MODMED

WP1/D1: Preliminary Definition of a Domain Specific Specification Language

Version 0.10
November 10, 2016

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 this DSL (Domain Specific 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.

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. From this observation, we reconsidered the priorities of the language and postponed time handling to a future version.

3 Related Work on Property Specification

Property specification is an important challenge in software verification and much work has been dedicated to specification logics and languages. Most of them are designed to be verified automatically and exhaustively through techniques such as model checking. Their expressiveness must be carefully limited so that they remain decidable in such frameworks. However, by trading exhaustiveness, runtime verification allows verifying properties formulated in a much more expressive specification language. Over the last decade, many specification languages for runtime verification have emerged. They are based on formalisms such as state machines, regular expressions, temporal logic, grammars or rule systems.

The linear temporal logic (LTL) [Pnu77] is widely used as a reference formalism in runtime verification. Most notably, Bauer *et al.* proposed alternatives LTL semantics for finite traces that are suitable for monitoring [BLS10]. However, LTL's syntax and typical minimality make the formulation of concise and correct specifications difficult, even for specialists. SALT [BLS06] is a domain independent abstraction over LTL's syntax providing specification patterns, nested temporal scopes and real-time constructs. Although it is much more convenient to use than LTL, it does not support parametric events and, therefore, is not suitable for the MODMED project. Many other LTL extensions handling parametric events have been proposed such as *counting quantifiers* [MJBF14] or SQL-like aggregation [BKMZ15], but they do not attempt to facilitate property specification.

Rule-based systems also received a lot of attention. They feature the most powerful logics and parametric monitoring. One of the most notable proposals is RULER [BRH10] where specifications are written as conditional rules. In its simplest form, a rule-based specification consists of a set of rules of the form:

$$\mathit{condition}_1, \ldots, \mathit{condition}_n \implies \mathit{action}$$

When all the conditions of a rule are respected, the right-hand side action is executed, typically updating the state of the monitoring system. RULER's poor efficiency led to the development of LOGFIRE [Hav15], an internal Scala DSL attempting to achieve efficient monitoring for the rule-based approach. Despite these efforts, LOGFIRE performance is still behind the state-of-the-art runtime verification system MOP [MJG$^+$12] by several orders of magnitude. We also notice that rule-based systems favor an unnatural control flow in specifications, making them arguably difficult to understand. Because of this, and because of the efficiency issues in underlying implementations, we dismissed ruled-based systems for property specification.

Our approach was mostly influenced by the work of Dwyer *et al.* on *specification patterns* [DAC99]. They identified a set of frequently used patterns in real-world specifications and elaborated a pattern system, much like design patterns in software engineering. Basically, a pattern is a *requirement* (*e.g.*, an event triggers another one) coupled with a *temporal scope* (*e.g.*, after a certain event). Dwyer *et al.* defined an LTL translation for each and every couple requirement/scope. The pattern system enables inexperienced users to write formal specifications in a natural language. However, it suffers from a limited expressiveness mostly due to the fact that scopes cannot be nested. Extending the system with a new requirement or scope requires writing the LTL translation for every possible couple containing the new construct, which is inconvenient and error-prone. Taha *et al.* shown that even the original pattern translations are inconsistent [TJD+15].

3.1 Examples

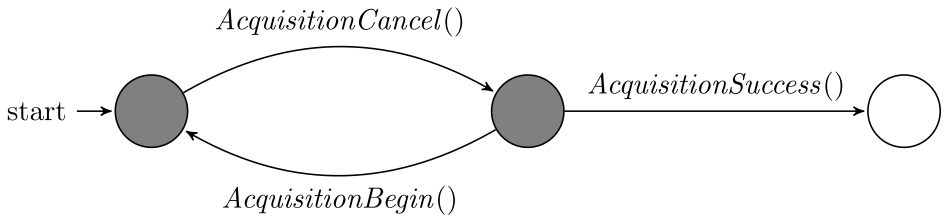
In this section we demonstrate the specification of some of the selected properties with two powerful formalisms handling parametric events: the FO-LTL$_f$, a first-order discrete time linear temporal logic with finite-trace semantics proposed by Reger and Rydeheader [RR15], and the Quantified Event Automata (QEA) formalism from Barringer *et al* [BFH+12]. We assume that both formalisms are extended to support basic arithmetic and boolean expressions.

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

- FO-LTL$_f$:

$$\Box(\mathit{AcquisitionCancel} \implies \neg \mathit{AcquisitionSuccess}\ \mathcal{U}\ \mathit{AcquisitionBegin})$$

- QEA:



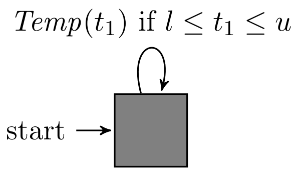
$\mathit{AcquisitionCancel}()$
start
$\mathit{AcquisitionSuccess}()$
$\mathit{AcquisitionBegin}()$

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

- FO-LTL$_f$:

$$\forall t_1.\Box(\mathit{Temp}(t_1) \implies l \leq t_1 \leq u)$$

- QEA:

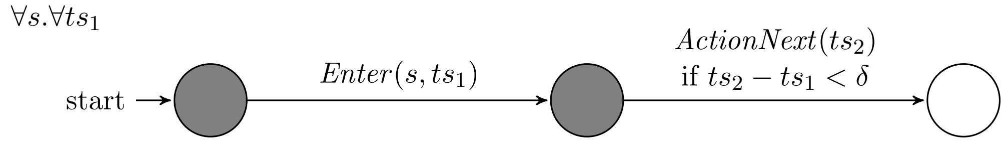


Property 6: The user never skips a screen.

- FO-LTL$_f$:

$$\forall s.\forall ts_1.\Box(\mathit{Enter}(s, ts_1) \implies \neg(\exists ts_2.\Diamond(\mathit{ActionNext}(ts_2) \wedge ts_2 - ts_1 < \delta)))$$

- QEA:

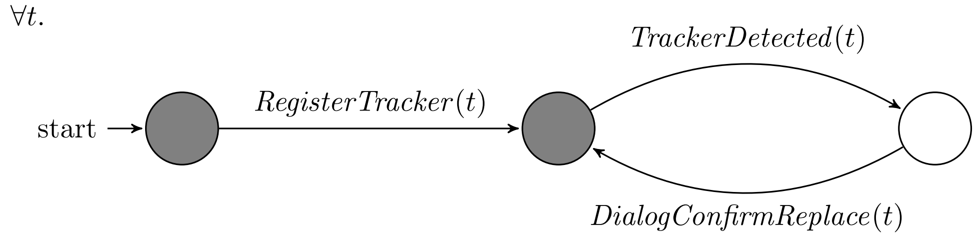


Property 9: Detecting a new tracker produces a dialog asking for replacement confirmation.

- FO-LTL$_f$:

$$\forall t.\Box((RegisterTracker(t) \wedge \Diamond TrackerDetected(t)) \\ \implies \Diamond(TrackerDetected(t) \wedge \Diamond DialogConfirmReplace(t)))$$

- QEA:



Although those two formalisms allow specifying the properties concisely, they have several drawbacks. Both of them are difficult to write and to understand, especially when one or several quantifiers are involved in a specification. Also, they require to explicitly specify the parameters of each event, even if they are not relevant to the current property. This is not practical since real-world events tends to have many parameters. We also notice that the FO-LTL$_f$ is even harder to understand because of its cryptic notations whereas QEA's graphical representation helps on this matter. However, QEA specifications do not convey the intent of the properties.

4 Basic Concepts: Event, Traces and Environment

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:

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

Let *Val* be the set of values that includes:

- atomic litterals: booleans, integers, strings and floating-point numbers,
- homogeneous sequence of values, and
- records of named values ($F \rightharpoonup Val$ where F is a set of field names).

We are also considering the addition of an atomic type to represent dates and times in a future version.

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

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

where Σ and P are finite sets of event names and parameter names, respectively. For convenience we define the two following projections on $e = \langle n, ps \rangle$: $\mathit{name}(e) = n$ and $\mathit{parameters}(e) = ps$.

Finally, an *environment* is a map from variables to values:

$$\mathit{Env} = \mathit{Var} \rightharpoonup \mathit{Val}$$

where *Var* is a set of variable names.

5 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 1.

⟨*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*⟩	::=	‘`<`’ | ‘`<=`’ | ‘`==`’ | ‘`>`’ | ‘`>=`’ | ‘`!=`’ | ‘`and | ‘or`’ | ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’ | ‘`%`’	
⟨*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 1: 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:

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

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

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

$$\frac{}{\eta \vdash f(e_1, \ldots, e_n) \downarrow v}\ \textsc{E-ExtCall}$$

6 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.

6.1 Syntax

The syntax of the proposed DSL is described by the grammar in Figure 2.

$\langle prop \rangle$ ::= $\langle pattern \rangle$
| $\langle scope \rangle$ '`,`' $\langle prop \rangle$
| '`forall`' $\langle ident \rangle$ '`in`' $\langle expr \rangle$, $\langle prop \rangle$
| '`(`' $\langle prop \rangle$ '`)`'
| '`not`' $\langle prop \rangle$
| $\langle prop \rangle$ ('`and`' | '`or`' | '`equiv`' | '`implies`') $\langle prop \rangle$

$\langle scope \rangle$ ::= ('`after`' | '`before`') ('`each`' | '`first`' | '`last`') $\langle event \rangle$
| '`between`' $\langle event \rangle$ '`and`' $\langle event \rangle$
| '`since`' $\langle event \rangle$ '`until`' $\langle event \rangle$

$\langle pattern \rangle$ ::= '`absence_of`' $\langle event \rangle$
| '`occurrence_of`' $\langle expr \rangle$ $\langle event \rangle$
| $\langle event \rangle$ '`followed_by`' $\langle event \rangle$
| $\langle event \rangle$ '`enables`' $\langle event \rangle$
| $\langle event \rangle$ '`disables`' $\langle event \rangle$

$\langle event \rangle$ ::= $\langle ident \rangle$ [$\langle ident \rangle$ ['`where`' $\langle expr \rangle$]]

Figure 2: Syntax of the proposed DSL

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 enables D)`

6.2 Informal semantics

6.2.1 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`.

6.2.2 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 enables B` and `A disables 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 enables B` holds if and only if each occurrence of the event `B` is eventually preceded by an occurrence of the event `A`. Therefore, an occurrence of the event `A` *enables* the event `B` to occur. Finally, `A disables B` prevents the event `B` from occuring after an occurence 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`	✗	✓	✓	✗	✗
`A enables B`	✓	✗	✓	✗	✓
`A disables B`	✓	✓	✗	✓	✗

Table 2: Examples of pattern satisfaction for various traces

6.2.3 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 3, are the basic building blocks.

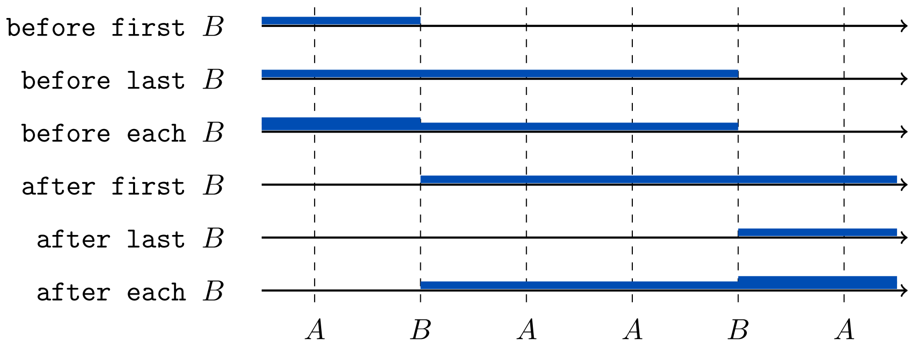


Figure 3: Graphical representation of the unary scopes

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 4.

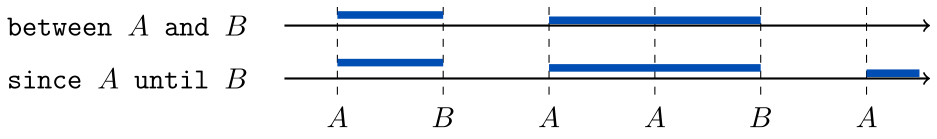


Figure 4: Graphical representation of the binary scopes

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 subsubsection 6.3.2), 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.

6.3 Semantics

Before defining the satisfaction relation between a trace, an environment and a property, we need to introduce a couple of helpful functions.

6.3.1 Preliminary Definitions

The predicate *matches* : $\mathit{Event} \times \Sigma \times \mathit{Env} \times \mathit{Var} \times \mathit{Expr} \to \mathbb{B}$ holds when an event matches a description and is defined as follows:

$$\mathit{matches}(e, E, \eta, x, c) = (\mathit{name}(e) = E) \wedge (\eta[x \mapsto \mathit{parameters}(e)] \vdash c \downarrow \mathbf{true})$$

For instance, given the event $e = \langle E, [p \mapsto 1] \rangle$, the proposition

$$\mathit{matches}(e, E, [y \mapsto 1], x, x.p \le y)$$

holds since the name of e is indeed E and the guard can be reduced to **true** in the environment $\eta' = [y \mapsto 1][x \mapsto \mathit{parameters}(e)] = [x \mapsto [p \mapsto 1], y \mapsto 1]$:

$$\dfrac{\dfrac{\dfrac{\eta'(x) = [p \mapsto 1]}{\eta' \vdash x \downarrow [p \mapsto 1]}\text{E-Lookup} \qquad [p \mapsto 1](p) = 1}{\eta' \vdash x.p \downarrow 1}\text{E-FieldAccess} \qquad \dfrac{\eta'(y) = 1}{\eta' \vdash y \downarrow 1}\text{E-Lookup} \qquad 1 \le 1}{\eta' \vdash x.p \le y \downarrow \mathbf{true}}$$

Let $\mathit{Occ} = \{\texttt{each}, \texttt{first}, \texttt{last}\}$. The function *occurrences* : $\mathit{Trace} \times \Sigma \times \mathit{Env} \times \mathit{Var} \times \mathit{Expr} \times \mathit{Occ} \to \mathcal{P}(\mathbb{N})$ returns the indexes of the events of a trace matching a description and an occurrence:

$$\mathit{occurrences}(\tau, E, \eta, x, c, o) = \begin{cases} S & \text{if } o = \texttt{each} \\ \{\mathit{min}\ S\} & \text{if } o = \texttt{first} \text{ and } S \neq \emptyset \\ \{\mathit{max}\ S\} & \text{if } o = \texttt{last} \text{ and } S \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

where $S = \{i \mid \mathit{match}(\tau_i, E, \eta, x, c)\}$

The function *elems* : $A^* \to \mathcal{P}(A)$ converts a finite sequence into a set:

$$\mathit{elems}((s_n)_{n \in D}) = \{s_i \mid i \in D\}$$

Finally, we define the *counting existential* quantifier $\exists_{\ge n} x.\varphi$ where $n \in \mathbb{N}$ and read as "there exist at least n different x such that φ". Formally it can be defined as follows:

$$\exists_{\ge n} x.\varphi \equiv |\{x \mid \varphi\}| \ge n$$

6.3.2 Satisfaction Relation

Properties are evaluated on pairs composed of a finite trace and an environment. The satisfaction relation between a trace τ, an environment η and a property p is the smallest relation $(\tau, \eta) \vDash p$ satisfying the following rules:

$$\frac{\langle \tau, \eta \rangle \not\models P}{\langle \tau, \eta \rangle \models \texttt{not}\ P}\ \text{S-NEG}$$

$$\frac{\langle \tau, \eta \rangle \models P_1 \vee \langle \tau, \eta \rangle \models P_2}{\langle \tau, \eta \rangle \models P_1\ \texttt{or}\ P_2}\ \text{S-CONJ}$$

$$\frac{\eta \vdash e \downarrow S \quad \forall v \in \mathit{elems}(S).\langle \tau, \eta[x \mapsto v] \rangle \models P}{\langle \tau, \eta \rangle \models \texttt{forall}\ x\ \texttt{in}\ e\texttt{,}\ P}\ \text{S-FORALL}$$

$$\frac{\eta \vdash n_e \downarrow n \quad \exists_{\geq n} i.\mathit{matches}(\tau_i, E, \eta, x, c)}{\langle \tau, \eta \rangle \models \texttt{occurrence\_of}\ n_e\ E\ x\ \texttt{where}\ c}\ \text{S-OCC}$$

$$\frac{\forall i \in \mathit{occurrences}(\tau, E, \eta, x, c, o).\langle (\tau_j)_{j>i}, \eta[x \mapsto \mathit{parameters}(\tau_i)] \rangle \models P}{\langle \tau, \eta \rangle \models \texttt{after}\ o\, E\, x\ \texttt{where}\ c\texttt{,}\ P}\ \text{S-AFTER}$$

$$\frac{\forall i \in \mathit{occurrences}(\tau, E, \eta, x, c, o).\langle (\tau_j)_{j<i}, \eta[x \mapsto \mathit{parameters}(\tau_i)] \rangle \models P}{\langle \tau, \eta \rangle \models \texttt{before}\ o\, E\, x\ \texttt{where}\ c\texttt{,}\ P}\ \text{S-BEFORE}$$

We say that a trace τ *satisfies* a property P when $(\tau, [\,]) \models P$.

The previous rules allow to define the additional logical operators as follows:

- 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)

The additional temporal operators are defined as follows:

- `absence_of` $E \equiv$ `not` (`occurrence_of` E)
- E_1 `followed_by` $E_2 \equiv$ `after last` E_1`,` `occurrence_of` E_2
- E_1 `enables` $E_2 \equiv$ `before first` E_2`,` `occurrence_of` E_1
- E_1 `disables` $E_2 \equiv$ `after first` E_1`,` `absence_of` E_2
- `between` E_1 `and` E_2`,` $P \equiv P_1$ `and` P_2, where
  $P_1 =$ `after first` E_1`,` `before first` E_2`,` `P`
  $P_2 =$ `after each` E_2`,` P_1
- `since` E_1 `until` E_2`,` $P \equiv$ (`between` E_1 `and` E_2`,` P) `and`
  (`after last` E_2`,` `after first` E_1`,` P)

7 Examples

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.

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

where `dist` is the same function than the one presented in the previous property. We notice some form of redundancy in the specification that motives the need for a construct capturing unordered events:

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

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

```
(after each MedialMalleolus m, LateralMalleolus l where
  norm(l.point) < norm(m.point) followed_by WarningMalleolusInverted)
and
(after each LateralMalleolus l, MedialMalleolus m 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. With a construction similar to the previous property:

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

Property 6: The user never skips a screen.

```
after each Enter e, absence_of ActionNext a where a.ts - e.ts <= d
```

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.

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

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 TrackersVisibCheck tvc, forall t in tvc.trackers,
  occurrence_of TrackerDetected td where td.type == t
```

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.

8 Future Work

The language proposed here is a preliminary definition of the MODMED DSL. As such, it is still incomplete and likely to change significantly.

In particular, the missing important features identified so far are the support for physical time and a construction for capturing unordered events (*collections*) to avoid redundancy in specifications. Gruhn and Laue proposed interesting patterns [GL06] for these two features.

The trace format used here implies that each event has a well defined structure. Obtaining such a trace will require to process untyped execution traces according to a set of event descriptions. We will develop a library and a description format to do so.

Finally, we also want to study the practical interest of adding non-strict scope delimiters.

References

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

[BFH+12] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, pages 68–84, 2012.

[BKMZ15] David A. Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zalinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods in System Design*, 46(3):262–285, 2015.

[BLS06] Andreas Bauer, Martin Leucker, and Jonathan Streit. SALT - structured assertion language for temporal logic. In *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, pages 757–775, 2006.

[BLS10] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL semantics for runtime verification. *J. Log. Comput.*, 20(3):651–674, 2010.

[BRH10] Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from eagle to ruler. *J. Log. Comput.*, 20(3):675–706, 2010.

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

[GL06] Volker Gruhn and Ralf Laue. Patterns for timed property specifications. *Electr. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006.

[Hav15] Klaus Havelund. Rule-based runtime verification revisited. *STTT*, 17(2):143–170, 2015.

[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.

[MJBF14] Ramy Medhat, Yogi Joshi, Borzoo Bonakdarpour, and Sebastian Fischmeister. Accelerated runtime verification of LTL specifications with counting semantics. *CoRR*, abs/1411.2239, 2014.

[MJG+12] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977.

[RR15] Giles Reger and David E. Rydeheard. From first-order temporal logic to parametric trace slicing. In *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, pages 216–232, 2015.

[TJD+15] Safouan Taha, Jacques Julliand, Frédéric Dadeau, Kalou Cabrera Castillos, and Bilal Kanso. A compositional automata-based semantics and preserving transformation rules for testing property patterns. *Formal Asp. Comput.*, 27(4):641–664, 2015.