Contents

1 Related Work

Just a collection of thoughts for now, needs to be polished a lot

As Runtime monitoring and verification is a widely researched field, multiple approaches to attain it's goals were developed.

As stated in [3] most approaches are geared towards software written in Java, while many critical systems are written in C and there are countless other systems that could benefit from monitoring and verification written in all kinds of programming languages. With TeSSLa as a specification Language over Streams, which has no assumptions on the environment of the system that produces the Streams, as the base for our monitoring approach, we recognized the possibility to abstract the monitoring plattform from the monitored program. This means that the developed runtime for TeSSLa specifications is not restricted to monitor programs written in a specific language but can monitor anything that can produce Streams.

To show that the runtime is valuable in the context of existing approaches, we will show ways to generate traces from systems that were used to evaluate other monitoring techniques. Afterwards we will compare the expressiveness of TeSSLa and the runtime with other approaches, based on the generated traces, to show what kinds of specifications can be monitored with TeSSLa and where the Language or the runtime can be extended.

The following Chapter will highlight the systems against which TeSSLa and the runtime is evaluated, furthermore it will also give insights into other work that TeSSLa and this thesis is based on.

1.1 TeSSLa

The implemented Runtime and the theoretic work of this thesis is built upon the TeSSLa project. For this project the syntax and a formal semantic for a specification language was defined.

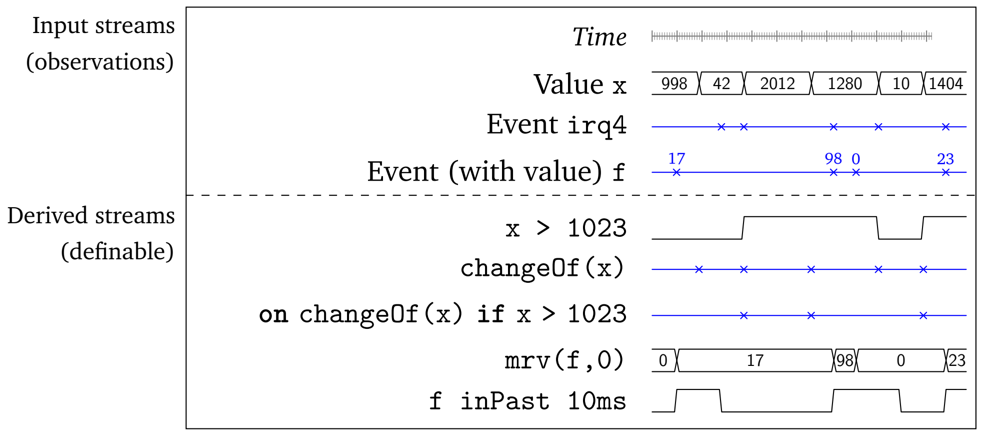

Fig. 1.1: Visualization of TeSSLas Stream model, taken from [2]

1.2 LOLA

The concepts of LOLA [1] are very similar to the ones of TeSSLa. Both approaches built upon streams of events. The biggest difference in the modelation is, that while streams in Lola are based on a discrete model of time TeSSLa uses a continuous timing model.

The specification language of Lola is very small (expressions are built upon three operators) but the expressiveness surpasses Temporal Logics and many other Formalisms [1]. Expressions in Lola are built by manipulating existing streams to form new ones. Therefore streams depend on other streams, so they can be arranged in weighted dependency graph, where the weight describes the amount of steps a generated Stream is delayed compared to the parent.

Based on this graph a notion of efficiently monitorable properties is given and an algorithm to monitor them is presented.

TeSSLa takes concepts of LOLA and applies them to a continuous model of time and introduces a language and a rich set of functions that can be applied to streams. The dependency graph is a core concept of TeSSLa and is used to check if specifications are valid (e.g. cycle free) and is also the core concept to evaluate specifications over traces in this thesis.

1.3 Distributed Verification Techniques

While most implementations of RV systems don't consider or use modern ways of parallelism and distribution and focus on programs running locally, in [4] a way to monitor distributed programs is given. To do this distributed monitors, which have to communicate with one another, are implemented.

As stated earlier, the TeSSLa runtime doesn't care about the environment of the monitored program, so it doesn't distinguish between traces from distributed and non distributed programs. But the runtime itself is highly concurrent and can be distributed easily to many processors or event different computers. Therefore many of the definitions for distributed monitors can be used to reason about the behaviour of the Runtime.

1.4 Copilot

The realtime runtime monitor system Copilot was introduced in [6]. Copilot is designed to overcome the shortcomings of existing RV tools in regards to hard-realtime software written in C.

To do so they first define characteristics a monitoring approach has to fullfill to be considered valuable for this domain. The four principles are:

Functionality Monitors cannot change the functionality of the observed program unless a failure is observed.

Schedulability Monitors cannot alter the schedule of the observed program.

Certifiability Monitors must minimize the difficulty in re-validating the observed program; in particular, we make it our goal to avoid modifying the observed programs source code.

SWaP overhead Monitors must minimize the additional overhead required including size, weight, and power (SWaP).

The monitors follow a sampling based approach, where at specified steps the values of global variables are observed and the monitors are evaluated on that values. While sampling based approaches are widely disregarded in RV, because they can lead to both false positives and false negatives, they argue:

> In a hard real-time context, sampling is a suitable strategy. Under the assumption that the monitor and the observed program share a global clock and a static periodic schedule, while false positives are possible, false negatives are not. [6]

A special detail of Copilot is that monitors aren't inlined into the program but can be scheduled as independet processes. The implementation of the TeSSLa runtime in this thesis follows a similar approach: It is a totally independent program, and therefore also has some of the gains in regard to the specified four characteristics. Because the runtime works with all kinds of traces, it is insignificant how they are produced: It can work with traces based on sampling, working in a similar fashion as Copilot, or by actually instrumenting code to generate traces, which alters the semantics of the program.

1.5 RMoR

RMoR is another approach on monitoring C programs. It does so by transforming C code into an *armored* version, which includes monitors to check conformance to a specification.

Specifications are given as a textual representation of state machines. The specifications are then interweaved into the programm using CIL [5]. Specifications work on the level of function calls and state properties like *write may never be called before open was called*. Because Software Developers are often working at the same abstraction level (in contrast to e.g. assembler or machine instructions), they can define specifications without having to learn new concepts. For the TeSSLa runtime support for traces at the same abstraction level (function calls, variable reads and writes) is present and used in most of the tests in Section **??**.

Because RMoR specifications are interweaved into the program, their observations can not only be reported but also used to recover the program or even to prevent errors by calling specified functions when some condition is encountered. The TeSSLa runtime doesn't support this out of the box, as it's primary purpose is testing and offline monitoring, but in Section **??** we will look at ways to support this.

1.6 MaC

Bibliography

[1]Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, et al. „LOLA: Runtime monitoring of synchronous systems“. In: *Proceedings of the International Workshop on Temporal Representation and Reasoning* (2005), pp. 166–175 (cit. on p. 2).

[2]Normann Decker, Daniel Thoma, and Jannis Harder. „TESSLA A Temporal Stream-based Specification Language“. 2016 (cit. on p. 2).

[3]Klaus Havelund. „Runtime Verification of C Programs“. In: (2008) (cit. on p. 1).

[4]Menna Mostafa and Borzoo Bonakdarpour. „Decentralized Runtime Verification of LTL Specifications in Distributed Systems“. In: *2015 IEEE International Parallel and Distributed Processing Symposium* (2015), pp. 494–503 (cit. on p. 2).

[5]George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. „CIL: Intermediate language and tools for analysis and transformation of C programs“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2304 (2002), pp. 213–228 (cit. on p. 4).

[6]Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. „Copilot: A hard real-time runtime monitor“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6418 LNCS.Rv (2010), pp. 345–359 (cit. on p. 3).

List of Figures

List of Tables

List of Theorems