MODMED

WP1/D2 & WP1/D3: Complete Definition of a Domain Specific Specification Language

Version 1.1
May 3, 2018

MODMED (ANR-15-CE25-0010) 2015-2018

Yoann Blein	LIG	Author
Lydie du Bousquet	LIG	Reviewer
Roland Groz	LIG	Reviewer
Yves Ledru	LIG	Reviewer
Fabrice Bertrand	BlueOrtho	Reviewer
Arnaud Clère	MinMaxMedical	Reviewer

Contents

1 Introduction

Medical Cyber-Physical Systems (MCPS) provide support for complex medical interventions. Despite their increasing complexity, the MCPS industry is reluctant to use formal methods [Lee08]. Fortunately, these systems can be easily instrumented to provide execution traces enabling us to understand their use and behavior in the field. Based on this observation, the MODMED initiative strive at developing an environment for partial Model-Based Verification of execution traces for MCPS.

The properties to verify can be written in a specification language and analyzed automatically on a corpus of execution traces. Such an environment presents several interests to MCPS manufacturers:

- validating the correctness and the robustness of an implementation when used in real conditions,
- validating the hypotheses made on the environment and the conditions of use of a device, and
- understanding the way a device is used in a perspective of product enhancement.

One of the challenges faced by MODMED is to make formal requirements writable by software engineers with no training in formal methods and readable by domain experts. For this purpose, we are developing a high-level language dedicated to property specification for MCPS. This document presents a preliminary definition of ParTraP (Parametric Trace Property language) based on the ExactechGPS-TKA case study, which is further detailed in [BBdB+16].

2 The ExactechGPS-TKA Case Study

This case study focuses on a computer assisted navigation system for total knee arthroplasty: ExactechGPS-TKA, designed BlueOrtho company for Exactech implant manufacturer. Total knee arthroplasty is a surgical procedure that involves replacing parts of the knee joint with a prosthesis. The purpose of this operation is to relieve the pain of an arthritic knee, while maintaining or improving its functionality. To install the prosthesis, it is necessary to cut off part of the tibia and the femur. ExactechGPS-TKA helps the surgeon achieve these cuts precisely through the guided installation of cutting guides in the *right* position. This position is automatically determined by combining the target objective given by the surgeon and the spatial reconstruction of the scene established by the system. This system is currently used worldwide.

ExactechGPS-TKA is a turnkey solution that provides both the knee prosthesis and the guidance system. The latter consists of several components: a machine able to communicate with the surgeon via a touch screen, a three dimensions camera, a set of trackers visible by the camera and a set of mechanical instruments for attaching trackers and cutting guides to the tibia and the femur.

The installation of cutting guides is carried out through a succession of steps to be performed by the surgeon. The nature of these steps and the order in which they are performed are, to some extent, configurable. This configuration, called *profile*, is determined according to the surgeon's operating preferences. In every case, the sequence of steps takes the following macroscopic form: sensor calibration, acquisition of anatomical points, checking acquisitions, adjustment of target parameters, and finally, cutting guides setting.

ExactechGPS-TKA is equipped with a recording system for execution trace. For each performed surgery, the corresponding execution trace is sent to BlueOrtho. To this day, the company collected a corpus of about 7,000 traces of surgeries that took place in real conditions. Each trace consists of an event log, a hierarchical description of the stages of surgery containing the acquired and calculated values, and all the screenshots performed at each stage. This set of information allows understanding the course of a surgery and possibly identifying failures.

2.1 Properties identified

As expected with a medical cyber-physical system, the range of properties collected in the ExactechGPS-TKA case study is very wide. In this section, we present a set of properties that we identified as representative. Here is the list of the 15 properties in no particular order:

Property 1: The trace contains a step "redo acquisitions".

The "redo acquisition" step allows the surgeon to correct his previous acquisition. It is not part of the standard procedure flow and, therefore, interesting to detect.

Property 2: The temperature of the camera stays within $[l, u]$.

If used in proper conditions, the camera temperature should not deviate from the range where its precision is guaranteed.

Property 3: The distance between pairs of hip centers is less than d.

This property asserts that the algorithm computing the hip center is stable.

Property 4: The distance between the hip center and the knee center is greater than d.

A violation of this property could reveal an abnormal positioning of the patient or the sensors.

Property 5: If the medial malleolus is further than the lateral one, a warning is issued.

A violation of this property could reveal that the 3D camera was installed on the wrong side of the patient.

Property 6: The user never skips a screen.

The surgeon is expected to spend sufficient time to appreciate the information showed at each step of the procedure.

Property 7: The acquisition of a point succeeds if and only if the probe is stable.

If the surgeon moves the probe during an acquisition, it should not be accepted.

Property 8: The protocol "redo acquisitions" proposes only already performed acquisitions.

The system should not offer the user to redo acquisitions that were never performed.

Property 9: Detecting a new tracker produces a dialog asking for replacement confirmation.

Property 10: The state `TrackersConnection` is unreachable until the camera is connected.

The system should not reach a state dependant on the camera until the camera is connected.

Property 11: A replaced tracker is not used until it is registered again.

If a tracker is replaced, the system should not try to use it until it is registered again.

Property 12: The action "previous" cancels the current points cloud acquisition.

Acquiring a points cloud takes a few seconds and can be cancelled. In this case, the current acquisition should not succeed.

Property 13: All the necessary trackers are seen before entering the state `TrackersVisibCheck`.

To proceed, the system requires a set of trackers depending on the profile in use. All these trackers should be seen at least once before entering the state `TrackersVisibCheck`. Note that if we go back to the beginning and change the profile, the trackers already seen do not have to be seen again.

Property 14: On the trackers connection screen, a tracker is shown if and only if it is necessary.

Only the set of required trackers is shown to the user.

Property 15: In the state `TrackersConnection`, not detecting any new tracker for 2 minutes produces an error message.

2.2 Analysis

A property can be characterized by the number of different events it involves and whether it:

- is *parametric*, *i.e.* it involves event parameters,
- is *temporal*, *i.e.* it constrains the order of two or more occurrences of events,
- applies to a restricted scope of the trace,
- has geometric predicates on data,
- has GUI predicates on screenshots, and
- involves physical-time.

Table 1 summarizes the characteristics of each property according to the previous criteria.

Property	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
No. of event types	1	1	1	2	3	2	2	2	2	2	3	3	4	2	4
Parametric		✓	✓	✓	✓	✓	✓	✓	✓		✓		✓	✓	✓
Temporal						✓	✓	✓	✓	✓	✓	✓	✓		✓
Restricted scope								✓	✓	✓	✓		✓		✓
Geometric predicate			✓	✓	✓	✓									
GUI predicate														✓	
Physical time						✓									✓

Table 1: Characterization of the selected properties

First, we observe that nearly all properties involve at least two different types of events and that the parameters of the events are heavily used. This confirms that the properties are not simple invariance constraints and there is a need for an expressive language to formalize them.

The key observation is that most of the properties are temporal. Therefore, the DSL should focus on the temporal relations between events. Non-temporal properties (invariance, occurrence, ...) should be expressible as degenerate cases. Scope restrictions are also common and should be coupled with temporal relations in the DSL.

```
[
  { "time": 1, "id": "TrackerDetected", "type": "F" },
  { "time": 4, "id": "SearchTrackers", "types": ["P", "F"] },
  { "time": 6, "id": "TrackerDetected", "type": "P" },
  { "time": 9, "id": "StartAcquisitions" }
]
```

Figure 1: Example of JSON input trace

Then, we notice that one third of the properties relies on geometry or GUI predicates. Thus the language should provide a mechanism to call external predicates defined by other means, such as a Python or C++ library.

Finally, despite our expectations, the physical time is rarely involved in the properties. However, we still believe that supporting it would be useful for applications beyond this case study.

3 WP1/D2: ParTraP Definition

3.1 Basic Concepts: Event, Traces and Environment

The traces are formatted as JSON[1] files, where the top-level element is an array of records, representing events. Each event is composed of a mandatory name and time keys, and can have other parameters. Figure 1 shows a simple example of such trace. In this section, we formalize this format after introducing some notations.

We use $X \to Y$ and $X \rightharpoonup Y$ to denote sets of total and partial functions from X to Y, respectively. We write maps (partial functions) as $[x_0 \mapsto v_0, \ldots, x_i \mapsto v_i]$ and the empty map as $[\,]$. We note $m[x \mapsto v]$ the map which is the same as m except that the mapping for x is updated to refer to v:

$$m[y \mapsto v](x) = \begin{cases} v & \text{if } x = y \\ m(x) & \text{otherwise} \end{cases}$$

If Z is a set, let Z^* be the set of *finite* sequences of elements in Z.

Equipped with these notations, we may now define the traces content and format. The set of *values* is the smallest set Val such that:

1. atomic literals (booleans, integers, strings and floating-point numbers) are values;
2. if $v_1, \ldots, v_n$ are values, then the sequence $(v_k)_{k=1}^n$ is a value;
3. if $v_1, \ldots, v_n$ are values and $f_1, \ldots, f_n$ are names, then the map, or record, $[f_0 \mapsto v_0, \ldots, f_i \mapsto v_i]$ is a value.

An *environment* is a map from variable names to values:

$$\text{Env} = \text{Var} \rightharpoonup \text{Val},$$

where Var is a set of variable names.

[1] `https://tools.ietf.org/html/rfc7159`

An *event* is characterized by a *name* and a set of named *parameters*. Formally an event is defined as a pair:

$$\text{Event} = \Sigma \times (P \rightharpoonup \text{Val}),$$

where Σ and P are finite sets of event names and parameter names, respectively. Note that definition permits events to have the same name and yet different parameters. This provide more flexibility with the input traces and allows, for instance, to have optional parameters. For convenience, we define the two following projections on events: $\text{name}(\langle n, ps\rangle) = n$ and $\text{param}(\langle n, ps\rangle) = ps$.

Finally, a *trace* $\tau = (e_i, t_i)_{1=i}^{n}$ is an element of $(\text{Event} \times \mathbb{N})^*$, where $(t_i)_{1=i}^{n}$ is a non-decreasing sequence representing time, and

$$\text{untime}((e_1, t_1) \dots (e_n, t_n)) = e_1 \dots e_n.$$

3.2 Simple Expressions

It is useful to be able to manipulate simple to moderately complex expressions in properties, while delegating more complex expressions to an external language (Python, C++, ...) through a Foreign Function Interface (FFI). Since simple expressions are orthogonal to the definition of the DSL for property specification, we will first introduce them in this dedicated section.

Let Expr be the set of expressions that can derived from the grammar in Figure 2.

⟨*expr*⟩	::=	⟨*literal*⟩	
		| ⟨*unop*⟩ ⟨*expr*⟩	
		| ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩	
		| '`(`' ⟨*expr*⟩ '`)`'	
		| ⟨*ident*⟩	(variable lookup)
		| ⟨*expr*⟩ '`.`' ⟨*ident*⟩	(record field access)
		| ⟨*expr*⟩ '`[`' ⟨*expr*⟩ '`]`'	(sequence indexing)
		| ⟨*ident*⟩ '`(`' ⟨*args*⟩ '`)`'	(external function call)
⟨*unop*⟩	::=	'`not`' | '`-`'	
⟨*binop*⟩	::=	'`<`' | '`<=`' | '`==`' | '`>`' | '`>=`' | '`!=`' | '`&&`' | '`||`' | '`+`' | '`-`' | '`*`' | '`/`' | '`%`'	
⟨*args*⟩	::=	[(⟨*expr*⟩ '`,`')* ⟨*expr*⟩]	
⟨*ident*⟩	::=	(_ | ⟨*letter*⟩) (_ | ⟨*letter*⟩ | ⟨*digit*⟩)*	
⟨*letter*⟩	::=	'`a`' | ... | '`z`' | '`A`' | ... | '`Z`'	
⟨*digit*⟩	::=	'`0`' | ... | '`9`'	
⟨*literal*⟩	::=	usual set of boolean, integer, floating-point and string literals	

Figure 2: Syntax of expressions

To formalize expression evaluation, we will use the judgement form $\eta \vdash e \downarrow v$, read as "in the environment η, expression e reduces to value v". Evaluation of basic expressions (literals, unary expressions and binary expressions) is defined as usual and not detailed here. The interesting

rules are the following:

$$\frac{\eta(x) = v}{\eta \vdash x \downarrow v}\ \text{E-Lookup}$$

$$\frac{\eta \vdash e \downarrow r \quad r(p) = v}{\eta \vdash e.p \downarrow v}\ \text{E-FieldAccess}$$

$$\frac{\eta \vdash e_1 \downarrow s \quad \eta \vdash e_2 \downarrow i \quad s_i = v}{\eta \vdash e_1[e_2] \downarrow v}\ \text{E-Indexing}$$

$$\frac{\eta \vdash e_1 \downarrow v_1 \quad \ldots \quad \eta \vdash e_n \downarrow v_n \quad \eta \vdash f(v_1, \ldots, v_n) \downarrow v}{\eta \vdash f(e_1, \ldots, e_n) \downarrow v}\ \text{E-ExtCall}$$

The rule E-Lookup resolves variables names to their values in the current environment, E-FieldAccess accesses the value associated a key in a record, and E-Indexing addresses the element at a given index in a sequence. The last rule, E-ExtCall, handles external function calls with a *call-by-value* strategy.

3.3 DSL Definition

The proposed DSL is an event-based formalism with a natural language syntax. The proposed DSL is mostly influenced by the specification patterns proposed by Dwyer *et al* [DAC99]. However, we extended their expressiveness significantly by allowing them to be combined and nested, and to operate on parametrized events.

3.3.1 Syntax

The syntax of the proposed DSL is described by the grammar in Figure 3.

The following expressions are typical examples that can be derived from this grammar:

- `after each A, B followed_by C`
- `after first A a where a.x != 0, absence_of B or absence_of C"`
- `before last A a, forall v in a.set, occurrence_of 2 B where b.p == v`
- `between A a and B b where a.v == b.v, not (C preceded_by D)`
- `within 2min before first A, B prevents C for 2s`

3.3.2 Informal semantics

Events Events are designed simply by their type such as in `absence_of A` where `A` is the type of the event. Additionally, an event can be bound to a variable `x` like in `A x`. In this case, it is also possible to add a condition on the event thanks to the `where` construct: `A x where c`. This expression describes the set of events having the type `A` and respecting the condition `c` when bound to the variable `x`.

It is also possible to designate an unordered collections of events with the `set` construct: `set(E1 x1, ..., En xn) where c`. This event will be triggered at the last event of any set of events `E1...En` that respects the guard `c` when the events `E1...En` are bound to `x1...xn`, respectively.

⟨*prop*⟩ ::= ⟨*pattern*⟩
| ⟨*scope*⟩ ‘`,`’ ⟨*prop*⟩
| (‘`forall`’ | ‘`exists`’) ⟨*ident*⟩ ‘`in`’ ⟨*expr*⟩ ‘`,`’ ⟨*prop*⟩
| ‘`given`’ (‘`each`’ | ‘`first`’ | ‘`last`’) ⟨*event*⟩ ‘`,`’ ⟨*prop*⟩
| ‘`(`’ ⟨*prop*⟩ ‘`)`’
| ‘`not`’ ⟨*prop*⟩
| ⟨*prop*⟩ (‘`and`’ | ‘`or`’ | ‘`equiv`’ | ‘`implies`’) ⟨*prop*⟩

⟨*scope*⟩ ::= [‘`within`’ ⟨*duration*⟩] (‘`after`’ | ‘`before`’) (‘`each`’ | ‘`first`’ | ‘`last`’) ⟨*event*⟩
| ‘`between`’ ⟨*event*⟩ ‘`and`’ ⟨*event*⟩
| ‘`since`’ ⟨*event*⟩ ‘`until`’ ⟨*event*⟩

⟨*pattern*⟩ ::= ‘`absence_of`’ ⟨*event*⟩
| ‘`occurrence_of`’ ⟨*expr*⟩ ⟨*event*⟩
| ⟨*event*⟩ ‘`followed_by`’ ⟨*event*⟩ [‘`within`’ ⟨*duration*⟩]
| ⟨*event*⟩ ‘`preceded_by`’ ⟨*event*⟩ [‘`within`’ ⟨*duration*⟩]
| ⟨*event*⟩ ‘`prevents`’ ⟨*event*⟩ [‘`for`’ ⟨*duration*⟩]

⟨*event*⟩ ::= ⟨*ident*⟩ [⟨*ident*⟩ [‘`where`’ ⟨*expr*⟩]]
| ‘`set`’ ‘`(`’ ⟨*ident*⟩ [⟨*ident*⟩] (‘`,`’ ⟨*ident*⟩ [⟨*ident*⟩])* ‘`)`’ [‘`where`’ ⟨*expr*⟩]

⟨*duration*⟩ ::= ⟨*expr*⟩ (‘`ms`’ | ‘`s`’ | ‘`min`’ | ‘`h`’ | ‘`d`’)

Figure 3: Syntax of the proposed DSL

Patterns A pattern is the mandatory basic component of a property. It is impossible to derive a property that does not contain at least one pattern. They describe and rule the occurrences of events in the current scope of the trace.

The first unary pattern is `occurrence_of n A`, where `n` is optional, and which requires the occurrence of *at least* `n` events `A` in the current scope. If `n` is not specified, it defaults to `1`, as expected. The second unary pattern is simply the dual: `absence_of A`.

There are also three binary patterns: `A followed_by B`, `A preceded_by B` and `A prevents B`. `A followed_by B` holds if and only if every occurrence of the event `A` is eventually *followed by* an occurrence of the event `B`. Conversely, `A preceded_by B` holds if and only if each occurrence of the event `A` is eventually preceded by an occurrence of the event `B`. Finally, `A prevents B` prevents the event `B` from occurring after an occurrence of the event `A`. Examples of pattern satisfaction for various traces are given in Table 2.

	$\langle A \rangle$	$\langle B \rangle$	$\langle A, A, C, B \rangle$	$\langle B, A \rangle$	$\langle A, B, A \rangle$
`absence_of A`	✗	✓	✗	✗	✗
`occurrence_of A`	✓	✗	✓	✓	✓
`occurrence_of 2 A`	✗	✗	✓	✗	✓
`A followed_by B`	✗	✓	✓	✗	✗
`B preceded_by A`	✓	✗	✓	✗	✓
`A prevents B`	✓	✓	✗	✓	✗

Table 2: Examples of pattern satisfaction for various traces

Scopes Scopes are a mean to designate the range of a trace where a property should hold. They are delimited by optionally bound events. If a delimiter event is bound, it will be made available in the environment under the bound name for the current property. For instance, the property `after first A v where v.x != 0, P` will evaluate `P` on the scope starting after the first event `A` with a non-null `x` parameter, and in a environment where the value of the variable `v` is this first event.

The scopes we propose can be classified according to their arity, *i.e.* the number of events types they expect. Unary scopes, illustrated Figure 4, are the basic building blocks.

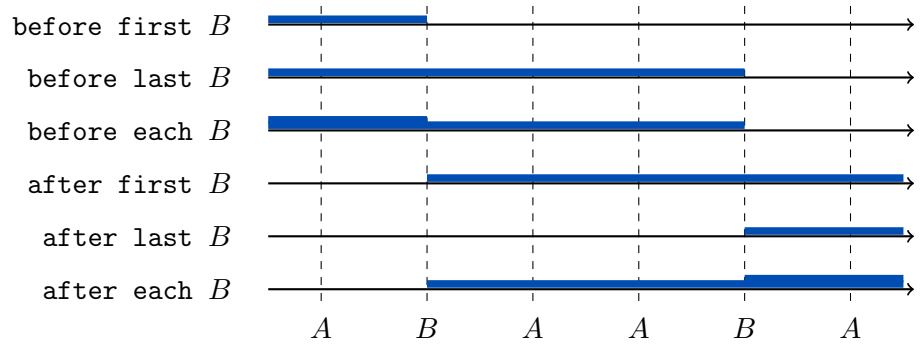

Figure 4: Graphical representation of the unary scopes

Note that all scopes are strict, i.e., delimiter events are not included in the interval they define. Although most of them are trivial, the "each" variants may describe multiple intervals. They happen to be powerful combinators.

An important consequence of the grammar definition is that scopes can be nested. For instance, `after last A, after each B, P` will hold if and only if `P` holds after each `B` occurring after the last occurrence of `A`. Nesting scopes properly allows defining more abstract scopes such as the two binary scopes illustrated Figure 5.

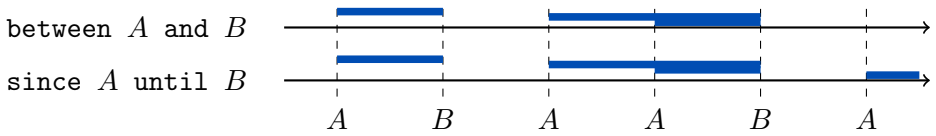

Figure 5: Graphical representation of the binary scopes

Binary scopes can be defined solely in terms of the previous unary scopes and, therefore, could be omitted from the language definition. However, we decided to include them considering that their definitions are difficult to read (see section 3.3.3), and to allow for further optimizations in their implementation.

Although we considered adding a "global" scope, we chose not to, since a property without scope restriction already implicitly refers to the whole trace.

Timed Variants Unary scopes and binary patterns may be additionally constrained with a duration expressed in common time units.

Both unary scopes can be prefixed with the `within` keyword and a duration expression, like in `within 2ms before each A, absence_of B`. The inner property only has to hold for the

given duration starting immediately at the delimiter event for the `after` scope, or ending exactly at the delimiter event in the `before` case. In the previous example, the event B cannot occur during the two milliseconds preceding any occurrence of an event A.

Binary patterns, built upon unary scopes, may also be extended with a suffix and a duration expression. For instance, the response pattern becomes bounded in time: `A followed_by B within 2s`.

Quantifiers The trace format allows compound values in event parameters. In particular, they can be lists of values. The language allows exploiting them through quantified properties.

The universal quantifier takes the following form: `forall a in L, P`, where `a` is an identifier and `L` is a list. Unsurprisingly, for each value in `L`, `P` must be satisfied in an environment where `a` is bound to that value. The existential quantifier (`exists`) is also defined as usual.

Since quantified properties are themselves properties, they can be arbitrarily nested. In particular, it is convenient to use a quantifier inside a scope property, so that the list quantified over can be a parameter of the event delimiting the scope.

Event Selection The only way to extract the parameters of an event so far is to bind the event in a scope. However, the associated restriction of the trace range might not be wanted.

The `given` expression takes the same syntactic form as scopes, i.e. suffixed with an occurrence specifier (`first`, `last` or `each`) and an event descriptor. It wraps another property that will be evaluated in a environment extended with the selected event. In the `each` case, the property must be true for all events matching the event descriptor. Like scopes, a property constructed with `given` will be true if no events match the descriptor.

3.3.3 Semantics

Preliminary Definitions If (e_i) is an untimed trace, (σ_k, x_k) is a sequence of event names and variable names, c is a condition expression and η is an environment, let

$$M_{\text{all}}((e_i)_{i=1}^m, (\sigma_k, x_k)_{k=1}^n, c, \eta) = \{(i_k)_{k=1}^n \mid \text{name}(e_{i_k}) = \sigma_k \wedge \eta' \vdash c \downarrow \textbf{true}\},$$

where

$$\eta' = \eta[x_1 \mapsto \text{param}(e_{i_1}), \ldots, x_n \mapsto \text{param}(e_{i_n})], \tag{1}$$

be the set of index sequences of (e_i) such that each event sequence *matches* the event names (σ_k), and respects the condition c when evaluated in the environment η extended with the event parameters. Put simply, M_{all} computes all the event sets matching an event set description for a given trace and a given environment.

Let S be a finite set of sequences of equal length. We write $\min S$ (resp. $\max S$) to denote the minimal (resp. maximal) element of S in colexicographic order, which is a variant of the lexicographical order obtained by comparing sequences from the right to the left. If o is an occurrence specifier, i.e. an element of $\{\texttt{first}, \texttt{last}, \texttt{each}\}$,

$$\text{select}(o, S) = \begin{cases} S & \text{if } o = \texttt{each} \\ \{\min\ S\} & \text{if } o = \texttt{first} \text{ and } S \neq \emptyset \\ \{\max\ S\} & \text{if } o = \texttt{last} \text{ and } S \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

restricts S according to the occurrence specifier o. The colexicographic ensures that the first element is the event set with the earliest last event and that the last element is the event set with the latest first event. This order is total for S, which guarantees the existence and uniqueness of the minimal and maximal elements.

If (t_i, e_i) is a trace, (σ_k, x_k) is a sequence of event names and variable names, c is a condition expression, η is an environment and o an occurrence specifier, let

$$M((t_i, e_i)_{i=1}^m, (\sigma_k, x_k)_{k=1}^n, c, \eta, o) = \\ \{(\min_k i_k, \max_k i_k, \eta') \mid (i_k)_{k=1}^n \in \text{select}(o, M_{\text{all}}((e_i)_{i=1}^m, (\sigma_k, x_k)_{k=1}^n, c, \eta))\},$$

where η' is defined as in (1), be the set of triplets that, for each event set matching the given description and occurrence specifier, is composed of the beginning index of the match in the trace, its ending index and the updated environment with the matched event parameters.

Finally, if (e_i, t_i) is a trace and l a natural, the function

$$\text{upto}((e_i, t_i)_{i=1}^n, l) = (e_i, t_i)_{i=1}^{\max(\{j \mid t_j < l\} \cup n)}$$

slices the trace (e_i, t_i) from its beginning and up to the time limit l.

Semantics Rules Properties are evaluated over finite traces and in a specific environment. The satisfaction relation between a trace τ, an environment η and a property p is the smallest relation $\tau \vDash_\eta p$ satisfying the following rules:

$$\frac{\tau \nvDash_\eta P}{\tau \vDash_\eta \texttt{not } P} \textsc{Neg} \qquad \frac{\tau \vDash_\eta P_1 \vee \tau \vDash_\eta P_2}{\tau \vDash_\eta P_1 \texttt{ or } P_2} \textsc{Disj}$$

$$\frac{\eta \vdash e \downarrow L \quad \forall v \in L. \tau \vDash_{\eta[x \mapsto v]} P}{\tau \vDash_\eta \texttt{forall } x \texttt{ in } e, P} \textsc{Forall}$$

$$\frac{\eta \vdash n_e \downarrow n \quad n \leq |M(\tau, (E_i, x_i)_{i=1}^n, c, \eta, \texttt{each})|}{\tau \vDash_\eta \texttt{occurrence\_of } n_e \texttt{ set(}E_1\ x_1\texttt{, ...,}E_n\ x_n\texttt{) where } c} \textsc{Occ}$$

$$\frac{\forall (j, k, \eta') \in M(\tau, (E_i, x_i)_{i=1}^n, c, \eta, o). (\tau_i)_{i>k} \vDash_{\eta'} P}{\tau \vDash_\eta \texttt{after } o \texttt{ set(}E_1\ x_1\texttt{, ...,}E_n\ x_n\texttt{) where } c, P} \textsc{Aft}$$

$$\frac{\eta \vdash \delta_e \downarrow \delta \quad \forall (j, k, \eta') \in M(\tau, (E_i, x_i)_{i=1}^n, c, \eta, o). \text{ upto}((\tau_i)_{i>k}, t_0 + \delta) \vDash_{\eta'} P}{\tau \vDash_\eta \texttt{within } \delta_e \texttt{ after } o \texttt{ set(}E_1\ x_1\texttt{, ...,}E_n\ x_n\texttt{) where } c, P} \textsc{AftT}$$

$$\frac{\forall (j, k, \eta') \in M(\tau, (E_i, x_i)_{i=1}^n, c, \eta, o). \tau \vDash_{\eta'} P}{\tau \vDash_\eta \texttt{given } o \texttt{ set(}E_1\ x_1\texttt{, ...,}E_n\ x_n\texttt{) where } c, P} \textsc{Given}$$

The rules Bef and BefT for the `before` scope are symmetrical to Aft and AftT, respectively, and are omitted here. We say that a trace τ *satisfies* a property P when $\tau \vDash_{[]} P$.

The first two rules are straightforward. The next one, Forall, handles universally quantified properties by first evaluating the expression that represents the quantification domain, and then evaluating the subsequent property for all values in that domain. Occ, the only non-recursive rule, asserts the occurrence of a particular event set description by counting the number of event

sets that match this description in the current trace. The rules for the `after` scope are AFT, and its timed variant AFTT. They rely on the results of the M function to slice the trace after the end of each event set matching the description and to update the evaluation environment. Additionally, AFTT evaluates a duration expression and slices the trace so that it lasts at most for this duration. The rule for the `given` expression is the same as AFT without the range restriction.

The previous rules allow defining the additional logical expressions with the usual identities:

- P_1 `and` $P_2 \equiv$ `not` (`not` P_1 `or not` P_2)
- P_1 `implies` $P_2 \equiv$ `not` P_1 `or` P_2
- P_1 `equiv` $P_2 \equiv (P_1$ `implies` $P_2)$ `and` $(P_2$ `implies` $P_1)$
- `exists` x `in` e, $P \equiv$ `not (forall` x `in` e, `not` $P)$,

and the additional temporal expressions:

- `absence_of` $E \equiv$ `not` (`occurrence_of` E)
- A `followed_by` B `within` $\delta \equiv$
  `within` δ `after each` A, `occurrence_of` B
- A `preceded_by` B `within` $\delta \equiv$
  `within` δ `before each` A, `occurrence_of` B
- A `prevents` B `within` $\delta \equiv$
  `within` δ `after each` A, `absence_of` B
- `between` A `and` B, $P \equiv$ `after each` A, `before first` B, P
- `since` A `until` B, $P \equiv$
  `(between` A `and` B, $P)$ `and after last` A, `(absence_of` B `implies` $P)$

Finally, simple event descriptions are translated into event set descriptions with a single event, unbound events are bound to the empty variable name, and omitted guards defaults to `true`.

4 WP1/D3: Examples of specifications

The following list of examples illustrates the idiomatic expression of the studied properties in the proposed DSL.

Property 1: The trace contains a step "redo acquisitions".

```
occurrence_of Enter e where e.state == 'redo'
```

Property 2: The temperature of the camera stays within $[l, u]$.

```
absence_of Temp t where not (a <= t.t1 and t.t1 < b)
```

Property 3: The distance between pairs of hip centers is less than d.

```
after each HipCenter h1,
  absence_of HipCenter h2 where dist(h1.point, h2.point) >= d
```

where `dist` : $\mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$ is an external function returning the euclidean distance between two points.

Property 4: The distance between the hip center and the knee center is greater than d.

```
absence_of set(HipCenter hc, KneeCenter kc)
  where dist(hc.point, kc.point) <= d
```

where `dist` is the same function than the one presented in the previous property.

Property 5: If the medial malleolus is further than the lateral one, a warning is issued.

```
set(MedialMalleolus m, LateralMalleolus l) where
  norm(l.point) < norm(m.point) followed_by WarningMalleolusInverted
```

where `norm` : $\mathbb{R}^3 \to \mathbb{R}$ is an external function returning the norm of a vector.

Property 6: The user never skips a screen.

```
Enter prevents ActionNext for 100 ms
```

Property 7: Acquire a point succeed if and only if the probe is stable.

```
AcquirePoint ap where isStable(ap.cloud) followed_by PointAcquired
```

where `isStable` : $\mathcal{P}(\mathbb{R}^3) \to \mathbb{B}$ is an external predicate that holds if the given set of points is *stable*. Issue: the events `AcquirePoint` and `PointAcquired` might not be correlated.

Property 8: The protocol "redo acquisitions" proposes only already performed acquisitions.

```
before each Redo r, forall o in r.options,
  occurrence_of Enter e where e.state == o
```

Property 9: Detecting a new tracker produces a dialog asking for replacement confirmation.

```
after each RegisterTracker rt,
  TrackerDetected td where td.type == rt.type followed_by
    DialogConfirmReplace dc where dc.type == rt.type
```

Property 10: The state `TrackersConnection` is unreachable until the camera is connected.

```
Enter e where e.state == 'TrackersConnection' preceded_by CameraConnected
```

Property 11: A replaced tracker is not used until it is registered again.

```
since (Unregister u) until (Register r where r.id == u.id),
  absence_of (Activate a where a.id == u.id)
```

Property 12: The action "previous" cancels the current points cloud acquisition.

```
since AcquisitionCancel until AcquisitionBegin,
  absence_of AcquisitionSuccess
```

Property 13: All the necessary trackers are seen before entering the state `TrackersVisibCheck`.

```
before each EnterState e
  where e.state == "mainCasp.TrackingConnection.TrackersVisibCheck",
given last SearchTrackers st,
forall ty in st.types,
occurrence_of TrackerDetected td where td.ty == ty
```

Property 14: On the trackers connection screen, a tracker is shown if and only if it is necessary.

```
since SearchTrackers st1 until SearchTrackers,
  absence_of ScreenshotTrackersConnection stc
    where stc.trackers != st1.requiredTrackers
```

Property 15: In the state `TrackersConnection`, not detecting any new tracker for 2 minutes produces an error message.

This property cannot be expressed in the proposed language yet.

References

[BBdB+16] Fabrice Bertrand, Yoann Blein, Lydie du Bousquet, Arnaud Clère, Roland Groz, and Yves Ledru. MODMED WP6/D1: Requirements Analysis. Technical report, BlueOrtho, LIG, MinMaxMedical, 2016.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 411–420. ACM, 1999.

[Lee08] Edward A. Lee. Cyber physical systems: Design challenges. In *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008), 5-7 May 2008, Orlando, Florida, USA*, pages 363–369, 2008.