Specifying Source Code and Signal-based Behaviour of Cyber-Physical System Components

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

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

Abstract. Specifying properties over the behaviour of components of Cyber-Physical Systems usually focuses on the behaviour of signals, i.e., the behaviour of the *physical* part of the system, leaving the behaviour of the cyber components implicit. There have been some attempts to provide specification languages that enable more explicit reference to the behaviour of cyber components, but it remains awkward to directly express the behaviour of both cyber and physical components in the same specification, using one formalism. In this paper, we introduce a new specification language, Source Code and Signal Logic (SCSL), that 1) provides syntax specific to both signals and events originating in source code; and 2) does not require source code events to be abstracted into signals. We introduce SCSL by giving its syntax and semantics, along with examples. We then provide a comparison between SCSL and existing specification languages, using an example property, to show the benefit of using SCSL to capture certain types of properties.

Keywords: Specification Language · Temporal Logic · Source Code · Signals

1 Introduction

Analysing the behaviour of components of Cyber-Physical Systems (CPS) often involves analysing the behaviour of the signals generated by such components. Analysis can begin with capturing the expected behaviour of these signals using formal specifications. For example, one could write a specification to capture the *temporal property* that, if a given signal falls below a certain threshold then, no more than ten units of time later, the same signal has risen back to a safe value again. Ultimately, deciding whether a given system satisfies such a specification is the goal of Runtime Verification (RV) [8].

In the context of RV, a number of languages have been introduced for the CPS setting, one of the most notable examples being Signal Temporal Logic (STL) [23]. STL is based on Metric Temporal Logic (MTL) [22], which allows one to capture temporal properties over *atomic propositions*, that is, symbols that can be associated with a Boolean value. STL extends these propositions to be predicates over real-valued signals. For example, one can capture the property that a signal x should always be strictly less than 10 by writing $\Box\, x < 10$. Here, the *always* temporal operator $\Box$ is inherited from MTL, while the ability to compare the value of a signal x with another quantity is a novelty of STL.

Another example of a specification language introduced in the context of CPS is the Hybrid Logic of Systems (HLS) [24]. This language allows explicit reference to

the behaviour of *cyber* and *physical* components of a CPS by providing access to both timestamps and indices. Here, a cyber component is often a component that contains software, and a physical component is one that measures a physical process. While the timestamps provided by HLS allow one to refer, as usual, to the behaviour of some physical process, the indices allow one to capture properties of the behaviour of cyber components. The capture of the behaviour of cyber components assumes that behaviour like program variable changes has been abstracted into Boolean signals, which can then be accessed using the indices provided by HLS. Ultimately, describing the behaviour of cyber components is more natural when using HLS than when using STL.

In the domain of cyber components, in particular capturing properties over source code-level behaviour, recent contributions include Inter-procedural Control-Flow Temporal Logic (iCFTL) [16]. iCFTL allows one to write constraints over events such as program variable changes and function calls by providing specific syntax for doing so.

Ultimately, iCFTL and HLS are complementary: HLS allows one to capture behaviour of signals and cyber components of CPS, but ultimately assumes that everything is encoded in signals. iCFTL does not support signals, but provides syntax specific to source code-level behaviour.

Hence, in this paper we introduce *Source Code and Signal Logic* (SCSL), which is a combination of iCFTL and HLS. In particular, SCSL provides syntax specifically designed for dealing with signal and source code-level behaviour. We also introduce a semantics that deals with traces representing CPS runs that are on-going. In line with existing work, such as that by Bauer at al. [10,9], our semantics uses an extended truth domain in order to deal with situations in which information that is required by a given specification may not yet be available. Once SCSL is introduced, we then provide a comparison of SCSL with existing specification languages, by attempting to capture a property using those languages, along with SCSL.

Related Work. Since we provide a comparison of SCSL with existing specification languages, our description of related work does not go into more depth with respect to contributions from the RV community. Instead, we focus on a subset of contributions from Model Checking, since these languages served as a starting point for a lot of contributions from the RV community. An example of a contribution is that by Alur et al. [6], where *hybrid systems* are assumed to be represented by automata augmented with time [7]. Other contributions are numerous, ranging from providing specification languages for model checking of systems that contain both cyber and physical components [13] to falsification of specifications [19,5].

In contrast with the RV community, contributions from model checking focus on static analysis of a model of the system under scrutiny. When a system involves continuous behaviour, such as signals, this can give rise to probabilistic model checking approaches, such as the COMPASS project [12]. Our work assumes that a trace has already been generated, either by the real system or by a simulation. Hence, we do not consider the probabilistic setting.

Paper structure. Section 2 motivates our new language and defines its design goals. We describe a notion of trace for CPS in Section 3. In Section 4, we introduce the concept of hybrid traces. Section 5 presents the syntax of SCSL; Section 6 illustrates its semantics.

We present a comparison of SCSL with existing specification languages in Section 7. Finally, Section 8 describes on-going further work, and Section 9 concludes the paper.

2 Motivation and Language Design Goals

Our main goal is to introduce a language that enables engineers to capture properties concerning the behaviour of multiple components in the system under scrutiny. To this end, we assume that the expression of properties concerning both the *cyber* and *physical* components of a system can involve placing constraints over the behaviour of 1) signals and 2) source code components (that control the signals). Hence, a specification language that allows one to capture such properties must provide syntax (and a semantics) specific to both of these domains.

Considering the signal and source code domains separately, the properties that one could aim to express can be taken from HLS and iCFTL respectively. For example, over signals, one might express the property that the value of a signal `temperature` never exceeds 100. Over source code, one might express the property that one statement is reached from another within a certain amount of time.

Let us consider a property that requires us to talk about both signals and source code behaviour at the same time, $\mathcal{P}1$: "*if the value of the signal* `temperature` *exceeds 100, then the time taken to reach the next call of the function* `fix` *should not exceed one second*". Here, we can express the two components of this property in two separate specification languages, but it would be useful to express the property as a whole using a single language.

Consider a further example, $\mathcal{P}2$: "*if the value of the signal* `temperature` *exceeds 100, then the next change of the variable* `adjustment` *must leave it with a value that is proportional to the value of* `temperature`". For this property, there are no distinct components that we could express in HLS and iCFTL; we must be able to relate a quantity taken from source code directly to a quantity taken from a signal.

These two examples define the key design goals for our new specification language:

G1 One must be able to define constraints over signals and source code within the same specification.
G2 One must be able to define *hybrid* constraints, that is, single constraints that relate quantities from signals with quantities from source code.

Ultimately, RV serves as a supplement to existing testing approaches. Hence, while these design goals are met by many existing specification languages, the language features that allow them to be met introduce a lot of additional effort to the software verification and validation process. This is discussed in depth in Section 7, which highlights the extra effort required (for a representative set of specification languages) if one wants to capture properties like $\mathcal{P}1$ and $\mathcal{P}2$.

3 Background

In line with other approaches in RV, we refer to our representation of a system's execution as a *trace*. Given our focus on the behaviour of source code-level behaviour in control

components, and signal behaviour, our notion of trace must contain information from source code-level events, and signals. Hence, we begin by describing the traces used by iCFTL and HLS.

3.1 Traces for Signals used by HLS [24]

The traces used by HLS are intended to represent a set of signals with the assumption that a value of each signal in the system is available at each timestamp being considered. This assumption is encoded in the *records* used by HLS, which are tuples of the form $\langle ts, \mathsf{index}, s_1, \ldots, s_n \rangle$, for ts a real-numbered timestamp, index an integer index and each s_i representing the value of the signal s_i at the timestamp ts. A signal is a sequence of such records, with strictly increasing timestamps and consecutive indices.

3.2 Trace for Source Code used by iCFTL [16]

The traces introduced for iCFTL are tightly coupled with the source code that generated them, in that a trace can be seen as a path through a program. To support this idea, iCFTL introduces *concrete states*, which are intuitively the *states* reached by a program execution after the execution of individual program statements.

Formally, a concrete state c is a triple $\langle ts, \mathsf{pPoint}, \mathsf{values} \rangle$, for ts a real-numbered timestamp, pPoint a *program point*, and values a map from program variables to values. A program point pPoint is the unique identifier of the program statement whose execution generated the concrete state. Hence, program points capture the intuition that concrete states represent instantaneous checkpoints, within a single procedure, that a program can reach.

For a program point pPoint, we define the predicate changed(pPoint, x) to indicate whether the statement at the program point pPoint assigns a value to the program variable x. Similarly, we define the predicate called(pPoint, f) to indicate whether the statement involves a call of the function f[1].

Returning to concrete states, a concrete state $c = \langle ts, \mathsf{pPoint}, \mathsf{values} \rangle$ can be said to *be attained* at time ts, which we denote by $\textsc{time}(\langle ts, \mathsf{pPoint}, \mathsf{values} \rangle)$. Since a concrete state holds a map m from program variables to their values, we will denote values(x), for a program variable x, by $c(\mathtt{x})$.

When concrete states are arranged in a sequence (ordered by timestamps ascending), we call a pair of consecutive concrete states a *transition*, often denoted by tr, because one can consider the computation required to move from one concrete state, to the other. For a transition $tr = \langle ts, \mathsf{pPoint}, \mathsf{values} \rangle, \langle ts', \mathsf{pPoint}', \mathsf{values}' \rangle$, we denote by $\textsc{time}(tr)$ the timestamp ts. Since transitions represent the computation performed to move between concrete states, we can also talk about the *duration* of the transition tr, which we denote by $\textsc{duration}(tr)$ and define as $ts' - ts$.

Ultimately, a sequence of concrete states generated by a single procedure in code is referred to as a *dynamic run* $\mathcal{D}$. We remark that, since we consider dynamic runs as being generated by individual procedures, concrete states from the same procedure execution cannot share timestamps. Further, we can consider a system consisting of

[1] These predicates can be computed via static analyses of source code.

multiple procedures and group the dynamic runs (generated by each procedure) together into a triple $\langle\{\mathcal{D}_i\}, \mathsf{Procs}, \mathsf{runToProc}\rangle$, where $\{\mathcal{D}_i\}$ is a set of dynamic runs, Procs is a set of names of procedures in the program, and runToProc is a map that labels each dynamic run in $\{\mathcal{D}_i\}$ with the name of a procedure in Procs. We call this triple an *inter-procedural dynamic run*.

Inside an inter-procedural dynamic run, given a concrete state c from some dynamic run, we write $\mathsf{proc}(c)$ to mean $\mathsf{runToProc}(\mathcal{D})$ for $\mathcal{D}$ being the dynamic run in which c is found. Similarly, for a transition tr, we write $\mathsf{proc}(tr)$.

4 Hybrid Traces

Based on the notions of traces introduced by iCFTL and HLS, we must now combine them to yield a kind of trace that contains both source code events, and signal entries. Such a notion of trace will serve as the basis for SCSL. Further, we will assume that all traces are generated by a CPS whose execution is on-going.

Our first step in combining the two notions of trace is to assume a global clock from which all timestamps (whether they be attached to concrete states or records) can be taken. We highlight that our approach is so far being developed in the context of simulators, so the existence of a global clock is a reasonable assumption. With this global clock, we collect the system's inter-procedural dynamic run and its sequence of records into a tuple $\langle\mathsf{signalNames}, \mathsf{records}, \mathsf{sigID}, \{\mathcal{D}_i\}, \mathsf{Procs}, \mathsf{runToProc}\rangle$, whose elements are as such:

- signalNames is a set of names of signals in the CPS.
- records is a sequence of records, each containing signal entries for each of the signals represented in signalNames.
- sigID is a map that sends each signal name s in signalNames to a sequence of triples $\langle ts, \mathsf{index}, s\rangle$ derived from records. These triples are obtained by projecting the records in records with respect to the signal name s. Assuming that one has used $\mathsf{sigID}(s)$ to obtain a sequence of triples, we denote by $\mathsf{sigID}(s)(ts)$ the signal value held in the triple whose timestamp is ts. This map is included in the tuple (i.e., it is part of the trace) so that the correspondence between signal names and sequences of triples is fixed for a given trace.
- $\{\mathcal{D}_i\}$, Procs and runToProc are as introduced earlier.

Ultimately, we refer to the tuple introduced above as a *trace*, which we will denote by $\mathcal{T}$. Using this notation, we will often write $\mathcal{T}(s)$ instead of $\mathsf{sigID}(s)$, with the understanding that we are implicitly referring to the map sigID held in $\mathcal{T}$. Hence, to refer to the value of a signal s at the timestamp ts in the trace $\mathcal{T}$, we write $\mathcal{T}(s)(ts)$.

Remark on obtaining traces. We highlight that, in practice, the parts of a trace that are dynamic runs can be obtained by instrumenting the relevant source code of a CPS [15]. Further, obtaining signal entries depends heavily on the use case (i.e., whether signals are generated by a simulator, or by physical sensors).

Entry point

$$\phi \longrightarrow \forall v \in P_Q : \phi \mid \mathit{true} \mid \phi \vee \phi \mid \neg\phi \mid A$$

Atomic constraints

$$\mathit{cmp} \longrightarrow <\mid>\mid=$$
$$A \longrightarrow V\ \mathit{cmp}\ V$$

Terms

$$V \longrightarrow V_{Ts} \mid V_C \mid V_{Tr} \mid V_{\mathsf{m}} \mid n$$
$$V_{Ts} \longrightarrow \mathsf{signal.at}(Ts) \mid f(V_{Ts}) \mid \textsc{time}(C) \mid \textsc{time}(Tr)$$
$$V_C \longrightarrow C(x) \mid f(V_C)$$
$$V_{Tr} \longrightarrow \textsc{duration}(Tr) \mid f(V_{Tr})$$
$$V_{\mathsf{m}} \longrightarrow \textsc{TimeBetween}(Ts, Ts)$$

Expressions

$$Ts \longrightarrow ts \mid n_{\mathsf{pos}} \mid Ts + Ts \mid \textsc{time}(C) \mid \textsc{time}(Tr)$$
$$C \longrightarrow c \mid ts.\textsc{next}(P_C) \mid C.\textsc{next}(P_C) \mid Tr.\textsc{next}(P_C) \mid \textsc{before}(Tr) \mid \textsc{after}(Tr)$$
$$Tr \longrightarrow tr \mid ts.\textsc{next}(P_{Tr}) \mid C.\textsc{next}(P_{Tr}) \mid Tr.\textsc{next}(P_{Tr})$$

Predicates

$$P_Q \longrightarrow P_{QTs} \mid P_{QC} \mid P_{QTr}$$
$$P_{QTs} \longrightarrow [Ts, Ts] \mid (Ts, Ts)$$
$$P_{QC} \longrightarrow P_C \mid P_C.\mathsf{after}(Ts)$$
$$P_C \longrightarrow \mathsf{changes}(\mathtt{var}).\mathsf{during}(\mathtt{proc})$$
$$P_{QTr} \longrightarrow P_{Tr} \mid P_{Tr}.\mathsf{after}(Ts)$$
$$P_{Tr} \longrightarrow \mathsf{calls}(\mathtt{proc}_1).\mathsf{during}(\mathtt{proc}_2)$$

Fig. 1: The syntax of SCSL.

5 SCSL syntax

We now introduce the syntax of SCSL, which allows one to construct specifications over traces, as defined in Section 4. We give the syntax as a context-free grammar in Figure 1, in which all uses of non-terminal symbols are highlighted in blue. The rules given in the grammar are divided into groups that cover the key roles of certain parts of specifications. Further, n is a constant, and n_{pos} is a real number that is greater than zero.

Entry point. A specification is constructed by first using the rule ϕ. This allows one to generate a quantifier, along with a subformula that will be subject to the quantification. The role of a quantifier is to capture events from a trace, including concrete states, transitions, and timestamps of signal entries. Hence, quantifiers consist of a *predicate* P_Q and a *variable* v; P captures values from a trace, which are each bound to v ready to be used elsewhere in the specification.

Aside from the form of quantifiers enforced by the grammar, we place two additional constraints: 1) if a quantifier is not the root quantifier in a specification, it must depend on its closest parent quantifier (i.e., it must use the timestamp of the event captured by its parent quantifier to capture events that occur after that event); and 2) a specification can have no free variables.

Predicates. The predicates that one can use are arranged in the grammar by what they capture: timestamps (P_{QTs}), concrete states (P_{QC}), or transitions (P_{QTr}).

Atomic constraints. Once a quantifier has been used to assign values to a variable, the next step is to define constraints over those values. This is done using the rule A, which generates a comparison between two *terms*, that is, strings obtained using the rule V. These terms represent values that are obtained by extracting certain information from

concrete states, transitions or timestamps. For example, from a concrete state c, one might wish to refer to the value of the program variable `x` in that state, so one would use V_C to generate $c(\texttt{x})$.

Expressions. Using variables from quantifiers, one can either place a constraint on the immediate value held by those variables, or one can search forwards in time using a specific criterion. For example, given a concrete state c, one could denote the next transition after c in the trace by $c.\textsc{next}(P_{Tr})$. A predicate could then be generated using P_{Tr} to reflect the desired criterion.

5.1 Examples

We now present two sets of example SCSL specifications, and show how the design goals introduced in Section 2 are met.

Artificial examples Our first set consists of artificial examples that have been constructed to showcase the expressive power of SCSL.

Example 1 "*Whenever the signal* temperature *drops below 100 during the first 10 minutes of a system run, the time until the variable* `flag` *in the procedure* `monitor` *is changed should be no more than 1 second.*"

$$\begin{aligned}&\forall ts \in [0, 60 * 10] : \mathsf{temperature}.\textsc{at}(ts) < 100 \rightarrow \\ &\quad \textsc{TimeBetween}(ts, \textsc{time}(ts.\textsc{next}(\mathsf{changes}(\texttt{flag}).\mathsf{during}(\texttt{monitor})))) \leq 1\end{aligned} \tag{E1}$$

In this specification, we refer to the value of the signal temperature at the timestamp held in the variable *ts*, and we refer to the time taken to reach a specific variable change, from the time *ts*.

This specification shows that we have met design goal **G1**, because we can refer to both signals and source code events in the same specification.

Example 2 "*If the procedure* `adjust` *is called by the procedure* `control`*, then the signal* temperature *should be equal to 100 within 1 second.*"

$$\begin{aligned}&\forall tr \in \mathsf{calls}(\texttt{adjust}).\mathsf{during}(\texttt{control}) : \\ &\quad \textsc{duration}(tr) < 1 \wedge \exists ts \in [\textsc{time}(tr), \textsc{time}(tr) + 1] : \mathsf{temperature}.\textsc{at}(ts) = 100\end{aligned} \tag{E2}$$

In this specification, we use the time at which a given function was called to select timestamps, and then refer to a signal value at each of the timestamps identified.

Example 3 "*Within the first ten seconds of a CPS execution, the value of the program variable* `x` *in the procedure* `p` *should reflect the value of the most recent value of the signal* signal"

$$\forall ts \in [0, 10] : \mathsf{signal}.\textsc{at}(ts) = ts.\textsc{next}(\mathsf{changes}(\texttt{x}).\mathsf{during}(\texttt{p}))(\texttt{x}), \tag{E3}$$

This specification involves the comparison of a value extracted from a signal, and a value extracted from a program variable.

This specification shows that we have met design goal **G2**, because we can write a single atomic constraint that uses information from a signal entry, and from a source code event.

The ArduPilot system The ArduPilot [3] system acts as an autopilot for various types of vehicle both in the simulation setting and in the real-world setting. Here, we give examples derived by inspecting the source code found in their GitHub repository [4]. We have simplified the names of some program variables and procedures to save space.

Example 4 This property is derived from the code in the file `fence.cpp` [2]. In this code, the procedure `fence_check` checks for the copter leaving some *safe region*, called a *fence*. To this end, a program variable `new_breaches` holds each example of a breach of the fence that has been detected. If the copter strays more than 100 m outside the fence, its *mode* is set to *landing* by a call of the function `set_mode`. Hence, the property "*If a copter strays more than* 100 m *outside a fence, the mode should be changed within 1 unit of time*" can be expressed as follows:

$$\begin{aligned}&\forall q \in \mathsf{changes}(\texttt{new\_breaches}).\mathsf{during}(\texttt{fence\_check}) :\\ &\quad (q(\texttt{new\_breaches}) \neq \texttt{null} \wedge \mathsf{distFence}.\textsc{at}(\textsc{time}(q)) > 100)\\ &\quad \rightarrow \textsc{TimeBetween}(\textsc{time}(q),\\ &\qquad \textsc{time}(\textsc{before}(q.\textsc{next}(\mathsf{calls}(\texttt{set\_mode}).\mathsf{during}(\texttt{fence\_check}))))) \leq 1\end{aligned} \tag{E4}$$

We have introduced the signal distFence, which we assume contains a value representing the distance of the copter from the fence in metres.

Example 5 This property is derived from the code in the file `crash_check.cpp` [1]. In this file, the procedure `thrust_loss_check` checks the behaviour of the copter's thrust. This involves checking the attitude, the throttle, and the vertical component of the velocity. If the checks reveal a problem, `set_thrust_boost` is called.

A property capturing this behaviour could be "*If (a) The attitude is less than or equal to an allowed deviation; (b) The throttle satisfies the predicate P; and (c) The vertical component of velocity is negative; then* `set_thrust_boost` *should be called within 1 unit of time*", which could be captured by the specification

$$\begin{aligned}&\forall ts \in [0, L] : (\mathsf{att}.\textsc{at}(ts) < \mathsf{maxDev} \wedge P(\mathsf{thr}.\textsc{at}(ts)) \wedge \mathsf{vel}_z.\textsc{at}(ts) < 0) \rightarrow\\ &\quad \exists c \in \mathsf{calls}(\texttt{set\_thrust\_boost}).\mathsf{during}(\texttt{thrust\_loss\_check}).\mathsf{after}(ts) :\\ &\qquad \textsc{TimeBetween}(ts, \textsc{time}(\textsc{before}(c))) \leq 1.\end{aligned} \tag{E5}$$

for L some positive real number. Here, we assume that att (attitude), thr (throttle), and vel_z are signals. In order to refer to their values over time, we quantify over the interval $[0, L]$ using the variable *ts*. We take maxDev to be a constant, and P to be some atomic constraint allowed by the syntax in Figure 1 (both to be decided by the engineer).

$\mathcal{T}, \beta, ts \vdash [Ts_1, Ts_2]$ iff $\mathsf{getVal}(\mathcal{T}, \beta, Ts_1) \leq ts \leq \mathsf{getVal}(\mathcal{T}, \beta, Ts_2)$
$\mathcal{T}, \beta, ts \vdash (Ts_1, Ts_2)$ iff $\mathsf{getVal}(\mathcal{T}, \beta, Ts_1) < ts < \mathsf{getVal}(\mathcal{T}, \beta, Ts_2)$
$\mathcal{T}, \beta, \langle ts, \mathsf{pPoint}, \mathsf{values}\rangle \vdash \mathsf{changes}(\mathtt{x}).\mathsf{during}(\mathtt{func})$
iff $\mathsf{changed}(\mathsf{pPoint}, \mathtt{x})$ and $\mathsf{proc}(\langle ts, \mathsf{pPoint}, \mathsf{values}\rangle) = \mathtt{func}$
$\mathcal{T}, \beta, q \vdash \mathsf{changes}(\mathtt{x}).\mathsf{during}(\mathtt{func}).\mathsf{after}(Ts)$
iff $\textsc{time}(q) > \mathsf{eval}(\mathcal{T}, \beta, Ts)$ and $\mathcal{T}, \beta, q \vdash \mathsf{changes}(\mathtt{x}).\mathsf{during}(\mathtt{func})$
$\mathcal{T}, \beta, tr \vdash \mathsf{calls}(\mathtt{f}).\mathsf{during}(\mathtt{func})$ iff $\mathsf{called}(\mathsf{pPoint}', \mathtt{f})$ and $\mathsf{proc}(tr) = \mathtt{func}$
$\mathcal{T}, \beta, tr \vdash \mathsf{calls}(\mathtt{f}).\mathsf{during}(\mathtt{func}).\mathsf{after}(Ts)$
iff $\textsc{time}(tr) > \mathsf{eval}(\mathcal{T}, \beta, Ts)$ and $\mathcal{T}, \beta, tr \vdash \mathsf{calls}(\mathtt{f}).\mathsf{during}(\mathtt{func})$

Fig. 2: The valuation relation for SCSL. Timestamps *ts* are assumed to be in some record. Transitions *tr* denote, as defined in Section 3.2, pairs of concrete states $\langle ts, \mathsf{pPoint}, \mathsf{values}\rangle, \langle ts', \mathsf{pPoint}', \mathsf{values}'\rangle$.

6 Semantics

We now introduce a function that takes a trace and an SCSL specification, and gives a *truth value* reflecting the status of the trace, with respect to the specification. For example, if the trace satisfies the specification (that is, holds the property captured by the specification), then our function should give a value that indicates as such. The situation should be similar if the trace does not satisfy the specification. However, if there is not enough information in the trace to decide whether it satisfies the specification, then our function should give a value reflecting this.

Our semantics function makes use of multiple components, including 1) a way to extract the information from a trace that is needed by a term (§ 6.1); and 2) a way to determine the truth value of an atomic constraint and, from there, the specification as a whole (§ 6.2).

6.1 Determining values of terms

We will support our introduction of the first components of our semantics for SCSL using the specification in Example E2.

This example includes the atomic constraints $\textsc{duration}(tr) < 1$ and $\mathsf{temperature}.\textsc{at}(ts) = 100$, along with a quantifier whose predicate is $[\textsc{time}(tr), \textsc{time}(tr) + 1]$. In each case, there is an expression, or a term, whose value we must determine, given either the transition held in the variable *tr*, or the timestamp held in the variable *ts*. To this end, we introduce the eval and getVal functions. Leaving a complete definition to Appendix A, we consider how these functions would be applied to the various components of our running example:

First, consider the atomic constraint $\textsc{duration}(tr) < 1$. Deciding a truth value for this atomic constraint requires us to determine 1) the transition held by the variable *tr*, and 2) the duration of that transition. In this case, the getVal function is responsible for deriving the final value of $\textsc{duration}(tr)$, while the eval function is used to determine the transition held by *tr*. For $\mathsf{temperature}.\textsc{at}(ts) = 100$, the getVal function must determine the value to which the term $\mathsf{temperature}.\textsc{at}(ts)$ evaluates. This then requires the eval

function to determine the timestamp held by the variable *ts*. For $[\text{TIME}(tr), \text{TIME}(tr)+1]$, in order for the getVal function to determine the relevant values, the eval function is needed to determine the value stored in the variable *tr*, whose timestamp can then be extracted.

In all three cases, we need the eval function to determine the value held by a variable. Hence, we need some structure that quantifiers can use to communicate the values that they capture with the eval function. We call such a structure a *valuation*, which is a map that associates with each variable in a specification a concrete state, transition or timestamp.

Using this idea, we can say that the eval function will take a trace, a valuation, and an expression, and return a unique result. In addition, we say that the getVal function will take a trace, a valuation, and a term, and return a unique result. Hence, we will write $\mathsf{eval}(\mathcal{T}, \beta, expr)$ and $\mathsf{getVal}(\mathcal{T}, \beta, term)$.

This notation is used by Figure 2, which gives a recursive definition of the *valuation relation*. This relation defines which concrete states, transitions, or timestamps are captured by a predicate, hence providing a way to construct valuations. The relation also makes use of the changed(pPoint, x) and called(pPoint, f) predicates defined in Section 3.2.

We conclude our description of the eval function and the valuation relation with two remarks.

Returning null. We do not assume that traces will always contain the information that a given term references. For example, a specification might refer to $c.\text{NEXT}(\mathsf{changes}(\mathtt{x})$ .during(p)), but we might work with a trace that does not contain the relevant concrete state. To deal with such cases, we allow the eval and getVal functions to return null.

Interpolation of signals. Suppose that a specification contains the atomic constraint $\mathsf{signal}.\text{AT}(Ts) < 1$, where *Ts* is some expression that yields a timestamp. Depending on the expression *Ts*, the signal signal is not certain to have a value at that timestamp. Hence, we must interpolate. Our strategy in this work is to find the closest timestamp *in the future* at which the signal has a value. Interpolation, a common practice in CPS monitoring [24], is required because, otherwise, one would have to know the precise timestamps of events for a given CPS run.

6.2 A semantics function

We next introduce a semantics function that takes a trace, along with an SCSL specification, and yields a truth value reflecting the status of that trace with respect to the specification. Our semantics function assumes that the trace given represents a CPS execution that is on-going. In particular, our semantics function holds the *impartiality* property [9], that is, it does not generate a definitive verdict, rather a *provisional* one, since processing further events can lead to a change in verdict. Specifically, our semantics function declares *falseSoFar*, *inconclusive*, or *trueSoFar*. These truth values have the total ordering $\mathit{falseSoFar} < \mathit{inconclusive} < \mathit{trueSoFar}$. We also have that $\overline{\mathit{trueSoFar}} \equiv \mathit{falseSoFar}$, and $\overline{\mathit{inconclusive}} \equiv \mathit{inconclusive}$. In order to generate truth values in either of these domains, our semantics function works as follows: for a given

$$[\mathcal{T},\beta,\forall v \in P : \varphi] = \bigsqcap_{v \vdash P} [\mathcal{T},\beta \dagger [c \mapsto v],\varphi] \qquad [\mathcal{T},\beta,\varphi_1 \vee \varphi_2] = [\mathcal{T},\beta,\varphi_1] \sqcup [\mathcal{T},\beta,\varphi_2]$$

$$[\mathcal{T},\beta,\neg\varphi] = \overline{[\mathcal{T},\beta,\varphi]} \qquad [\mathcal{T},\beta,\mathit{true}] = \mathit{true}$$

$$[\mathcal{T},\beta,V_1\ \mathit{cmp}\ V_2] = \begin{cases} \mathit{trueSoFar} & \mathsf{getVal}(\mathcal{T},\beta,V_1) \neq \mathsf{null} \text{ and } \mathsf{getVal}(\mathcal{T},\beta,V_2) \neq \mathsf{null} \\ & \quad \text{and } \mathsf{getVal}(\mathcal{T},\beta,V_1)\ \mathit{cmp}\ \mathsf{getVal}(\mathcal{T},\beta,V_2) \\ \mathit{falseSoFar} & \mathsf{getVal}(\mathcal{T},\beta,V_1) \neq \mathsf{null} \text{ and } \mathsf{getVal}(\mathcal{T},\beta,V_2) \neq \mathsf{null} \\ & \quad \text{and } \neg(\mathsf{getVal}(\mathcal{T},\beta,V_1)\ \mathit{cmp}\ \mathsf{getVal}(\mathcal{T},\beta,V_2)) \\ \mathit{inconclusive} & \text{otherwise} \end{cases}$$

Fig. 3: The semantics function for SCSL.

trace, an appropriate truth value is computed by recursing on the structure of a specification, computing a truth value for each subformula. Truth values come from atomic constraints, by deciding the truth value of each constraint with respect to the information available in the trace. These truth values are then propagated up through the specification by the recursion. In line with this intuition, a semantics function is presented in Figure 3. The function takes a trace $\mathcal{T}$, a valuation β, and a subformula φ, and computes a truth value. We now give a brief description of the approach taken by each case in Figure 3 to evaluate a given part of a specification.

- Computing $[\mathcal{T},\beta,\forall v \in P : \varphi]$ consists of computing the greatest-lower-bound of the set of truth values $[\mathcal{T},\beta \dagger [v \mapsto e],\varphi]$, for each e (whether it be a concrete state, a transition or a timestamp) that satisfies P, according to the relation defined in Figure 2. The $\dagger$, or *map amend*, operator is used to extend valuations with new values. For example, $a \dagger [v \mapsto n]$ refers to the map that agrees with the map a on all values except v, which it sends to n. This operator is used to extend a valuation once a v that satisfies P has been found.
- Computing $[\mathcal{T},\beta,\varphi_1 \vee \varphi_2]$ consists of computing the truth values of the two disjuncts, and then computing their least-upper-bound.
- Computing $[\mathcal{T},\beta,\neg\varphi]$ consists of taking the complement of $[\mathcal{T},\beta,\varphi]$.
- Computing $[\mathcal{T},\beta,\mathit{true}]$ consists of deciding on a truth value for this case requires no further computation, other than taking the truth value already used in the subformula.
- Computing $[\mathcal{T},\beta,V_1\ \mathit{cmp}\ V_2]$ involves the weight of the work performed by the semantics, and is responsible for generating truth values that are propagated up through the specification. Specifically, *provisional* truth values are generated, including *trueSoFar*, *inconclusive*, and *falseSoFar*, depending on whether 1) the information necessary was found in the trace; and 2) that information satisfies the constraint in question.

Table 1: Comparison of specification languages and their features.

	$\mathcal{S}_{\mathsf{ts}}$	$\mathcal{S}_{\mathsf{index}}$	$\mathcal{X}_{\mathsf{sig}}$	$\mathcal{X}_{\mathsf{code}}$	$\mathcal{X}_{ll}$	$\mathcal{X}_{\mathsf{het}}$	$\mathcal{X}_{\mathsf{index}}$	$\mathcal{X}_{\mathsf{ts}}$	$\mathcal{X}_{\mathsf{metric}}$
SCSL	✓	×	✓	✓	✓	✓	×	✓	×
LTL	×	✓	×	×	×	×	×	×	×
MTL	✓	×	×	×	×	×	×	×	✓
TLTL	×	✓	×	×	×	×	×	✓	×
HyLTL	×	✓	✓	×	×	✓	×	×	×
STL	✓	×	✓	×	✓	×	×	×	✓
STL*	✓	×	✓	×	✓	×	×	×	✓
STL-MX	✓	✓	✓	×	✓	✓	×	×	✓
HLS	✓	✓	✓	×	✓	✓	✓	✓	×
SB-TemPsy	✓	×	✓	×	✓	×	×	✓	✓

7 Language Comparison

We now present a comparison of SCSL with existing specification languages, in order to demonstrate the novelty of this new language. Table 1 presents an initial comparison by highlighting a number of key features, which are defined as follows:

$\mathcal{S}_{\mathsf{ts}}$ Whether a specification language's semantics is defined using timestamps to refer to entries in the trace. The form of such *entries* differs according to the specification language. For example, STL's semantics considers the pair (s, t) of a signal s and a timestamp t, and uses the timestamp t to refer to the signal s at the given timestamp.

$\mathcal{S}_{\mathsf{index}}$ Whether a specification language's semantics is defined using indices to refer to entries in the trace. For example, LTL's semantics considers the pair (ω, i) for a trace ω and index i.

$\mathcal{X}_{\mathsf{sig}}$ Whether a specification language provides syntax specific to signals. For example, HLS provides the `@t` operator, which enables one to write (s `@t` t) to refer to the value of the signal s at time t.

$\mathcal{X}_{\mathsf{code}}$ Whether a specification language provides syntax specific to events generated at the source code level. For example, SCSL enables one to easily measure the duration of a function call with DURATION(tr) (for tr holding a *transition*).

$\mathcal{X}_{ll}$ Whether a specification language's syntax is at a low level of abstraction with respect to the system being monitored. For example, LTL abstracts behaviour into *atomic propositions*, whereas SCSL assumes that traces contain explicit representations of key events, such as program variable value changes and function calls.

$\mathcal{X}_{\mathsf{het}}$ Whether a specification language provides explicit support for heterogeneity (components of multiple types, such as sensors and source code-based control components). For example, with SCSL one can write the constraint that $q(\mathtt{x}) = \mathtt{s}.\text{AT}(t)$, which involves measurements taken from both signal behaviour and source code execution. Hence, SCSL can be said to support heterogeneity.

$\mathcal{X}_{\mathsf{index}}$ Whether a specification language allows reference to events in a trace by their index. For example, HLS enables one to get the event in a signal based on its index by writing (s `@i` i) for a signal s and an index i.

$\mathcal{X}_{\mathsf{ts}}$ Whether a specification language allows explicit reference to timestamps. For example, TLTL provides the $\triangleleft$ operator, which gives the timestamp of the most recent occurrence of some atomic proposition.

$\mathcal{X}_{\mathsf{metric}}$ Whether a specification language's temporal operators are augmented with metrics. For example, MTL attaches a metric to its temporal operator $\mathcal{U}$, yielding the operator $\mathcal{U}_{[a,b]}$.

Justifications of our classification of each language are presented in Appendix B. Ultimately, Table 1 demonstrates the key feature of SCSL: syntax specific to the domain of application, which is signal and source code-based behaviour. In particular, though displaying only a representative set of languages, the table illustrates that the languages introduced by or adapted for the RV community offer a high level of abstraction with respect to the system being analysed. When considering specifications that talk about the behaviour of cyber components, this leads to the need to define a correspondence between runtime events and symbols used in a specification. While this approach often enables a language to be highly expressive (with the addition of complex modal operators), the use of generalised syntax means that expressing simple properties (such as the time taken by a function call) requires effort beyond simply writing the specification.

We now demonstrate the usefulness of SCSL by attempting to express the property "*whenever the program variable* `x` *is changed, eventually there is a call of the function* `f` *that takes less than 1 unit of time*" in each of the languages previously discussed. We remark that we have opted to use a property that does not require reference to signals so that we can include a wider range of languages in our comparison. In addition, we will consistently make use of the following *atomic propositions*:

- $\mathsf{changed}_{\mathtt{x}}$ to represent whether the program variable `x` has been changed.
- $\mathsf{called}_{\mathtt{f}}$ to represent whether the function `f` has been called.
- $\mathsf{returned}_{\mathtt{f}}$ to represent whether the function `f` has returned.

Linear Temporal Logic (LTL) [25]. This language has a high level of abstraction and provides complex modal operators. Its semantics is over *untimed words*, that is, sequences of atomic propositions that encode discrete time.

While expressing the example property is possible in LTL, effort would be required to define the correspondence and ensure that the specification was properly written to capture the variable change and function call behaviour (such as the combination of passing control to a function, and control being returned to the caller). Such a correspondence would make use of $\mathsf{changed}_{\mathtt{x}}$, $\mathsf{called}_{\mathtt{f}}$ and $\mathsf{returned}_{\mathtt{f}}$, but would also include $\mathsf{timeLessThan}_1$, representing whether the time that elapsed since the last call to `f` is less than 1 unit of time. We might then write the specification

$$\Box\,(\mathsf{changed}_{\mathtt{x}} \rightarrow \Diamond\,(\mathsf{called}_{\mathtt{f}} \rightarrow \Diamond\,(\mathsf{returned}_{\mathtt{f}} \wedge \mathsf{timeLessThan}_1)))$$

with globally, $\Box$, and eventually, $\Diamond$, having the expected semantics.

One can see that much of the actual computation required for checking the property would be migrated to the definition of the correspondence between runtime events and atomic propositions.

Metric Temporal Logic (MTL) [22]. This language extends LTL by attaching metrics to modal operators, allowing time constraints to be placed on the occurrence of events. The semantics of MTL is defined over *timed words*, which are sequences of atomic propositions with timestamps attached. Using the correspondence defined at the beginning of this section, we could then write the specification

$$\Box\,(\mathsf{changed}_{\mathtt{x}} \rightarrow \Diamond\,(\mathsf{called}_{\mathtt{f}} \rightarrow (\Diamond_{[0,1]}\ \mathsf{returned}_{\mathtt{f}}))).$$

A new operator is $\Diamond_I$, which is the *metric eventually* operator. For example, $\Diamond_{[a,b]}\ p$ means that, *eventually*, after a number of units of time in the interval $[a, b]$, p will become true.

Timed Linear Temporal Logic (TLTL) [10]. This language extends LTL with clock variables, which take the form of additional syntax used to check the time since/until an event occurred/will occur. Using the correspondence defined at the beginning of this section, we could then write the specification

$$\Box\,(\mathsf{changed}_{\mathtt{x}} \rightarrow \Diamond\,(\mathsf{called}_{\mathtt{f}} \rightarrow \Diamond\,(\mathsf{returned}_{\mathtt{f}} \wedge \lhd_{\mathsf{called}_{\mathtt{f}}} < 1))).$$

Here, $\lhd_{\mathsf{called}_{\mathtt{f}}}$ refers to the time at which the atomic proposition $\mathsf{called}_{\mathtt{f}}$ was most recently true.

*Hybrid Linear Temporal Logic (*HyLTL*) [13].* This language supports *hybrid behaviour*, meaning a combination of discrete and continuous behaviour, by extending LTL. Expressing the example property would be similar to LTL.

Signal Temporal Logic (STL) and variants [23,14,20]. Signal Temporal Logic [23], Signal Temporal Logic with a freeze quantifier (STL*) [14], and Mixed Time Signal Temporal Logic (STL-MX) [20] are all temporal logics whose semantics are defined over real-valued functions. While STL-MX is aimed at the heterogeneous setting (supporting both dense and discrete time), STL and STL* do not provide direct support for heterogeneity.

Since the behaviour of heterogeneous systems could be supported via instrumentation, one could capture the example property by abstracting the relevant system behaviour into signals, and using STL or its variants to express properties over that abstraction. However, this approach would require effort to 1) abstract complex behaviour into signals; and 2) correctly capture properties over such behaviour as properties over signals.

The Hybrid Logic of Systems (HLS) [24]. This language is a linear time, temporal logic whose semantics is defined over sequences of *records*, which are tuples $\langle t, i, v_1, \ldots, v_n\rangle$ for t a timestamp, i an index, and v_i signal values. Expressing the example property would require abstraction of the variable change and function call/return behaviour required by the property into Boolean signals. It would then be possible to use timestamp and index-based quantifiers to imitate the semantics of the modal operators provided by the other temporal logics considered so far. Hence, the atomic propositions used in previous examples would be interpreted as Boolean signals (to use the terminology in Section 4, sequences of *records* that associate timestamps with truth values). One further signal,

$\mathsf{timeSinceCall}_{\mathtt{f}}$, would be necessary, to capture the amount of time since the most recent call of the function `f`. We assume that this signal would be computed given the other three signals.

We must then translate the modal operator $\Box$ into HLS, which can be expressed by $\forall t \in [0, L]$, for L the length of the trace being considered. Further, we can translate $\Diamond$ into HLS by writing $\exists t \in [t', L]$, for some *starting timestamp* t' and L again the length of the trace. Using this translation, the example property can be expressed as

$$\begin{aligned}
&\forall t \in [0, L] : ((\mathsf{changed}_{\mathtt{x}}\ \mathtt{@t}\ t) = 1 \rightarrow \exists t' \in [t, L] : \\
&\quad ((\mathsf{called}_{\mathtt{f}}\ \mathtt{@t}\ t') = 1 \rightarrow \exists t'' \in [t', L] : \\
&\qquad ((\mathsf{returned}_{\mathtt{f}}\ \mathtt{@t}\ t'') = 1 \wedge (\mathsf{timeSinceCall}_{\mathtt{f}}\ \mathtt{@t}\ t'') < 1) \\
&\quad) \\
&).
\end{aligned}$$

SB-TemPsy-DSL [11]. This language is a domain-specific, pattern-based language designed for expressing properties such as spiking, oscillation, undershoot and overshoot of signals. Its syntax follows the "scope" and "pattern" structure proposed by Dwyer et al. [18]. Its semantics is defined over traces which are assumed to be functions from timestamps to valuations of all signals being considered.

While runtime events can be extracted into signals, it would be non-trivial to express the property under consideration in SB-TemPsy-DSL, since the syntax focuses on a specific set of behaviours that a continuous signal could demonstrate.

Source Code and Signal Logic. Given our classification of SCSL, we highlight that:

- The lack of explicit referencing of indices ($\mathcal{X}_{\mathsf{index}}$) is not a disadvantage because SCSL provides syntax specific to certain behaviour of cyber components of systems (namely source code level behaviour).
- The lack of metrics ($\mathcal{X}_{\mathsf{metric}}$) does not pose a problem because one can make explicit reference to timestamps.

The example property could be expressed as

$$\begin{aligned}
&\forall q \in \mathsf{changes}(\mathtt{x}).\mathsf{during}(\mathtt{p}) : \exists ts \in [\textsc{time}(q), L] : \\
&\quad \textsc{duration}(ts.\textsc{next}(\mathsf{calls}(\mathtt{f}).\mathsf{during}(\mathtt{p}))) < 1.
\end{aligned}$$

where `p` is a procedure in the source code of the CPS under scrutiny, and L is the length of the trace.

Importantly, here there is no need for definition of a correspondence between runtime events and symbols used in the specification. We acknowledge that this specialisation of the syntax means that SCSL can only be used to express properties concerning the behaviour for which it was specifically designed. However, we argue that this enables a more intuitive language to be developed.

Ultimately, SCSL is a language with which one can capture source code and signal-based properties by referring directly to the events with which the properties are concerned.

7.1 Implications for Software Verification and Validation processes

Throughout this section, we have seen that, while many existing languages allow one to capture the types of specifications with which this work is concerned, considerable additional work is usually required.

Taking TLTL as an example, if events that take place during an execution of a CPS are correctly encoded as atomic propositions, it is indeed possible to capture properties that concern both signals and source code-level events. However, this places considerable pressure on engineers to correctly define this correspondence. SCSL, on the other hand, is designed specifically for the signal and source code-level of granularity, meaning that there is no effort in the software verification and validation process beyond writing (and maintaining) the specification.

Similarly to the argument used in the initial introduction of CFTL [17] (the language that inspired iCFTL), we observe that, in some cases, the requirement to define a correspondence between runtime events and atomic propositions in a specification can be beneficial. In fact, such an approach can lead to a specification language that can be used to capture properties across a wide range of behaviours. This is indeed the case for the JAVA-MAC framework [21], in which one must first construct a specification, and then use a separate language to define the correspondence between runtime events and atomic propositions in the specification. However, as we have seen in this section, for a specific domain of application, it can be beneficial to use a language with specific features.

8 Ongoing Work

Our current work involves evaluating monitoring algorithms that we have developed for SCSL, based on the semantics given in Section 6. The evaluation has two objectives: investigating the performance of our monitoring algorithms in various situations (i.e., for various specifications and traces); and investigating the expressiveness of SCSL. We will test the expressiveness by selecting open source projects and attempting to capture representative requirements from those projects using SCSL.

Preliminary evaluations have given promising results, showing that it is feasible to construct algorithms for monitoring for SCSL specifications in settings where 1) the trace is still being observed, as the system under scrutiny continues executing; and 2) the entire trace has already been observed. When the trace is still being observed, our preliminary results show that our online monitoring algorithm that can *keep up* with high event rates generated by systems under scrutiny. Alternatively, when the entire trace has already been observed, our results show that our offline monitoring algorithm scales approximately linearly with the trace length, in terms of time taken and memory consumed. Ultimately, due to space restrictions, we cannot include descriptions of these algorithms or preliminary results in this paper.

9 Conclusion

In this paper, we have introduced the new specification language SCSL, which allows engineers to explicitly specify the behaviour of source code-based components

and signal-generating components of CPS. Our introduction of this new language has included a syntax, a semantics (suitable for the online and offline settings), and a comparison with existing specification languages using an example property. This comparison highlighted the benefits of SCSL: a syntax specialised to source code and signal-level behaviour, along with a semantics that assumes traces that contain information specific to signals and source code-level events.

As part of future work, we plan to investigate characteristics of SCSL, such as monitorability and satisfiability of specifications, along with diagnostics of specification violations. We also plan to carry out an extensive evaluation of the expressiveness of SCSL in the CPS domain.

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.

References

1. `Copter::crash_check` function - ArduPilot. `https://github.com/ArduPilot/ardupilot/blob/a40e0208135c73b9f2204d5ddc4a5f281000f3f1/ArduCopter/crash_check.cpp#L100`, accessed: 2022-04-13
2. `Copter::fence_check` function - ArduPilot. `https://github.com/ArduPilot/ardupilot/blob/36f3fb316acf71844be80e0337fdc66515b4cf50/ArduCopter/fence.cpp#L9`, accessed: 2022-04-13
3. The ArduPilot autopilot. `https://ardupilot.org`, accessed: 2022-04-13
4. The ArduPilot GitHub repository. `https://github.com/ArduPilot/ardupilot`, accessed: 2022-04-13
5. Abbas, H., Fainekos, G., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Probabilistic Temporal Logic Falsification of Cyber-Physical Systems. ACM Trans. Embed. Comput. Syst. **12**(2s), 95:1–95:30 (2013). https://doi.org/10.1145/2465787.2465797, `https://doi.org/10.1145/2465787.2465797`
6. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The Algorithmic Analysis of Hybrid Systems. Theor. Comput. Sci. **138**(1), 3–34 (1995). https://doi.org/10.1016/0304-3975(94)00202-T, `https://doi.org/10.1016/0304-3975(94)00202-T`
7. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theor. Comput. Sci. **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8, `https://doi.org/10.1016/0304-3975(94)90010-8`
8. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to Runtime Verification. In: Lectures on Runtime Verification. Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457, pp. 1–33. Springer (Feb 2018). https://doi.org/10.1007/978-3-319-75632-5_1, `https://hal.inria.fr/hal-01762297`
9. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL Semantics for Runtime Verification. Journal of Logic and Computation **20**(3), 651–674 (02 2010). https://doi.org/10.1093/logcom/exn075, `https://doi.org/10.1093/logcom/exn075`
10. Bauer, A., Leucker, M., Schallhart, C.: Runtime Verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4) (sep 2011). https://doi.org/10.1145/2000799.2000800, `https://doi.org/10.1145/2000799.2000800`

11. Boufaied, C., Menghi, C., Bianculli, D., Briand, L., Parache, Y.I.: Trace-Checking Signal-Based Temporal Properties: A Model-Driven Approach. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. p. 1004–1015. ASE '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3324884.3416631, https://doi.org/10.1145/3324884.3416631
12. Bozzano, M., Bruintjes, H., Cimatti, A., Katoen, J.P., Noll, T., Tonetta, S.: COMPASS 3.0. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 379–385. Springer International Publishing, Cham (2019), https://doi.org/10.1007/978-3-030-17462-0_25
13. Bresolin, D.: HyLTL: a temporal logic for model checking hybrid systems. Electronic Proceedings in Theoretical Computer Science **124**, 73–84 (Aug 2013). https://doi.org/10.4204/eptcs.124.8, http://dx.doi.org/10.4204/EPTCS.124.8
14. Brim, L., Dluhos, P., Safránek, D., Vejpustek, T.: STL*: Extending signal temporal logic with signal-value freezing operator. Inf. Comput. **236**, 52–67 (2014). https://doi.org/10.1016/j.ic.2014.01.012, https://doi.org/10.1016/j.ic.2014.01.012
15. Dawes, J.H.: Towards Automated Performance Analysis of Programs by Runtime Verification (2021), https://cds.cern.ch/record/2766727
16. Dawes, J.H., Bianculli, D.: Specifying Properties over Inter-procedural, Source Code Level Behaviour of Programs. In: Feng, L., Fisman, D. (eds.) Runtime Verification. pp. 23–41. Springer International Publishing, Cham (2021), https://doi.org/10.1007/978-3-030-88494-9_2
17. Dawes, J.H., Reger, G.: Specification of Temporal Properties of Functions for Runtime Verification. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. pp. 2206–2214. SAC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3297280.3297497, https://doi.org/10.1145/3297280.3297497
18. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: Proceedings of the 21st International Conference on Software Engineering. p. 411–420. ICSE '99, Association for Computing Machinery, New York, NY, USA (1999). https://doi.org/10.1145/302405.302672, https://doi.org/10.1145/302405.302672
19. Fainekos, G., Hoxha, B., Sankaranarayanan, S.: Robustness of Specifications and Its Applications to Falsification, Parameter Mining, and Runtime Monitoring with S-TaLiRo. In: Finkbeiner, B., Mariani, L. (eds.) Runtime Verification. pp. 27–47. Springer International Publishing, Cham (2019), https://doi.org/10.1007/978-3-030-32079-9_3
20. Ferrère, T., Maler, O., Ničković, D.: Mixed-Time Signal Temporal Logic. In: André, É., Stoelinga, M. (eds.) Formal Modeling and Analysis of Timed Systems. pp. 59–75. Springer International Publishing, Cham (2019), https://doi.org/10.1007/978-3-030-29662-9_4
21. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: A Run-time Assurance Approach for Java Programs. Formal Methods in System Design **24**, 129–155 (03 2004). https://doi.org/10.1023/B:FORM.0000017719.43755.7c
22. Koymans, R.: Specifying Real-Time Properties with Metric Temporal Logic. Real-Time Syst. **2**(4), 255–299 (Oct 1990). https://doi.org/10.1007/BF01995674, https://doi.org/10.1007/BF01995674
23. Maler, O., Nickovic, D.: Monitoring Temporal Properties of Continuous Signals. In: Lakhnech, Y., Yovine, S. (eds.) Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems. pp. 152–166. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)

24. Menghi, C., Viganò, E., Bianculli, D., Briand, L.: Trace-Checking CPS Properties: Bridging the Cyber-Physical Gap. In: Proceedings of the 43rd International Conference on Software Engineering (ICSE'21), May 23–29, 2021, Virtual Event, Spain. pp. 847–859. IEEE, Los Alamitos, CA, USA (May 2021)
25. Pnueli, A.: The temporal logic of programs. In: 2013 IEEE 54th Annual Symposium on Foundations of Computer Science. pp. 46–57. IEEE Computer Society, Los Alamitos, CA, USA (oct 1977). https://doi.org/10.1109/SFCS.1977.32, https://doi.ieeecomputersociety.org/10.1109/SFCS.1977.32

A The eval and getVal functions

Here, we define the eval and getVal functions that are used by the semantics in Section 6. Intuitively, eval function identifies concrete states, transitions and timestamps from a trace, based on an expression and a valuation. The getVal function then extracts values from those objects, in order to determine the value indicated by a term in a specification.

A recursive definition of the eval function is given in Figure 4, and a recursive definition of the getVal function is given in Figure 5. In the following sections, we describe how these functions work.

A.1 The eval function

Since the eval function is responsible for identifying concrete states, transitions and timestamps, given expressions, this function is defined for each possible expression that can be generated by the grammar in Figure 1.

For example, the timestamp given by the expression $\textsc{time}(C)$ (with respect to a trace $\mathcal{T}$ and a valuation β) is obtained by 1) determining the unique concrete state identified by C; and 2) determining the timestamp held by that concrete state. This process requires a recursive call of eval on the expression C. Further, we also distinguish between the information being found in the trace, and not being found, by either returning the actual timestamp that is needed, or null.

A.2 The getVal function

Since the getVal function is responsible for extracting appropriate values from concrete states, transitions or timestamps, this function generally follows the pattern of 1) obtaining the relevant object using the eval function; and 2) accessing the relevant information in this object.

For example, suppose that we have the term $\mathsf{signal}.\textsc{at}(Ts)$, along with a trace $\mathcal{T}$ and a valuation β. The first step in determining a value for the term is to determine the timestamp to which the expression Ts evaluates under the valuation β. For this, we use the eval function. Once we have the relevant timestamp, we can refer to the value given by the signal signal in $\mathcal{T}$ at time $\mathsf{eval}(\mathcal{T}, \beta, Ts)$. Notice that, if eval gives null, then we must also evaluate the value of the overall term to null.

B Language comparison

We now justify the characterisation of specification languages given in Section 7.

LTL

$\mathcal{S}_{\mathsf{ts}}$ (×) The semantics of LTL assumes that traces are sequences of *untimed* atomic propositions; the only notion of order comes from the index that one can assign to each atomic proposition based on its position in the sequence. Hence, timestamps are not involved.

$$\mathsf{eval}(\mathcal{T}, \beta, ts) = \beta(ts)$$

$$\mathsf{eval}(\mathcal{T}, \beta, c) = \beta(c)$$

$$\mathsf{eval}(\mathcal{T}, \beta, tr) = \beta(tr)$$

$$\mathsf{eval}(\mathcal{T}, \beta, \textsc{time}(C)) = \begin{cases} \textsc{time}(\mathsf{eval}(\mathcal{T}, \beta, C)) & \mathsf{eval}(\mathcal{T}, \beta, C) \neq \mathsf{null} \\ \mathsf{null} & \text{otherwise} \end{cases}$$

$$\mathsf{eval}(\mathcal{T}, \beta, \textsc{time}(Tr)) = \begin{cases} \textsc{time}(\mathsf{eval}(\mathcal{T}, \beta, Tr)) & \mathsf{eval}(\mathcal{T}, \beta, Tr) \neq \mathsf{null} \\ \mathsf{null} & \text{otherwise} \end{cases}$$

$$\mathsf{eval}(\mathcal{T}, \beta, \textsc{before}(Tr)) = \begin{cases} s \text{ such that } \mathsf{eval}(\mathcal{T}, \beta, Tr)) = \langle s, s' \rangle & \mathsf{eval}(\mathcal{T}, \beta, Tr) \neq \mathsf{null} \\ \mathsf{null} & \text{otherwise} \end{cases}$$

$$\mathsf{eval}(\mathcal{T}, \beta, \textsc{after}(Tr)) = \begin{cases} s' \text{ such that } \mathsf{eval}(\mathcal{T}, \beta, Tr)) = \langle s, s' \rangle & \mathsf{eval}(\mathcal{T}, \beta, Tr) \neq \mathsf{null} \\ \mathsf{null} & \text{otherwise} \end{cases}$$

$\mathsf{eval}(\mathcal{T}, \beta, C.\textsc{next}(\mathsf{calls}(f).\mathsf{during}(p))) =$

> tr, if there is a tr such that:
> $\eta(\mathcal{T}, \beta, , tr, C)$ and $\mathcal{T}, tr \vdash \mathsf{calls}(f).\mathsf{during}(p)$
> and there is no tr' such that:
> $\textsc{time}(\mathsf{eval}(\mathcal{T}, \beta, C)) < \textsc{time}(tr') < \textsc{time}(tr)$ and $\mathcal{T}, tr' \vdash \mathsf{calls}(f).\mathsf{during}(p)$
>
> null otherwise

where $\eta(\mathcal{T}, \beta, tr, C) = (\textsc{time}(tr) \geq \textsc{time}(\mathsf{eval}(\mathcal{T}, \beta, C)))$

$\mathsf{eval}(\mathcal{T}, \beta, Tr.\textsc{next}(\mathsf{calls}(f).\mathsf{during}(p))) =$

> tr, if there is a tr such that:
> $\eta(\mathcal{T}, \beta, tr, Tr)$ and $\mathcal{T}, tr \vdash \mathsf{calls}(f).\mathsf{during}(p)$
> and there is no tr' such that:
> $\textsc{time}(\mathsf{eval}(\mathcal{T}, \beta, Tr)) < \textsc{time}(tr') < \textsc{time}(tr)$ and $\mathcal{T}, tr' \vdash \mathsf{calls}(f).\mathsf{during}(p)$
>
> null otherwise

where $\eta(\mathcal{T}, \beta, tr, Tr) =$

$$\begin{cases} \textsc{time}(tr) > \textsc{time}(\mathsf{eval}(\mathcal{T}, \beta, Tr)) & \mathcal{T}, \mathsf{eval}(\mathcal{T}, \beta, Tr) \vdash \mathsf{calls}(f).\mathsf{during}(p) \\ \textsc{time}(tr) \geq \textsc{time}(\mathsf{eval}(\mathcal{T}, \beta, Tr)) & \text{otherwise} \end{cases}$$

$\mathsf{eval}(\mathcal{T}, \beta, ts.\textsc{next}(\mathsf{calls}(f).\mathsf{during}(p))) =$

> tr, if there is a tr such that:
> $\eta(\mathcal{T}, \beta, tr, ts)$ and $\mathcal{T}, tr \vdash \mathsf{calls}(f).\mathsf{during}(p)$
> and there is no tr' such that:
> $\mathsf{eval}(\mathcal{T}, \beta, ts) < \textsc{time}(tr') < \textsc{time}(tr)$ and $\mathcal{T}, tr' \vdash \mathsf{calls}(f).\mathsf{during}(p)$
>
> null otherwise

where $\eta(\mathcal{T}, \beta, tr, ts) = (\textsc{time}(tr) \geq \mathsf{eval}(\mathcal{T}, \beta, ts))$

Fig. 4: The eval function for SCSL.

$$\mathsf{getVal}(\mathcal{T}, \beta, n) = n, \text{ a constant}$$

$$\mathsf{getVal}(\mathcal{T}, \beta, \textsc{time}(C)) = \begin{cases} \textsc{time}(\mathsf{eval}(\mathcal{T}, \beta, C)) & \mathsf{eval}(\mathcal{T}, \beta, C) \neq \mathsf{null} \\ \mathsf{null} & \text{otherwise} \end{cases}$$

$$\mathsf{getVal}(\mathcal{T}, \beta, \textsc{time}(Tr)) = \begin{cases} \textsc{time}(\mathsf{eval}(\mathcal{T}, \beta, Tr)) & \mathsf{eval}(\mathcal{T}, \beta, Tr) \neq \mathsf{null} \\ \mathsf{null} & \text{otherwise} \end{cases}$$

$$\mathsf{getVal}(\mathcal{T}, \beta, \mathsf{signal}.\textsc{at}(Ts)) = \begin{cases} \mathcal{T}(\mathsf{signal})(\mathsf{eval}(\mathcal{T}, \beta, Ts)) & \mathsf{eval}(\mathcal{T}, \beta, Ts) \neq \mathsf{null} \\ \mathsf{null} & \text{otherwise} \end{cases}$$

$$\mathsf{getVal}(\mathcal{T}, \beta, C(x)) = \begin{cases} \mathsf{eval}(\mathcal{T}, \beta, C)(x) & \mathsf{eval}(\mathcal{T}, \beta, C) \neq \mathsf{null} \\ \mathsf{null} & \text{otherwise} \end{cases}$$

$$\mathsf{getVal}(\mathcal{T}, \beta, \textsc{duration}(Tr)) = \begin{cases} \textsc{duration}(\mathsf{eval}(\mathcal{T}, \beta, Tr)) & \mathsf{eval}(\mathcal{T}, \beta, Tr) \neq \mathsf{null} \\ \mathsf{null} & \text{otherwise} \end{cases}$$

$$\mathsf{getVal}(\mathcal{T}, \beta, \textsc{TimeBetween}(Ts_1, Ts_2)) =$$

$$\begin{cases} \mathsf{eval}(\mathcal{T}, \beta, Ts_2) - \mathsf{eval}(\mathcal{T}, \beta, Ts_1) & \mathsf{eval}(\mathcal{T}, \beta, Ts_1) \neq \mathsf{null} \text{ and} \\ & \quad \mathsf{eval}(\mathcal{T}, \beta, Ts_2) \neq \mathsf{null} \\ \mathsf{null} & \text{otherwise} \end{cases}$$

Fig. 5: The getVal function for SCSL.

$\mathcal{S}_{\mathsf{index}}$ (✓) The LTL semantics is purely index-based, since this is the only information held in traces that can be used to infer an ordering.

$\mathcal{X}_{\mathsf{sig}}$ (×) LTL does not provide specific syntax for dealing with signals; it only deals with atomic propositions.

$\mathcal{X}_{\mathsf{code}}$ (×) While it was introduced for expressing properties over program behaviour, LTL does not provide specific syntax for source code level properties. Such properties must be captured by defining a correspondence between runtime events and atomic propositions.

$\mathcal{X}_{ll}$ (×) LTL uses atomic propositions, therefore LTL specifications have a high level of abstraction with respect to the system whose behaviour they describe.

$\mathcal{X}_{\mathsf{het}}$ (×) LTL does not explicitly support expression of properties concerning the behaviour of heterogeneous systems.

$\mathcal{X}_{\mathsf{index}}$ (×) While LTL's semantics considers indices, one cannot refer directly to these indices in specifications.

$\mathcal{X}_{\mathsf{ts}}$ (×) LTL does not deal with timestamps, so one cannot refer directly to timestamps (or events in traces at those timestamps).

$\mathcal{X}_{\mathsf{metric}}$ (×) LTL does not provide temporal operators with metrics attached, for example, to capture the property "*b should be true within at most 5 indices of a being true*".

MTL

$\mathcal{S}_{\mathsf{ts}}$ (✓) The semantics of MTL assumes that traces are *timed words*, that is, each entry in a trace is a combination of atomic propositions and timestamps.

$\mathcal{S}_{\mathsf{index}}$ (×) The MTL semantics is purely timestamp-based, using timestamps to refer to *moments* in which a set of atomic propositions are true.

$\mathcal{X}_{\mathsf{sig}}$ (×) MTL does not provide specific syntax for dealing with signals; it only deals with atomic propositions.

$\mathcal{X}_{\mathsf{code}}$ (×) MTL does not provide specific syntax for source code level properties. Such properties must be captured by defining a correspondence between runtime events and atomic propositions.

$\mathcal{X}_{ll}$ (×) MTL uses atomic propositions, therefore MTL specifications have a high level of abstraction with respect to the system whose behaviour they describe.

$\mathcal{X}_{\mathsf{het}}$ (×) MTL does not explicitly support expression of properties concerning the behaviour of heterogeneous systems.

$\mathcal{X}_{\mathsf{index}}$ (×) MTL does not allow one to refer directly to indices and use these to refer to events in a trace.

$\mathcal{X}_{\mathsf{ts}}$ (×) MTL assumes that traces contain timing information, but this cannot be obtained directly and cannot be used to refer to events in a trace.

$\mathcal{X}_{\mathsf{metric}}$ (✓) MTL provides metric temporal operators that allow one to not only specify modal constraints on sequences of events, such as "*b appears eventually if a appears*", but also to put a constraint on *when b* should appear.

TLTL

$\mathcal{S}_{\mathsf{ts}}$ (×) The semantics of TLTL assumes that traces are *timed words*, that is, each entry in a trace is a combination of atomic propositions and timestamps. However, these timestamps are only used in combination with clock variables, and not as a way to order events in the semantics.

$\mathcal{S}_{\mathsf{index}}$ (✓) The TLTL semantics is purely index-based, with timestamps held in timed words only being used by metric temporal operators.

$\mathcal{X}_{\mathsf{sig}}$ (×) TLTL does not provide specific syntax for dealing with signals; it only deals with atomic propositions.

$\mathcal{X}_{\mathsf{code}}$ (×) TLTL does not provide specific syntax for source code level properties. Such properties must be captured by defining a correspondence between runtime events and atomic propositions.

$\mathcal{X}_{ll}$ (×) TLTL uses atomic propositions, therefore TLTL specifications have a high level of abstraction with respect to the system whose behaviour they describe.

$\mathcal{X}_{\mathsf{het}}$ (×) TLTL does not explicitly support expression of properties concerning the behaviour of heterogeneous systems.

$\mathcal{X}_{\mathsf{index}}$ (×) TLTL does not allow one to refer directly to indices and use these to refer to events in a trace.

$\mathcal{X}_{\mathsf{ts}}$ (✓) TLTL assumes that traces contain timing information, and allows one to obtain the timestamp at which an atomic proposition was most recently true, or will next be true.

$\mathcal{X}_{\mathsf{metric}}$ (×) TLTL does not provide metric temporal operators. Instead, it provides access to *clock variables* in syntax, which enables the expression of timing constraints by referring to the value of clock variables.

HyLTL

$\mathcal{S}_{\mathsf{ts}}$ (×) The semantics of HyLTL assumes that traces are infinite sequences that combine discrete and continuous behaviour, but still uses indices to refer to each instance of behaviour.

$\mathcal{S}_{\mathsf{index}}$ (✓) The HyLTL semantics is purely index-based, with timestamps only being used to check whether the continuous component of the system being checked satisfies some *flow constraints*.

$\mathcal{X}_{\mathsf{sig}}$ (✓) HyLTL enables one to construct constraints over *trajectories*, which describe the behaviour of the continuous component of a system. Since these trajectories could often be signals, we say that HyLTL does indeed provide signal-specific syntax.

$\mathcal{X}_{\mathsf{code}}$ (×) HyLTL was introduced to extend LTL to the hybrid system setting. The extension is performed assuming a higher level of abstraction than source code.

$\mathcal{X}_{ll}$ (×) HyLTL uses atomic propositions, therefore HyLTL specifications have a high level of abstraction with respect to the system whose behaviour they describe.

$\mathcal{X}_{\mathsf{het}}$ (✓) HyLTL supports expression of properties concerning the behaviour of both continuous and discrete behaviour of hybrid systems.

$\mathcal{X}_{\mathsf{index}}$ (×) HyLTL does not provide syntax to refer directly to the index of an event in a trace.

$\mathcal{X}_{\mathsf{ts}}$ (×) HyLTL does not provide syntax to refer directly to timestamps associated with any event in a system.

$\mathcal{X}_{\mathsf{metric}}$ (×) HyLTL does not provide metric temporal operators, only providing the same temporal operators as those provided by LTL.

STL, STL* and STL-MX

$\mathcal{S}_{\mathsf{ts}}$ (✓) The semantics of STL (and each variant) assumes real-valued functions, and performs evaluation by referring to each timestamp at which these functions are defined. STL-MX combines dense and discrete time, but ultimately still assumes signals to be real-valued functions.

$\mathcal{S}_{\mathsf{index}}$ **(mixed)** The semantics of STL and STL* are purely timestamp-based. The semantics of STL-MX are both timestamp and index-based.

$\mathcal{X}_{\mathsf{sig}}$ (✓) STL and its variants provide syntax to refer to values of real-valued functions. If these functions are seen as representing signals, then one can say that STL and its variants support signals.

$\mathcal{X}_{\mathsf{code}}$ (×) Neither STL nor its variants provide the ability to refer to events in source code.

$\mathcal{X}_{ll}$ (✓) All three variants can be used to capture properties directly over signals, so we say that they have a low level of abstraction. However, it should be noted that STL-MX supports heterogeneity (via support for dense and discrete time), but the syntax used for discrete time is taken from LTL.

$\mathcal{X}_{\mathsf{het}}$ **(mixed)** Neither STL nor STL* explicitly enables the expression of properties concerning behaviour of multiple types of components (e.g., cyber and physical components). However, STL-MX enables expression of properties concerning systems with both cyber (discrete time) and physical (dense time) components.

$\mathcal{X}_{\mathsf{index}}$ (×) None of the variants of STL that we consider deal with indices of events; values of real-valued functions are referred to via timestamps (which are not explicit, instead being computed by the semantics).

$\mathcal{X}_{\mathsf{ts}}$ (×) Despite each of the variants being considered working with real-valued functions, there is no syntax provided for explicitly referring to timestamps.

$\mathcal{X}_{\mathsf{metric}}$ (✓) All variants of STL provide metric temporal operators.

HLS

$\mathcal{S}_{\mathsf{ts}}$ (✓) The semantics of HLS involves recursing over the structure of a specification, while maintaining a map from variables used in quantifiers to timestamps or indices. Hence, one can say that HLS uses timestamps in its semantics.

$\mathcal{S}_{\mathsf{index}}$ (✓) HLS allows quantification over indices.

$\mathcal{X}_{\mathsf{sig}}$ (✓) HLS provides syntax specific to referring to signals at both indices and timestamps.

$\mathcal{X}_{\mathsf{code}}$ (×) While HLS provides indices to enable one to reference the behaviour of cyber components, it does not allow explicit references to source code.

$\mathcal{X}_{ll}$ (✓) Since HLS assumes traces to be signals, we say that it has a low level of abstraction with respect to signal-based behaviour. However, it should be noted that HLS has a high level of abstraction with respect to the behaviour of cyber components (that it indirectly supports via indices), because that behaviour must be encoded into signals.

$\mathcal{X}_{\mathsf{het}}$ (✓) HLS enables timestamp-based referencing of events generated by physical components, and index-based referencing of events generated by cyber components.

$\mathcal{X}_{\mathsf{index}}$ (✓) HLS provides access to indices, either via quantifiers or via accessing the indices of individual records in a trace.

$\mathcal{X}_{\mathsf{ts}}$ (✓) HLS provides access to timestamps, either via quantifiers or via accessing the timestamps of individual records in a trace.

$\mathcal{X}_{\mathsf{metric}}$ (×) HLS does not provide modal operators, hence does not provide model operators with metrics.

SB-TemPsy-DSL

$\mathcal{S}_{\mathsf{ts}}$ (✓) The semantics of SB-TemPsy-DSL is defined based on intervals of timestamps.

$\mathcal{S}_{\mathsf{index}}$ (×) The semantics of SB-TemPsy-DSL is defined based on intervals of timestamps, so does not refer to indices.

$\mathcal{X}_{\mathsf{sig}}$ (✓) SB-TemPsy-DSL is a language for expressing properties over signals, so provides syntax specific to signals.

$\mathcal{X}_{\mathsf{code}}$ (×) SB-TemPsy-DSL does not provide any specific syntax for source code but, as with other purely signal-based languages, one could use it to express such properties by mapping events generated by source code at runtime onto Boolean signals.

$\mathcal{X}_{ll}$ (✓) SB-TemPsy-DSL assumes traces to be signals, and provides syntax such as `exists spike in` s (for some signal s) for capturing specific properties of signal-based behaviour. Hence, given that it is a domain-specific language, we say that it has a low level of abstraction.

$\mathcal{X}_{\mathsf{het}}$ (×) SB-TemPsy-DSL does not provide explicit support for heterogeneous systems; its focus is signals. Though one can place constraints over multiple signals, the focus must still always be on signals.

$\mathcal{X}_{\mathsf{index}}$ (×) SB-TemPsy-DSL does not provide access to indices.

$\mathcal{X}_{\mathsf{ts}}$ (✓) SB-TemPsy-DSL provides explicit access to timestamps in its syntax. Because of the intrinsic restriction due to the pattern-based syntax, timestamps can only be referred to in the context of a (specific type of) scope.

$\mathcal{X}_{\mathsf{metric}}$ (✓) SB-TemPsy-DSL does not provide temporal operators, therefore also does not provide metric temporal operators. However, we say that SB-TemPsy-DSL does indeed use metrics because of its `if` p `then` ... syntax, where `then` can be followed by a `within` operator. The options for this operator include further syntax such as `exactly`, `at most` and `at least`. The presence of this syntax gives one the ability to assert that some behaviour in the signal should be observed, subject to some timing constraints. Hence, this is a form of metric.

SCSL

$\mathcal{S}_{\mathsf{ts}}$ (✓) The semantics of SCSL involves recursing over the structure of a specification and building up a map from variables in quantifiers, to timestamps, concrete states, or transitions. Hence, the semantics can be said to involve timestamps.

$\mathcal{S}_{\mathsf{index}}$ (×) The semantics of SCSL does not use indices to refer to events in a trace.

$\mathcal{X}_{\mathsf{sig}}$ (✓) SCSL provides syntax specific to expressing constraints over signals.

$\mathcal{X}_{\mathsf{code}}$ (✓) SCSL provides syntax specific to expressing constraints over events generated by source code.

$\mathcal{X}_{ll}$ (✓) SCSL specifications refer directly to the events that they consider, so no correspondence between runtime events and symbols in the specification need be defined. Hence, we say that SCSL specifications have a low level of abstraction.

$\mathcal{X}_{\mathsf{het}}$ (✓) SCSL provides syntax for capturing properties concerning both the behaviour of signals and the behaviour of source code-based components. Hence, we say that SCSL supports heterogeneous behaviour.

$\mathcal{X}_{\mathsf{index}}$ (×) SCSL does not provide syntax for referring directly to indices.

$\mathcal{X}_{\mathsf{ts}}$ (✓) SCSL provides syntax specifically for referring to timestamps (including quantifiers and operators for obtaining the timestamp of a given concrete state or transition in a trace).

$\mathcal{X}_{\mathsf{metric}}$ (×) SCSL does not provide the temporal operators seen in other languages, such as LTL and MTL, hence does not provide metric temporal operators.