Extending Specification Patterns for Verification of Parametric Traces

Yoann Blein
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG
Grenoble, France
yoann.blein@imag.fr

Yves Ledru
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG
Grenoble, France
yves.ledru@imag.fr

Lydie du-Bousquet
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG
Grenoble, France
lydie.du-bousquet@imag.fr

Roland Groz
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG
Grenoble, France
roland.groz@imag.fr

ABSTRACT

This article proposes a temporal and parametric specification language (ParTraP) developed for the verification of execution traces. The language extends specification patterns with nested scopes, real-time and first-order quantification over the data inside a JSON trace, while remaining pragmatic. Its design was directed by a case study in the medical field (computer aided surgery). The paper briefly presents the case study and details the design rationale, syntax and semantics of the language. The language has been implemented and several properties have been successfully evaluated over a corpus of 100 surgery traces.

CCS CONCEPTS

• **Software and its engineering → Specification languages**; *Formal software verification*;

KEYWORDS

Runtime Verification, Parametric Events, Temporal Specification, Trace Analysis

ACM Reference Format:
Yoann Blein, Yves Ledru, Lydie du-Bousquet, and Roland Groz. 2018. Extending Specification Patterns for Verification of Parametric Traces. In *FormaliSE '18: FormaliSE '18: 6th Conference on Formal Methods in Software Engineering , June 2, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3193992.3193998

1 INTRODUCTION

Numerous industrial fields (air and space, transportation, energy, ...) have made significant progress in the last decades in order to improve the quality of their systems and software. Although formal methods appear as a way to achieve high quality, their adoption is limited to critical applications. Daniel Jackson and Jeannette Wing have advocated the adoption of lightweight formal methods [20]. Runtime Verification of execution traces appears as one way to achieve such a transition. In this paper, we report on the design of a trace specification language in the context of the design of Medical Devices, which provide support for complex medical surgeries. Fortunately, these systems can easily be instrumented to provide execution traces enabling us to observe their usage in the field.

In this work, we consider traces of surgical operations. They record the surgery workflow, the inputs captured from sensors and the interactions with the surgeon. The verification of execution traces will serve the following purposes: verify the correctness and the robustness of an implementation when used in real conditions, validate the hypotheses made on the environment and the conditions of use of a device, and understand the device usage in a perspective of product improvement.

One of the main challenges is to make trace properties writable by software engineers with no training in formal methods and readable by domain experts. For this purpose, we are developing ParTraP, a language dedicated to property specification for finite traces. The language extends the specification patterns originally proposed by Dwyer et al. [15] with parametrized constructs, nested scopes, real-time and first-order quantification over the data inside a JSON trace. A prototype interpreter has been implemented and experimented on traces provided by our industrial partner.

Section 2 motivates the need for a parametric language for property specification by presenting our industrial case study. We introduce the ParTraP language by giving an overview of its major features and some real-world examples in section 3. Section 4 formally describes the semantics of ParTraP and section 5 discusses a prototype implementation of the language. Finally, we discuss related work in section 6 and draw conclusions in section 7.

2 CONTEXT AND MOTIVATION

This work originates from an industrial collaboration aimed at pushing runtime verification techniques into the Medical Devices industry. After briefly presenting the case study, we describe the trace format in use and illustrate the nature of the requirements.

2.1 Case Study Presentation

The case study focuses on a computer assisted guidance system for total knee arthroplasty: TKA, designed by the BlueOrtho company for Exactech implant manufacturer. Total knee arthroplasty is a surgical procedure that involves replacing parts of the knee with

FormaliSE '18, June 2, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5718-0/18/06...$15.00
https://doi.org/10.1145/3193992.3193998

a prosthesis. To install the prosthesis, it is necessary to cut off parts of the tibia and the femur. TKA helps the surgeon achieve these cuts with the required precision, through the supervised installation of cutting guides at the “right” position. The position of cutting guides is automatically computed on the basis of several measurements performed by the surgeon using the sensors of TKA, and of additional parameters chosen by the surgeon. This system is currently used worldwide and thousands of surgeries have been conducted with it.

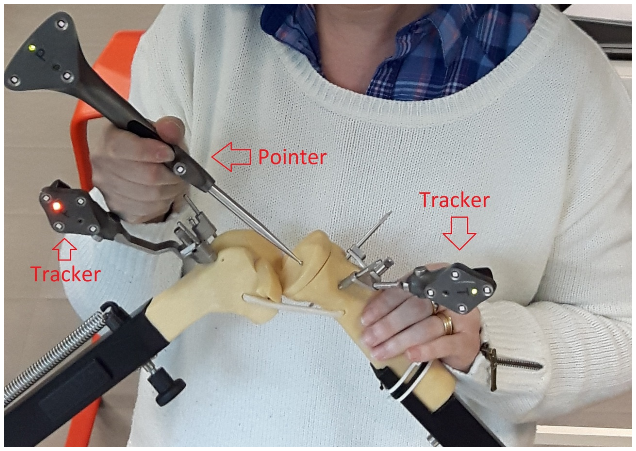

Figure 1: Pointer and trackers

TKA consists of several components: a computer able to communicate with the surgeon via a touch screen, a stereoscopic camera, a set of sensors, named trackers, that can be localized in 3D by the stereoscopic camera, and a set of mechanical instruments to attach trackers and cutting guides to the bones of the patient. Fig. 1 shows several trackers, during a tutorial session, using a fake skeleton. The “trackers” have a diamond shape with four blinking LEDs. The stereoscopic camera is synchronized with the LEDs and is able to capture the 3D position, and the orientation of the tracker. The “pointer” is a hand-held tracker whose length has been calibrated. The surgeon uses the pointer to acquire the position of anatomic points of the patient. The other trackers are mechanically attached to the bones, so that they provide a reference when placing the cutting guides.

The surgery process involves a sequence 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. In every case, the sequence of steps takes the following macroscopic form: sensor calibration, acquisition of anatomical points using the pointer, check of acquisitions, adjustment of target parameters, positioning of the cutting guides, bones cut, and finally, digitization of the performed cut.

2.2 Traces

BlueOrtho systematically collects the logs of performed surgeries. Each log is composed of a sequence of about 3000 parametric events. The nature of these events is very diverse and includes hardware events (communication with the sensors), user input or complex computation results. They carry values of various types: numeric values, lists, 3D points. . . To this day, the company has collected a corpus of about 10 000 traces of surgeries that took place in real conditions.

At the conceptual level, a trace is viewed as a timed sequence of events. Each event is characterized by a name and a set of named parameters. Those parameters can be simple literals such as strings, or compounds values such as records. The original traces are mainly textual and poorly structured. Therefore, original traces are transformed into JSON files, where the top-level element is an array of objects, representing events. Each object must include the “time” and “name” keys, and may have other parameters.

Fig. 2 shows a simplified example of such a trace. The first event registers a tracker of type F (attached to the Femur), whose id is 0. This event took place at time 5. The second event declares that the current surgery needs trackers P and F. The third event corresponds to the registration of the P tracker (P stands for Pointer). The fourth event records the beginning of the acquisition phase. During this phase, the coordinates of the medial and the lateral malleolus are recorded (at times 9 and 20). Also, from 14 to 16, the pointer is replaced by another pointer which is registered and then activated.

```
[
  ...
  {"time": 5, "name": "RegisterTracker", "type": "
      F", "id": 0 },
  {"time": 6, "name": "SearchTrackers", "types":
      ["P", "F"]},
  {"time": 7, "name": "RegisterTracker", "type": "
      P", "id": 1 },
  {"time": 8, "name": "StartAcquisitions"},
  {"time": 9, "name": "MedialMalleolus", "point":
      [0.5, 1.0, 0.8] },
  ...
  {"time": 14, "name": "ReplaceTracker", "id": 1
      },
  {"time": 15, "name": "RegisterTracker", "id": 2,
       "type": "P" },
  {"time": 16, "name": "ActivateTracker", "id": 2
      },
  ...
  {"time": 20, "name": "LateralMalleolus", "point
      ": [0.6, 0.9, 0.9] }
  ...
]
```

Figure 2: Example of JSON trace

2.3 Requirements for a Trace Property Language

To design a property specification language accessible to engineers with no training in formal methods, we gathered a corpus of properties that BlueOrtho developers would like to verify on the traces produced by TKA. These properties come from three sources: interviews with BlueOrtho developers, reviews of the technical specification documents of TKA, and reviews of some usual and unusual existing traces. Usual traces correspond to surgeries which followed the expected process, and where no problem was reported. Unusual traces were mainly identified after a report of the surgeon, signalling that something went wrong during the surgery. These reports and the associated traces are analysed by the Blue Ortho engineers. Traces help understand what went wrong during the surgery, and what was the cause of the problem. For example, this analysis may reveal that the camera was positioned on the same side of the patient than the knee being cured (see property 1).

The resulting corpus is composed of about 50 properties. They include constraints on event ordering, invariant checking on 3D points and soft real-time constraints. The following examples illustrate some types of properties found in this corpus.

Runtime Property 1. *If the medial malleolus is further away from the camera than the lateral malleolus, a warning is issued within 100 milliseconds.*

The inversion of malleolus distances could reveal that the 3D camera was installed on the wrong side of the patient (the one of the leg being cured). This property requires the occurrence of a warning event in response to the observation of a set of events that fulfills a predicate. Also, this response is bounded in time and the two malleolus events may happen in any order. Note that geometric computation is out of scope of the language, and we assume that functions such as 3D euclidean distance are defined externally.

Runtime Property 2. *A replaced tracker is no longer used until it is registered again.*

The system should no longer try to use a tracker that is not currently registered. Trackers registration, replacement and usage can be observed through the events `RegisterTracker`, `ReplaceTracker` and `ActivateTracker`, respectively. Each tracker has a unique identifier that appears as parameter in all those events.

Runtime Property 3. *All the necessary trackers are detected before starting the acquisitions.*

To proceed, the system requires a set of trackers which depends on the profile in use. This set is logged in the parameters of the `SearchTrackers` event. Each of these trackers should be detected at least once before starting the acquisitions. The set of required trackers may be modified during the surgery (according to the selected profile) and, upon search for the new set, any tracker already detected should not be detected again.

As said previously, we identified about 50 such properties during the requirements analysis. Considering this corpus, we conclude that the specification language should have at least the following properties:

- **Parametric** From the example properties, one can notice that using event parameters is necessary. For instance, Runtime Property 3 relies on a list of values given as parameter of events `SearchTrackers`. Moreover, it should be possible to access the values of structured parameters such as list and records.
- **Temporal** The three example properties constrain the occurrence and ordering of events. Most of the properties from the collected corpus also have this trait.
- **Timed** A few properties from the corpus, such as Runtime Property 1, refer to physical time. Although time can be treated as regular data, a first class support for it can help write properties that are easier to understand and more efficient to process.

3 LANGUAGE FEATURES

ParTraP (Parametric Trace Property language) is a new property specification language for finite parametric traces, designed to meet the properties stated in the previous section. Although it was mostly influenced by the specification patterns proposed by Dywer et al. [15], it differs from them by being event-based, featuring parametric events, allowing nested scopes and providing timed properties.

In spite of the inspiration from specification patterns, the language does not restrict properties to pairs composed of a scope and a pattern. Instead, most of the property expressions are built on top of other properties, and can be freely nested. The inner property and the outer part are always separated by commas. Nested properties are evaluated from the outermost property towards the innermost. This is important since each one may modify the evaluation environment that will be used to evaluate any inner property. This mechanism allows relating different events, possibly according to their parameters.

3.1 Events Descriptors

Event descriptors allow matching events from a trace. At the simplest, events are designated by their name, such as in `absence_of A` where `A` is the name of the event. Additionally, an event can be bound to a variable `x` as 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 name `A` and fulfilling the condition `c` when bound to the variable `x`.

It is also possible to designate an unordered collection of events with the `set` construct: `set(E1 x1, ..., En xn) where c`. This event set will be triggered after seeing all of the events `E1. . . En` in any order, provided that they respect the guard `c` when `E1. . . En` are bound to `x1. . . xn`, respectively.

3.2 Temporal Properties

3.2.1 Scopes. The range of a trace where a property should hold can be restricted through scopes. In the absence of such construct, a property must hold on the whole trace. Scopes are delimited by events and, as properties themselves, can be nested. If the event delimiting a scope never occurs, the scope does not exist and the whole property is considered satisfied. Scopes can be classified according to their arity, i.e. the number of events they involve. Unary scopes, illustrated in Fig. 3, are the building blocks of the language.

All scopes are strict, i.e., delimiter events are not included in the interval they define. The "each" variants may describe several intervals and happen to be useful combinators. For instance, we can express that the property `P` should be true for all the segments of a trace between an event `A` and an event `B` by nesting two scope expressions: `after each A, before first B, P`. For convenience, we included this scope in the language as `between A and B`, as well as its weak variant `since A until B`. Those two binary scopes are

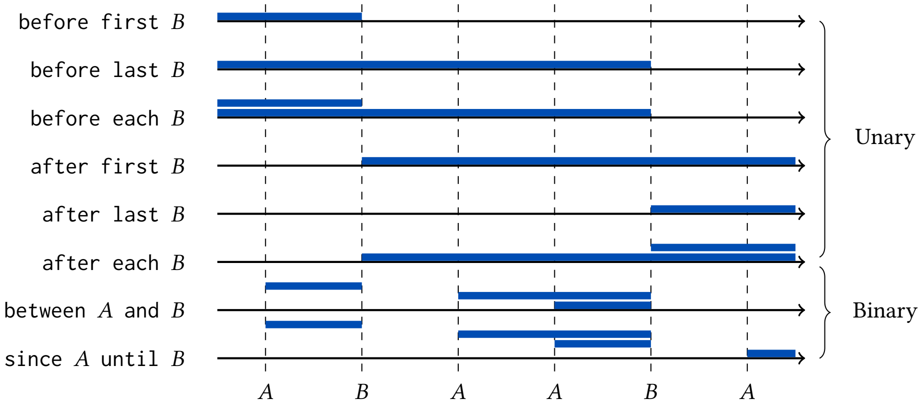

Figure 3: Graphical representation of scopes

illustrated in Fig. 3 and formally defined later. As a consequence of being defined on top of nested scopes, the event delimiting the end of a binary scope may depend on the one starting it.

Scopes implicitly extend the evaluation environment for the inner property with all the events that are associated to a variable name. 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-zero `x` parameter, and in an environment where the value of the variable `v` is this first event. This is especially useful with the `each` variants of `after` and `before` as the sub-property to verify may depend on the events delimiting each scope. For instance, the following ParTraP property encodes the fact that each entered state should be exited eventually:

```
after each EnterState e,
occurrence_of ExitState x where e.state == x.state
```

By delimiting the range of a trace where a property should hold, and providing access to the parameters of a delimiter event at the same time, scopes allow to concisely express temporal properties.

3.2.2 Patterns. The only mandatory components of properties are patterns. They rule the occurrences of events in a trace. There are two unary patterns: `occurrence_of n A`, where `n` is optional, and its dual `absence_of A`. The first one requires the occurrence of *at least* `n` events `A` in the current range of the trace. If `n` is omitted, it defaults to `1`. The second one simply prevents an event from occurring. As usual, the event descriptor `A` may be bound to a variable and filtered with the `where` construct.

Following Dwyer's specification patterns, the language also includes the response pattern `A followed_by B` and the precedence pattern `A precedes B`. The first one requires an occurrence of the event *B* after each occurrence of an event *A*, while the second one prevents the event *B* from occurring until the event *A* occurs (but *A* may occur without *B*). The language also features another convenient pattern: `A prevents B`, which prevents any occurrence of the event *B* after an occurrence of the event *A*. Note that the three binary patterns are later defined as combinations of scopes and unary patterns. A direct consequence is that the event descriptor on the right-hand side may depend on the event matched on the left-hand side.

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

Unary scopes can be prefixed with the `within` keyword and a duration expression, like in the property

```
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* should not occur during the two milliseconds preceding any occurrence of an event *A*.

Binary patterns 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`.

3.3 Non-Temporal Properties

3.3.1 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. 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: if a parameter of the event delimiting the scope is a list, it can be used as a quantification domain.

3.3.2 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 an 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 event matches the descriptor.

3.4 Examples

Let us illustrate the language features by formalizing the examples introduced in subsection 2.3. Runtime Property 1 can be expressed as follows:

```
set(LateralMalleolus l, MedialMalleolus m)
    where norm(m.point) > norm(l.point)
followed_by WarningMalleolusInverted within 100 ms
```

This expression states that any unordered pair of misplaced malleolus events should result in a warning.

Runtime Property 2 simply combines a `since`/`until` scope delimiting the range of a trace where a specific tracker is replaced, with an absence pattern preventing the activation of this tracker:

```
since ReplaceTracker rep
until RegisterTracker reg where reg.id == rep.id,
absence_of ActivateTracker act
    where act.id == rep.id
```

Note that the event delimiting the end of the scope depends on the one starting it. If several trackers are replaced, the property will be enforced for all of them because of the `since`/`until` semantics (see Fig. 3).

Finally, Runtime Property 3 is captured by the following expression:

```
before each StartAcquisitions,
given last SearchTrackers st,
forall type in st.types,
occurrence_of TrackerDetected td
    where td.type == type
```

This expression captures the property concisely by combining four constructs: a past scope, an event selector, an universal quantifier and a pattern.

4 LANGUAGE SEMANTICS

This section formalizes the semantics of ParTraP. It is defined in terms of inference rules over traces instead of translating it into an existing formalism because there is no well-established formalism for parametric specification.

For space reasons, we describe the semantics of all ParTraP constructs except events sets. This construct adds a layer of complexity but does not change the global shape of the semantic rules. The simpler and shorter version presented here is close to the complete rules, which can be found in the language reference [10]. The additional complexity mainly comes from the two following points:

(1) Contrary to single events, events sets last in time; scopes must be updated accordingly.
(2) The `first` and `last` events sets matching a set description must be defined carefully.

Some elements of the language such as the `where` clause rely on predicate expressions. Since the language producing those expressions is orthogonal to the definition of ParTraP, we will not specify it here. Instead, we will assume that any well-typed expression e can be evaluated in the environment η to the value v with the judgement form

$$\eta \vdash e \downarrow v.$$

Additionally, we require this language to provide at least an infix "`.`" operator allowing to access the fields of a record, as in `record.field`. Such a construct is necessary to inspect the parameters of an event.

4.1 Preliminary Definitions

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

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

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

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

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

where Var is a set of variable names.

An *event* is characterized by a *name*, an occurrence time and a set of named *parameters*. Formally an event is defined as a triplet:

$$\mathrm{Event} = \Sigma \times \mathbb{N} \times (P \rightharpoonup \mathrm{Val}),$$

where Σ and P are finite sets of event names and parameter names, respectively. Note that this definition permits events to have the same name and yet different parameters. This provides more flexibility with the input traces and allows, for instance, to have optional parameters. For convenience, we define the three following projections on an event $e = \langle \sigma, t, p \rangle$: $\mathrm{name}(e) = \sigma$, $\mathrm{time}(e) = t$ and $\mathrm{param}(e) = p$.

Finally, a *trace* is a sequence of events $(e_i)_{i=1}^n$ with non-decreasing occurrence times. We can formally define the set of possible traces as follows:

$$\mathrm{Trace} = \{\, (e_i)_{1=i}^n \in \mathrm{Event}^* \mid \forall i \in 1 \mathinner{..} n{-}1,\ \mathrm{time}(e_i) \le \mathrm{time}(e_{i+1}) \,\}.$$

In the following, traces are also denoted as τ when the indices are irrelevant.

4.2 Events Extraction and Time Slicing

Finding events that satisfy some constraints expressed in ParTraP properties is a basic necessity to define the semantics of the language. We first introduce a dedicated function that handles that matter. The semantic rules of ParTraP are built upon that function and focus on the temporal aspect.

The function M computes the events of a trace that *match* an event description with a name and a condition on the event, and respect an occurrence specifier, i.e. an element of {`first`, `last`, `each`}. More precisely, given a trace $(e_i)_{i=1}^m$, $M((e_i)_{i=1}^m, \sigma, x, c, \eta, o)$ is the set of indices i of the trace such that:

- e_i has the name σ,
- the condition c evaluates to **true** in the environment η extended with x associated to e_i, and
- e_i respects the occurrence specifier o.

This set can be computed in two steps: finding M_{desc}, the set of all the indices that match the event description, and then selecting the ones that respect the occurrence specifier. The definition of the function M is based on that idea:

$$M((e_i)_{i=1}^m, \sigma, x, c, \eta, o)$$

$$= \begin{cases} M_{\text{desc}} & \text{if } o = \texttt{each} \\ \{\min\ M_{\text{desc}}\} & \text{if } o = \texttt{first} \text{ and } M_{\text{desc}} \neq \emptyset \\ \{\max\ M_{\text{desc}}\} & \text{if } o = \texttt{last} \text{ and } M_{\text{desc}} \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

where

$$M_{\text{desc}} = \{ j \in 1\,..\,n \mid \text{name}(e_j) = \sigma \wedge \eta[x \mapsto \text{param}(e_j)] \vdash c \downarrow \textbf{true} \}.$$

M_{desc} is the result of the first step, i.e. it is the set of indices that match the event description given by the name σ and the condition c. The subset of M_{desc} that is actually returned is computed according to the occurrence specifier o. For instance, if $o = \texttt{first}$, only the minimal index in M_{desc} is returned, which indeed corresponds to the first event of the trace that matches the description.

We also need the ability to slice a trace according to a time limit. If $(e_i)_{i=1}^m$ is a trace and l a natural, the function

$$\text{upto}((e_i)_{i=1}^m, l) = (e_i)_{i=1}^{\max(\{ j \in [1..m] \mid \text{time}(e_j) < l \} \cup \{0\})}$$

slices the trace $(e_i)_{i=1}^m$ from its beginning and up to the time limit l. The union with the singleton $\{0\}$ in the upper bound ensures that an empty sequence is returned if there are no events occurring before the time limit.

4.3 Semantic 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 8 rules given in Fig. 4. We say that a trace τ *satisfies* a property P when $\tau \vDash_{[\,]} P$.

The semantics of the original pattern system proposed by Dwyer et al. was given through translation rules manually defined for each pair of pattern and scope. Because of the number of combinations, there are numerous rules and they have been shown to be inconsistent by Taha et al. [28]. Since ParTraP scopes can be arbitrarily nested, the number of combinations is infinite and defining an exhaustive set of rules is impossible. Instead, the satisfaction relation $\vDash$ is defined through recursive rules derived from its informal meaning, and where each rule only handle a single construct.

The first three 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. The rule Occ asserts the occurrence of a particular event description by measuring the size of M, i.e. counting the number of events that match this description in the current trace, and checking that is is greater than the computed value of n_e. The rules for the `after` scope are Aft and AftT for its timed variant. 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 slicing the trace. The rules Bef and BefT for the `before` scope are not given here for space reasons. They are symmetrical to Aft and AftT, respectively.

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_1 $\equiv$ `not (forall` x `in` e`, not` P_1`)`,

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 `precedes` B `within` $\delta \equiv$
  `within` δ `before each` B`, occurrence_of` A
- 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` B`, after first` A`,` P`)`

When events that are not explicitly associated to a variable name, they are bound to the empty variable name. Finally, omitted `where` conditions on events default to `true`.

5 IMPLEMENTATION AND VALIDATION

We implemented a trace verification tool based on the proposed language. It is written in Haskell and freely available online[1]. It checks the conformance of JSON-formatted traces to property expressions. All the proposed features have been implemented. However, the expressions allowed in `where` clauses are currently limited to simple boolean and arithmetic expressions along with access to event parameters. We plan to complete them with a foreign function interface to call routines written in Python, which is well suited for data manipulation and numeric computations.

Since we are performing offline verification only, we consider complete execution traces and not a prefix of them as in online monitoring. Having access to a full trace allows exploring it back and forth and not just sequentially. A direct consequence is that it is possible to interpret properties naively, in the sense that an interpreter may follow their syntactic structure. For instance, to check the property `before first A, P`, the interpreter may simply

[1] https://gricad-gitlab.univ-grenoble-alpes.fr/modmed/partrap

$$\frac{\tau \vDash_\eta P_1}{\tau \vDash_\eta P_1 \ \texttt{or} \ P_2}\ \textsc{DisjL} \qquad \frac{\tau \vDash_\eta P_2}{\tau \vDash_\eta P_1 \ \texttt{or} \ P_2}\ \textsc{DisjR}$$

$$\frac{\tau \nvDash_\eta P}{\tau \vDash_\eta \texttt{not} \ P}\ \textsc{Neg} \qquad \frac{\eta \vdash \mathit{expr} \downarrow L \quad \forall v \in L,\ \tau \vDash_{\eta[x \mapsto v]} P}{\tau \vDash_\eta \texttt{forall} \ x \ \texttt{in} \ \mathit{expr}\texttt{,} \ P}\ \textsc{Forall}$$

$$\frac{\eta \vdash n_e \downarrow n \quad n \leq |M(\tau, \sigma, x, c, \eta, \mathsf{each})|}{\tau \vDash_\eta \texttt{occurrence\_of} \ n_e \ \sigma \ x \ \texttt{where} \ c}\ \textsc{Occ}$$

$$\frac{\forall j \in M(\tau, \sigma, x, c, \eta, o),\ (\tau_i)_{i>j} \vDash_{\eta[x \mapsto \mathrm{param}(\tau_j)]} P}{\tau \vDash_\eta \texttt{after} \ o \ \sigma \ x \ \texttt{where} \ c\texttt{,} \ P}\ \textsc{Aft}$$

$$\frac{\eta \vdash \delta_e \downarrow \delta \quad \forall j \in M(\tau, \sigma, x, c, \eta, o),\ \mathrm{upto}((\tau_i)_{i>j}, \mathrm{time}(\tau_j)+\delta) \vDash_{\eta[x \mapsto \mathrm{param}(\tau_j)]} P}{\tau \vDash_\eta \texttt{within} \ \delta_e \ \texttt{after} \ o \ \sigma \ x \ \texttt{where} \ c\texttt{,} \ P}\ \textsc{AftT}$$

$$\frac{\forall j \in M(\tau, \sigma, x, c, \eta, o),\ \tau \vDash_{\eta[x \mapsto \mathrm{param}(\tau_j)]} P}{\tau \vDash_\eta \texttt{given} \ o \ \sigma \ x \ \texttt{where} \ c\texttt{,} \ P}\ \textsc{Given}$$

Figure 4: Operational semantics for property satisfaction

process the input trace forward, looking for an event A, and then process the trace until A again to check for P. This approach avoids the typical bookkeeping overhead of parametric monitoring and performs well in our context. For instance, verifying the moderately complex Runtime Property 3 on a trace of 11 120 events takes 202.63 µs on a first generation mobile Intel Core i5 (M520). This result does not include trace parsing, which account for about 6 ms.

We used the ParTraP interpreter to verify a dozen properties over the same corpus. Those properties encode requirements for TKA, hypotheses made on the device usage or usage queries. No errors were found in TKA software, but we discovered abnormal events and suspicious behaviors in the usage of the system. For instance, the periodically reported temperature of the 3D camera occasionally reaches the extreme value of −273. It is the result of a failure when querying the temperature sensor. Although not critical, this error illustrates an issue between the software and its execution environment.

Using a temporal usage query we also noticed that for 11 % of the traces in the corpus, the user performs an action within less than 100 ms after a new screen is displayed. Even experienced users could miss information in such a short time. After further investigation, the manufacturer attributed this issue to a defect in the pointer device or a poor handling of that same device. This defect is not critical because the surgeon may always go back to the missed screen.

6 RELATED WORK

We first present several specification formalisms classified by style and then offer some elements of comparison with ParTraP.

6.1 Overview

6.1.1 Pattern Systems. An important challenge in the design of ParTraP was to make temporal specification accessible. A famous technique to assist users with finite-state verification is to provide high-level temporal patterns with a verbose syntax. Dwyer et al. proposed a major step in that direction with a pattern system covering the temporal requirements of a very large study [15]. Those patterns are domain agnostic and relatively simple to use. Propel refines them with additional mandatory clauses which help to remove ambiguities due to the english-like syntax [26]. The Requirement Specification Language (RSL) is another pattern system with limited temporal relations but a high emphasis on real-time [13].

While being relatively easy to use, pattern systems have a severely limited expressiveness. The Salt language [9] addresses that issue with composable patterns and common temporal operators behind a natural syntax. It features many constructs, ranging from timed temporal operators to star-free regular expressions. All those systems share a common limitation as a consequence of supporting finite-state verification: they do not support parametric events.

6.1.2 Temporal Logics. The Linear Temporal Logic (LTL) is a widely accepted temporal logic with a precise semantics [22]. However, it targets infinite traces and its semantics is not adapted to runtime verification. Bauer et al. offer a formal treatment on adapting LTL for runtime verification, LTL_f, as well as its timed variant, TLTL_f, and discuss different trade-offs and their implications [7, 8]. The Counting Fluent Temporal Logic (CFLTL) is another interesting adaptation of LTL to event-based traces [25]. It features a simple and powerful mean of counting event occurrences and comparing them. However, none of those adaptations support parametric traces.

To deal with parametric traces, many first order temporal logics have been proposed, most of them also deriving from LTL. Stolz introduced free variables and quantification in next-free LTL with parametrized propositions [27]. $\mathrm{FO\text{-}LTL}^+$ [17] is another approach adding quantification to LTL. It targets data-rich XML traces and events parameters may exhibit a hierarchical structure, similarly to ParTraP. Parameters can be accessed through quantifiers over a domain described with XPath queries. In spite of the additional capabilities, $\mathrm{FO\text{-}LTL}^+$ retains the simplicity and conciseness of LTL.

However, we argue that the resulting elegance impairs pragmatism: the only way to access events is through quantification over their parameters. In consequence, single-valued parameters must also be quantified, which makes formulas harder to understand. Basin et al. proposed a semantics and algorithm for the Metric First Order Temporal Logic (MFOTL) [6] and implemented it with the MONPOLY tool [11]. They further extended MFOTL with SQL-like aggregation operators [5]. Eagle [2] is a radically different and powerful temporal logic as it does not derive from LTL, but still encompasses it. This very succint logic also features parametric events and data quantifiers.

6.1.3 Finite State Machines. Finite State Machines (FSM) form another popular specification formalism in runtime verification. Transitions are labelled with event descriptions and taken whenever a matching event is observed. Whereas pattern systems and temporal logics are declarative, FSM encode specifications in an operational style.

FSM are especially popular in the *parametric trace slicing* framework [12], where events are composed of a name and a tuple of parameters. For instance, FSM constitute the core of the expressive Quantified Event Automata [1] and its efficient implementation MarQ [23]. Mufin [14] also uses parametric trace slicing and relies exclusively on FSM. It is faster but less expressive than QEA/MarQ.

6.1.4 Rule-Based Systems. The use of rule-based systems for runtime verification has also received some interest. This model seems well suited for processing data rich events, as exemplified by the RuleR system [4] or the more efficient Logfire [18]. Rules take the form

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

where the conditions depend on a memory of facts and the action can add or remove facts, or yield a verdict. Constantly managing a database of facts through action rules makes rule-based system highly operational.

6.1.5 Hybrid. Some formalisms or monitoring tools support several specification styles.

LogScope [3] is a solution proposed in a context similar to the one of ParTraP. It targets offline verification for data-rich traces. LogScope is composed of a higher-level pattern language that compiles down to FSM, which can also be used to encode more complex properties. Despite some common characteristics with ParTraP, the higher-level language is different in many ways: its scoping mechanism is much more limited as there is a single scope (equivalent to our `before first` scope), quantification is completely implicit, time is not handled at language-level and specification of event sequences is much easier. This kind of two-levels systems have the advantage of providing a very simple high-level language. However, they also force the user to learn two formalisms, including a low-level one, often with different paradigms (e.g. declarative and operational for LogScope).

JavaMOP [21] also supports several formalisms, called “logics”. It is another incarnation of parametric trace slicing, being famous for making it efficient. Contrary to the two-level hierarchy found in LogScope, logics in JavaMOP are not cleary ordered and complement each other.

6.1.6 Stream Computation. All the solutions presented so far check properties on the execution of a system by observing events or state changes of the system. Lola is a runtime monitor for synchronous systems which takes a radically different approach: specification are written as computations over the stream of values manipulated by a synchronous system. This unique approach allows specification to resemble a synchronous program. Besides checking logical properties, Lola also supports numeric queries over streams. Lola was specifically designed for synchronous systems and adapting it for event traces is not obvious. For this reason, we omit Lola in the rest of this section.

6.2 Comparison and Discussion

Comparing the expressiveness of various temporal specification formalisms is difficult. For instance, Reger and Rydeheard showed that the relationship betwen First-Order Linear Temporal Logic for finite traces and parametric trace slicing is not trivial [24]. Instead of establishing a formal relation between ParTraP and the aforementioned formalisms, we propose to compare several aspects that are important when writing specifications.

Parametric Events. Supporting parametric events was a critical requirement for ParTraP. Although pattern systems and adaptations of LTL for finite traces do not support parametric events, most temporal specification formalisms designed for runtime verification do.

Compound Values. Parametric events may carry compound values such as lists and records. To the best of our knowledge, only a few of the formalism presented previously, including ParTraP, support further inspection of compound values.

Local vs. Global Quantification. We can distinguish two types of quantification in formalisms for parametric monitoring. In *global* quantification the domain value of a quantified variable is defined as the values taken by this variable in a whole trace. On the contrary, in *local* the quantification domain of a variable may only depend on the current state. Local quantification is mostly useful in combination with support for compound values as it allows quantifying over lists that are carried as event parameters. While ParTraP and a few others uses local quantification, all approaches based on parametric trace slicing use global quantification.

Reference to Past Data. It is well-known that adding past operators to LTL does not increase its expressiveness [16]. However, this result no longer holds when LTL is extended with quantifiers. To see why, consider the following informal specification: "for each value x in the parameter p_1 of an event e_1, there must be an occurrence of an event e_2 before e_1 and with a parameter p_2 equal to x" (this specification constitutes the core of Runtime Property 3). Because the occurrences of the event e_2 are constrained by the parameters of another event that is yet to occur, this property cannot be captured in a future-only LTL extended to first order. Monitoring such a property is expensive and many specification formalisms do not include past operators for efficiency and simplicity reasons. A possible approach – as taken in QEA [1] – is to explicitly store the data for future use, e.g. the set of values taken by the parameter p_2. In ParTraP, there is no notion of past or future because properties

Table 1: Comparison of several temporal specification languages

Language	Parametric	Comp. Values	Quantification	Ref. Past Data	Real-Time	Paradigm
Dwyer's Patterns	✗	n/a	n/a	n/a	✗	declarative
Propel	✗	n/a	n/a	n/a	✗	declarative
RSL	✗	n/a	n/a	n/a	✓	declarative
Salt	✗	n/a	n/a	n/a	✓	declarative
LTL$_f$	✗	n/a	n/a	n/a	✗	declarative
TLTL$_f$	✗	n/a	n/a	n/a	✓	declarative
CFLTL	✗	n/a	n/a	n/a	✗	declarative
Eagle	✓	✗	global	✗	✓	declarative
Stolz's Param. Prop.	✓	✗	local	✗	✗	declarative
FO-LTL$^+$	✓	✓	local	✗	✓	declarative
MFOTL/MONPOLY	✓	✗	global	✓	✓	declarative
JavaMOP	✓	✗	global	✓	✗	mixed
QEA/MarQ	✓	✗	global	✓	✗	operational
Mufin	✓	✗	global	✓	✗	operational
Ruler	✓	✗	n/a	✓	✗	operational
Logfire	✓	✗	n/a	✓	✗	operational
LogScope	✓	✓	global	✗	✗	mixed
ParTraP	✓	✓	local	✓	✓	declarative

are evaluated over whole slices of trace. In consequence, events and their data never belong to the past and always remain accessible. The aforementioned specification can be formalised with ParTraP as demonstrated in the encoding of Runtime Property 3.

Language Support for Real-Time. Real-time support can be classified according to three levels: unsupported, supported as regular data, supported at language-level. We argue that supporting time at language-level is critical since it allows much clearer specifications and more efficient implementation thanks to the monotonicity of time. For instance, one can stop checking a bounded safety property once the time bound has passed.

Paradigm. Specification formalisms may be classified according to their paradigm: declarative or operational. Even if they both have their strengths, as exemplified by Havelund and Reger [19], we argue that an operational style introduces additional complexity for the user. For instance, in rule-based system each rule may impact the behavior of the others by adding or removing shared facts, which requires careful attention. This problem is similar to imperative code routines where side-effects interplay is often critical. ParTraP uses a declarative style to operate at a higher level.

Summary. Table 1 summarises the comparison between ParTraP and the other specification languages mentioned previously. It appears from this table that ParTraP has a unique combination of features.

7 CONCLUSION

ParTraP is a specification language for parametric execution traces that is designed to be simple to use by software engineers thanks to a declarative style and a verbose syntax. Its development originated from an industrial cooperation with Medical Devices manufacturers. A large majority of the requirements of the studied system involves temporal constraints and data values. ParTraP allows formalising these requirements concisely, yet remaining readable. It was inspired by specification patterns and extends them with nested scopes, real-time and first-order quantification. Event parameters have a centric role and all ParTraP constructs may extract or exploit them.

A prototype implementation for verification of JSON traces is available online. This implementation has been used to verify several representative requirements for a medical device over a hundred surgery traces. It also showed promising results in spotting unusual behaviors of the system or the user.

Although ParTraP was developed in the context of offline runtime verification, we believe it could be adapted for online monitoring. This will require to rework the language semantics or remove some operators from the language. For example, constructs such as `last` may refer to different events while the trace is produced. Also, an `absence_of` property will only have a definitive verdict when the trace is completed (unless the events appears in the trace, falsifying the property). Moreover, intuitiveness and flexibility had higher priority than efficiency in the development of the language. While this is fine for offline verification, it may induce a significant overhead over the monitored system in the online case.

Although the design of ParTraP was driven by logs and requirements from the medical field, it is not specific to that particular field. The generation and gathering of execution traces is already a standard practice in many industries, but their systematic and automatic exploitation is not generalized. ParTraP should be transposable to any of them, and would be particularly useful for traces with complex structured data. We are currently working on a case study in the field of home automation, where traces include parametric information from sensors, and actions performed by the inhabitants and by the control system. At longer term, we have contacts with another medical devices manufacturer, and also envision log analysis of access control systems.

Finally, temporal specification in a flexible language remains delicate regardless of the efforts put into its ease of use. To help with this issue, we believe that the language should be complemented with tools to help writing and understanding properties. We are working on providing the means to analyze the verdict of ParTraP properties and to evaluate their coverage over a set of traces. In particular, we are studying the transformation of properties into disjunctive normal form, in order to measure the coverage of the sub-properties and get a finer understanding of the property.

ACKNOWLEDGMENTS

This work is funded by the ANR MODMED project (ANR-15-CE25-0010). We also thanks our partners Arnaud Clère (MinMaxMedical) and Fabrice Bertrand (BlueOrtho) for the fruitful discussions about TKA and the design of software for medical devices.

REFERENCES

[1] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. 2012. Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings (Lecture Notes in Computer Science)*, Dimitra Giannakopoulou and Dominique Méry (Eds.), Vol. 7436. Springer, 68–84. https://doi.org/10.1007/978-3-642-32759-9_9

[2] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. 2004. Rule-Based Runtime Verification. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings.* 44–57. https://doi.org/10.1007/978-3-540-24622-0_5

[3] Howard Barringer, Alex Groce, Klaus Havelund, and Margaret H. Smith. 2010. Formal Analysis of Log Files. *JACIC* 7 (2010), 365–390.

[4] Howard Barringer, David E. Rydeheard, and Klaus Havelund. 2010. Rule Systems for Run-time Monitoring: from Eagle to RuleR. *J. Log. Comput.* 20, 3 (2010), 675–706. https://doi.org/10.1093/logcom/exn076

[5] David A. Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zalinescu. 2015. Monitoring of temporal first-order properties with aggregations. *Formal Methods in System Design* 46, 3 (2015), 262–285. https://doi.org/10.1007/s10703-015-0222-7

[6] David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. 2015. Monitoring Metric First-Order Temporal Properties. *J. ACM* 62, 2 (2015), 15. https://doi.org/10.1145/2699444

[7] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2010. Comparing LTL Semantics for Runtime Verification. *J. Log. Comput.* 20, 3 (2010), 651–674. https://doi.org/10.1093/logcom/exn075

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

[9] Andreas Bauer, Martin Leucker, and Jonathan Streit. 2006. SALT - Structured Assertion Language for Temporal Logic. In *ICFEM (Lecture Notes in Computer Science)*, Vol. 4260. Springer, 757–775.

[10] Yoann Blein, Yves Ledru, Lydie du Bousquet, Roland Groz, Arnaud Clère, and Fabrice Bertrand. 2017. *MODMED WP1/D1: Preliminary Definition of a Domain Specific Specification Language.* Technical Report. LIG, MinMaxMedical, BlueOrtho.

[11] David A. Bversiooasin, Matús Harvan, Felix Klaedtke, and Eugen Zalinescu. 2011. MONPOLY: Monitoring Usage-Control Policies. In *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers (Lecture Notes in Computer Science)*, Sarfraz Khurshid and Koushik Sen (Eds.), Vol. 7186. Springer, 360–364. https://doi.org/10.1007/978-3-642-29860-8_27

[12] Feng Chen and Grigore Rosu. 2009. Parametric Trace Slicing and Monitoring. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science)*, Stefan Kowalewski and Anna Philippou (Eds.), Vol. 5505. Springer, 246–261. https://doi.org/10.1007/978-3-642-00768-2_23

[13] Werner Damm, Hardi Hungar, Bernhard Josko, Thomas Peikenkamp, and Ingo Stierand. 2011. Using contract-based component specifications for virtual integration testing and architecture design. In *Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14-18, 2011.* IEEE, 1023–1028. https://doi.org/10.1109/DATE.2011.5763167

[14] Normann Decker, Jannis Harder, Torben Scheffel, Malte Schmitz, and Daniel Thoma. 2016. Runtime Monitoring with Union-Find Structures. In *TACAS (Lecture Notes in Computer Science)*, Vol. 9636. Springer, 868–884. https://doi.org/10.1007/978-3-662-49674-9_54

[15] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. 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.*, Barry W. Boehm, David Garlan, and Jeff Kramer (Eds.). ACM, 411–420. http://portal.acm.org/citation.cfm?id=302405.302672

[16] Dov M. Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. 1980. On the Temporal Basis of Fairness. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*, Paul W. Abrahams, Richard J. Lipton, and Stephen R. Bourne (Eds.). ACM Press, 163–173. https://doi.org/10.1145/567446.567462

[17] Sylvain Hallé and Roger Villemaire. 2008. Runtime Monitoring of Message-Based Workflows with Data. In *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, 15-19 September 2008, Munich, Germany.* IEEE Computer Society, 63–72. https://doi.org/10.1109/EDOC.2008.32

[18] Klaus Havelund. 2015. Rule-based runtime verification revisited. *STTT* 17, 2 (2015), 143–170. https://doi.org/10.1007/s10009-014-0309-2

[19] Klaus Havelund and Giles Reger. 2015. Specification of Parametric Monitors. In *SyDe Summer School.* Springer, 151–189. https://doi.org/10.1007/978-3-658-09994-7_6

[20] Daniel Jackson and Jeannette Wing. 1996. Lightweight Formal Methods. *ACM Comput. Surv.* 28, 4 (1996), 121. https://doi.org/10.1145/242224.242380

[21] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Rosu. 2012. JavaMOP: Efficient parametric runtime monitoring framework. In *ICSE.* IEEE Computer Society, 1427–1430.

[22] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977.* IEEE Computer Society, 46–57. https://doi.org/10.1109/SFCS.1977.32

[23] Giles Reger, Helena Cuenca Cruz, and David E. Rydeheard. 2015. MarQ: Monitoring at Runtime with QEA. In *TACAS (Lecture Notes in Computer Science)*, Vol. 9035. Springer, 596–610. https://doi.org/10.1007/978-3-662-46681-0_55

[24] Giles Reger and David E. Rydeheard. 2015. From First-order Temporal Logic to Parametric Trace Slicing. In *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings.* 216–232. https://doi.org/10.1007/978-3-319-23820-3_14

[25] Germán Regis, Renzo Degiovanni, Nicolás D'Ippolito, and Nazareno Aguirre. 2015. Specifying Event-Based Systems with a Counting Fluent Temporal Logic. In *ICSE (1).* IEEE Computer Society, 733–743. https://doi.org/10.1109/ICSE.2015.86

[26] Rachel L. Smith, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. 2002. PROPEL: an approach supporting property elucidation. In *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA.* 11–21. https://doi.org/10.1145/581339.581345

[27] Volker Stolz. 2007. Temporal Assertions with Parametrised Propositions. In *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers (Lecture Notes in Computer Science)*, Oleg Sokolsky and Serdar Tasiran (Eds.), Vol. 4839. Springer, 176–187. https://doi.org/10.1007/978-3-540-77395-5_15

[28] Safouan Taha, Jacques Julliand, Frédéric Dadeau, Kalou Cabrera Castillos, and Bilal Kanso. 2015. A compositional automata-based semantics and preserving transformation rules for testing property patterns. *Formal Asp. Comput.* 27, 4 (2015), 641–664. https://doi.org/10.1007/s00165-014-0328-5