An environment for the ParTraP trace property language (tool demonstration)*

Ansem Ben Cheikh, Yoann Blein, Salim Chehida, German Vega, Yves Ledru✉, and Lydie du Bousquet

Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000, Grenoble, France
{Yoann.Blein,Yves.Ledru}@univ-grenoble-alpes.fr

Abstract. We present ParTraP and its associated toolset, supporting a lightweight approach to formal methods. In critical systems, such as medical systems, it is often easy to enhance the code with tracing information. ParTraP is an expressive language that allows to express properties over traces of parametric events. It is designed to ease the understanding and writing of properties by software engineers without background in formal methods. In this tool demonstration, we will present the language and its toolset: compiler, syntax directed editor, and a prototype generator of examples and counter-examples.

1 Introduction

Many applications, such as software intensive medical devices, require high quality software but their criticality does not mandate the use of formal proofs. Therefore, the formal methods community has promoted a lightweight approach to formal methods [17]. The ParTraP language [8] goes into that direction. Most computer systems can easily be augmented to produce traces of their activity. ParTraP allows to express properties of these traces which are evaluated by monitoring. ParTraP and its associated toolset are designed to support software engineers not trained in formal methods. The language supports a unique combination of features: it is parametric and provides temporal operators to increase expressiveness; it is declarative and uses descriptive keywords to favour user-friendliness. It reuses Dwyer's specification patterns [14] to express intuitive properties. Properties of parameters can be expressed in Python, a language familiar to our target users. Its toolset includes a syntax-directed editor. It generates detailed explanations to help understand why a property evaluates to true or false on a given trace. We also prototyped a generator of examples and counter-examples, to help users understand the meaning of their properties.

Section 2 gives an overview of the ParTraP language. Section 3 presents its associated toolset, and section 4 draws the conclusions and perspectives of this study. On the ParTraP web site[1], you will find a companion video demonstrating the tool, links to reference documents describing the syntax and semantics of ParTraP, and instructions on how to download the ParTraP eclipse plugin.

* This work is funded by the ANR MODMED project (ANR-15-CE25-0010).

[1] http://vasco.imag.fr/tools/partrap/

```
00. [{"state": "Connect", "id": "EnterState", "time": 4042.7},
01.  {"state": "Connect", "id": "ExitState", "time": 4042.7},
...
04.  {"id": "CameraConnected", "time": 4747.34},
...
13.  {"ty": "P", "id": "TrackerDetected", "time": 4783.8},
14.  {"ty": "T", "id": "TrackerDetected", "time": 4798.0},
15.  {"v1": 41.44, "id": "Temp", "time": 4816.6},
16.  {"state": "Save", "id": "EnterState", "time": 4816.611},
...
20.  {"v1": 55.62, "id": "Temp", "time": 4847.6}]
```

Fig. 1. A trace excerpt in JSON Format

2 The ParTraP Language

Context. The design of ParTraP was performed in cooperation with Blue Ortho, a medical devices manufacturer, and MinMaxMedical, a software company. Together, we considered a medical system (TKA) that guides Total Knee Arthroplasty surgeries, i.e. the replacement of both tibial and femoral cartilages with implants. TKA produces traces of the sensors acquisitions and interactions with the surgeon. Over time, more than 10 000 traces of actual surgeries have been collected by Blue Ortho. Each trace counts about 500 significant events. Fig. 1 gives a simplified excerpt of such a trace, in JSON format.

We identified 15 properties, listed in [19], representative of such medical devices for assisted surgery. These properties specify the TKA traces. A careful analysis of the properties revealed that they express temporal relations between events, involve event parameters, and may apply to a restricted scope of the trace. A few properties also refer to physical time or involve 3D calculations. Based on these properties, we designed ParTraP (Parametric Trace Property language)[8, 9]. The language is aimed at being used for *offline* trace checking. In [19], we discussed how the language can be used during development to express and check properties of traces produced by system tests, but also during exploitation in order to identify how the system is used or mis-used.

2.1 Structure of ParTraP Properties

A ParTraP temporal property is described by its scope in the trace, and a temporal pattern over events satisfying some predicate. For example, the following property expresses that *"once the camera is connected, the device temperature does not go below 45^{o}C"*.

```
VAlidTemp1 : after first CameraConnected,
                absence_of Temp t where t.v1 < 45.0
```

The first line defines the scope of the property, here it is the suffix of the trace starting after the first event of type `CameraConnected`. The second line expresses a temporal pattern. Here, it is an absence pattern, stating that no event should be a `Temp` whose `v1` parameter is less than 45.0. `t` is a local variable designating an event of type `Temp`. The `where` clause refers to this event and its parameter. The evaluation of this property on the trace of Fig. 1 yields false and returns the following message[2]:

```
[WARNING] False on trace unit-tests_Trace.json:
-In the scope from 5 to 20 with the environment:{}
found 1 event that should not occur:
{trace_occ_index=15, time=4816.6, id=Temp, v1=41.44}
```

Actually, event `CameraConnected` appears on line 4 of Fig. 1. So the scope of the property ranges from lines 5 to 20 which corresponds to the last event of the trace. The error message tells that line 15 features a `Temp` event which violates the property because its `v1` has value `41.44` which is actually below 45^{o}C.

ParTraP allows to express more complex properties. For example, property `ValidTrackers` in Fig. 3 features nested scopes and universal quantification to express that all types of trackers have been detected before entering a state whose name includes *'TrackersVisibCheck'*.

ParTraP offers a variety of operators to express scopes and patterns. Scopes refer to events located before or after an event or between two events. The scope may consider the first or the last occurrence of the event, but also be repeated for each occurrence of the event. Patterns may refer to the absence or occurrence of an event, but may also refer to pairs of events where one event enables or disables the occurrence of the other. Moreover, it is possible to express physical time constraints stating that a property holds within a time interval. Finally, expressions occurring in the `where` clause may be written either in a basic language with support for numeric and string expressions, or in Python as in [3]. Python allows to take advantage of software libraries to express complex or domain dependent properties, e.g. properties based on 3D calculations.

Basic ParTraP temporal properties can be combined using propositional logic connectors (`and`, `or`, `implies`, `equiv`) or quantifiers (`exists` and `forall`). ParTraP properties are defined on finite traces and evaluated after completion of the trace. A detailed description of the language is given in [8] or [9].

2.2 Related Work

Several languages based on temporal logic have been proposed to express trace properties. In [8], we compared ParTraP to several temporal specification languages using multiple criteria. Table 1 summarizes this study and groups languages with similar features combination. Please refer to [8] for detailed explanations about this table. The "Parametric" column indicates whether a language

[2] The message has been slightly simplified to fit in the size of the paper. See the console in Fig. 3 for the actual message.

Table 1. Comparison of ParTraP with several temporal specification languages

Language	Para-metric	Comp. values	Quan-tifica-tion	Ref. past data	Wall-clock time	Style
Dwyer's patterns [14], Propel [23], LTL_f [6], CFLTL [21]	✗	n/a	n/a	n/a	✗	decl.
RSL [22], Salt [7], TLTL_f [6]	✗	n/a	n/a	n/a	✓	decl.
Eagle [2]	✓	✗	global	✓	✗	decl.
Stolz's Param. Prop. [24]	✓	✗	local	✗	✗	decl.
FO-LTL^+ [15]	✓	✓	local	✗	✗	decl.
MFOTL/MONPOLY [5, 11]	✓	✗	global	✓	✓	decl.
JavaMOP [18]	✓	✗	global	✓	✗	mixed
QEA/MarQ [1, 20], Mufin [13]	✓	✗	global	✓	✗	oper.
Ruler [4], Logfire [16]	✓	✗	n/a	✓	✗	oper.
LogScope [3]	✓	✗	global	✗	✗	mixed
ParTraP	✓	✓	local	✓	✓	decl.

supports parametric events, i.e. events carrying data. If so, "Comp. values" specify if *compound values* in parameters (e.g. records or lists) may be exploited. If quantification is supported, it may be *global*, i.e. the domain value of a quantified variable is defined as the values taken by this variable in a whole trace, or *local*, where the quantification domain may only depend on the current state. The "Ref. past data" column indicates whether it is possible to use parameters values of past events. We also consider if physical time ("wall-clock time") is supported at the language level, in which case specifications involving timing constraints are easier to express. Finally, the specification style of a language can be *declarative*, *operational*, or *mixed* between the two and offers the choice to the user. As shown in Table 1, ParTraP supports a unique combination of these features, appropriate for our industrial context and motivated by the need for expressiveness. Other approaches, like [12], have similar goals as ours, and use a controlled natural language. However, the resulting language is domain specific and can not be applied to our industrial application.

3 Associated Toolset

ParTraP-IDE is a toolset designed to edit and execute the ParTraP language directly on a set of trace files. Given a set of properties, the tool provides the set of traces violating them and an explanation of the error causes.

ParTraP-IDE relies on the Eclipse IDE and the XText framework[3]. Xtext provides a complete infrastructure including: parser, lexer, typechecker and a compiler generator. Fig. 2 shows the ParTraP-IDE architecture. Part A presents how XText generates the toolset. Part B presents the usage of the tool.

[3] https://www.eclipse.org/Xtext

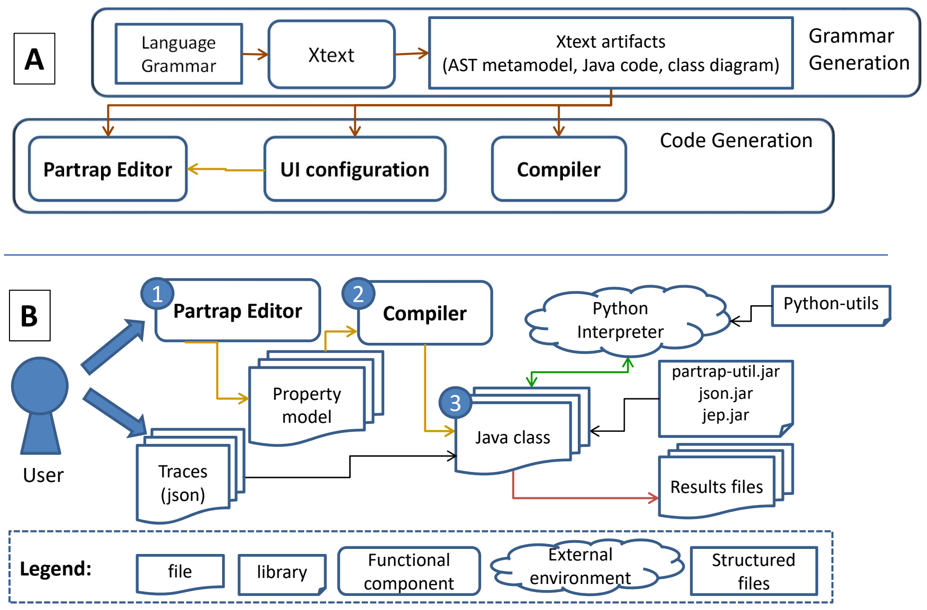


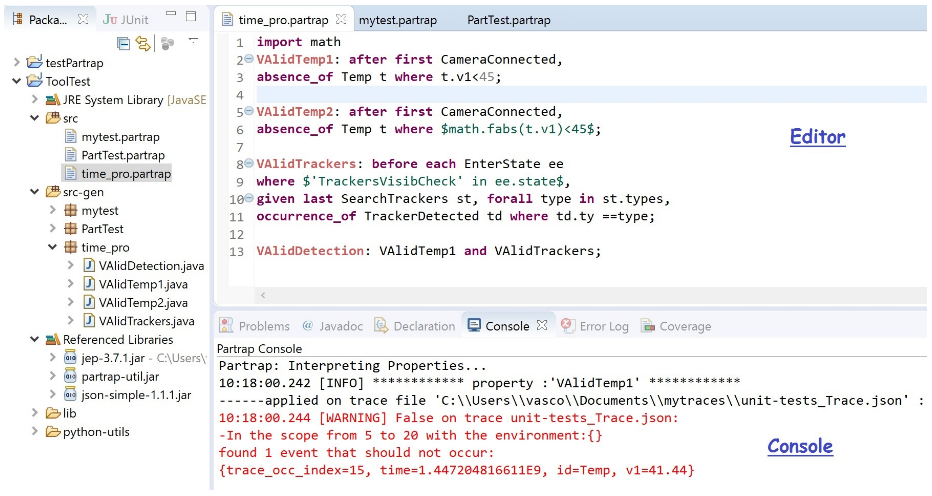
Fig. 2. Architecture of the ParTraP toolset

3.1 Tool Generation (Part A of Fig. 2)

This section deals with the tool generation architecture (Part A). The ParTraP Language grammar is defined in EBNF (XText's default grammar language). After being parsed, a set of language models is generated (AST metamodel, Java code and class diagram). These Xtext artifacts are used to configure the language editor and to generate a compiler that transforms each ParTraP property to a Java monitor. When a large set of properties is considered, ParTraP-IDE allows to compute the whole set of properties at the same time. It is less time consuming than executing separate Java classes for each property.

3.2 ParTraP-IDE by Example (Part B of Fig. 2)

The ParTraP *Editor* (1) helps the user to write syntactically correct properties. The configured editor provides syntax coloring according to concepts (name, keyword, python script,..) as shown in Fig. 3 under the editor window. Python expressions are delimited by dollars signs ('$') as featured by property `VAlidTemp2` in Fig. 3. Moreover, some validation constraints are enforced by the editor in order to forbid undesired language expressions like double use of property names or recursivity when referencing properties. Saving the file automatically calls the ParTraP compiler (2) and produces the set of Java classes under package 'src-gen' (see project explorer in Fig. 3).



```
import math
VAlidTemp1: after first CameraConnected,
absence_of Temp t where t.v1<45;

VAlidTemp2: after first CameraConnected,
absence_of Temp t where $math.fabs(t.v1)<45$;

VAlidTrackers: before each EnterState ee
where $'TrackersVisibCheck' in ee.state$,
given last SearchTrackers st, forall type in st.types,
occurrence_of TrackerDetected td where td.ty ==type;

VAlidDetection: VAlidTemp1 and VAlidTrackers;
```

```
Partrap: Interpreting Properties...
10:18:00.242 [INFO] ************ property :'VAlidTemp1' ************
------applied on trace file 'C:\\Users\\vasco\\Documents\\mytraces\\unit-tests_Trace.json' :
10:18:00.244 [WARNING] False on trace unit-tests_Trace.json:
-In the scope from 5 to 20 with the environment:{}
found 1 event that should not occur:
{trace_occ_index=15, time=1.447204816611E9, id=Temp, v1=41.44}
```

Fig. 3. Screen capture of the environment

Execution and Results. Execution takes two forms: running individual java classes or evaluating all properties simultaneously. The user provides a set of traces to be evaluated. Executing the property (3) produces a set of results files. Short logs only display the result (true or false) and an explanation for false cases. Detailed logs give information on calculated scopes, patterns and expressions results. A summary of false and true traces is provided for each property.

The example property presented in section 2.1 is typed in the editor of Fig. 3 and named `VAlidTemp1`. When applied on the trace of Fig. 1, the console reports that one event having the temperature value '41.44' violates this property.

Python Expressions Property `VAlidTemp2` is an alternate expression of the same property whose `where` clause is expressed in Python and uses the Python `math` module. As it is important for our envisioned users to define complex calculations in properties expressions, ParTrap properties support the integration of Python expressions using declared Python libraries. As a consequence, the designed IDE allows the import of Python modules and the execution of Python scripts. This is made possible by the use of JEP (Java Embedded Python)[4] which is a Python package generating a jar file 'jep.jar' added in the Java build path to exchange data and scripts between the JVM and the Python Interpreter.

Performance Although we traded performance off against expressiveness of the language, we carried out several experiments to check that the generated monitors featured sufficient performance in the context of our industrial partner, who typically collects and analyzes several dozens of traces every day. Therefore, we collected 100 traces from our partner[5]. The traces range from 304 to 1163

[4] https://github.com/ninia/jep
[5] These traces are not publicly available for confidentiality reasons.

Table 2. Performance evaluation (times in seconds)

Property	100 traces		1 trace with 521 events	
		with Python		with Python
1	0.307	2.988	0.064	0.105
2	0.326	2.378	0.064	0.088
3	n/a	1.736	n/a	0.070
4	17.605	51.639	0.255	0.728
5	0.674	2.170	0.065	0.116
6	1.517	4.969	0.074	0.118

events, with an average of 530 events. We evaluated the 6 ParTraP properties presented in chapter 5 of [10]. These properties typically combine a scope with a temporal pattern. For each property, we constructed a variant whose `where` clause is expressed in Python (except property 3 which already has its assertion in Python). We led the experiments on a Windows 10 machine with an Intel(R) Core(TM) i7-6600U CPU @ 2.60GHz, and 16 Go of RAM. Each experiment was performed 50 times and the average execution times are reported in table 2.

Column 2 reports the time in milliseconds to evaluate the property on 100 traces. The 100 traces are covered in a few seconds for each property. Property 4 takes longer because it features a complex scope involving pairs of events. Column 3 reports on the same properties but with their `where` clause expressed in Python. Their evaluations are slower because of the extra cost of interactions between the java monitor and the Python interpreter.

Columns 4 and 5 report on the time needed to evaluate a property on a single trace. Actually, the initialisation of the monitor involves some overhead independently of the number of traces. Hence, we arbitrarily selected one of the traces whose length, 521 events, was close to the average length of our set. As expected, the average time to evaluate each property is significantly longer than one hundredth of the time of columns 2 or 3.

In summary, these experiments show that ParTraP monitors perform well on traces provided by our partner. Most results are computed in less than one second, even if they involve Python assertions. These performances match the needs of our industrial partner. But further experiments should be led to evaluate how these performances scale up for much longer traces.

3.3 Example and Counter-Example Generator

To help software engineers understand or write ParTraP formulae, we are working on a prototype that generates examples and counter-examples and lets engineers check that they match their intuition of the meaning of the formula. We use the Z3 SMT solver[6]. Its input is a script composed of declarations (constants or functions) and assertions. Z3 computes whether the current assertions are satisfiable or not, and gives a valuation of the variables, for satisfiable formulae.

[6] https://github.com/Z3Prover/z3

We have defined the semantics of ParTraP operators as Z3 functions. ParTraP properties are translated as Z3 assertions which use these functions. Z3 then checks these assertions for satisfiability and, if possible, produces a trace satisfying the property. For example, the following Z3 assertion expresses property `VAlidTemp1` (absence of temperature below 45^{o}C after the camera is connected).

```
(assert (afterFirst "CameraConnected"
            (absence_of_where "Temp" t (< (v1 t) 45.0))))
```

This property is satisfiable and the solver generates the following example, which trivially satisfies the property by avoiding `Temp` events.

```
[{"id": "CameraConnected", "time": 5263, "v1": 2.0},
 {"id": "C",               "time": 5264, "v1": 0.0},
 {"id": "CameraConnected", "time": 5853, "v1": 4.0},]
```

Counter-examples are generated by evaluating satisfiability of the negation of the property. In our example, it produces the following trace where a `Temp` event with low temperature (`0.0`) is generated after the `CameraConnected` event.

```
[{"id": "C",               "time": 8,    "v1": 5.0},
 {"id": "a",               "time": 9,    "v1": 7.0},
 {"id": "CameraConnected", "time": 2436, "v1": 2.0},
 {"id": "Temp",            "time": 2437, "v1": 0.0},]
```

This part of the tool is currently at a prototyping stage. It will be included in the ParTraP distribution in the coming months. A major limitation of this tool is that it does not support Python expressions.

4 Conclusion

This paper has briefly presented ParTraP and its associated toolset. ParTraP is aimed at software engineers with poor knowledge of formal methods. Hence, we designed a keyword oriented language based on intuitive constructs such as Dwyer's patterns. We also integrated Python expressions in ParTraP properties to give access to domain specific libraries. Moreover, the evaluation of ParTraP expressions produces detailed logs to explain why a property was verified or failed. An examples and counter-examples generator is currently prototyped to help engineers understand the meaning of their formulae.

The companion video of this paper illustrates the main constructs of the language and shows how the toolset helps to edit them, generates Java monitors, evaluates properties and explains the result of their evaluation. The tool was successfully experimented on 6 properties evaluated on 100 traces of surgical operations, provided by our partner. Work in progress applies the tool to home automation traces. Future work will apply the tool to other medical devices.

References

1. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified event automata: Towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7436, pp. 68–84. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_9
2. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings. pp. 44–57 (2004). https://doi.org/10.1007/978-3-540-24622-0_5
3. Barringer, H., Groce, A., Havelund, K., Smith, M.H.: Formal analysis of log files. JACIC **7**(11), 365–390 (2010). https://doi.org/10.2514/1.49356
4. Barringer, H., Rydeheard, D.E., Havelund, K.: Rule systems for run-time monitoring: from Eagle to RuleR. J. Log. Comput. **20**(3), 675–706 (2010). https://doi.org/10.1093/logcom/exn076
5. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15 (2015). https://doi.org/10.1145/2699444
6. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14 (2011). https://doi.org/10.1145/2000799.2000800
7. Bauer, A., Leucker, M., Streit, J.: SALT - structured assertion language for temporal logic. In: Liu, Z., He, J. (eds.) Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4260, pp. 757–775. Springer (2006). https://doi.org/10.1007/11901433_41
8. Blein, Y., Ledru, Y., du Bousquet, L., Groz, R.: Extending specification patterns for verification of parametric traces. In: Proceedings of the 6th Conference on Formal Methods in Software Engineering, FormaliSE 2018, collocated with ICSE 2018, Gothenburg, Sweden, June 2, 2018. pp. 10–19. ACM (2018). https://doi.org/10.1145/3193992.3193998
9. Blein, Y., Ledru, Y., du Bousquet, L., Groz, R., Clère, A., Bertrand, F.: MODMED WP1/D1: Preliminary Definition of a Domain Specific Specification Language. Tech. rep., LIG, MinMaxMedical, BlueOrtho (2017)
10. Blein, Y., Tabikh, M.A., Ledru, Y.: MODMED WP4/D1: Test assessment - preliminary study and tool prototype. Tech. rep., LIG, MinMaxMedical, BlueOrtho (2017)
11. Bversiooasin, D.A., Harvan, M., Klaedtke, F., Zalinescu, E.: MONPOLY: monitoring usage-control policies. In: Khurshid, S., Sen, K. (eds.) Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7186, pp. 360–364. Springer (2011). https://doi.org/10.1007/978-3-642-29860-8_27
12. Calafato, A., Colombo, C., Pace, G.J.: A controlled natural language for tax fraud detection. In: Controlled Natural Language - 5th International Workshop, CNL 2016, Aberdeen, UK, July 25-27, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9767, pp. 1–12. Springer (2016). https://doi.org/10.1007/978-3-319-41498-0_1
13. Decker, N., Harder, J., Scheffel, T., Schmitz, M., Thoma, D.: Runtime monitoring with union-find structures. In: TACAS. Lecture Notes in Computer Science,

vol. 9636, pp. 868–884. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_54

14. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Boehm, B.W., Garlan, D., Kramer, J. (eds.) Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999. pp. 411–420. ACM (1999). https://doi.org/10.1145/302405.302672
15. Hallé, S., Villemaire, R.: 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. pp. 63–72. IEEE Computer Society (2008). https://doi.org/10.1109/EDOC.2008.32
16. Havelund, K.: Rule-based runtime verification revisited. STTT **17**(2), 143–170 (2015). https://doi.org/10.1007/s10009-014-0309-2
17. Jackson, D., Wing, J.: Lightweight formal methods. ACM Comput. Surv. **28**(4), 121 (1996). https://doi.org/10.1145/242224.242380
18. Jin, D., Meredith, P.O., Lee, C., Rosu, G.: JavaMOP: Efficient parametric runtime monitoring framework. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland. pp. 1427–1430. IEEE Computer Society (2012). https://doi.org/10.1109/ICSE.2012.6227231
19. Ledru, Y., Blein, Y., du Bousquet, L., Groz, R., Clère, A., Bertrand, F.: Requirements for a trace property language for medical devices. In: 2018 IEEE/ACM International Workshop on Software Engineering in Healthcare Systems, SEHS@ICSE 2018, Gothenburg, Sweden, May 28, 2018. pp. 30–33. ACM (2018), http://ieeexplore.ieee.org/document/8452638
20. Reger, G., Cruz, H.C., Rydeheard, D.E.: MarQ: Monitoring at runtime with QEA. In: TACAS. Lecture Notes in Computer Science, vol. 9035, pp. 596–610. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_55
21. Regis, G., Degiovanni, R., D'Ippolito, N., Aguirre, N.: Specifying event-based systems with a counting fluent temporal logic. In: ICSE (1). pp. 733–743. IEEE Computer Society (2015). https://doi.org/10.1109/ICSE.2015.86
22. Reinkemeier, P., Stierand, I., Rehkop, P., Henkler, S.: A pattern-based requirement specification language: Mapping automotive specific timing requirements. In: Reussner, R.H., Pretschner, A., Jähnichen, S. (eds.) Software Engineering 2011 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik, 21.-25.02.2011, Karlsruhe. LNI, vol. 184, pp. 99–108. GI (2011), http://subs.emis.de/LNI/Proceedings/Proceedings184/article6316.html
23. Smith, R.L., Avrunin, G.S., Clarke, L.A., Osterweil, L.J.: 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. pp. 11–21 (2002). https://doi.org/10.1145/581339.581345
24. Stolz, V.: Temporal assertions with parametrised propositions. In: Sokolsky, O., Tasiran, S. (eds.) Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers. Lecture Notes in Computer Science, vol. 4839, pp. 176–187. Springer (2007). https://doi.org/10.1007/978-3-540-77395-5_15