(qTerm) of form changes(x).during(p) then
4          M′ ← addTree(M, s);
5      else
6          prevState ← execToPrevState(containingExecution);
7          M′ ← addTree(M, ⟨prevState, s⟩);
8  end
9  return M′;
```

Algorithm 2: An algorithm for the function updateEvents.

```
1  Input: A state s = ⟨t, pp, m⟩, a monitoring tree M, a map execToPrevState,
     and the current state's containing execution contExec
   Result: A modified monitoring tree M′
2  for evtTerm ∈ evtTerms(pp) do
3      if isStateTerm(evtTerm) then
4          M′ ← setEvent(M, s);
5      else
6          prevState ← execToPrevState(contExec);
7          M′ ← setEvent(M, ⟨prevState, s⟩);
8  end
9  return M′, execToPrevState;
```

by calls(f).during(p).after(v). We would then have a second binding, $\beta' = [v \mapsto s, v' \mapsto tr']$, and we would have to compute evalValue(Π, β', DURATION(v')). However, we would not need to recompute evalValue($\Pi, \beta, v(\mathsf{x})$), because this relies on the state associated with the variable v, which is shared by the two bindings.

Such repeated work is avoided by using monitoring trees (§4.2.1) to ensure that values whose computation is the same for multiple bindings are only computed once, for the common part of those bindings. Specifically, suppose that a state $s = \langle t, pp, m\rangle$ is identified by a *Pred* term P. Assuming a monitoring tree $M(\Pi, \phi)$, we proceed as follows:

(1) Find each node $n = \langle QF, \beta, v\rangle$ in N, where QF is a QF term of the form $\forall v \in P : \phi$, for v a *Var* term and ϕ a F term (i.e., QF contains P).
(2) For each node $n = \langle QF, \beta, v\rangle$, construct a monitoring tree $M'(\Pi, \phi) = \langle N', E'\rangle$ representing the F term ϕ, whose nodes all have the binding $\beta \dagger [v \mapsto \langle t, pp, m\rangle]$. If we are considering instead the transition $tr = \langle s, s'\rangle$ for the state s' immediately after s in Π, then we create nodes with the binding $\beta \dagger [v \mapsto tr]$.
(3) For each node $n = \langle QF, \beta, v\rangle$, attach the tree $M'(\Pi, \phi) = \langle N', E'\rangle$ to $M(\Pi, \varphi) = \langle N, E\rangle$ by setting $N = N \cup N'$ and $E = E \cup E' \cup \{\langle n, n_r\rangle\}$ where n_r is the root node of M'.

We represent this transformation with the function addTree, which takes as input a monitoring tree $M(\Pi, \varphi)$ and a state s and outputs a monitoring tree $M^*(\Pi, \varphi)$, obtained by modifying $M(\Pi, \varphi)$ with respect to the procedure described above.

We then wrap this function in another function, updateQs, which takes as input a monitoring tree $M(\Pi, \varphi)$, a state s, and a map from executions to states, and outputs a monitoring tree $M^*(\Pi, \varphi)$. Specifically, this function represents applying addTree for each *Pred* term in qTerms(pp). We given an algorithm for this function in Algorithm 1. The algorithm makes use of the functions getPredicate, which extracts the *Pred* term from a given Q term. Further, we assume that the map qTerms is globally accessible, and computing during our instrumentation process.

4.2.3 Storing states or transitions in monitoring trees. Suppose now that a state $s = \langle t, pp, m\rangle$ (or a transition of which it is a part) is required by a *St* (or *Tr*) term *Evt*. In this case, we find each node $n = \langle Evt, \beta, v\rangle$ in N, and set $v = s$. We represent this transformation with the function setEvent, which, similarly to addTree, takes as input a monitoring tree $M(\Pi, \varphi)$ and a state s and outputs a monitoring tree $M^*(\Pi, \varphi)$ obtained by modifying $M(\Pi, \varphi)$ as described above.

$$
\begin{aligned}
&\text{resolve}(\langle true, \beta, v\rangle) = true \qquad \text{resolve}(\langle \forall Var \in Pred : F, \beta, v\rangle) = \textstyle\bigwedge_{\langle F, \beta', v'\rangle} v' \\
&\text{resolve}(\langle F_1 \vee F_2, \beta, v\rangle) = \text{resolve}(\langle F_1, \beta, v_1\rangle) \vee \text{resolve}(\langle F_2, \beta, v_2\rangle) \\
&\text{resolve}(\langle \neg F, \beta, v\rangle) = \neg \text{resolve}(\langle F, \beta, v\rangle) \\
&\text{resolve}(\langle Val_1\ Cmp\ Val_2, \beta, v\rangle) = \\
&\quad \begin{cases} \text{getValue}(\langle Val_1, \beta, v\rangle)\ \text{evalOp}(Cmp) & \text{getValue}(\langle Val_1, \beta, v\rangle) \neq \text{null} \wedge \\ \quad \text{getValue}(\langle Val_2, \beta, v\rangle) & \quad \text{getValue}(\langle Val_2, \beta, v\rangle) \neq \text{null} \\ false & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 7: The resolve **function for monitoring trees.**

Similarly to the quantifier case, we then wrap the setEvent function with the function updateEvents. This function takes as input a monitoring tree $M(\Pi, \varphi)$, a state s, a map from executions to states, and the execution containing the current state being considered by the monitoring algorithm; and outputs a monitoring tree $M^*(\Pi, \varphi)$. The function applies setEvent for each *St* or *Tr* term in evtTerms(pp). The algorithm for this function is given in Algorithm 2. The algorithm makes use of isStateTerm, which is a predicate that tells us whether a given term is a *St* term.

4.2.4 Resolving monitoring trees. Suppose that we have the final monitoring tree $M(\Pi, \varphi)$. Then we decide whether $\Pi \models \varphi$ by recursing on the monitoring tree and computing a final truth value. We encode this operation in a function called resolve, which takes as input a monitoring tree node and gives as output a truth value. This function is defined in Figure 7, and makes use of the getValue function, which takes a node $\langle Val, \beta, v\rangle$ and computes its value by inspecting its children. For example, getValue($\langle q(\mathsf{x}), \beta, v\rangle$) will inspect the child node $\langle q, \beta, s\rangle$ of $\langle q(\mathsf{x}), \beta, v\rangle$, to which some state s has been assigned while processing the trace, and derive the value $s(\mathsf{x})$. We highlight that this may be null if no information could be extracted from the trace. In particular, if the getValue function gives null, then nodes containing *Constr* terms are evaluated to *false*, matching the policy described by the semantics (§3).

4.2.5 A monitoring algorithm. We now use the machinery that we have developed to present Algorithm 3.

The algorithm begins by initialising a monitoring tree M, and initialising a map execToPrevState. This map allows the algorithm to keep track of the previous state that was processed in a given execution in Π (since there may be multiple). Since the input trace consists only of states, to check transitions we need to pair the previous state with the current one. However, when multiple procedure executions are involved, we must ensure that the states that we use to form a transition are from the same execution. The for-loop on line 4 iterates over the states in the trace Π in order of

Algorithm 3: An algorithm for deciding whether $\Pi \models \varphi$.

Input: A trace Π and a VFL specification φ
Result: *true* or *false*, indicating whether $\Pi \models \varphi$
execToPrevState : map $\leftarrow$ [ ];
M : monitoring tree $\leftarrow \langle \{\langle \varphi, [\,], \text{null} \rangle\}, \emptyset \rangle$;
for *state* $s = \langle t, pp, m \rangle$ *in* Π, *ordered by time* **do**
  contExec $\leftarrow$ exec(s);
  if $pp \in$ dom(qTerms) **then**
    $M \leftarrow$ updateQs(s, M, execToPrevState);
  end
  if $pp \in$ dom(evtTerms) **then**
    M, execToPrevState $\leftarrow$ updateEvents(s, M, execToPrevState, contExec);
  end
  execToPrevState(contExec) $\leftarrow s$;
end
return resolve(n_r) for n_r the root node of M;

time. Inside the loop, two cases are considered: that the state (or transition) being processed is identified by a *Pred* term, or that the state (or transition) being processed is required by a *St* or *Tr* term. Once the trace has been processed, the monitoring tree is resolved and the algorithm returns a truth value.

5 EVALUATION

In this section, we evaluate the scalability of our monitoring algorithm (and our prototype implementation) by answering the following research question: *how does our monitoring procedure scale (in terms of the total time taken, and the memory consumed) as the trace being checked increases in length?*

5.1 Implementation

We answer our research question using our prototype framework, which provides machinery for three key processes: instrumenting the source code of the system under scrutiny (assumed to be written in Python), generating a trace by executing the system under scrutiny, and checking that trace for satisfaction of a SCSLc specification.

The instrumentation process makes use of a significant optimisation: traces are generated by code that is inserted at program points identified by the procedure described in §4.1. This means that traces contain only the states that are actually needed by a specification.

5.2 Scalability

We now describe the experiments that we carried out to answer our research question.

5.2.1 Test subject. Our test subject, *AeroBenchVVPython* [1], is an open-source, Python-based aircraft simulator that supports multiple scenarios. While not being the most complex simulator that one can work with, it was an appropriate choice for us because 1) it is written in Python, so immediately useful to the initial version of our framework, and 2) it could be modified to generate traces of any length that we needed.

5.2.2 Specifications. The SCSLc specifications that we use are representative of categories of specifications that exercise different parts of our monitoring algorithm. Hence, in considering these specifications, we demonstrate the performance of our monitoring algorithm in various circumstances. The specifications that we consider are given in Table 1.

Table 1: Properties expressed in English with their corresponding SCSLc specifications.

ID	Natural language	SCSLc
1	Whenever the 1) the program variable n is equal to True, and 2) the program variable t is equal to True, then the next time the program variable m is changed during the procedure adv, it should be set to standby.	$\forall q \in \text{changes(n).during(adv)} :$ $(q(\text{n}) = \text{True} \wedge q(\text{t}) = \text{True}) \rightarrow$ $q.\text{NEXT}(\text{changes(m).during(adv)}))$ $(\text{m}) = \text{standby}$
2	The time taken to reach a change of the program variable rv, from a change of the program variable pm, in the procedure adv, should be less than one second.	$\forall q \in \text{changes(pm).during(adv)} :$ $\text{TIMEBETWEEN}(q, q.\text{NEXT}($ $\text{changes(rv).during(adv))}) < 1$
3	For every change of the variable pm, if the new value of pm is pull, then eventually there should be a call of the function nh that takes less than 2 seconds.	$\forall q \in \text{changes(pm).during(adv)} :$ $q(\text{pm}) = \text{pull} \rightarrow$ $\exists c \in \text{calls(nh).during(adv).after}(c) :$ $\text{DURATION}(c) < 2$
4	All calls of the function nh should take less than 2 seconds.	$\forall c \in \text{calls(nh).during(adv)} :$ $\text{DURATION}(c) < 2$

Specification 1 tests our monitoring procedure's ability to handle multiple atomic constraints, combined using Boolean operators. Specification 2 tests our procedure's ability to deal with the TIMEBETWEEN operator. Specification 3 allows us to evaluate the performance of our monitoring procedure when dealing with an existential quantifier. Finally, Specification 4 allows us to see how well our monitoring procedure handles a simple duration constraint. While this final specification is simple, our instrumentation approach will yield a trace containing only measurements relevant to the duration constraint. Hence, for traces containing up to one million events, we can see how well our monitoring tree evaluation procedure scales.

5.2.3 Traces. For evaluating our monitoring algorithm, we instrumented and executed the test subject to generate traces with lengths ranging from $\approx$ 200 000 to $\approx$ 1 000 000 in steps of $\approx$ 200 000. We give approximate numbers for trace lengths because the number of states was not controllable by a parameter given to the simulator. Instead, the desired number of states had to be generated by varying the time step used during the simulation, which led to an approximate result.

5.2.4 Settings. We ran all of our experiments on a machine running Ubuntu 20.04.4 LTS, with 6 GB of RAM and four CPUs of type Intel(R) Xeon(R) Gold 6146 at 3.20 GHz. Timing was measured using the `default_timer` method, provided by the `timeit` Python library. This method chooses the timing method with the highest resolution

available on the machine on which the code is running. Memory consumption was measured by recursively computing the memory required to store the monitoring tree (i.e., the number of bytes taken up by each object in the tree).

We evaluated the scalability of our monitoring algorithm by:

- Measuring the total time taken to iterate over all events in the trace being considered. This is reasonable because event processing is done in sequence.
- Computing the memory consumed by the monitoring tree once all states in a trace have been processed. This measurement is reasonable because the monitoring tree is never shrunk at runtime.

For each specification, we instrumented the test subject and executed it to generate a trace for each number of states as defined in §5.2.3 (200 000 up to 1 000 000 in steps of 200 000). Monitoring on a given trace/specification combination was then executed three times. The tables presented show the maximum, minimum, and mean of the relevant measurements taken across the three executions.

Finally, we remark that we do not compare our results with baseline monitoring procedures. This is justified by considering the novelty of the specification language that we have used. In particular, since SCSLc is a syntactic fragment of SCSL, we first highlight our previous in-depth comparison of SCSL with existing specification languages in [15]. This comparison demonstrated the additional effort that engineers would have to go to when using other languages, in order to capture properties for which SCSL (and therefore SCSLc) was designed. Now, one could argue that we could have compared our monitoring algorithm to those for other languages, in which one could capture the same properties. However, such a comparison would aim at determining the cases in which it may be better to use existing approaches, or our own. Since the properties of interest in this work are captured more easily in SCSLc, we argue that the additional effort required to use other languages removes the need to consider any performance differences in monitoring algorithms.

5.2.5 Results. We now discuss Table 2. Discussing the results per specification, we have the following:

Specification 1. This specification has a single quantifier. For each state captured by the quantifier, monitoring is a case of determining the other states that are relevant to the atomic constraints in the specification (and adding a constant number of nodes to the tree for each state captured by the quantifier). With this in mind, the time taken and memory consumed by monitoring should scale linearly with the number of states, which is confirmed by Table 2.

Specification 2. This specification also has a universal quantifier as its root and, while it uses a different operator than specification 1, monitoring for it still involves processing a fixed number of states from the trace for every state captured by the quantifier. Hence, we expect the same linear scaling for both time and memory, which is confirmed by the relevant rows in Table 2.

Specification 3. This specification uses nested quantifiers; the existential quantifier meaning that, for each state satisfying the universal quantifier, there may be a variable number of other states to consider. Table 2 shows linear scaling for both time and memory, meaning that all existential quantifiers were satisfied quickly. If existential quantifiers are not satisfied quickly (especially when nested inside other quantifiers), this can lead to simultaneous checking of many quantifiers, and running times worse-than-linear in the number of states being processed.

Table 2: Memory consumed and time taken by our framework to check traces generated by both test subjects.

Spec	Events (10^5)	Memory (10^7 bytes) Min	Avg	Max	Time (s) Min	Avg	Max
1	2.00	27.89	27.89	27.89	9.1	9.93	10.99
	4.00	55.78	55.78	55.78	17.98	18.93	20.03
	6.00	83.67	83.67	83.67	26.95	27.91	29.79
	8.00	111.56	111.56	111.56	33.7	35.45	36.83
	10.00	139.45	139.45	139.45	43.79	44.81	46.02
2	2.11	3.54	3.54	3.54	4.55	4.86	5.26
	4.21	7.08	7.08	7.08	8.53	9.12	9.48
	6.32	10.61	10.61	10.61	14.11	14.35	14.63
	7.90	13.27	13.27	13.27	17.09	17.57	18.26
	10.00	16.81	16.81	16.81	20.89	21.85	22.72
3	2.37	21.34	21.34	21.34	6.25	10.1	14.49
	4.15	37.35	37.35	37.35	16.47	16.71	17.01
	5.92	53.36	53.36	53.36	22.4	23.48	24.54
	8.29	74.70	74.70	74.70	32.78	33.09	33.44
	10.66	96.04	96.04	96.04	41.06	42.26	44.07
4	1.93	2.32	2.32	2.32	3.23	4.16	5.78
	3.86	4.64	4.64	4.64	6.44	6.79	7.03
	6.28	7.53	7.53	7.53	10.41	10.95	11.41
	8.21	9.85	9.85	9.85	13.35	13.91	14.39
	10.14	12.17	12.17	12.17	16.77	17.06	17.21

Specification 4. For this specification, since there is a constant amount of information to be processed for each state satisfying the quantifier, we expect linear scaling in both time and memory. As expected, Table 2 shows that our monitoring procedure scales linearly in terms of both time and memory, since a constant number of checks are performed for each state identified by the quantifier.

Ultimately, Table 2 shows that our implementation scales linearly with the length of the trace being checked, with the actual time taken in each case always within reasonable bounds. The maximum time taken across all specifications, for ≈1 000 000 events, is ≈46.02 s, when checking traces for specification 1. Further, the maximum memory consumed by our framework for traces of length ≈1 000 000 was ≈1.39 GB, for specification 1.

We conclude that our offline trace checking implementation scales linearly with the number of events being processed, both in terms of total time taken, and memory consumed, for the representative specifications that we considered.

5.3 Threats to validity

In terms of external validity, we argue that our approach (and prototype framework) is generalisable by highlighting that, not only has it been applied to an open source test subject, but it has also been used out-of-the-box by two industry partners (a process we describe below in § 6).

Another threat to validity is the scale of the system under scrutiny. We argue that this does not have much bearing on the results that we have shown because specifications ensure that only a small part of system behaviour is analysed, so the size of the system outside what is covered by that specification is not relevant. However, a

larger system could give rise to different properties that would challenge either the expressive power of SCSLc, or the efficiency of our monitoring algorithm.

A further threat to validity is found in the lack of a comparison with a baseline monitoring implementation. We justified the lack of a baseline in §5.2.4, but include the fact here for completeness.

Finally, we remark that we have assumed correctness of our implementation throughout our experiments. However, there can of course be undiscovered bugs in our implementation. While we have tested our framework thoroughly, on various programs and with a variety of specifications, and industry partners have applied our framework to their own systems, there may still be bugs.

5.4 Data Availability

We make available on Figshare our specifications, traces, and experimental results [11] as well as our implementation [12].

6 APPLICABILITY IN INDUSTRY

In this section, we report on the applicability of our prototype framework to two industry systems. We show that our framework can be used to check properties of interest to engineers (taking acceptable amounts of time and using acceptable amounts of memory), and can be integrated into existing automated testing pipelines. We conclude the section discussing the lessons learned from this experience.

6.1 The Avionics Partner

GMV, our partner in the avionics sector with a world-wide presence, needed to analyse an actively developed Python-based project that retrieved data from a database and performed post-processing.

6.1.1 Integration. A key requirement from GMV was that, once a set of specifications had been defined, it must be possible to write unit tests whose oracles were provided by our framework. Specifically, it should be possible to write assertions within unit tests that check the truth value obtained when monitoring for satisfaction of specifications, during the unit test. Ultimately, the integration of our framework into GMV's use case enabled specification-based testing. In practice, our framework provides an API that can be used when writing unit tests to enable any code exercised by unit tests to be subject to the relevant specifications.

To derive an initial set of specifications, we investigated the most recent version of the project's code with GMV's engineers and derived 6 specifications. This process took place during an 8 hour on-site meeting with GMV, and involved us giving an introduction to the specification language, and discussing with GMV's engineers the kinds of properties that were of interest that could be captured using our language (while looking through the source code).

6.1.2 Initial Results. To obtain experimental results, GMV ran their project (with our framework attached) on a machine running Ubuntu 20.04.6 LTS 64-bit, with 4 GB of RAM and 2 CPUs of type Intel(R) Core(TM) i7-10610U at 1.80 GHz

Since the specifications focused on individual lines of code that were executed a small number of times at runtime, the trace generated for each specification when running GMV's code was short (containing no more than 10 states).

GMV worked with traces containing an average of 4.83 states (ranging from 0 states, generated whenever the execution of code took a path on which there was no instrumentation, up to 30 states). Across all traces, instrumentation had a significant impact. Across all specifications, our monitoring algorithm took an average of 0.001 s to process the trace generated for each specification (ranging from 0 s, when no states were generated by instrumentation, up to 0.019 s). Further, our algorithm needed an average of 1.53 kB (ranging from 348 B, which is the space required by only the root node of a monitoring tree to which no nodes were added, up to 9.23 kB).

6.2 The Acoustics Partner

Unparallel Innovations, our partner in the acoustics sector which is an SME, had a Python-based project that was responsible for updating the embedded software deployed to some of their hardware.

6.2.1 Integration. Unparallel Innovations were exposed to our work via a project presentation, and subsequently applied our framework independently using our documentation (this included writing an initial set of 5 specifications and running their project with monitoring). Similarly to GMV's use of our framework, Unparallel Innovations also performed specification-based testing. In contrast, Unparallel Innovations chose to use our framework for integration tests. For this, we provided an API with which Unparallel Innovations were able to write simple Python scripts to test their system with different inputs, under different specifications.

Since Unparallel Innovations applied our framework independently, they did report a property that they could not capture; one that constrained the number of times a specific function was called. In order to capture such a constraint in our language, one would need either 1) a way to count the number of states or transitions captured by a quantifier, or 2) an atomic constraint designed specifically for counting states or transitions. Since the language did not provide either of these features, and this property was not a high priority, we agreed with the partner to leave it out.

6.2.2 Initial Results. Since we did not have the ability to run the project ourselves (it required specific hardware), we modified our framework to report additional values, and deployed that version for Unparallel Innovations to use. They then ran their project on two machines (to test how their code interacted with different hardware): a BeagleBone Green Wireless running AM3358 Debian 10.3, and a MacBook Pro 16-inch, 2023, running macOS Ventura 13.5.1, with 32 GB of RAM and an Apple M2 Pro processor. Similarly to the GMV case, they worked with traces containing an average of 229.9375 states (ranging from 3 states up to 727 states), meaning once again that instrumentation had a significant impact. Across all specifications, our monitoring algorithm took an average of 0.099 s to process the trace generated for each specification (ranging from 2.875×10^{-5} s up to 0.558 s). Further, our algorithm needed an average of 32.21 kB (ranging from 306 B up to 286.52 kB).

6.3 Lessons Learned and Discussion

When first working with GMV, we provided documentation for SCSLc, and example specifications. We demonstrated that SCSLc provides a high degree of control, while still giving instrumentation

and monitoring for free. However, this level of control could lead to specifications that are difficult to read and write. To remedy the situation, we worked with GMV *to develop a high-level DSL* for capturing common patterns seen in SCSLc specifications. For more complex properties, the full SCSLc syntax was still available. Our DSL was *validated* when Unparallel Innovations were able to independently apply our framework, while using only the DSL that was developed with GMV. The first lesson learned was that *providing a high-level, pattern-based DSL for specifying properties of interest can greatly facilitate the adoption of RV approaches.* This is aligned with existing reports on pattern-based specifications [9, 10].

Once our framework was integrated into automated testing pipelines, we saw that additional work was needed to enable engineers to analyse the results of specification-based testing. For example, our framework stores enough information to identify the line of code that generated a violating measurement at runtime; visualising this has already proven useful in previous work [16]. The second lesson learned was that *the verdicts of an RV approach should be easily accessible to engineers and facilitate diagnosis*, prompting directions for future work (see also § 8).

Finally, one could argue that, with such short traces, one could monitor for specifications manually. However, we remark that this work aims at providing a framework for use in automated testing pipelines, in which case manually checking even small traces for satisfaction of specifications would become tedious and possibly infeasible (especially across multiple executions of a pipeline).

7 RELATED WORK

Instrumentation. An example of static analysis for instrumentation is found in [26], which proposes a finer grained version of Aspect-Oriented Programming (AOP). The work extends AOP with the ability to attach instrumentation to basic blocks and loop back edges. This work shares some similarity to ours in that it focuses on fine-grained instrumentation (although our instrumentation is at the line-of-code level).

In the direction of using instrumentation to optimise monitoring, Clara reduces automata used in monitoring by progressively proving program points to be safe via static analysis [5]. The reduced automata are shown to capture the same violations as non-reduced automata. While this is similar to our work in its efforts to optimise monitoring, our work does not attempt to decide whether parts of a specification can be checked statically.

Other work introduces a *sampling-based* approach to monitoring, which involves determining the longest period that a monitor can wait to sample a system's state (without losing information important to a specification) [6]. In contrast, our instrumentation approach looks for program points that may generate information relevant to a specification.

The Java-MaC tool for Java [24] has the policy of specifications being defined separately from the rules that indicate how runtime events should be mapped onto specifications. A key difference between this work and ours is that our specification language admits specifications that already contain enough information for instrumentation to take place, meaning that no separate definition is required.

Monitoring. Automata are commonplace when checking traces for satisfaction of specifications captured using temporal logics. Monitoring LTL and its variants lends itself well to the use of automata [4, 8, 21, 23, 25, 31] because automata can be synthesised that accept precisely the languages of traces that LTL specifications do. In contrast, our monitoring algorithm uses a *monitoring tree* (§4.2) to keep track of the values extracted from a trace.

Other work uses dynamic programming to compute the truth values of subformulae of specifications [22]. Another approach [18] uses dynamic programming to compute *robustness* (a quantitative verdict). While our work has not considered robustness, giving engineers additional understanding of monitoring results is something we are actively working towards.

In the online monitoring setting (checking a system's execution for satisfaction of a specification while the execution is ongoing), some approaches focus on *keeping up* with the events that are being generated by the system under scrutiny [2]. Our industry partners have not needed online monitoring, but our framework does support it (omitted because of space limitations).

Some authors have considered monitoring traces by *slicing* them into separate traces and monitoring the resulting set of traces in parallel [3, 28, 29]. Our traces are already sliced at the implementation level: only states that are relevant to a specification are actually written to a trace.

For automata-based specification formalisms such as Quantified Event Automata [27], the specification is actually executable, since it takes the form of an automaton that one can pass events from a trace through. In contrast, our work uses a monitoring tree that is maintained by a separate algorithm.

Finally, our monitoring algorithm is defined over traces that are assumed to be complete. For the cases where the information necessary to decide a truth value has not been found in the trace, our policy is to make the affected constraint *false*. Existing work considers different policies [14, 19].

8 CONCLUSION

In this paper, our first contribution was a new semantics for a source code fragment, SCSLc, of SCSL. Based on this new semantics, we introduced a monitoring procedure for checking traces to which no additional information can be added (for example, because of observing further information from a program's execution). We also presented an evaluation showing that our monitoring algorithm scales well for longer traces (containing up to one million states). Finally, we demonstrated the applicability of our framework by describing its use by two industry partners. Through our collaborations with the industry partners, we have seen that our monitoring algorithm can be applied to real systems, checking traces for satisfaction of properties of interest to engineers in a reasonable amount of time (greatly aided by our instrumentation approach).

As part of future work, we plan to extend our work on trace checking for SCSLc to support full SCSL. In addition, we plan to address the trace diagnostic problem [7, 20, 30] in the context of SCSLc (and later, full SCSL) specifications. We also plan to conduct a thorough evaluation of our work with multiple case studies.

REFERENCES

[1] Stanley Bak. 2023. The AeroBenchVVPython GitHub Repository. https://github.com/stanleybak/AeroBenchVVPython. Accessed: 2023-08-11.

[2] David Basin, Bhargav Nagaraja Bhatt, Srđan Krstić, and Dmitriy Traytel. 2019. Almost event-rate independent monitoring. *Formal Methods in System Design* 54, 3 (2019), 449–478. https://doi.org/10.1007/s10703-018-00328-3

[3] David Basin, Germano Caronni, Sarah Ereth, Matúš Harvan, Felix Klaedtke, and Heiko Mantel. 2016. Scalable Offline Monitoring of Temporal Specifications. *Form. Methods Syst. Des.* 49, 1–2 (oct 2016), 75–108. https://doi.org/10.1007/s10703-016-0242-y

[4] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* 20, 4, Article 14 (sep 2011), 64 pages. https://doi.org/10.1145/2000799.2000800

[5] Eric Bodden, Patrick Lam, and Laurie Hendren. 2010. Clara: A Framework for Partially Evaluating Finite-State Runtime Monitors Ahead of Time. In *Runtime Verification*, Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–197.

[6] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. 2011. Sampling-Based Runtime Verification. In *FM 2011: Formal Methods*, Michael Butler and Wolfram Schulte (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 88–102.

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

[8] Agnishom Chattopadhyay and Konstantinos Mamouras. 2020. A Verified Online Monitor for Metric Temporal Logic with Quantitative Semantics. In *Runtime Verification*. Springer International Publishing, Cham, 383–403.

[9] Christoph Czepa, Amirali Amiri, Evangelos Ntentos, and Uwe Zdun. 2019. Modeling compliance specifications in linear temporal logic, event processing language and property specification patterns: a controlled experiment on understandability. *Software and Systems Modeling* 18, 6 (2019), 3331–3371. https://doi.org/10.1007/s10270-019-00721-4

[10] C. Czepa and U. Zdun. 2018. On the Understandability of Temporal Properties Formalized in Linear Temporal Logic, Property Specification Patterns and Event Processing Language. *IEEE Trans. Softw. Eng.* 46 (2018), 1–13. doi: 10.1109/TSE.2018.2859926.

[11] Joshua Dawes and Domenico Bianculli. 2024. Checking Complex Source Code-level Constraints using Runtime Verification - FSE 2024 (Industry papers track) - Artefact. https://doi.org/10.6084/m9.figshare.25783266.v1

[12] Joshua Dawes and Alexander Vatov. 2024. SCSL-Trace-Checker - Source Code. https://doi.org/10.6084/m9.figshare.25784355.v1

[13] Joshua Heneage Dawes and Domenico Bianculli. 2021. Specifying Properties over Inter-procedural, Source Code Level Behaviour of Programs. In *Runtime Verification*, Lu Feng and Dana Fisman (Eds.). Springer International Publishing, Cham, 23–41.

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

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

[16] Joshua Heneage Dawes, Marta Han, Omar Javed, Giles Reger, Giovanni Franzoni, and Andreas Pfeiffer. 2020. Analysing the Performance of Python-Based Web Services with the VyPR Framework. In *Runtime Verification: 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6–9, 2020, Proceedings* (Los Angeles, CA, USA). Springer-Verlag, Berlin, Heidelberg, 67–86. https://doi.org/10.1007/978-3-030-60508-7_4

[17] Joshua Heneage Dawes and Giles Reger. 2019. Specification of Temporal Properties of Functions for Runtime Verification. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing* (Limassol, Cyprus) *(SAC '19)*. Association for Computing Machinery, New York, NY, USA, 2206–2214. https://doi.org/10.1145/3297280.3297497

[18] Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos. 2014. On-Line Monitoring for Temporal Logic Robustness. *CoRR* abs/1408.0045 (2014). arXiv:1408.0045 http://arxiv.org/abs/1408.0045

[19] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. 2003. Reasoning with Temporal Logic on Truncated Paths. In *Computer Aided Verification*, Warren A. Hunt and Fabio Somenzi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–39.

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

[21] M.C.W. Geilen. 2001. On the Construction of Monitors for Temporal Logic Properties. *Electronic Notes in Theoretical Computer Science* 55, 2 (2001), 181–199. https://doi.org/10.1016/S1571-0661(04)00252-X RV'2001, Runtime Verification (in connection with CAV '01).

[22] Klaus Havelund and Grigore Roşu. 2002. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 342–356.

[23] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. 2014. Online Monitoring of Metric Temporal Logic. In *Runtime Verification*. Springer International Publishing, Cham, 178–192.

[24] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. 2004. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Form. Methods Syst. Des.* 24, 2 (mar 2004), 129–155. https://doi.org/10.1023/B:FORM.0000017719.43755.7c

[25] Konstantinos Mamouras, Agnishom Chattopadhyay, and Zhifu Wang. 2021. A Compositional Framework for Quantitative Online Monitoring over Continuous-Time Signals. In *Runtime Verification*. Springer International Publishing, Cham, 142–163.

[26] Amjad Nusayr and Jonathan Cook. 2009. Using AOP for Detailed Runtime Monitoring Instrumentation. In *Proceedings of the Seventh International Workshop on Dynamic Analysis* (Chicago, Illinois) *(WODA '09)*. Association for Computing Machinery, New York, NY, USA, 8–14. https://doi.org/10.1145/2134243.2134246

[27] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. 2015. MarQ: Monitoring at Runtime with QEA. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 596–610.

[28] Joshua Schneider, David Basin, Frederik Brix, Srdjan Krstic, and Dmitriy Traytel. 2019. Adaptive Online First-Order Monitoring. In *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11781)*. Springer, 133–150. https://doi.org/10.1007/978-3-030-31784-3_8

[29] Joshua Schneider, David Basin, Frederik Brix, Srundefinedan Krstić, and Dmitriy Traytel. 2021. Scalable Online First-Order Monitoring. *Int. J. Softw. Tools Technol. Transf.* 23, 2 (apr 2021), 185–208. https://doi.org/10.1007/s10009-021-00607-1

[30] Cristina Stratan, Joshua Dawes, and Domenico Bianculli. 2024. Diagnosing Violations of Time-based Properties Captured in iCFTL. In *Proceedings of the 2024 International Conference on Formal Methods in Software Engineering (FormaliSE 2024), co-located with ICSE 2024, Lisbon, Portugal*. ACM.

[31] Moshe Y. Vardi. 1996. An Automata-Theoretic Approach to Linear Temporal Logic. In *Proceedings of the VIII Banff Higher Order Workshop Conference on Logics for Concurrency : Structure versus Automata : Structure versus Automata* (Banff, Canada). Springer-Verlag, Berlin, Heidelberg, 238–266.

Received 2024-02-08; accepted 2024-04-18