Kalpa Publications in Computing
Volume 3, 2017, Pages 81–88
RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools

Event Stream Processing with BeepBeep 3

Sylvain Hallé and Raphaël Khoury

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

Abstract

This paper is a short introduction to the BeepBeep 3 event stream processor. It highlights the main design decisions that informed its development, and the features that distinguish it from other Runtime Verification tools.

1 Introduction

This paper is a very brief introduction to the BeepBeep 3 event stream processing engine. Although BeepBeep can be used as a Runtime Verification (RV) tool, many of its features extend beyond classical RV, and borrow from the related field of Complex Event Processing (CEP). A recent tutorial highlights the similarities and differences between these two domains, and illustrates how BeepBeep 3 straddles the line that separates them [10].

Version 1 of BeepBeep was developed from 2008 to 2013 and has been the subject of numerous papers and case studies [17–19, 23]. The main distinguishing point of this first version was the handling of complex events with a nested structure (such as XML documents), and an input language extending propositional Linear Temporal Logic with XML path expressions and first-order quantifiers. BeepBeep 1 is no longer under active development and is considered obsolete for all practical purposes. Version 2 was an attempt at implementing the same concepts as BeepBeep 3, which has been scrapped at an early stage of development and was never officially released. One can hence consider BeepBeep 3 as the second "real" incarnation of BeepBeep. It benefits from a complete redesign of the platform, which includes and significantly extends most of the 1.x features.

BeepBeep 3 has been under development since 2014, and is still actively maintained and extended. It is freely available online under the GNU Lesser General Public License.[1] Its online code repository lists a total of more than 350 commits over a two-year span. In addition to the aforementioned tutorial, several research papers have already covered various aspects of the tool's design [9, 12–15]. For detailed information about BeepBeep, including an extensive review of related work and numerous examples, the reader is referred to a recent technical report [11] and to BeepBeep's online documentation.

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

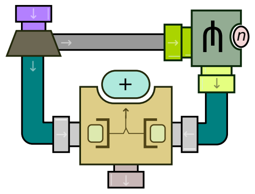

Figure 1: A simple composition of processors, represented graphically

2 Related Work

Current event processing tools and techniques can roughly be divided into two groups. The first group of software originates from the database community, and includes tools like Cayuga [3], Borealis [1], TelegraphCQ [4], Esper[2], LINQ[3], VoltDB[4] and StreamBase SQL[5]. While their input languages vary, most can best be seen as special cases of database query languages, with added support for computation of aggregate functions (average, minimum, etc.) over sliding windows of events (e.g. all events of the last minute). The second group of software, while not labelled specifically as such, comes from the runtime verification community. Indeed, runtime monitors such as BeepBeep, JavaMOP [21], LARVA [5], Tracematches [2], J-Lo [22], PQL [20], PTQL [8], SpoX [6] and PoET [7] are designed with the purpose of detecting violations of some sequential pattern of events generated by a system in realtime.

It was observed in earlier work [16] that these two classes of systems have complementary strengths. The handling of aggregate functions over events provided by CEP tools is notably lacking in virtually all existing runtime monitors. Conversely, monitors generally allow the expression of intricate sequential relationships between events, using finite-state automata or temporal languages, that go far beyond CEP's traditional capabilities.

3 Design Decisions

BeepBeep aims at reconciling Complex Event Processing and Runtime Verification into a uniform, consistent and expressive query language for event streams. Its current architecture is the result of conscious design decisions, which have informed the development of the tool since the beginning. We list below a few of the most important decisions.

3.1 Focus on composition and modularity

The most important design decision relates to the way properties are being evaluated on an input trace. Traditionally, RV tools (including BeepBeep 1) start from a specification language, and implement a generic algorithm that can evaluate an expression from this language on a given input trace. Instead, BeepBeep 3 divides the computation of a result on an input trace into simple units, called *processors*.[6] Roughly speaking, processors transform an input trace into another output trace. The desired result is obtained by composing (i.e. piping) the outputs of a processor into the inputs of another, possibly forming a complex chain.

[2]http://espertech.com
[3]http://msdn.microsoft.com/en-us/library/bb397926.aspx
[4]http://voltdb.com
[5]http://streambase.com
[6]Similar to the mathematical concept of *transducer*.

The piping of processors can be represented graphically, as Figure 1 illustrates. In this case, an input trace (of numbers) is duplicated into two copies; the first is sent as the first input of a 2:1 processor labelled "+"; the second is first sent to the decimation processor, whose output is connected to the second input of "+". The end result is that output event i will contain the value $e_i + e_{ni}$.

According to this modularity principle, BeepBeep is organized as a simple platform providing the basic functionalities for creating and connecting processors and handling event buffers. The rest of the architecture consists of a number of relatively independent, self-contained *palettes*, each of which providing a toolbox of predefined processors focusing on a specific purpose. The end result is a very modular system, where only the necessary palettes can be included by a user, and where extensions unforeseen by the authors can easily be included in the system.

3.2 No single event type

Many existing RV and CEP tools assume a fixed structure for the events they process; in many cases, these events are *tuples*, i.e. maps from names (strings) to scalar values (generally strings or numbers). BeepBeep 1 already supported richer types of events, in the form of XML documents. BeepBeep 3 further relaxes these restrictions, and can handle as events any descendent of Java's `Object` class. To this end, BeepBeep divides the processing of an event trace in two parts.

- Trace Manipulation Functions (TMF) are processing units that manipulate events without accessing their content. A simple example of a TMF would be a processor that returns every n-th event of an input trace, and discards the others. In such a case, the actual content of each event is irrelevant.
- Event Manipulation Functions (EMF) are processing units that read or write data to/from an individual event. Contrarily to TMFs, EMFs are specific to the type of event that is being manipulated. BeepBeep includes EMFs to manipulate sets, XML documents, tuples, and allows users to create their own functions for any special event object that needs to be processed.

This separation between TMFs and EMFs presents the advantage of avoiding the "square peg in a round hole" problem common to many other RV systems. Instead of trying to fit every conceivable problem into the single available event type offered by the tool, BeepBeep allows the user to choose the objects most appropriate for the situation, and use the corresponding EMFs to access and manipulate their content. In the scenarios where BeepBeep has been used so far, event types encountered include sets, bitmap images, two-dimensional arrays, XML documents, Apache log entries, simple Booleans and numbers, Gnuplot input files, tuples, and IP network packets. In contrast, TMFs are generally agnostic to the type of events they manipulate. For example, the `CountDecimate` processor removes every n-th event from an input stream, irrespective of what these events may be. In other words, TMFs can be considered as polymorphic stream processors, as opposed to concrete type processors in EMFs.

3.3 No single input language

In the same way, BeepBeep does not impose a unique language to express the properties to be verified on a trace. Rather, it offers multiple means of creating the desired chain of processors. A first one is programmatically, by directly instantiating and connecting the Java objects corresponding to each processor.

```
Fork f = new Fork(2);
FunctionProcessor sum =
  new FunctionProcessor(Addition.instance);
CountDecimate decimate = new CountDecimate(n);
Connector.connect(fork, LEFT, sum, LEFT)
        .connect(fork, RIGHT, decimate, INPUT)
        .connect(decimate, OUTPUT, sum, RIGHT);
```

Figure 2: Java code creating the chain of processors corresponding to Figure 1.

For example, Figure 2 shows the Java code required to create the processor chain of Figure 1. The first three lines create the three processors that appear in the figure: a `Fork` that duplicates the input trace, a `FunctionProcessor` that computes the pairwise sum of its two inputs, and a `CountDecimate` processor that keeps every n-th event (for some unspecified variable `n`). The last instruction is a chained call to `Connector`'s static method `connect()`. It is responsible for creating the "piping" between these processors, again following the illustration in Figure 1. For example, the first ("`LEFT`most") output of the `fork` is connected to the first ("`LEFT`most") input of `sum`. Similarly, the only `OUTPUT` of `decimate` is connected to the second ("`RIGHT`most") input of `decimate`.

Another way of creating queries is by using BeepBeep's query language, called the Event Stream Query Language (eSQL). eSQL is the result of a careful process that went along with the development of BeepBeep's processors.

To this end, BeepBeep defines a top-level abstract class called `Interpreter`. When instantiated, an interpreter is instructed to load a set of grammar rules in Backus-Naur Form (BNF), generally from some internal text file. The interpreter must also be given a set of instances of another class called `Buildable`. Such an object must implement two methods. The first, `appliesTo()`, receives a node form a parse tree as an argument, and is expected to return `true` if the handling of this node should be done by this instance of `Buildable`. The second method is called `build()`. Its task is to push on the stack the object that is supposed to be built from the current parse node. In order to do so, method `build()` most often needs to pop elements from the stack that have been pushed by the previous action of other `Buildable` instances.

Equipped with a BNF grammar and a set of `Buildable` objects, the interpreter is now ready to parse an expression contained in a text string. It first creates the parse tree corresponding to that expression, according to the grammar that was loaded beforehand. The next step is then to build and connect processor instances based on its contents. To this end, the interpreter instantiates an empty stack of objects, and performs a postfix traversal of the parse tree, using the Visitor design pattern. The end result is a set of processors that have been instantiated and piped together through the traversal of a parse tree for a given expression. Once the processor chain has been created, it is no different from any other group of processors created directly with Java code.

As one can see, the eSQL grammar is entirely soft-coded. It is loaded at runtime from a file, and the code for instantiating a chain of processors from a parse tree can be completely overridden by the user. It allows eSQL to be extended, and ultimately re-designed from scratch, to become a Domain-Specific Language (DSL).

3.4 Queries, not properties

Runtime Verification has mostly focused on the evaluation of *properties*, i.e. assertions that must hold on a sequence of events. There do exist monitors whose specification language involves advanced data computing capabilities (numerical aggregation functions, mostly), but they still compute the answer to what is fundamentally a yes/no question. Yet, if one sees a monitor, in the broader sense of the term, as a diagnostics tool for discovering and understanding bugs, then it should provide the possibility to compute results beyond a single Boolean verdict.

In contrast, BeepBeep borrows from the field of Complex Event Processing, and generalizes the concept of property by allowing the user to compute arbitrary *queries* on a trace of events. The result of a query is not necessarily Boolean, and can be an output sequence of data elements of an arbitrary type. Examples of queries detailed in BeepBeep's latest technical report include non-Boolean traces such as the average of a sequence of numerical values over a sliding window, the detection of peaks in a numerical signal, and the generation of sets of tuples used to produce two-dimensional plots [11].

4 A Few Use Cases

The current implementation of BeepBeep has been used in a variety of scenarios; which we shortly describe two of them in the following.

4.1 Video Games

BeepBeep has been used to speed up the testing phase of a system, such as a video game under development, by automating the detection of bugs when the game is being played [23]. We take as an example the case of a game called *Pingus*, a clone of Psygnosis' *Lemmings* game series. The game is divided into levels populated with various kinds of obstacles, walls, and gaps. Between 10 and 100 autonomous, penguin-like characters (the Pingus) progressively enter the level from a trapdoor and start walking across the area. The player can give special abilities to certain Pingus, allowing them to modify the landscape to create a walkable path to the goal. For example, some Pingus can become Bashers and dig into the ground; others can become Builders and construct a staircase to reach over a gap.

Equipped with processors for parsing XML events, as well as Linear Temporal Logic (LTL) and first-order quantifiers, BeepBeep can be used to specify various properties about the expected behaviour of each Pingu in a level. For example, one can make sure that a walking Pingu that encounters a Blocker turns around and starts walking in the other direction. As a matter of fact, this particular property has been the subject of a benchmark at the 2016 Competition on Runtime Verification.

BeepBeep has been used to successfully monitor properties in a variety of video game types, ranging from classical arcade games to first-person shooters and 2D platformers.

4.2 Signal Processing

The next scenario touches on the concept of ambient intelligence, which is a multidisciplinary approach that consists of enhancing an environment (room, building, car, etc.) with technology (e.g. infrared sensors, pressure mats, etc.), in order to build a system that makes decisions based on real-time information and historical data to benefit the users within this environment. A main challenge of ambient intelligence is activity recognition, which consists in raw data from sensors, filter it, and then transform that into relevant information that can be associated with a patient's activities of daily living using Non-Intrusive Appliance Load Monitoring (NIALM). Typically, the parameters considered are the voltage, the electric current and the power (active and reactive). This produces a stream of power readings, where an event consists of a timestamp, and numerical readings of each of the aforementioned electrical components.

The NIALM approach attempts to associate a device with a load signature extracted from a single power meter installed at the main electrical panel. This signature is made of abrupt variations in one or more components of the electrical signal, whose amplitude can be used to determine which appliance is being turned on or off [12]. An example of query in this context could be: "Produce a *Toaster On* event whenever a spike of 1,000 W is observed on Phase 1 and the toaster is currently off." This can be done through the use of processors for signal processing (e.g. peak detection), mixed with finite-state machines and plain arithmetic functions.

5 The Road Ahead

BeepBeep's goal is to occupy a currently vacant niche among event stream processing engines: it lies somewhere in between low-level command line scripts for small trace crunching tasks, on one end, and heavy distributed event processing platforms on the other. The variety of proposed palettes, combined with a simple computational model, makes it suitable for the definition of clean and readable processing chains at an appropriate level of abstraction. While top-notch performance was not the first design goal, iin the use cases where it has been applied, BeepBeep has shown "fast enough" performance: this means that it can evaluate the queries given to it, at least as fast as the target system produces input events.

Rather than try to compete with commercial-grade platforms like Storm or Kinesis, BeepBeep could best be viewed as a toolbox for creating expressive computations *within* these environments. As a matter of fact, the development of (straightforward) adapters from BeepBeep to these environments is currently under way.

Several research problems around BeepBeep's concepts of processors and event streams are also left unexplored. For example, BeepBeep currently does not support *lazy evaluation*; if the output of an n-ary processor can be determined by looking at fewer than n inputs, all inputs must still be computed and consumed. Implementing lazy evaluation in a stream processing environment could provide some performance benefits, but is considered at the moment as a non-trivial task.

Most importantly, it is hoped that BeepBeep's palette architecture, combined with its simple extension mechanisms, will help third-party users contribute to the BeepBeep ecosystem by developing and distributing extensions suited to their own needs. More than a genuine runtime monitor, BeepBeep should be seen as a platform that can accommodate a variety of monitoring algorithms. Thanks to its generic architecture based on modularity and composition, existing monitors could be encapsulated as special types of processors, and tap into BeepBeep's broad range of palettes for upstream and downstream event processing tasks. In time, it is hoped that BeepBeep will be adopted as a modular framework under which multiple event processing techniques can be developed and coexist, and that their potential for composition will make the sum greater than its parts.

References

[1] Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.B.: The design of the borealis stream processing engine. In: CIDR. pp. 277–289 (2005)

[2] Bodden, E., Hendren, L.J., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative runtime verification with Tracematches. J. Log. Comput. 20(3), 707–723 (2010)

[3] Brenna, L., Gehrke, J., Hong, M., Johansen, D.: Distributed event stream processing with non-deterministic finite automata. In: Gokhale, A.S., Schmidt, D.C. (eds.) DEBS. ACM (2009)

[4] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.A.: TelegraphCQ: Continuous dataflow processing for an uncertain world. In: CIDR (2003)

[5] Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time Java programs (tool paper). In: Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM). pp. 33–37. IEEE Computer Society (November 2009)

[6] Erlingsson, Ú., Pistoia, M. (eds.): Proceedings of the 2008 Workshop on Programming Languages and Analysis for Security, PLAS 2008, Tucson, AZ, USA, June 8, 2008. ACM (2008)

[7] Erlingsson, Ú., Schneider, F.B.: IRM enforcement of Java stack inspection. In: IEEE Symposium on Security and Privacy. pp. 246–255 (2000)

[8] Goldsmith, S., O'Callahan, R., Aiken, A.: Relational queries over program traces. In: OOPSLA. pp. 385–402 (2005)

[9] Hallé, S.: A declarative language interpreter for CEP. In: Kolb, J., Weber, B., Hallé, S., Mayer, W., Ghose, A.K., Grossmann, G. (eds.) 19th IEEE International Enterprise Distributed Object Computing Workshop, EDOC Workshops 2015, Adelaide, Australia, September 21-25, 2015. pp. 156–159. IEEE Computer Society (2015), http://dx.doi.org/10.1109/EDOCW.2015.19

[10] Hallé, S.: When RV meets CEP. In: Falcone, Y., Sánchez, C. (eds.) Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10012, pp. 68–91. Springer (2016), http://dx.doi.org/10.1007/978-3-319-46982-9_6

[11] Hallé, S.: From complex event processing to simple event processing. CoRR abs/1702.08051 (2017), http://arxiv.org/abs/1702.08051

[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), http://dx.doi.org/10.1109/BigData.2016.7840873

[15] Hallé, S., Khoury, R.: Runtime monitoring of stream logic formulae. In: García-Alfaro, J., Kranakis, E., Bonfante, G. (eds.) Foundations and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9482, pp. 251–258. Springer (2015), http://dx.doi.org/10.1007/978-3-319-30303-1_15

[16] Hallé, S., Varvaressos, S.: A formalization of complex event stream processing. In: Reichert, M., Rinderle-Ma, S., Grossmann, G. (eds.) 18th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2014, Ulm, Germany, September 1-5, 2014. pp. 2–11. IEEE Computer Society (2014), http://dx.doi.org/10.1109/EDOC.2014.12

[17] Hallé, S., Villemaire, R.: Runtime verification for the web - A tutorial introduction to interface

contracts in web applications. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6418, pp. 106–121. Springer (2010), http://dx.doi.org/10.1007/978-3-642-16612-9_10

[18] Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Services Computing 5(2), 192–206 (2012), http://dx.doi.org/10.1109/TSC.2011.10

[19] Khoury, R., Hallé, S., Waldmann, O.: Execution trace analysis using LTL-FO$^+$. 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), http://dx.doi.org/10.1007/978-3-319-47169-3_26

[20] Martin, M.C., Livshits, V.B., Lam, M.S.: Finding application errors and security flaws using PQL: a program query language. In: OOPSLA. pp. 365–383 (2005)

[21] Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. International Journal on Software Techniques for Technology Transfer (2011), to appear

[22] Stolz, V., Bodden, E.: Temporal assertions using AspectJ. Electr. Notes Theor. Comput. Sci. 144(4), 109–124 (2006)

[23] 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