Writing Domain-Specific Languages for BeepBeep: 18th International Conference, RV 2018, Limassol, Cyprus, November 10–13, 2018, Proceedings

Chapter · November 2018

DOI: 10.1007/978-3-030-03769-7_27

2 authors:

Sylvain Hallé
University of Québec in Chicoutimi
126 PUBLICATIONS **660** CITATIONS

SEE PROFILE

Raphaël Khoury
University of Québec in Chicoutimi
31 PUBLICATIONS **118** CITATIONS

SEE PROFILE

Writing Domain-Specific Languages for BeepBeep

Sylvain Hallé and Raphaël Khoury

Laboratoire d'informatique formelle
Université du Québec à Chicoutimi, Canada

Abstract. This paper describes a plug-in extension of the BeepBeep 3 event stream processing engine. The extension allows one to write a custom grammar defining a particular specification language on event traces. A built-in interpreter can then convert expressions of the language into chains of BeepBeep processors through just a few lines of code, making it easy for users to create their own domain-specific languages.

1 Introduction

The field of Runtime Verification (RV) has seen a proliferation of specification languages over the years. Among the various formal notations that have been put forward, we can mention logic-based specifications such as LTL-FO$^+$ [15] and MFOTL [3]; automata-based specifications like DATE [6] and QEA [2]; stream-based languages like ArtiMon [18], Lola [7], Lustre [10] and TeSSLa [8]. Each specification language seems to have a "niche" of problem domains whose properties they can be expressed more easily and more clearly than others. The recent trend towards the development of *domain-specific languages* (DSL) can be seen as a natural consequence of this observation. As its name implies, a DSL is a custom language, often with limited scope, whose syntax is designed to express properties of a particular nature in a succinct way.

Unfortunately, current RV tools are often implemented to evaluate expressions of a single language following a single grammar. They offer very few in the way of easily customizing their syntax to design arbitrary DSLs. In this respect, the BeepBeep event stream processor is designed differently, as it does not provide any imposed, built-in query language. However, a special extension to the system's core makes it possible for a user to define the grammar for a language of their choice, and to set up an interpreter that can build chains of processor objects for expressions of that language. In this paper, we describe BeepBeep's DSL plug-in, and illustrate its purpose by showing how to build interpreters allowing BeepBeep to read and evaluate specifications of multiple existing specification languages, each in less than a hundred lines of Java code.

2 An Overview of BeepBeep 3

BeepBeep 3 is an event stream processing engine implemented as an open source Java library.[1] It is organized around the concept of *processors*. In a nutshell, a processor is a

[1] `https://liflab.github.io/beepbeep-3`

basic unit of computation that receives one or more event traces as its input, and produces one or more event traces as its output. BeepBeep's core library provides a handful of generic processor objects performing basic tasks over traces; they can be represented graphically as boxes with input/output "pipes", as is summarized in Figure 1.

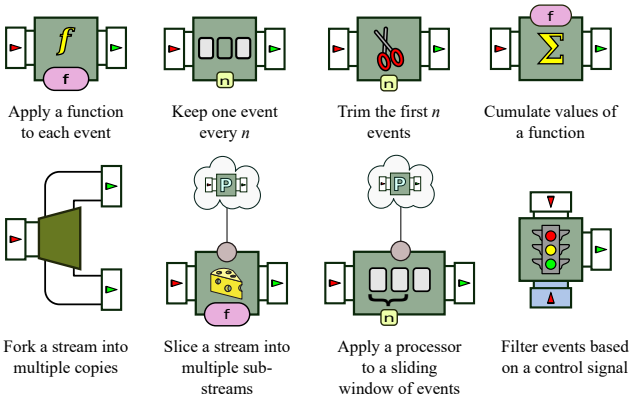


Fig. 1: BeepBeep's basic processors.

In order to create custom computations over event traces, BeepBeep allows processors to be *composed*; this means that the output of a processor can be redirected to the input of another, creating complex processor chains. Events can either be *pushed* through the inputs of a chain, or *pulled* from the outputs, and BeepBeep takes care of managing implicit input and output event queues for each processor. In addition, users also have the freedom of creating their own custom processors and functions, by extending the `Processor` and `Function` objects, respectively. Extensions of BeepBeep with predefined custom objects are called *palettes*; there exist palettes for various purposes, such as signal processing, XML manipulation, plotting, and finite-state machines.

Over the past few years, BeepBeep has been involved in a variety of case studies [4, 12–14, 17, 19], and provides built-in support for multi-threading [16]. For a complete description of BeepBeep, the reader is referred to a recent tutorial [11] or to BeepBeep's detailed user manual [1].

3 The DSL Palette

Like many other extensions, BeepBeep's DSL capabilities come in the form of an auxiliary JAR library (a *palette*) for creating languages and parsing expressions. It can be freely downloaded from the palette repository[2]. This palette provides a special object for creating domain-specific languages called the `GrammarObjectBuilder`. The operation of the `GrammarObjectBuilder` can be summarized as follows: 1. The object builder

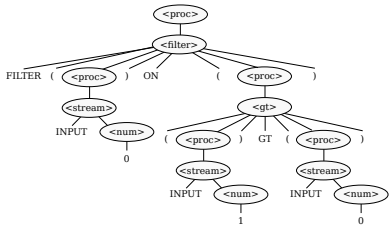
[2] `https://github.com/liflab/beepbeep-3-palettes/`

is given the syntactical rules of the language in the form of a Backus-Naur grammar. 2. Given an expression of the language in a character string, the object builder parses the expression according to the grammar and produces a parsing tree. 3. The builder then performs a postfix traversal of the tree and progressively builds the object represented by the expression.

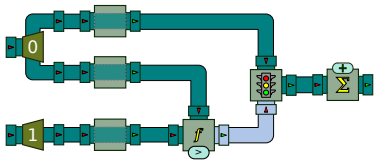
We shall illustrate the operation of the `GrammarObjectBuilder` using a very simple example, and then show how it has been used to implement interpreters for various existing languages. The first step consists of defining the grammar for the targeted language and expressing it in Backus-Naur Form (BNF). In our simple example, suppose the language only supports a few constructs: filtering, comparing numbers with the greater-than operator, summing numbers, and referring to an input stream by a number. A possible way to organize these functionalities into a language would be the small grammar shown in Figure 2a. Given such a grammar, an expression like "FILTER (INPUT 0) ON ((INPUT 1) GT (INPUT 0))" is syntactically correct; parsing it results in the tree shown in Figure 2b.

⟨*proc*⟩ := ⟨*filter*⟩ | ⟨*sum*⟩ | ⟨*gt*⟩ | ⟨*stream*⟩
⟨*filter*⟩ := FILTER (⟨*proc*⟩) WITH (⟨*proc*⟩)
⟨*sum*⟩ := THE SUM OF (⟨*proc*⟩)
⟨*gt*⟩ := (⟨*proc*⟩) > (⟨*proc*⟩)
⟨*stream*⟩ := INPUT ⟨*num*⟩ ;
⟨*num*⟩ := ^[0-9]+;

(a)



(b)



(c)

Fig. 2: A simple grammar (a); the parsing tree for the expression "FILTER (INPUT 0) ON ((INPUT 1) GT (INPUT 0))" (b); the chain of processors created by the `GrammarObjectBuilder` from this expression.

The next step is to create a new interpreter, which extends `GrammarObjectBuilder` and implements handler methods for the various symbols of the grammar. The parsing tree is traversed in postfix fashion using the Visitor design pattern; Java objects are pushed and pulled from a persistent stack. By default, the `GrammarObjectBuilder` treats any terminal symbol of the tree as a character string. Therefore, when visiting a leaf of the parsing tree, the builder puts on its stack a `String` object whose value is the contents of that specific literal. When visiting a parse node that corresponds to

a non-terminal token, such as <gt>, the builder looks for a method that handles this symbol. "Handling" a symbol generally means popping objects from the stack, creating one or more new objects, and pushing some of them back onto the stack.

Let us start with a simple case, that of the <gt> symbol. When a <gt> node is visited in the parsing tree, as per the postfix traversal we described earlier, we know that the top of the stack contains two strings with the constants that were parsed earlier. The task of the handler method is to create a new processor evaluating the "greater than" function, pipe into the inputs of this processor the two objects popped from the stack, and push this new processor back onto the stack. Therefore, we can create a method called `handleGt` that reads as follows:

```
@Builds(rule="<gt>")
public void handleGt(ArrayDeque<Object> stack) {
  ApplyFunction af = new ApplyFunction(Numbers.greaterThan);
  Connector.connect((Processor) stack.pop(), 0, af, 1);
  Connector.connect((Processor) stack.pop(), 0, af, 0);
  stack.push(af);
}
```

The `ApplyFunction` and `Connector` objects are part of BeepBeep's core library. The method's name is irrelevant; the `Builds` annotation at the beginning of the method is used to signal the object builder what non-terminal symbol of the grammar this method handles.

Special attention must be given to the manipulations corresponding to the <stream> grammar rule. This rule refers to an input stream from which the events are expected to be produced. Internally, the `GrammarObjectBuilder` maintains a set of `Fork` processors for each of the inputs referred to in the query. A call to a special method `forkInput` fetches the fork corresponding to the input pipe at position n, adds one new branch to that fork, and connects a `Passthrough` processor at the end of it. This `Passthrough` is then returned.

As an example, Figure 2c shows the chain of processor objects created through manipulations of the stack for the expression we mentioned earlier. As we can see, the `GrammarObjectBuilder` takes care of a good amount of the tedious task of parsing a string and performing specific actions for each non-terminal symbol of a grammar. In the example shown here, a complete running interpreter for expressions of the language can be obtained for 6 lines of BNF grammar and 30 lines of Java code[3]. With some experience, such an interpreter can be written in a few minutes.

4 Extending Existing Specification Languages

The example shown in the previous section is only meant to illustrate the operation of the `GrammarObjectBuilder` through a very simple case. Several features available in the DSL palette have been left out due to lack of space. For instance, the `GrammarObjectBuilder` can also be used to build BeepBeep's `Function` objects instead of

[3] The code for the interpreter can be found in the BeepBeep example repository: `https://liflab.github.io/beepbeep-3-examples/classdsl_1_1_simple_processor_builder.html`

processors, and additional annotations can be appended to handler methods in order to further simplify the manipulations of the object stack.[4].

Obviously, the syntax for the language does not need to look like the example we have shown earlier, and it is not necessary to impose a one-to-one correspondence between grammar rules and BeepBeep's `Processor` objects. A single rule can spawn and push on the stack as many objects as one wishes, making it possible for a short grammatical construct to represent a potentially complex chain of processors under the hood. In the following, we briefly describe interpreters for three existing languages that have been implemented using BeepBeep's DSL palette. In all three cases, additional functionalities have been included into the original language "for free", by taking advantage of BeepBeep's available palettes and generic event model. All the interpreters described in this section are freely available online.[5]

4.1 Linear Temporal Logic

The four basic operators of Linear Temporal Logic (**F**, **G**, **X** and **U**) can easily be accommodated by a simple LTL palette that was already discussed in BeepBeep's original tutorial [11, §5.1]. The grammar and stack manipulations for handling these operators, as well as Boolean connectives, is straightforward and results in an interpreter with around 50 lines of Java code. We shall rather focus on the extensions to that have been added to the original LTL by leveraging BeepBeep's architecture.

Arbitrary Ground Terms Special syntactical rules for ground terms can be added to LTL's syntax, depending on the underlying trace's type. For example, if events in a trace are made of numeric values, the ground terms of the language can be defined as arithmetic operations over numbers (many of which are already included in BeepBeep's core); if events are XML or JSON documents, ground terms can be XPath or JPath expressions fetching fragments of these events, using BeepBeep's XML and JSON palettes, respectively.

First-order Quantification BeepBeep's `Slice` processor performs exactly the equivalent of LTL-FO$^+$'s first-order quantifiers. This places the expressiveness of BeepBeep 3's LTL interpreter to the level of its ancestor, BeepBeep 1 [15].

New Operators Additional temporal operators can easily be defined as new `Processor` objects and added to LTL's syntax. As an example, our LTL interpreter adds an operator **C**, which counts the number of times a formula is true on a given stream. For example, the expression $\mathbf{G}\,(\neg c\,\mathbf{U}\,(\mathbf{C}\,(a \wedge \mathbf{X}\,b) = 3)$ expresses the fact that c cannot hold until a has been immediately followed by b three times.

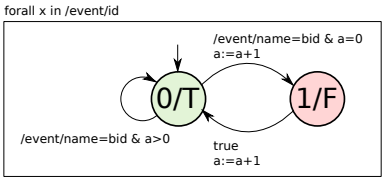
4.2 Quantified Event Automata

Quantified Event Automata (QEAs) is the formalism used by the MarQ runtime monitor [2]. BeepBeep has a palette called FSM that allows users to define generalized Moore

[4] More details can be found in a dedicated chapter of BeepBeep's user manual [1]

[5] `https://github.com/liflab/beepbeep-3-polyglot`

Machines, whose expressiveness is similar to QEAs. In its current version, MarQ does not provide an input language, and QEAs need to be built programmatically using Java objects. Our interpreter proposes a tentative syntax, shown in Figure 3, reminiscent of the Graphviz library for drawing graphs [9].



(a)

```
forall x in /event/id {
  0 -> 1 [/event/name = bid & a=0 : a=a+1];
  1 -> 0 [true : a=a+1];
  0 -> 0 [/event/name = bid & a>0 : ];
  0 [true];
  1 [false];
}
```

(b)

Fig. 3: A simple QEA with two states (a); a textual notation that defines the same automaton (b).

The quantifier part of the QEA is taken care of by BeepBeep's `Slice` processor. The interpreter for this language is interesting in that the postfix traversal of the parsing tree does not create a chain of processors, but rather updates a single `MooreMachine` object (itself a descendant of `Processor`) with new transitions; the Moore machine is repeatedly popped and pushed on the stack at each new transition. The interpreter also generalizes the original QEAs in a few ways.

Event Types As with the LTL interpreter, the syntax for fetching event content is type-dependent. The code example above shows that quantifiers and guards on transitions are written as XPath conditions, suitable for events in the XML format.

State Symbols Since the automaton is a Moore machine, each state can be set to output a symbol when visited. Moreover, this symbol does not need to be a Boolean value. The quantifier part of the automaton is actually an aggregation function over the set of the last symbols output by each instance of the automaton; the `forall` statement is the special case of conjunction over Boolean values. For instance, one can associate a number to each state, and compute the `sum` of these numbers as the output of the quantified automaton.

4.3 Lola

Our last example is an interpreter for version 1.0 of Lola, which is the specification language for the runtime monitor of the same name [7]. A Lola specification is a set of equations over typed stream variables. Figure 4 shows an example of such a specification, taken from the original paper, and summarizing most of the language's features. It defines ten streams, based on three independent variables t_1, t_2 and t_3. A stream expression may involve the value of a previously defined stream. The values of the streams corresponding to s_1 to s_6 are obtained by evaluating their defining expressions place-wise at each position.

$$s_1 = t_1 \vee (t_3 \leq 1) \qquad s_4 = \text{ite}(t_1; t_3 \leq s_4; \neg s_3)$$
$$s_2 = ((t_3)^2 + 7) \mod 15 \qquad s_5 = t_1[+1; \text{false}]$$
$$s_3 = \text{ite}(s_3; s_4; s_4 + 1) \qquad s_6 = s_9[-1; 0] + (t_3 \mod 2)$$

Fig. 4: An example of a Lola specification showing various features of the language.

Using the DSL palette, the complete BeepBeep interpreter for Lola 1.0 has less than 100 lines of Java code. Since the original Lola language corresponds to a subset of BeepBeep's existing processors and features, we proceeded as in the previous examples and added a few new features to it.

Generalized Offset The original Lola construct $s[n, c]$ denotes the value of stream s, offset by n events from the current position, with n a fixed integer. In contrast, the construction of $s[x, c]$ in BeepBeep accepts constructs of the form $s[ax + n, c]$, where x is the index of the current position in the stream and $a \in \mathbb{N}$.

Filtering and Non-Uniform Processors In Lola, every stream is expected to be *uniform*: exactly one output is produced for each input. In BeepBeep however, processors do not need to be uniform. One example is the `Filter` processor, which may discard events from its input trace based on a condition. We can hence create the Lola construct "`filter` φ `on` ψ", where φ and ψ are two arbitrary stream expressions (with ψ of Boolean type). For example, to let out only events that are positive, one can write `filter` t `on` $t > 0$.

Scoping and Sub-streams Lola lacks a scoping mechanism for defining streams. If the same kind of processing (requiring intermediate streams) must be done on multiple input streams, this processing must be repeated for each input stream. Moreover, since stream specifications all live in the same global scope, intermediate streams must be given different names to avoid clashes. Our extended version of Lola provides a new construct, called `define`, that allows a user to create a processing chain and encapsulate it as a named object with parameters. Consider the following specification:

```
define $p(x1, x2)
  y1 = ite(x1 < x2, x2[1, 0], x2)
  y2 = ite(x2 < x1, x2[-1, 0], x1)
  out = x1 + x2
end define
s1 = $p(t1, t2)
s2 = $p(t2, t3)
```

It starts with a `define` block, which creates a new template stream called p, which takes two input streams called x_1 and x_2. The next two lines define two intermediate streams, and the last specifies the output of p, using the reserved stream name *out*. From then on, p can be used in an expression wherever a stream name is accepted. The next two lines show how streams s_1 and s_2 are defined by applying p to the input stream pairs (t_1, t_2) and (t_2, t_3), respectively. Streams y_1 and y_2 exist only in the local scope of p.

Generalized Windows User-defined blocks open the way to generic sliding window processors. Version 2.0 of Lola already supports classical aggregation functions over a sliding window, such as sum or average. In contrast, BeepBeep provides a generic `Window` processor, which can make a window out of any event trace and apply any processor to the contents of the window. Moreover, this window can be set as the input of an arbitrary chain of other processors, and receive input from an arbitrary chain of other processors. We can therefore create the generalized construct "`window($p(s),n)`". Here, *p* can be any sub-stream name, defined according to the syntax described above.

4.4 Additional features

In addition to BeepBeep's core processors and functions, functionalities of external palettes can also be added to an interpreter. For example, new grammatical constructs can be defined to use the Signal palette, which provides a peak-finding processor on an incoming stream of numerical values. Should these extensions prove to be insufficient, we remind the reader that any other extension can also be designed by creating new processors and functions, and adding them to an interpreter through the means described in this paper. Therefore, BeepBeep can turn out to be a convenient testbed for trying out new monitoring features, while leveraging the existing syntax of another language.

One last interesting feature of BeepBeep's DSL palette is the possibility to *mix* multiple languages in the same specification. Case in point, we compiled a "multi-interpreter", called *Polyglot*, which is able to read specifications from input files, and to dispatch them to the proper interpreter instance based on the file extension. If multiple files are specified, the output of processor chain built from file *n* is piped into the input of processor chain built from file $n + 1$. It is therefore possible to call the multi-interpreter from the command line as follows:

```
$ java -jar polyglot.jar spec1.qea spec2.lola spec3.ltl
```

This would in effect evaluate a specification that is a mix of a QEA, piped into a set of Lola equations, whose output is sent to an LTL formula. To the best of our knowledge, BeepBeep's Polyglot extension is one of few tools that provides such a flexible way of accepting specifications.[6]

5 Conclusion and Future Work

In this paper, we have seen how an extension of the BeepBeep event stream processing engine allows a user to easily define the syntax and construction rules for arbitrary domain-specific languages. The DSL palette provides facilities for parsing expressions according to a grammar, and takes care of many tedious tasks related to the processing of the parsing tree. Combined with BeepBeep's generic streaming model and large inventory of available processors and functions, we have seen how the DSL palette also makes it possible to write interpreters for a variety of *existing* specification languages, and even add new features to them.

[6] The other being MOP [5], which also handles specifications in multiple languages.

References

1. Event stream processing with BeepBeep 3, https://liflab.gitbooks.io/event-stream-processing-with-beepbeep-3
2. 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), http://dx.doi.org/10.1007/978-3-642-32759-9_9
3. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM 62(2), 15:1–15:45 (2015), http://doi.acm.org/10.1145/2699444
4. Boussaha, M.R., Khoury, R., Hallé, S.: Monitoring of security properties using beepbeep. In: Imine, A., Fernandez, J.M., Marion, J., Logrippo, L., García-Alfaro, J. (eds.) Foundations and Practice of Security - 10th International Symposium, FPS 2017, Nancy, France, October 23-25, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10723, pp. 160–169. Springer (2017), https://doi.org/10.1007/978-3-319-75650-9_11
5. Chen, F., Rosu, G.: Mop: an efficient and generic runtime verification framework. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S. (eds.) Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada. pp. 569–588. ACM (2007), http://doi.acm.org/10.1145/1297027.1297069
6. Colombo, C., Pace, G.J.: Runtime verification using LARVA. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA. Kalpa Publications in Computing, vol. 3, pp. 55–63. EasyChair (2017), http://www.easychair.org/publications/paper/Jwmr
7. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA. pp. 166–174. IEEE Computer Society (2005), https://doi.org/10.1109/TIME.2005.26
8. Decker, N., Gottschling, P., Hochberger, C., Leucker, M., Scheffel, T., Schmitz, M., Weiss, A.: Rapidly adjustable non-intrusive online monitoring for multi-core systems. In: da Costa Cavalheiro, S.A., Fiadeiro, J.L. (eds.) Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10623, pp. 179–196. Springer (2017), https://doi.org/10.1007/978-3-319-70848-5_12
9. Gansner, E.R., Koutsofios, E., North, S.: Drawing graphs with dot (2015), http://www.graphviz.org/pdf/dotguide.pdf
10. Halbwachs, N., Lagnier, F., Ratel, C.: Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. IEEE Trans. Software Eng. 18(9), 785–793 (1992)
11. Hallé, S.: When RV meets CEP. In: Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings. pp. 68–91 (2016), https://doi.org/10.1007/978-3-319-46982-9_6
12. Hallé, S., Gaboury, S., Bouchard, B.: Activity recognition through complex event processing: First findings. In: Bouchard, B., Giroux, S., Bouzouane, A., Gaboury, S. (eds.) Artificial Intelligence Applied to Assistive Technologies and Smart Environments, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016. AAAI Workshops, vol. WS-16-01. AAAI Press (2016), http://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12561

13. Hallé, S., Gaboury, S., Bouchard, B.: Towards user activity recognition through energy usage analysis and complex event processing. In: Proceedings of the 9th ACM International Conference on PErvasive Technologies Related to Assistive Environments, PETRA 2016, Corfu Island, Greece, June 29 - July 1, 2016. p. 3. ACM (2016), http://dl.acm.org/citation.cfm?id=2910707
14. Hallé, S., Gaboury, S., Khoury, R.: A glue language for event stream processing. In: Joshi, J., Karypis, G., Liu, L., Hu, X., Ak, R., Xia, Y., Xu, W., Sato, A., Rachuri, S., Ungar, L.H., Yu, P.S., Govindaraju, R., Suzumura, T. (eds.) 2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016. pp. 2384–2391. IEEE (2016), https://doi.org/10.1109/BigData.2016.7840873
15. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Services Computing 5(2), 192–206 (2012), https://doi.org/10.1109/TSC.2011.10
16. Hallé, S., Khoury, R., Gaboury, S.: Event stream processing with multiple threads. In: Havelund, K., Lahiri, S., Reger, G. (eds.) Runtime Verification - 8th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017. Proceedings. Lecture Notes in Computer Science, Springer (2017)
17. Khoury, R., Hallé, S., Waldmann, O.: Execution trace analysis using LTL-FO ˆ+. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9953, pp. 356–362 (2016), https://doi.org/10.1007/978-3-319-47169-3_26
18. Rapin, N.: Reactive property monitoring of hybrid systems with aggregation. In: Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings. pp. 447–453 (2016), https://doi.org/10.1007/978-3-319-46982-9_28
19. Varvaressos, S., Lavoie, K., Gaboury, S., Hallé, S.: Automated bug finding in video games: A case study for runtime monitoring. Computers in Entertainment 15(1), 1:1–1:28 (2017), http://doi.acm.org/10.1145/2700529