TeSSLa – An Ecosystem for Runtime Verification

Hannes Kallwies[1(✉)], Martin Leucker[1], Malte Schmitz[1], Albert Schulz[2], Daniel Thoma[1], and Alexander Weiss[2]

[1] Institute for Software Engineering and Programming Languages, University of Lübeck, Lübeck, Germany
{kallwies,leucker,schmitz,thoma}@isp.uni-luebeck.de
[2] Accemic Technologies GmbH, Kiefersfelden, Germany
{aschulz,aweiss}@accemic.com

Abstract. Runtime verification deals with checking correctness properties on the runs of a system under scrutiny. To achieve this, it addresses a variety of sub-problems related to monitoring of systems: These range from the appropriate design of a specification language over efficient monitor generation as hardware and software monitors to solutions for instrumenting the monitored system, preferably in a non-intrusive way. Further aspects play a role for the usability of a runtime verification toolchain, e.g. availability, sufficient documentation and the existence of a developer community. In this paper we present the TeSSLa ecosystem, a runtime verification framework built around the stream runtime verification language TeSSLa: It provides a rich toolchain of mostly freely available compilers for monitor generation on different hardware and software backends, as well as instrumentation mechanisms for various runtime verification requirements. Additionally, we highlight how the online resources and supporting tools of the community-driven project enable the productive usage of stream runtime verification.

1 Introduction

Runtime verification is the discipline of computer science that develops methods for verifying whether a system behavior adheres to its specification. To this extent the given specification in some specification language is typically translated into a *monitor* that analyzes the behavior in question. The analysis may be performed *online*, while the system is executing, or it may be analyzed *offline* when for example the trace is pre-recorded [1].

While the heart of a runtime verification framework consequently consists of the specification language itself and its synthesizers deriving monitors from given specifications, a practically viable tool suite has to support in many further aspects. One of the main challenges is how to get the observation of the system

T. Dang and V. Stolz (Eds.): RV 2022, LNCS 13498, pp. 314–324, 2022.
https://doi.org/10.1007/978-3-031-17196-3_20

under consideration. Most often, observing a system may slow it down or, more generally, may affect its timing. Even more, the monitor may affect the timing of the overall system. This may lead to both false positive and false negative verdicts which should of course be avoided.

Another aspect is the concrete application scenario. Runtime verification may be used as a form of debugging, for finding errors in a given system, or for showing (statistically) that the system is indeed correct. Depending on the application scenario, supporting tools either have to provide a quick turnaround time (i.e. the time to observing a new execution of the system once the specification has changed), or, have to be extremely efficient to support long-term observations. Finally, runtime verification may also be used for life-long supervision of the underlying system (and enforcing correctness of the system) such that the whole runtime verification machinery becomes a part of the system which, again, requires different properties to be fulfilled. An overview of the general stream runtime verification architecture with the required components and involved configuration documents, which are subject to these considerations is shown in Fig. 1.

However for practical applications, it is not only important to get the system right but likewise to get the specifications right. As such, supporting tools for writing meaningful specifications are helpful. Last but not least, a vivid community and open-source tools are a further plus when using runtime verification in industrial settings.

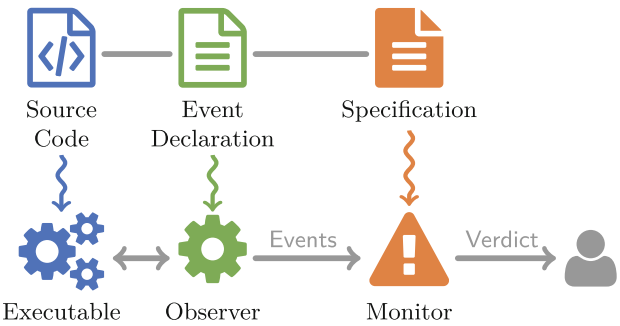

Fig. 1. General architecture of stream runtime verification.

Altogether we see that from a theoretical point of view runtime verification is often simplified to synthesizing monitors for your specification formalism, while a practically viable runtime verification framework has to meet a variety of different requirements and needs a variety of supporting tools.

In the following we focus on stream runtime verification (SRV) which has been pioneered by the specification language LOLA [2]. Later on RTLola [3], Striver [4] and TeSSLa [5,6] emerged. In this paper we will present TeSSLa's different compiler backends and supporting tools to meet the various requirements of the runtime verification process discussed above. While the TeSSLa language

itself [5,6] and some of its synthesizers [7,8] have been described before, this paper describes mainly the TeSSLa tool suite as a whole, which aims supporting software engineers and testers to achieve efficient and powerful verification.

This paper is organized as follows: In Sect. 2 we briefly recall the TeSSLa language by providing a specification that can be used for monitor synthesis. In Sect. 3 we give a broad overview of the backends such monitors may be compiled to. Section 4 presents the different instrumentation approaches that are compatible with the TeSSLa framework. In Sect. 5 we finally give an overview about additional tools and aspects connected to the TeSSLa ecosystem. We conclude the paper in Sect. 6.

2 The TeSSLa Specification Language

This section presents the TeSSLa language on the basis of an example to give a rough impression of the language features supported there: The specification in Listing 1.1, used as running example throughout the paper, checks that the time that passes between the activation of brakes of an automotive system and the reading of the brake sensors (which are used to supervise the braking process) is less than or equal to 4 ms.

```
# Inputs
@InstFunctionCall("read_brake_sensor")
in read_brake_sensor: Events[Unit]
@InstFunctionCall("activate_brakes")
in activate_brakes: Events[Unit]

# Trace Processing
def latency = measureLatency(read_brake_sensor, activate_brakes)
def error = latency > 4ms
def high = filter(latency, error) - 4ms
def is_critical = count(high) > 10
def critical = filter(high, is_critical)

# Output
@VisDots out high
@VisEvents out critical

# Macro
def measureLatency[A, B](a: Events[A], b: Events[B]) =
  time(b) - last(time(a), b)
```

Listing 1.1. TeSSLa specification for the *Brake Sensor* example.

The specification does so by defining two input streams `read_brake_sensor` and `activate_brakes`. The type of the events carried by these streams is Unit, i.e. they have no value, as they only represent calls to functions. The input streams are preceded by `@InstFunctionCall` annotations. During the following monitoring process, these annotations are extracted from the stream specification and passed to connected tools of the tool chain. In this specific case these annotations are meant for the instrumenter who is instructed to raise an event on the input streams, always when in the supervised system a call of the functions `read_brake_sensor` and `activate_brakes` happens. In the following lines five further streams are defined. The first one `latency` is defined as a call of the macro

`measureLatency`. This macro receives two streams `a` and `b` of generic types `A`, `B` and produces a new stream of events. It is defined at the end of the specification using two operators: `time(x)` provides access to the timestamps of the events on stream `x` and `last(y, z)` provides the last event on y for every event on z. The expression `time(b) - last(time(a), b)` calculates the difference between the timestamp of the current event on stream `b` and the timestamp of the last event on stream `a`. As a consequence the stream `latency` in our example always carries the latency between a call of `activate_brakes` and the subsequent call of `read_brake_sensor`. The other streams are defined based on this latency and via macros from the TeSSLa standard library. The stream `error` is *true* if the measured latency is higher than 4ms. If `error` is *true* then `high` contains the value by which amount the 4ms are surpassed. Stream `is_critical` counts the number of events on stream `high`, i.e. the number of breaches of the property and gets *true* if this number exceeds 10. `critical` finally filters the events of `high` if `critical` is true. In the third part the specification eventually defines which streams shall be printed out by the monitor (`high` and `critical`). Again these streams contain annotations which are passed to subsequent tools. In this case `@VisDots` and `@VisEvents` which indicate the graphical representation of the streams in a monitor GUI.

Note that the TeSSLa language, from a theoretical point of view, as presented in [6], only consists of six core operators. In practice, however, it provides several additional features, like annotations, macro definitions and access to macros from a standard library, which do not make the language more expressive, but are necessary for a comfortable usage of the tool chain and the language itself.

3 TeSSLa Compilers and Backends

The TeSSLa tool suite addresses different compilation targets for TeSSLa specifications. It comes with an interpreter that evaluates a TeSSLa specification on the JVM without compilation, compilers that synthesize the specification into software monitors that can be executed on different target platform, and a compiler for specialized event processing hardware.

The interpreter is written in Scala and available as a runnable Jar archive. It follows a straightforward evaluation strategy and serves as a reference implementation for TeSSLa, but is significantly slower than other backends (see measurements in Fig. 3). Still, it is a ready-to-use tool for simple experiments, e.g. when exploring the TeSSLa language. The interpreter provides results without compilation overhead, while the other software compilers translate TeSSLa to imperative languages first, which are then further compiled to binaries. The interpreter's direct evaluation supports the interactive process of writing new specifications and checking them on sample inputs. It also provides an API that can be used to integrate it with custom tools and trace sources.

The software compilers generate Scala or Rust code. The Scala code is compiled into a Jar which can be executed platform-independent on any JVM. Complex data structures like maps, sets and lists are implemented using the

immutable data structures provided by the Scala standard library. Additional Scala and Java data structures and functions can be used via native externs: They allow the declaration and utilization of TeSSLa functions that are implemented natively in the target language of the compilation. The Rust code is compiled into a native binary for all targets supported by the LLVM project. Complex data structures are implemented using immutable data structures for Rust provided by the library rust-im[1] and additional data structures and native externs are supported, too. Both software compilers generate a monitoring library and an exemplary command line application.

The TeSSLa framework also supports a specialized event processing hardware, Accemic's embedded processing units (EPUs) [8–13]. EPUs are implemented on an FPGAand allow data flow processing while maintaining short reconfiguration cycles: The EPUs are programmed by writing special commands into their memory. They can be reconfigured entirely without the need for a new FPGA synthesis. The TeSSLa EPU compiler generates such an EPU configuration which can be directly uploaded to EPU hardware. The maximal processing speed of the EPUs is 100 MEVent/s (million events per second).

The TeSSLa language is designed to be modular such that the requirements of different target platforms can be considered. For example, the EPUs do not support complex data structures to the same extent as the software compilers. The interpreter, the software compilers and the EPU compiler rely on the same compiler frontend, which compiles a TeSSLa specification into so-called TeSSLa Core. TeSSLa Core is a special form of a TeSSLa specification, representing the data flow graph of the TeSSLa specification. In TeSSLa Core every stream and every function has type annotations, and all macros are expanded. The compiler frontend can either print TeSSLa Core using the syntax for TeSSLa specifications, or provide the object graph as a data structure to compiler backends so that they do not need to parse it again.

The compiler frontend consists of an ANTLR-based parser, a type checker and a constant folder, which operates on macros and functions on statically known values and simplifies the translation for the further backends. The frontend is written in Scala and available as a library packaged as a Jar archive that the backends can use, for example as a Maven dependency. This makes it possible to extend the tool suite with further specialized synthesizer backends.

4 Observation and Instrumentation

The TeSSLa tool suite provides utilities for the entire runtime verification workflow: The previous section introduced several monitoring syntheses; this section discusses approaches to observe events from the system under test (SUT).

As already pointed out in the introduction, the requirements for the mechanism to do this are diverse and depend on the specific application scenario. While for some settings a powerful and highly customizable software instrumentation is

[1] https://docs.rs/im/latest/im/.

the desired mechanism, other scenarios may require a fully non-intrusive observation generation strategy, which has no interference with the SUT. Depending on the monitoring target (hardware or software) the TeSSLa tool chain is compatible with/offers different instrumentation utilities.

The software monitors can be used for online and offline monitoring. They can process trace data from text-based or binary files recorded earlier. In combination with instrumentation tools like AspectJ [14,15] they can be used for online monitoring, too: The instrumented executable sends a stream of events to the compiled monitor running as a separate process in order to reduce the influence of the monitoring on the SUT. The upper part of Fig. 2 shows this approach.

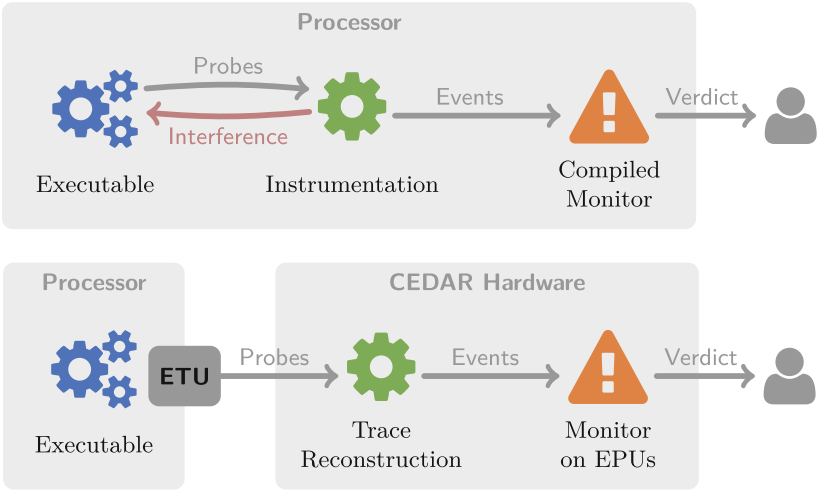

Fig. 2. Architecture of runtime verification with instrumented binary and compiled monitor (top) in comparison with dedicated CEDAR hardware for non-intrusive monitoring with the embedded tracing unit (ETU) of the processor (bottom).

The TeSSLa tool suite also comes with its own instrumentation tool for C code using the clang compiler tool chain. Instrumenting source code instead of binaries has the advantage that the instrumented source code is still human-readable and can manually be customized after the instrumentation by the user according to his needs and then be compiled with the existing compilation tool chain.

The C-Code instrumenter is available as a native binary that is integrated into the TeSSLa Jar package. It uses the information about the specification's input streams and annotations (e.g. `@InstFunctionCall` in Listing 1.1) to add dedicated calls to a logging library into the source code of the SUT. The logging library is also part of the TeSSLa tool suite. It uses multi-producer multi-consumer channels for message passing to allow multiple threads of the SUT to send messages to the monitor without any locking.

In contrast to the intrusive software monitoring approach, the TeSSLa tool suite also supports non-intrusive monitoring using Accemic's CEDAR hardware [8,11–13,16,17]. The lower part of Fig. 2 shows how non-intrusive monitoring utilises the processor's embedded tracing unit (ETU). The unmodified executable runs on the processor and the ETU provides a debugging trace. This trace contains information about the program counter, i.e. which instructions are currently executed by the processor. The ETU's trace is encoded: From time to time it contains absolute program counter addresses, but most of the time it only indicates if a conditional jump was taken or not. The trace reconstruction of the CEDAR hardware decodes the current program counter address online from the ETU's trace. Again, annotations in the TeSSLa specification are used to declare points of interest. If the program reaches such a point, the trace reconstruction adds an event into the event stream processed by the EPUs which were configured with the TeSSLa specification. A video demonstration of the usage of the TeSSLa tool suite non-intrusive monitoring using Accemic's CEDAR hardware with the specification from Listing 1.1 is available online.[2]

Figure 3 shows some exemplary throughputs of the specification *Brake Sensor* from Listing 1.1 and another specification *Scheduling* using complex data structures that are not supported on the EPUs. Both specifications are available in the playground in the menu item *RV Examples*.[3] One can clearly see that the interpreter is orders of magnitude slower than the compiled Scala program. The compiled Rust program and the EPUs are again an order of magnitude faster than the compiled Scala program.

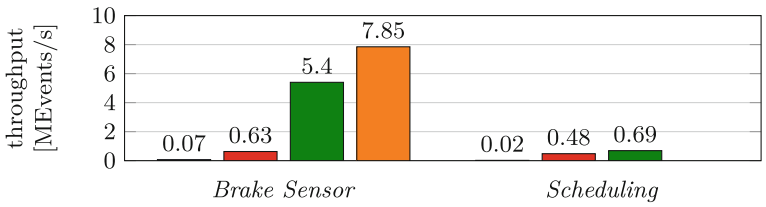

Fig. 3. Exemplary throughput of the interpreter, the compiled Scala monitor, the compiled Rust monitor, and the EPUs. (Color figure online)

5 The TeSSLa Ecosystem

The TeSSLa tool suite provides the necessary components for online and offline runtime verification: Instrumentation, logging, and monitor synthesis. However, the TeSSLa ecosystem goes beyond these software tools and covers further aspects that supports the practical application of runtime verification:

[2] www.youtube.com/watch?v=3AYVWK-X9nw.
[3] https://play.tessla.io/.

Playground. The TeSSLa website[4] contains an interactive playground (see Footnote 3) intended for a first exploration of the TeSSLa language and the runtime verification tools. TeSSLa specifications can be interpreted and C code can be instrumented and executed in a sand-boxed environment. Further, the stream visualizer provides a graphical intuition for TeSSLa streams. It helps to recognize event patterns and assists users with the interactive process of writing and testing TeSSLa specifications. The playground is shown in Fig. 4: Note how the annotations `@VisDots` and `@VisEvents` on the output stream declarations in Listing 1.1 determine the representation of the streams in the visualizer.

Documentation. Further, the TeSSLa website contains material on the formal semantics of the language, introductions and tutorials on writing TeSSLa specifications and using the instrumentation for runtime verification. The language specification precisely describes the syntax and semantics of the language. We developed TeSSLadoc to support documentation of TeSSLa specifications. The tool is inspired by Javadoc and used e.g. for the documentation of the standard library. The documentation includes examples which are graphically represented using the stream visualizer.

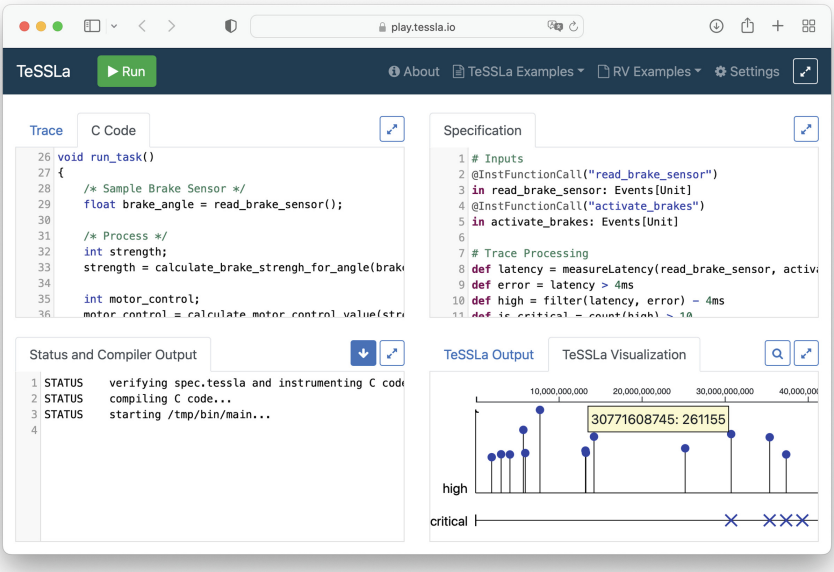

Fig. 4. The TeSSLa playground is a web-based IDE that can interpret TeSSLa specifications on instrumented C code or manually entered input traces. Output traces can be graphically visualized in the interactive stream visualizer.

[4] www.tessla.io.

Libraries. TeSSLa's macro system supports modular extensions for special application domains. There are currently libraries for AUTOSAR Timex [18,19], past-time LTL and timed dyadic deontic logic [20]. These documented user libraries are available for download on the TeSSLa website[5] and are contributed and maintained by the community.

Scientific Publications. TeSSLa itself is presented, analyzed and discussed in several publications [5,6,8,12,13,21–23] and used to implement and analyze advanced concepts for stream runtime verification like for example monitoring streams with partial information using ideas of abstraction [24] and new approaches to the aggregate update problem for multi-clocked data flow languages [7]. The application of TeSSLa for race detection is described in [25].

Community. The TeSSLa language, the language specification, the compiler frontend and several backends are available under a free license. TeSSLa is maintained and developed further by the TeSSLa community. It is used in several projects and the open source licensing allows all TeSSLa users to share their contributions with the growing community.

6 Conclusion

This paper provided an overview of the TeSSLa tool suite for runtime verification. We discussed typical challenges that come with the practical application of runtime verification and presented their solutions within the TeSSLa framework. We demonstrated how the main components work and how they can be used. Finally we sketched further accompanying aspects of the TeSSLa ecosystem and argued how they support the verification process further.

References

1. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebraic Program. **78**(5), 293–303 (2009)
2. D'Angelo, B., et al.: LOLA: runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME), pp. 166–174. IEEE Computer Society (2005)
3. Faymonville, P., et al.: StreamLAB: stream-based monitoring of cyber-physical systems. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 421–431. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_24
4. Gorostiaga, F., Sánchez, C.: Striver: stream runtime verification for real-time event-streams. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 282–298. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_16
5. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: Tessla: runtime verification of non-synchronized real-time streams. In: SAC, ACM, pp. 1925–1933 (2018)

[5] www.tessla.io/usrLibs/overview/.

6. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: temporal stream-based specification language. In: Massoni, T., Mousavi, M.R. (eds.) SBMF 2018. LNCS, vol. 11254, pp. 144–162. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03044-5_10
7. Kallwies, H., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: Aggregate update problem for multi-clocked dataflow languages. In: Symposium on Code Generation and Optimization (CGO), pp. 79–91. IEEE (2022)
8. Decker, N., et al.: Rapidly adjustable non-intrusive online monitoring for multi-core systems. In: Cavalheiro, S., Fiadeiro, J. (eds.) SBMF 2017. LNCS, vol. 10623, pp. 179–196. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70848-5_12
9. Weiss, A.: Event Processing US 2021081145 A1, March 18 (2021)
10. Weiss, A.: Event Processing EP 3792767 A1, March 17 (2021)
11. Weiss, A., et al.: Understanding and fixing complex faults in embedded cyberphysical systems. Computer **54**(1), 49–60 (2021)
12. Decker, N., et al.: Online analysis of debug trace data for embedded systems. In: DATE, pp. 851–856. IEEE (2018)
13. Convent, L., Hungerecker, S., Scheffel, T., Schmitz, M., Thoma, D., Weiss, A.: Hardware-based runtime verification with embedded tracing units and stream processing. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 43–63. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_5
14. Hilsdale, E., Hugunin, J., Kersten, M., Kiczales, G., Lopes, C.V., Palm, J.: AspectJ: the language and support tools. In: OOPSLA Addendum, ACM, p.163 (2000)
15. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45337-7_18
16. Weiss, A., Lange, A.: Trace-Data Processing and Profiling Device EP 2873983 A1, May 20 (2015)
17. Weiss, A., Lange, A.: Trace-Data Processing and Profiling Device US 9286186 B2, March 15 (2016)
18. Friese, M.J., Kallwies, H., Leucker, M., Sachenbacher, M., Streichhahn, H., Thoma, D.: Runtime verification of AUTOSAR timing extensions. In: International Conference on Real-Time Networks and Systems (RTNS), ACM, pp. 173–183 (2022)
19. Partnership, A.D.: Specification of timing extensions, version 1.0.0, release 4.0.1
20. Kharraz, K.Y., Leucker, M., Schneider, G.: Timed dyadic deontic logic. In: JURIX, Volume 346 of Frontiers in Artificial Intelligence and Applications, pp. 197–204. IOS Press (2021)
21. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: Runtime verification of real-time event streams under non-synchronized arrival. Software Qual. J. **28**(2), 745–787 (2020). https://doi.org/10.1007/s11219-019-09493-y
22. Kallwies, H., Leucker, M., Prilop, M., Schmitz, M.: Optimizing trans-compilers in runtime verification makes sense - sometimes. In: Ameur, Y. et al. (eds.) Theoretical Aspects of Software Engineering. TASE 2022. LNCS, vol. 13299, pp. 197–204. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10363-6_14
23. Kauffman, S.: `nfer` – a tool for event stream abstraction. In: Calinescu, R., Păsăreanu, C.S. (eds.) SEFM 2021. LNCS, vol. 13085, pp. 103–109. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92124-8_6
24. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Thoma, D.: Runtime verification for timed event streams with partial information. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 273–291. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_16

25. Ahishakiye, F., Jarabo, J.L.R., Pun, V., Stolz, V.: Hardware-assisted online data race detection. In: Bartocci, E., Falcone, Y., Leucker, M. (eds.) Formal Methods in Outer Space. LNCS, vol. 13065, pp. 108–126. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-87348-6_6