Hardware-Based Runtime Verification with Embedded Tracing Units and Stream Processing

Lukas Convent[1(✉)], Sebastian Hungerecker[1(✉)], Torben Scheffel[1(✉)], Malte Schmitz[1(✉)], Daniel Thoma[1(✉)], and Alexander Weiss[2(✉)]

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

Abstract. In this tutorial, we present a comprehensive approach to non-intrusive monitoring of multi-core processors. Modern multi-core processors come with trace-ports that provide a highly compressed trace of the instructions executed by the processor. We describe how these compressed traces can be used to reconstruct the actual control flow trace executed by the program running on the processor and to carry out analyses on the control flow trace in real time using FPGAs. We further give an introduction to the temporal stream-based specification language TeSSLa and show how it can be used to specify typical constraints of a cyber-physical system from the railway domain. Finally, we describe how light-weight, hardware-supported instrumentation can be used to enrich the control-flow trace with data values from the application.

1 Introduction

Software for embedded, hybrid and cyber-physical systems often operates under tight time and resource constraints. Therefore, testing, debugging and monitoring is particularly challenging in this setting. Strong limitations on timing and resource consumption prohibit usual approaches for the acquisition and analysis of execution information. First, comprehensive logging output during development built into the software (e.g. via instrumentation) decreases the performance significantly. Second, breakpoint-based debugging features of the processor are slow due to the potentially high number of interruptions. Both methods are highly intrusive as they modify the software temporarily for the analysis or interrupt the execution. This is especially problematic for concurrent programs running on multi-core processors or real-time applications. Errors due to race conditions or inappropriate timing may be introduced or hidden.

This work is supported in part by the European COST Action ARVI, the BMBF project ARAMiS II with funding ID 01 IS 16025, and the European Horizon 2020 project COEMS under number 732016.

C. Colombo and M. Leucker (Eds.): RV 2018, LNCS 11237, pp. 43–63, 2018.
https://doi.org/10.1007/978-3-030-03769-7_5

To allow for a non-intrusive observation of the program trace, many modern microprocessors feature an *embedded trace unit (ETU)* [3,11,13]. An ETU delivers runtime information to a debug port of the processor in a highly compressed format. State-of-the-art debugging solutions, such as ARM DSTREAM [4], allow the user to record this information for offline reconstruction and analysis.

The essential disadvantage of this technology is, however, that traces can be recorded for at most a few seconds because high-performance memory with very fast write access is required to store the delivered information. For example, the ARM DSTREAM solution offers a trace buffer of 4 GB for a recording speed of 10 Gbit/s or more which means that the buffer can only hold data of less than four seconds. While the majority of errors can be found immediately within a short program trace, some of them may only be observable on long-running executions or under specific, rarely occurring (logical or physical) conditions. It is therefore desirable for the developer and maintainer to be able to monitor the program execution for an arbitrary amount of time during development and testing and even in the field after deployment.

This paper is based on [9] which presented an earlier version of the monitoring techniques discussed in this paper. This tutorial gives a more extensive introduction into our monitoring approach and comprises the recent improvements made to the monitoring hardware, tools and specification language.

Related Work. For a general introduction into the field of runtime verification especially in comparison with static verification techniques such as model checking see [16,17].

Non-intrusive observation of program executions is a long-standing issue [21] and several approaches have been suggested. We rely on dedicated tracing interfaces as they are provided by many modern processors. Such interfaces have already been suggested in [26]. Another line of research focuses on the modification of processors [18] or complete systems on chips [25]. These approaches allow access to a wider range of information but require access to the processor or system hardware design and modifications have to be possible. In [6] a processor is monitored by synchronizing a second, emulated processor via a dedicated synchronisation interface. In [20] it is described that even side-channels can be used to monitor certain events on a processor.

There are also several approaches to execute monitors on FPGAs for various applications: synthesis for STL for observation of embedded systems is described in [14,15,24] and synthesis for past time LTL for observation of hardware buses is described in [22]. While these approaches directly synthesize FPGA designs from monitor specifications, we use processing units that are synthesized once and can be reconfigured quickly. Approaches also allowing for reconfiguration are described in [19] for past-time LTL and in [23] for past-time MTL.

The basic idea of stream-based runtime verification and stream transformations specified via recursive equations has been introduced with the language LOLA [8,10]. LOLA however is synchronous in the following sense: Events arrive in discrete steps and for every step, all input streams provide an event and all

output streams produce an event, which means that it is not suitable for handling events with arbitrary real-time timestamps arriving at variable frequencies.

Outline. The rest of this paper is organized as follows: Sect. 2 gives an overview of the general workflow and mechanism of hardware-based runtime monitoring as discussed in this paper. Section 3 describes how the program flow can be reconstructed online and how events are generated and fed into the monitoring engine. Section 4 describes how to specify monitors in the stream processing language TeSSLa. Section 5 introduces a simple cyber-physical system that is used as an example throughout the rest of the paper. Section 6 demonstrates how to check timing constraints and Sect. 7 shows how to check event ordering constraints using hardware-based runtime monitoring. Section 8 describes how the tracing of data values works and demonstrates how to check data-values. Finally, Sect. 9 describes the practical hardware setup in order to do hardware-based runtime monitoring.

2 Interactive Hardware Monitoring Workflow

To overcome the limitations of current technology we developed a novel runtime verification methodology for evaluating long-term program executions which is suitable for development, debugging, testing, and in-field monitoring. Based on the runtime information provided by the ETU, we perform a real-time reconstruction of the program trace. The latter is evaluated with respect to a specification formulated by the user in the stream-based specification language TeSSLa [7]. To deliver sufficient performance for online analysis, both the reconstruction and monitoring system are implemented using FPGA hardware.

FPGAs have become a very popular technology to implement digital systems. They contain thousands of programmable logic elements which can be configured to realize different boolean functions. These functions can be connected to each other in an arbitrary way by means of configurable routing elements. Additional features like flip-flops, digital signal processing blocks and blocks of RAM add more flexibility and performance to the implemented circuit. Designing digital circuits with FPGAs typically starts from hardware description languages like VHDL or System Verilog. Synthesis software is responsible for mapping such designs to the elements available in an FPGA and then these elements must be positioned and routed on the FPGA fabric. Even for moderately large designs, this process can take hours. In case the design should run at high clock speed, this time is dramatically increased. Additionally, a designer must be familiar with the FPGA elements and must have thorough experience in FPGA design to be able to create fast designs.

Our monitoring system therefore does not rely on synthesizing a specific FPGA-design for each property specification that has to be evaluated. Instead, it builds on a set of event processing units implemented on the FPGA. These units can be configured quickly via memory to evaluate arbitrary specifications.

We provide a tool chain for mapping TeSSLa specifications to these units automatically within seconds. This allows the user to focus on writing the correctness properties instead of working with the complex FPGA synthesis tool chain. Formulating hypotheses, adapting property specification and checking them on the target system can be iterated quickly without time-intensive synthesis.

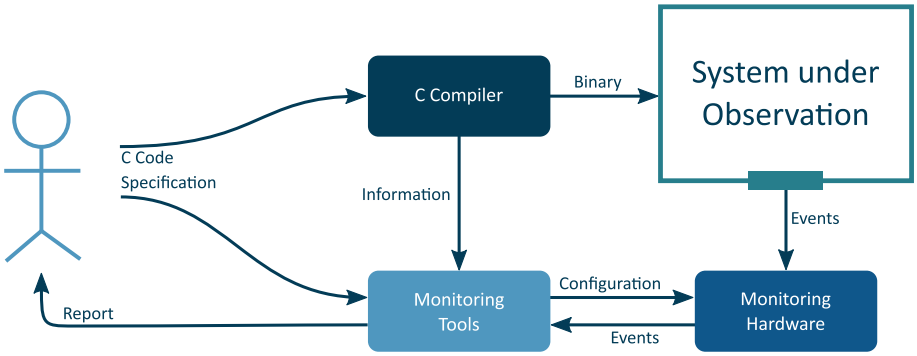


Fig. 1. General overview of the workflow cycle.

Figure 1 provides an overview of the proposed workflow based on our approach to rapidly adjustable embedded trace online monitoring. The user, e.g. the developer, tester, or maintainer, specifies the correct behaviour of the program under test based on program events such as function calls or variable accesses. The program is compiled and the binary is uploaded to the processor as usual. The property compiler automatically generates a corresponding configuration for the monitoring and trace reconstruction units that is then uploaded to the platform. When running the program on the processor, the monitoring platform reports the computed output stream to the user who can then use the information to adjust the program or the property.

Technically, all such events are represented in the reconstructed trace as so-called *watchpoints*, so the trace reconstruction provides a watchpoints stream to the monitoring platform. The reconstruction can already filter the full trace for those watchpoints (i.e., events) that are relevant for the property.

In this tutorial, we demonstrate how our approach can be applied to an example system from the railway domain. We first give an introduction to the specification language TeSSLa and show how it can be used to specify typical properties in such a setting. We then explain how our hardware implementation can be used to monitor these properties non-intrusively.

3 Monitoring Program Flow Trace

Figure 2 shows an overview of the program flow trace monitoring setup: The cores of the multi-core processor are communicating with periphery, such as the

memory, through the system bus. Every core is observed by its own tracer. The core is not affected at all by this kind of tracing. The trace data is sent through the trace buffer and concentrator to the trace port without affecting the core. This tracing is separated from the system bus and does not interfere with it. The trace port of the processor is connected to the monitoring hardware, i.e. the FPGA on which the program flow reconstruction, the interpretation and the actual monitoring are located.

The user of this system provides the C code of the system under observation and the specification for the monitoring. The specification contains information about the events of interest and the actual monitor expressed in terms of these events. The C compiler compiles the C source code, so that the resulting binary can be executed on the processor. The C compiler provides debug information which can be used to determine the instruction pointer addresses of the events of interest in the program, the so called *tracepoints*. The trace reconstruction is configured with the observation configuration which contains the tracepoints. The TeSSLa compiler compiles the monitor specification to the monitor configuration which is used to configure the actual trace monitoring.

The final monitoring output coming from the dedicated monitoring hardware undergoes some post-processing on a regular PC using metadata provided by the frontend such as the names and types of the output events. The final monitoring report is a sequence of events that can be either stored or processed further.

We explain the concept of the trace reconstruction with the ARM CoreSight [3] trace technology as a widely available example of an ETU, which is included in every current ARM processor (Cortex M, R and A). In particular, we use the Program Flow Trace (PFT) [2] to acquire trace data of the operations executed by the ARM processors.

As stated in the PFT manual [2] the "PFT identifies certain instructions in the program, and certain events as waypoints. A waypoint is a point where instruction execution by the processor might involve a change in the program flow." With PFT we only observe as waypoints conditional and unconditional direct branches as well as all indirect branches and all other events, e.g. interrupts and other exceptions, that affect the program counter other than incrementing it. In order to save bandwidth on the trace bus, the Program Flow Trace Protocol (PFTP) does not report the current program counter address for every cycle. Especially for direct branches, the target address is not provided but only the information on whether a (conditional) jump was executed or not. The full program counter address is sent irregularly for synchronization (I-Sync message). In case of an indirect branch those address bits that have changed since the last indirect branch or the last I-Sync message are output.

In typical state-of-the-art applications the trace is recorded and the actual program flow is reconstructed from the trace offline. This approach does not work well for the purpose of runtime verification because we want to

1. react to detected failures as early as possible and
2. watch the system under test for a long time.

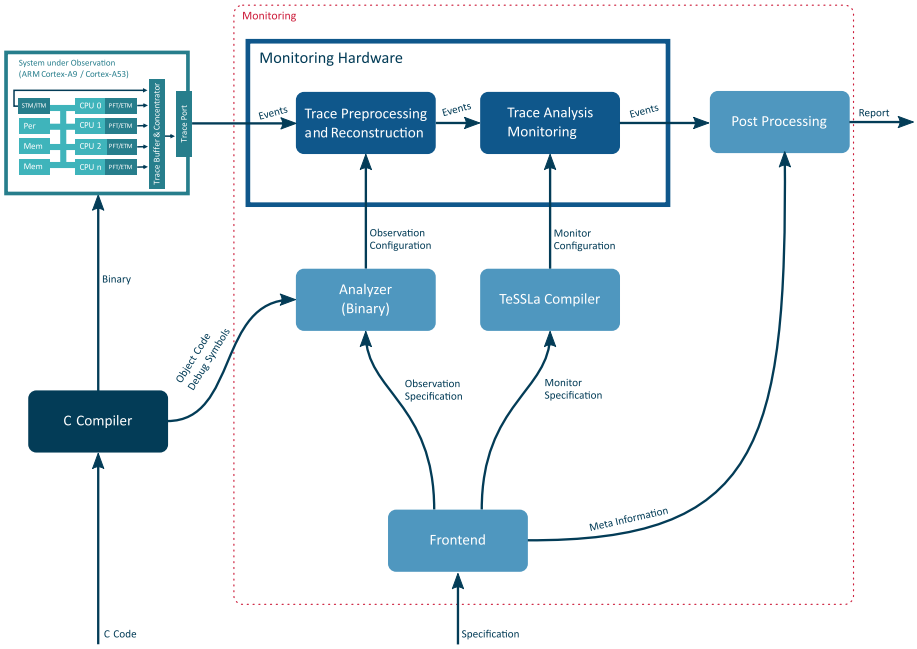


Fig. 2. Overview of the program flow trace monitoring setup. Operations of the cores are traced by the ETU, the trace is then reconstructed, filtered and monitored on the FPGA.

Currently, even with high technical effort, you can only record rather short sequences of trace data. For many real-world applications this might be not enough to spot errors, especially as you cannot start the recording based on complex events as you do not have the trace without reconstructing it from the waypoints.

Hence, we use an online (real time) program-flow-reconstruction method implemented on FPGA hardware [27,28]: From a static analysis of the binary running on the CPU we know all the jump targets of conditional direct jumps and can store those in a lookup table in the memory of the FPGA. Due to the high parallelism of the FPGA, we can split the trace data stream and reconstruct the program flow using the lookup table. The trace data stream can be split at the synchronization points that contain the full program counter address. A FIFO buffer stores the trace data stream until we reach the next synchronization point. For further processing we then immediately filter the reconstructed trace by comparing the reconstructed addresses to the tracepoints that correspond to the input events used in the TeSSLa specification. This comparison is realized by adding an additional tracepoint flag to the lookup table. After putting the slices back together in the right order we end up with a stream of tracepoints. Every tracepoint contains an ID and a timestamp. The timestamp is either assigned by the ARM processor if cycle accurate tracing is enabled or

during the reconstruction on the FPGA otherwise. Cycle accurate tracing is only available for certain processor architectures, because it requires high bandwidth on the trace port in order to attach timing information to every message.

Note that PFT traces logical addresses used in the CPU before the memory management unit (MMU) translates them to physical addresses, which are used to address concrete cells in the memory. The MMU is part of the cores and translates logical addresses to physical addresses. Because logical addresses are used in the program binary and by the CPU, we do not need to handle physical addresses.

In a typical multithreaded application, we have multiple threads running on different cores and multiple threads running on the same core using any kind of scheduling. While we can distinguish instructions traced from the different CPUs, we have to consider the actual thread ID in order to distinguish different threads running on the same core. This information is provided by a so-called context ID message [3], sent every time when the operating system changes the context ID register of the CPU. The logical addresses for different threads might be exactly the same, because the MMU is reconfigured in the context switch to point to another physical memory region. If we see a context switch to another thread, we have to change the lookup table for the program flow reconstruction information.

4 Monitoring Properties with Stream Processing

The specification language TeSSLa has been designed as a general purpose stream-based specification language, but with the prerequisites in mind that come with the setting of hardware-based monitoring of embedded systems. That is, technical prerequisites stemming from the processor architectures and the evaluation on FPGAs and prerequisites from the targeted use cases and targeted user groups relevant for the field of embedded systems. The TeSSLa compiler and interpreter are available online[1].

4.1 Technical Prerequisites

The most important technical prerequisite arises from the fact that due to the large amounts of trace data generated by multi-core CPUs, monitoring has to be performed in hardware. More specifically the monitoring specification has to be executed by a specialized engine implemented on an FPGA. This imposes several limitations that have to be addressed by the language design.

On an FPGA only a very limited amount of memory is available. Therefore the specification language should make it easy to specify monitors that are guaranteed to require only a small amount of memory. If memory is required to monitor a certain property, the user should have precise control of when and how memory is used.

[1] https://www.tessla.io.

The complexity of logic on an FPGA is limited. While a CPU can process programs of practically unlimited size, on an FPGA all logic has to be represented in hardware. Hence the basic operations of the specification language have to be simple. Also, the number of operations required to express the properties of interest should be relatively low.

Some properties and analyses are too complex to be evaluated on an FPGA but the amount of observation data is too high to evaluate them completely in software. They have to be split up in simpler parts suitable for evaluation on an FPGA and more complex parts that can be evaluated on an reduced amount of data. TeSSLa has been designed to be suitable for the restricted hardware setting but at the same time be flexible enough to not limit the user in the software setting.

It has to be easy to specify properties involving time in TeSSLa, because timing is a crucial aspect of embedded and cyber-physical systems.

Another important aspect is that of data. For many properties it is not enough to only specify the order and timing relation between certain events. It is also important to analyse and aggregate the associated data values.

4.2 Design Goals

TeSSLa's design goals described in this section are based on the prerequisites discussed in the previous section. On one hand, TeSSLa is a specification language rather than a programming language, to allow for simple system descriptions. TeSSLa should provide a different and perhaps more abstract perspective on the system under observation than its actual implementation. Specifying correctness properties in the same programming language that was also used to implement the system might lead to a repetition of the same mistakes in the implementation and the specification. On the other hand, TeSSLa should feel natural to programmers and should not be too abstract or require previous knowledge in mathematical logic. TeSSLa is a stream processing language, which can be used to describe monitoring algorithms, and not a mathematical logic describing valid runs of the systems. Aside from making TeSSLa easier to use for the practical software engineer this also allows to use TeSSLa for event filtering and quantitative and statistical analysis as well as live debugging sessions.

Time is a first-class citizen in TeSSLa, making the specification of timed properties as simple as possible. This does not change the expressiveness of TeSSLa specifications but makes it more natural to reason about time constraints. In cyber-physical systems events are often caused by external inputs of the physical system and hence not following regular clock ticks, but are appearing with variable frequency. In order to simplify specifications over such event streams, every event in TeSSLa always carries a time stamp and a data value.

TeSSLa has a very small set of basic operators which have a simple semantics. Such a small operator set simplifies the implementation of software and hardware interpreters and the corresponding compilers. TeSSLa transforms streams by deriving new streams from existing streams in a declarative functional way: One can define new streams by applying operators to the existing streams but

all streams are immutable. To gain the full expressiveness with such a small set of basic operators together with immutable streams we use recursive equation systems, which allow to express arbitrary computations over streams as combinations of the basic operators.

To allow adjustments of TeSSLa for different application domains and make it easier to use for practical software engineers without extending the set of basic operators we use a macro system. With TeSSLa's macro system we can build different libraries which support abstractions, code reuse and extension to allow the specification of more complex analyses. These libraries can use the domain knowledge and terms of the application knowledge without the need of adjusting the TeSSLa compiler and interpreter infrastructure.

TeSSLa's basic operators are designed to be implementable with limited memory, independent of the number of observed events. This allows for building TeSSLa evaluation engines in hardware without addressable memory. For every TeSSLa operator one only needs to store a fixed amount of data values, usually only one data value. TeSSLa allows to add additional data types to the language to adapt the language easily to different settings. Such data types can also be complex data types such as maps or sets, which then explicitly introduce infinite memory. To make the use of memory explicit via data types makes it very easy to identify the TeSSLa fragment that can be executed on hardware.

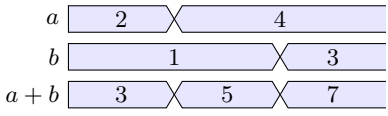
4.3 Basic Concepts

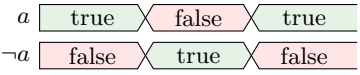
This section provides an overview on the monitoring specification language TeSSLa (Temporal Stream-based Specification Language).

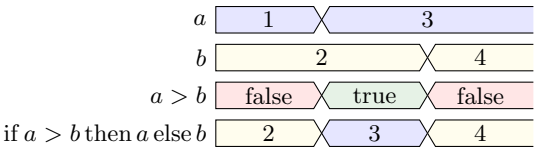
Its basic modelling primitives are timed event streams, i.e. sequences of events carrying a time stamp and some arbitrary data value. These streams are well suited to model the (timed) behaviour of all kinds of systems with one or multiple time sources.

Properties and analyses are then described by stream transformations. These transformations are expressed via recursive equations over a set of basic operators. We cover here the operators directly available on the hardware. See [7] for a complete, formal definition of TeSSLa's semantics.

The most central operator is signal lift which allows to lift operations on arbitrary data types to streams. For example, the addition on integer numbers can be lifted to streams of integers. This operator follows the intuition of piecewise constant signals, i.e. a stream of events is interpreted as a piecewise constant signal where events indicate changes in the signal value. Addition of two integer streams therefore results in a stream of events indicating the changes of the sum of two corresponding signals. Further examples are the negation of booleans that can be lifted to a stream of booleans and the ternary if-then-else function that can be lifted to a stream of booleans and two streams of identical type.

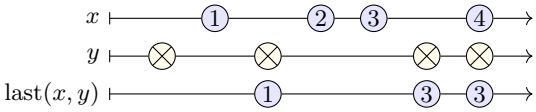




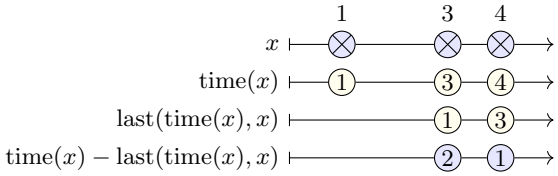


Note that the signal lift is implicitly applied when you use the built-in operators on integer numbers or booleans on streams of the corresponding types.

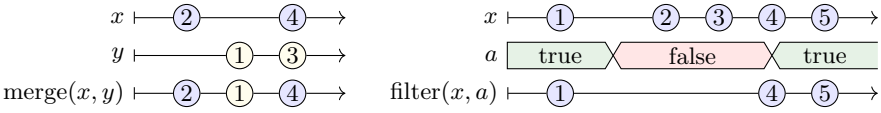
In order to define properties over sequences of events the operator last has been defined. It allows to refer to the values of events on one stream that occurred strictly before the events on another stream.



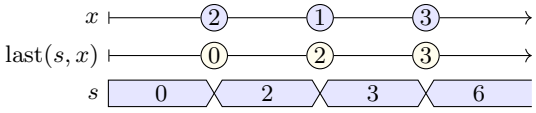
The time operator can be used to access the timestamp of events. It produces streams of events where the events carry their timestamps as data value. Hence all the computations available to data values can be applied to timestamps, too.



Furthermore, the language contains two operators to process streams in an event-oriented fashion, filter and merge. Filter allows to filter the events of one stream based on a second boolean stream interpreted as piecewise constant signal. Merge combines two streams into one, giving preference to the first stream when both streams contain identical timestamps.



Using the last operator in recursive equations, aggregation operations like the sum over all values of a stream can be expressed. The merge operation allows to initialize recursive equations with an initial event from an other stream, e.g. $s := \text{merge}(\text{last}(s, x) + x, 0)$.



Finally, there is the constant nil for the empty stream and the operator const converting a value to a stream starting with that value at timestamp 0.

These operators are enough to express arbitrary stream transformations. The recursion is limited to recursive expressions involving last. This guarantees that the specifications are directly executable and thereby efficiently monitorable. It also allows the user to think of such specifications as programs which can be more convenient for programmers than the mathematical view of recursive equations.

All of these operators can be implemented using finite memory, i.e. a small amount of memory independent of the amount of data that has to be monitored.

The specification language facilitates abstract specifications and extensibility through a macro system. Here, macros are functions that can be evaluated at compile time. Therefore specifications can be structured in a functional fashion without requiring memory for a call stack at runtime. These macros can also be used to provide a library of common specification patterns.

Another extension point is that of data types. The language is defined agnostically with respect to any time or data domain. Depending on the application domain and the restrictions of the execution domain, different data structures can be used to represent time and data. For hardware monitoring this will typically be restricted to different atomic data types such as integer and floating point numbers. For software monitoring this might also comprise more complex data structures like lists, trees and maps.

5 Example Scenario

In this paper we use a highly simplified engine controller of a train as an example system that we want to analyse and monitor using the hardware monitoring technique presented in this paper. One of the most important aspects of (autonomous) train driving is adhering to the speed limits and the railway signals. Hence we only consider the process of braking a train in front of a stop signal. Since the braking distance for trains is rather long, there are additional distant signals positioned in front of the actual stop signal which indicate caution if the train has to stop at an stop signal further down the track. To make sure that the train really stops in front of the stop signal an automatic speed supervision system checks if the train never exceeds the allowed maximal speed.

To keep our example scenario simple, we consider the speed limits of the *intermittent automatic train running control* system (in German *Punktförmige Zugbeeinflussung, PZB*): We consider passenger trains with a maximal allowed speed of 165 km/h. If the train passes a distant signal indicating caution, it has to reduce its speed to 85 km/h in 23 s. The actual stop signal is located 1000 m after the distant signal. 250 m in front of the stop signal the train must have a speed below 65 km/h which must be reduced to 45 km/h over the next 153 m.

The intermittent automatic train running control system detects special inductors mounted on the tracks which indicate distant and stop signals as well as the point 250 m in front of the stop system. The maximal allowed speed is shown in red in Fig. 4. The speed of an allowed execution of the system is shown in blue in the same diagram.

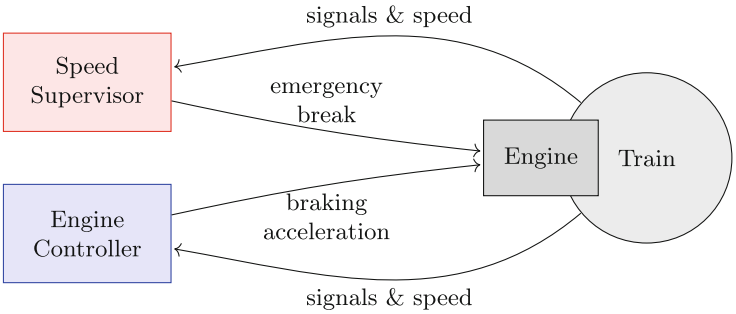


Fig. 3. Example scenario consisting of the train with its engine, the discrete speed supervisor and the engine controller.

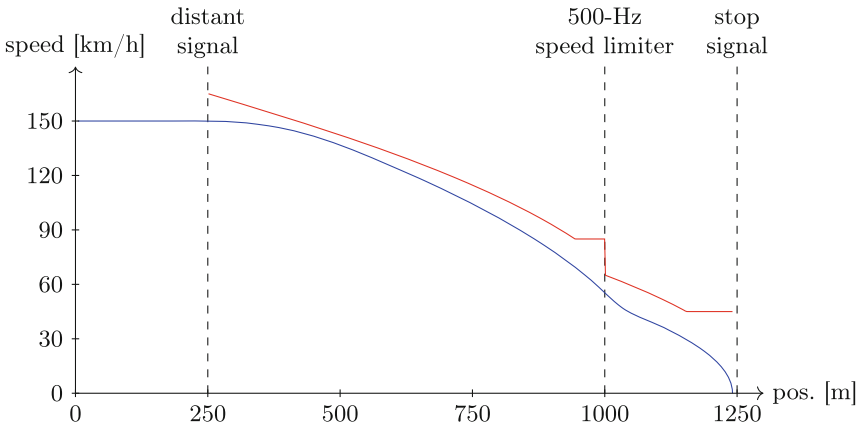


Fig. 4. Diagram showing the speed of the train (in blue) in relation to the train's position. The red curve shows the allowed speed of the train. (Color figure online)

The simulated system as depicted in Fig. 3 consists of three major components:

- The engine controller gets information about the current speed of the train and the signals and the 500 Hz inductor the train passed by. It controls the engine by setting the brake acceleration.
- The speed supervisor gets the same information about the train and computes the currently allowed speed. It compares this with the actual speed and performs an emergency brake if the current speed exceeds the currently allowed speed.
- The train is simulated using a highly simplified model with the initial speed and the acceleration set by the controller.

6 Measuring Timing Constraints

Timing is important for cyber-physical systems. Timing constraints encompass the runtime of tasks, functions or chains of dependant tasks, as well as response times and idle times. Checking such timing constraints or measuring the worst-case or average time consumed is an important task of hardware monitoring.

In this first example we want to measure the runtime of one execution of the speed supervisor. The supervisor is implemented in a function `supervisor`, hence we want to measure the runtime of this function. Therefore we have to specify that we are interested in the events of entering and leaving this function:

```
def call := function_call("supervisor")
def return := function_return("supervisor")
```

Using the `runtime` macro of the standard library we can now produce an event stream which has an event every time the functions returns. The data value of this event is the runtime of the function that did just return.

```
def supervisorRuntime := runtime(call, return)
```

The function `runtime` is defined in the standard library as follows:

```
def runtime(call: Events[Unit], return: Events[Unit]) :=
  at(return, time(return) - time(call))

def at[A,B](trigger: Events[A], values: Events[B]) :=
  filter(values, time(trigger) == time(values))
```

`Events[T]` is the type representing streams of elements of type `T`. The function `at` filters the `values` stream down to the events happening exactly when an event on the `trigger` stream happens. The runtime can then be defined as the difference of the timestamps of the last `return` and the last `call` event evaluated only at the `return` events.

Apart from measuring and checking timing constraints one can also use TeSSLa to aggregate statistical data. In this case lets compute the maximal runtime of the speed supervisor using the function `max` of the standard library:

```
def maxSupervisorRuntime := max(supervisorRuntime)
```

This maximum aggregation function is defined in the standard library by first defining the maximum of two integer values:

```
def maximum(a: Int, b: Int) := if a > b then a else b
```

This can now be aggregated in a similar recursive definition as the sum explained above:

```
def max(x: Events[Int]) := {
  def result := merge(maximum(last(result, x), x), 0)
  result
}
```

This pattern of aggregating functions can be generalized using TeSSLa's higher order functions. A fold function which takes a function and recursively folds it over an input stream is defined in TeSSLa as follows:

```
def fold[T,R](f: (Events[R], Events[T]) => Events[R],
    stream: Events[T], init: R) := {
  def result: Events[R] :=
    merge(f(last(result, stream), stream), init)
  result
}
```

Now we can define the aggregating `max` function simply by folding the `maximum` over the input stream `x`:

```
def max(x: Events[Int]) := fold(maximum, x, 0)
```

Especially these last examples show one of the strengths of the integrated hardware monitoring, where the online monitors are part of the system: We do not need to store the trace to analyse it. Hence we compute statistical data like the maximal runtime over very long executions of even multiple days or weeks since only the important events are stored and not the complete trace.

7 Checking Event Ordering Constraints

Another important class of properties of cyber-physical systems are event ordering constraints: Here we are not interested in the exact timing of the events, but in their order or presence. So for example one can check if certain events are always preceded or followed by other events.

As an example we again consider the speed supervisor which calls several local helper functions in order to compute the currently maximal allowed speed. The function `getAllowedSpeed` returns the currently allowed speed. Depending on the last seen signal or magnet it either calls `computeAllowedSpeedDistant` or `computeAllowedSpeedMagnet`. So first we want to assure that every call to `getAllowedSpeed` leads to a call of at least one of these two helper functions. To do so we have to declare these function calls as in the previous examples:

```
def call := function_call("getAllowedSpeed")
def return := function_return("getAllowedSpeed")
def computeDistant :=
  function_call("computeAllowedSpeedDistant")
def computeMagnet :=
  function_call("computeAllowedSpeedMagnet")
```

Using these three event streams we can now check every time the function `getAllowedSpeed` returns if one of the two other functions was called after the function `getAllowedSpeed` was entered. Such ordering constraints are expressed in TeSSLa as a comparison of the timestamps of the events:

```
def computation := on(return,
  time(computeDistant) > time(call) ||
  time(computeMagnet) > time(call))
```

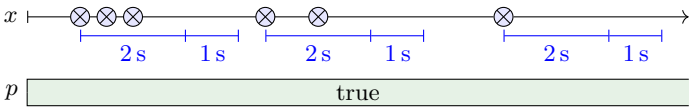
As another example we can analyse a complete execution of the braking sequence: Once the function `computeAllowedSpeedMagnet` was called for the first time we must be past the 500 Hz inductor and hence the function `computeAllowedSpeedDistant` must not be called any more. The following TeSSLa specification checks whether all events on the stream `computeMagnet` are happening after all events on the `computeDistant` stream:

```
def magnetAfterDistant :=
  time(computeMagnet) > time(computeDistant)
```

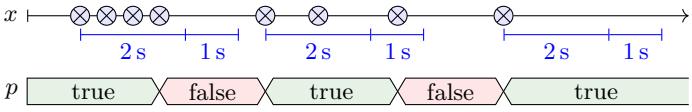
By combining timing and event ordering constraints one can express arbitrary complex constellations. As an example we consider the burst pattern known from automotive modelling languages such as the AUTOSAR Timing Extension [5] and the EAST-ADL2 timing extension TADL2 [12]. Such a pattern checks if events happen in bursts. The pattern is parametrized in the maximum number of events allowed in the burst, the maximal length of the burst and the minimum time without any event after the burst. In TeSSLa such a pattern can be implemented as macro and used as follows:

```
def p := bursts(x, burstLength = 2s,
                   waitingPeriod = 1s,
                   burstAmount = 3)
```

The following event pattern satisfies this burst pattern:



To violate the burst pattern you can either have too many events during one burst, or an event during the waiting period after the burst:



Such complex event patterns can be used to spot variations in event patterns of complex systems without a detailed knowledge of the dependencies of the individual events. For example in our scenario we can take the combination of all the function calls described above. If the supervisor is called roughly every second, this should adhere the following specification:

```
def e := merge3(call, computeDistant, computeMagnet)
def b := bursts(e, burstLength = 100ms,
                   waitingPeriod = 500ms,
                   burstAmount = 3)
```

With this specification you can spot abnormal behaviour, e.g. too many function calls during the computation. Such an abnormal behaviour is not necessarily a bug, but automatic detection of interesting parts of traces can be very helpful to speed up the debugging process, especially for partially unknown systems.

8 Monitoring Data Values

The previous examples were limited to the program flow trace, but in some situations one needs the actual values of variables or arguments to check the correct behaviour of the system under test. For example if we want to check that the allowed speed computed by the supervisor is equal to 85 km/h 23 s after the distant signal we need this computed value.

The *Instrumentation Trace Macrocell (ITM)* is part of the ARM CoreSight, see Chapter 12 of the CoreSight Technical Reference Manual [1], and allows programs to explicitly write small portions of data to the trace port. While the program flow trace can be monitored completely non-intrusively as described in the previous sections, one has to instrument the C code in order to use the ITM data trace. Figure 5 shows how the tooling and the workflow is adjusted in order to integrate such ITM trace messages into the system:

To make the usage of the ITM data trace comfortable, the instrumentation of the C code happens automatically. Therefore the C code and the given specification are analysed with regard to what data is used in the specification. The instrumenter than adds corresponding statements writing this information to the trace port.

Coming back to the example of verifying the computed allowed speed we need to know the last seen signal and the computed allowed speed. We can define TeSSLa signals of both values as follows:

```
def signal := function_argument("getAllowedSpeed", 1)
def allowed_speed := function_result("getAllowedSpeed")
```

`signal` now contains the value of the first argument of `getAllowedSpeed` and is updated with every function call. `allowed_speed` contains the return value of the same function and is updated every time the function returns. The instrumenter adds the following `debug_output` statements to the function:

```
double getAllowedSpeed(int signal, ...) {
  debug_output(1, (int64_t) signal);
  double result = ...
  tessla_debug(2, (int64_t) (result * 1000));
  return result;
}
```

The ITM tracing provides several data value slots, so in order to distinguish the two data values we are interested in we map them to the ITM slots 1 and 2. The TeSSLa specification is rewritten in terms of the current value and slot as follows:

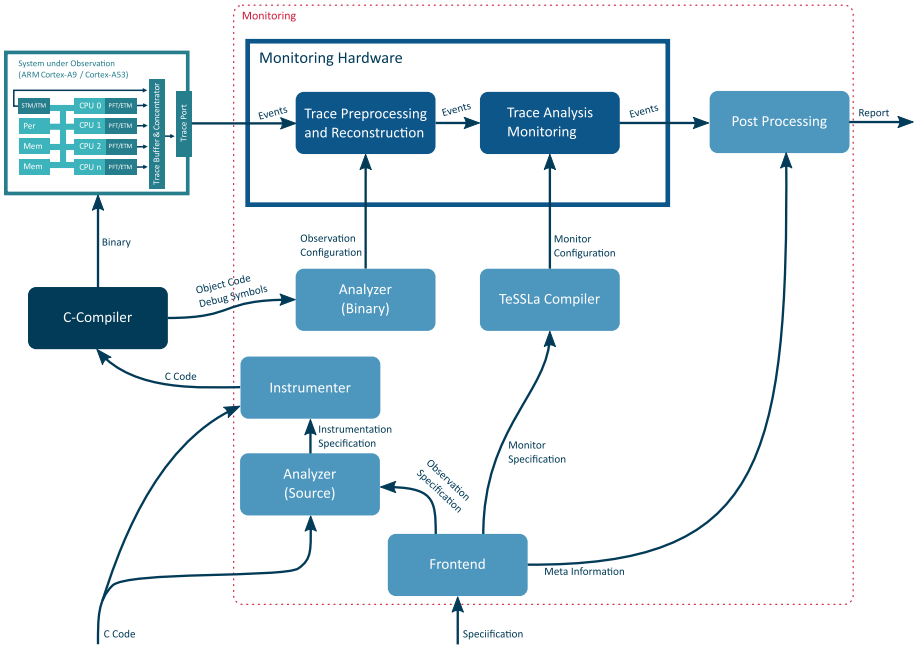


Fig. 5. Overview of the ITM trace setup.

```
in debug_slot: Events[Int]
in debug_value: Events[Int]
def signal := filter(debug_value, debug_slot == 1)
def allowed_speed := filter(debug_value, debug_slot == 2)
```

Now we can express the actual property: First we filter all changes of the `signal` stream for those where the value becomes `DISTANT_SIGNAL_CAUTION`. Then we check if the allowed speed is below 85 km/h if it was computed more than 23 s after we have seen the distant signal. We have to apply some unit conversions as the speed is internally represented in m/s.

```
def caution := filter(changes(signal),
                      signal == DISTANT_SIGNAL_CAUTION)
def valid :=
  if time(allowed_speed) - time(caution) > 23s
  then allowed_speed * 36 <= 85 * 1000
  else true
```

In the above specification we used the macro `changes` which returns only those events which have a different value than the previous one. Such a macro is defined in TeSSLa's standard library as follows:

```
def changes[A](signal: Events[A]) :=
  filter(signal, signal != last(signal, signal))
```

The above example of checking computation results is just one example where data traces are useful. The data trace can also be useful to check array indices and other indirect memory accesses. Further applications are additional instrumentations to enrich the program flow trace with extra information, e.g. additional timing data or clock values for synchronization purposes.

9 Practical Hardware Setup

The typical hardware setup comprises three components: a development board running the application under observation, the monitoring hardware and a personal computer running both the monitoring tools and the development tools for the application. Figure 6 depicts this setup. The development board is connected to the desktop computer via a USB cable. This connection is used to upload and start the application. The development board is connected to the monitoring hardware via an Aurora cable transmitting the compressed processor trace. The monitoring hardware is connected to the desktop computer via the second USB cable. This connection is used to configure the monitoring hardware and to receive the output events. The monitoring hardware has several LEDs that indicate its status and can also be used to display some status information of the monitor. Furthermore it has additional USB ports that are not used in normal operation and are only required to install updates.

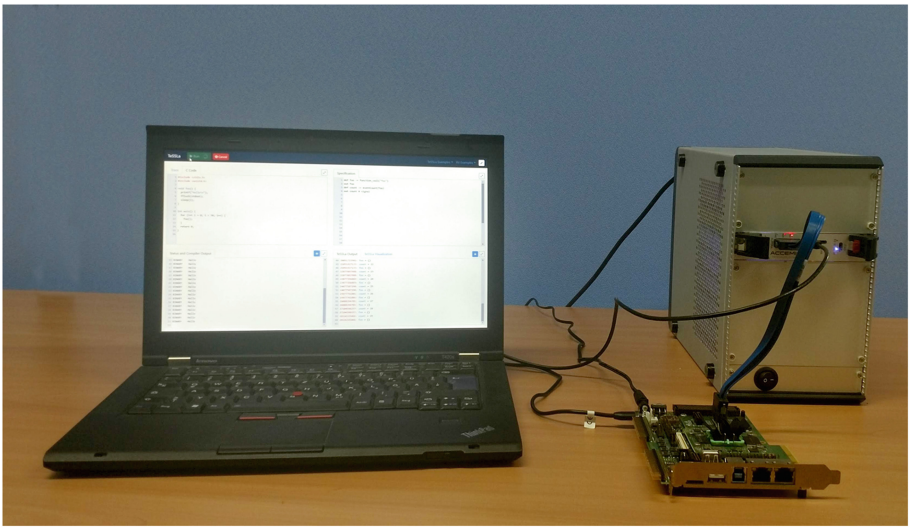


Fig. 6. The hardware setup comprising from left to right a personal computer, a development board and the monitoring hardware.

10 Conclusion

In this paper we demonstrated hardware-based runtime verification using the embedded tracing units of processors. While we discussed the technique in this paper using the example of the ARM CoreSight technology, other processor vendors recently developed similar tracing units: Intel's Processor Tracing IntelPT [13] supports program flow and data traces and for PowerPC the NEXUS tracing technology is already established. Combining the trace reconstruction discussed in this paper with stream-based online monitoring allows for long term monitoring of systems under observation without the need to store the processor traces. This technology can be seen as a milestone in the non-intrusive online-tracing of processors since all established solutions either need to modify the program and hence the timing of the system quite drastically or they can only analyse rather short executions, because they have to store the highly compressed processor trace in order to reconstruct it offline.

Online monitoring of processor traces can reduce costs for certification and development efforts as well as debugging costs. The ability to do long term analyses of systems in the field without the need to modify the source code, allows companies on the one hand to demonstrate the correct behaviour of their systems. On the other hand they can use this technology to identify root causes of bugs that occurred in productive systems faster. Currently bugs detected in productive systems often have to be reproduced in the lab to extract proper traces from the system which are needed to perform the root cause analysis. Using runtime verification of processor trace data can provide additional information already during the normal execution of the productive system and hence provide valuable information on errors faster and more accurate.

References

1. CoreSight Components: Technical Reference Manual. ARM DDI 0314H, July 2009. Issue H
2. ARM Limited, ARM IHI 0035B: CoreSight Program Flow Trace: PFTv1.0 and PFTv1.1 - Architecture Specification, March 2011. Issue B
3. ARM Limited, ARM IHI 0029B: CoreSightTM Architecture Specification v2.0 (2013). Issue D
4. ARM Limited: ARM DS-5 ARM DSTREAM User Guide Version 5.27 (2017)
5. AUTOSAR: Specification of Timing Extensions. Technical report, AUTOSAR (2017)
6. Backasch, R., Hochberger, C., Weiss, A., Leucker, M., Lasslop, R.: Runtime verification for multicore SoC with high-quality trace data. ACM Trans. Design Autom. Electr. Syst. **18**(2), 18:1–18:26 (2013)
7. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: temporal stream-based specification language. In: Proceedings of the Formal Methods: Foundations and Applications - 21st Brazilian Symposium, SBMF 2018, Recife, Brazil, 26–30 November 2018. Lecture Notes in Computer Science. Springer (2018)

8. D'Angelo, B., et al.: LOLA: runtime monitoring of synchronous systems. In: TIME, pp. 166–174. IEEE (2005)
9. 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
10. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 152–168. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_10
11. Freescale Semiconductor, Inc.: P4080 Advanced QorIQ Debug and Performance Monitoring Reference Manual, Rev. F (2012)
12. Goknil, A., DeAntoni, J., Peraldi-Frati, M., Mallet, F.: Tool support for the analysis of TADL2 timing constraints using timesquare. In: 2013 18th International Conference on Engineering of Complex Computer Systems, Singapore, 17–19 July 2013, pp. 145–154. IEEE Computer Society (2013)
13. Intel Corporation: Intel (R) 64 and IA-32 Architectures Software Developer's Manual (2016)
14. Jaksic, S., Bartocci, E., Grosu, R., Kloibhofer, R., Nguyen, T., Nickovic, D.: From signal temporal logic to FPGA monitors. In: MEMOCODE, pp. 218–227 (2015)
15. Jakšić, S., Bartocci, E., Grosu, R., Ničković, D.: Quantitative monitoring of STL with edit distance. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 201–218. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_13
16. Leucker, M.: Teaching runtime verification. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 34–48. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_4
17. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebr. Progr. **78**(5), 293–303 (2009)
18. Lu, H., Forin, A.: Automatic processor customization for zero-overhead online software verification. IEEE Trans. VLSI Syst. **16**(10), 1346–1357 (2008)
19. Moosbrugger, P., Rozier, K.Y., Schumann, J.: R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. Form. Methods Syst. Des. **51**(1), 31–61 (2017)
20. Moreno, C., Fischmeister, S.: Non-intrusive runtime monitoring through power consumption: a signals and system analysis approach to reconstruct the trace. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 268–284. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_17
21. Nutt, G.J.: Tutorial: computer system monitors. SIGMETRICS Perform. Eval. Rev. **5**(1), 41–51 (1976)
22. Pellizzoni, R., Meredith, P.O., Caccamo, M., Rosu, G.: Hardware runtime monitoring for dependable cots-based real-time embedded systems. In: Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November–3 December 2008, pp. 481–491. IEEE Computer Society (2008)
23. Reinbacher, T., Függer, M., Brauer, J.: Runtime verification of embedded real-time systems. Form. Methods Syst. Des. **44**(3), 203–239 (2014)
24. Selyunin, K., et al.: Runtime monitoring with recovery of the SENT communication protocol. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 336–355. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_17
25. Shobaki, M.E., Lindh, L.: A hardware and software monitor for high-level system-on-chip verification. In: ISQED, pp. 56–61. IEEE Computer Society (2001)

26. Tsai, J.J.P., Fang, K., Chen, H., Bi, Y.: A noninterference monitoring and replay mechanism for real-time software testing and debugging. IEEE Trans. Softw. Eng. **16**(8), 897–916 (1990)
27. Weiss, A., Lange, A.: Trace-Data Processing and Profiling Device, US 9286186 B2, 15 March 2016
28. Weiss, A., Lange, A.: Trace-Data Processing and Profiling Device, EP 2873983 A1, 20 May 2015