embeddedworld2020 Exhibition&Conference ... it's a smarter world

Debugging Complex Failures of Real-Time Multi-Core Systems

Albert Schulz
Accemic Technologies GmbH
Dresden, Germany
aschulz@accemic.com

Alexander Weiss
Accemic Technologies GmbH
Kiefersfelden, Germany
aweiss@accemic.com

Franz Münz
Airbus Defence and Space GmbH
Manching, Germany
franz.muenz@airbus.com

***Abstract*—This paper briefly examines common types and the nature of failures in complex, embedded multi-core environments. A novel tool CEDARtools® is presented, overcoming these issues by processing hardware-generated trace in real-time, providing complex triggers and variable monitoring scopes, facilitating a post-mortem analysis. The approach permits preventive monitoring even before the program fails and allows mastering the evolving complexity in embedded development.**

***Index Terms*—economic, multi-core, debugging, complex transient failures, sporadic failures, hardware trace, trace analysis**

I. Introduction

As the number of features and demands in modern embedded systems grows, the overall number of code lines in those projects and their complexity increases. While back in 2010, an average car had around 10 million *source lines of code* (SLOC), this number has multiplied by a factor of 15 to around 150 million SLOC by 2016 [1].

To satisfy the demand for more computational power, multi-core processors have been gaining more and more interest both in the automotive and the avionic industries. Although they allow for more powerful, higher-integrated and cost-effective implementations, they also increase the level of complexity in software development and the chances of the occurrence of *complex transient failures*, such as data races, deadlocks or resource starvation. Another more colloquial description may be *sporadic failures* that only occur under certain circumstances that are typically hard to reproduce and comprehend.

Research has shown that software defects are more likely with an increasing complexity, as shown in Fig. 1. The average defect potential increases as more functionality – measured in *function points* (FP) – is implemented. The source of the depicted statistics was published in 2011 [2], a time when multi-core processors were not yet widespread. Thus, the defect potential of these modern systems must even be assumed to be worse taking their complexity and concurrent operation into account.

Avoiding defects is the main goal of the development process in safety-critical applications where any malfunction could lead to costly product recalls or – even worse – serious injuries involving human life. Achieving this is hard and calls for conscientious work within all phases of the generic software development process shown in Fig. 2.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 732016.

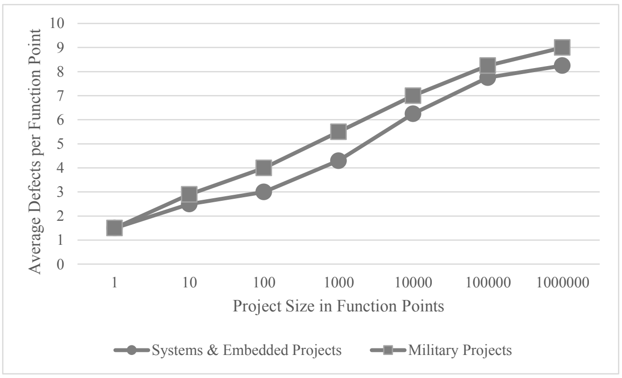


Fig. 1. Average defect potential by type and size of software projects (Source: [2])

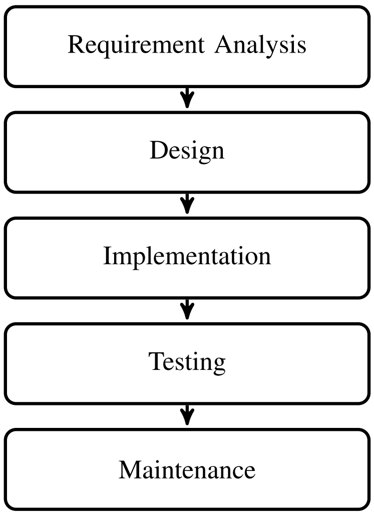


Fig. 2. Phases of software development

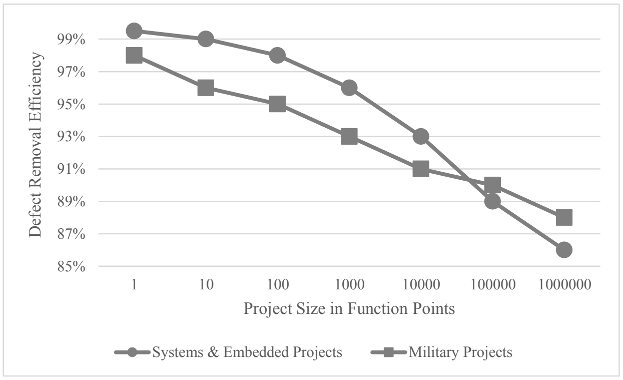


Fig. 3. Defect removal efficiency by type and size of software projects (Source: [2])

Despite all efforts, already tested and released software is likely to remain with defects, which were not detected in earlier phases during its development. For more complex software projects, the defect removal efficiency even lowers [2] as depicted in Fig. 3.

Both effects, the increased probability of defects and the reduced defect removal efficiency, lead, by trend, to a higher number of post-release defects as projects become larger. Since these defects were neither caught during the implementation nor in the testing phase, they are by nature more complex to trigger, involving various conditions, or only happen in very unlikely cases.

An exemplary abstract cause-effect chain is depicted in Fig. 4 showing a defective system. A code defect causes a *root infection* in state z_2, represented by its internal values and outputs. However, this might not be visible to the engineer due to the limited insight into the system provided by its observable outputs e_1 and e_2. Even worse, the actual root infection may be overwritten and masked regularly as in state z_5 before it eventually results in an observable failure in state z_6.

The remainder of this paper is structured as follows. In the next section, Sec. II, a list of requirements is given that a debugging tool should fulfill to efficiently track down transient faults. This is followed by a presentation and comparison of state-of-the-art debugging approaches in Sec. III, examining their suitability for the class of complex transient failures. In Sec. IV, a new approach is presented, explaining its working principles and detailing the differences to other approaches. Finally, a summary and conclusion are given.

II. Multi-core Debugging Requirements

Based on the experiences at Airbus Defence & Space GmbH, the following requirements for efficient multi-core debug tools have been derived. [4]

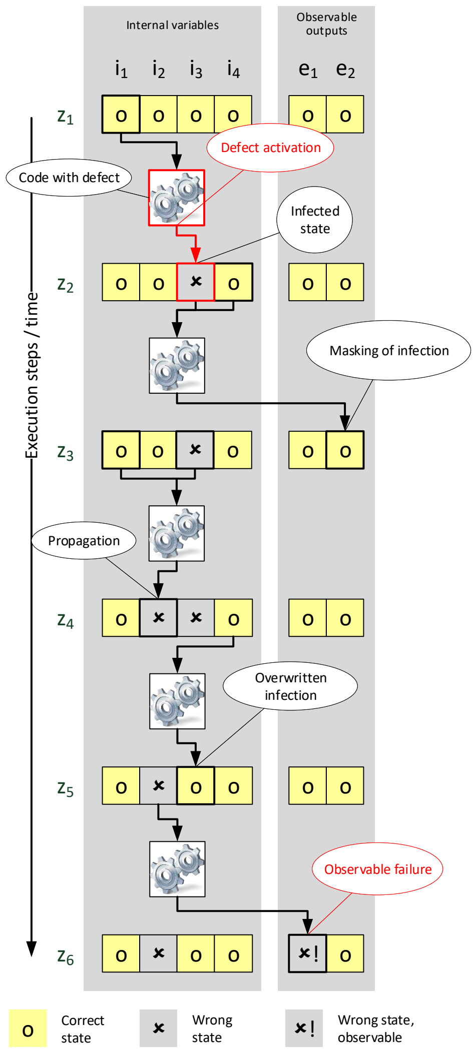


Fig. 4. A defective program execution as a succession of states (inspired by [3])

A. *Deep system insight*

It is favorable to be able to observe infections as soon as possible. Thus, a deep system insight is required, being able to notice any unusual behavior and react on that. Deep system insight means, from an access privilege point of view, that the observer must be able to see all data the running process may have access to as well.

B. *Non-instrusiveness*

A fundamental claim is that a tool does not change the behavior of the system under test during observation. This is in line with the behavior of bus tracers (Ethernet, PCI, MILSBUS, . . .) that do not interact with the exchanged signals. Non-intrusiveness is important in order to (a) not shadow any of the system behavior – often referenced in literature as Heisenbugs [5] – and (b) not to introduce new distracting infections.

C. *Decoupling from software*

A decoupling of the observation tool from the developed software also avoids the need for changes in the code so that the observation can be conducted on the actual release code. It can, thus, be seamlessly continued into the maintenance phase even with changing the observation focus variably without touching the system under observation. This is especially beneficial for safety-critical software, which is required to be certified in a time-consuming process and, hence, cannot be modified without serious ramifications to time and cost.

D. *Long observation period*

Issues under observation tend to happen very rarely in time. Hence, observation should not be restricted to a few seconds or minutes but should preferably be unlimited in time. This increases the probability of encountering and detecting the fault conditions to allow the observation of the sought misbehavior. It is also a prerequisite to satisfy the next criteria.

E. *Multiple focuses*

Debugging efficiency can be highly increased by the ability to analyze multiple independent failures in parallel due to the low occurrence rate of each single failure. The rationale here is that certain sporadic failures can only be reproduced on system integration level, which has the main purpose of performing formal testing. Dedicating an entire system integration facility (or perhaps even lower levels) entirely to chasing a sporadic failure, especially one that is not deemed safety-critical, is not economic. Therefore, it must be possible to analyze the root cause of such failures in parallel to normal testing activity.

F. *Multi-core support*

The tool should be able to debug parallel software running on multi-core processors with the aim of being able to catch failures introduced by this architecture.

G. *Cost effectiveness*

The debugging equipment and process has to be in proportion with the cost of the bug fixed in order to be applied sustainably in an industrial environment. In the avionics industry, a fix for a post-release defect is assumed to induce costs of around 500.000€ or even more depending on the project/program [6].

H. *Autonomous operation*

Once armed, the tool should be as autonomous as possible so that it can be used for long test runs in the real system environment where physical access might not be possible.

I. *Adaptability*

The tool should be rapidly adaptable in terms of the observation focus. Complex failures have the tendency to require several iterations of reproducing them before the root cause is understood. This may involve experimentation with hypotheses that can be verified or falsified. Continuously narrowing down the area, in which the root cause might be located, requires constant changes to the observation focus. The overall debug time can be reduced the faster and easier the tool can be adapted to a new focus.

III. State of the Art

`printf()` Debugging: Named after the `printf()` C function, this is the most basic form of debugging limited to writing debug information to a console output, such as RS232, by explicitly instrumenting the source code.

This approach can have massive impacts on timing for a single core and may introduce undesired synchronization in parallel programs when sharing the same output console. Besides, it is the least dynamic form for obtaining an understanding of what is happening inside a program. If the information of interest changes, the software must be changed and thus recompiled most of the time. With many iterations, this process can become very tedious and time-consuming. In the worst-case, this may threaten project goals if no other method is available. Even though this approach is archaic, it is still sometimes used, especially if no more advanced debug method is available.

Start/Stop Debugging: This very common debugging approach is based on the direct control of the program execution. One may break the execution at a certain point in time and analyze the current state.

However, the approach has some major drawbacks, when it comes to debugging complex transient failures:

1) The high interference with the program execution by directly controlling the execution progress changes the time behavior. This makes it hard to reproduce failures where timing matters. Besides, in cyber-physical systems where the software controls physical actuators, such as an engine, this approach might not even be applicable since halting the execution would cause physical damage to the engine.
2) It typically only allows stepping forward. Breakpoints have to be chosen thoughtfully to be early enough to reflect the root cause of the observed failure. This typically leads to a need of rerunning the software over and over again in an effort to trace back the manifestation of a failure.
3) Due to the cyclic debugging fashion, the system behavior is required to be fundamentally deterministic to enable the observation and investigation of a failure. This is hard or impossible to achieve in parallel or in real-time programs with dynamic asynchronous input data.

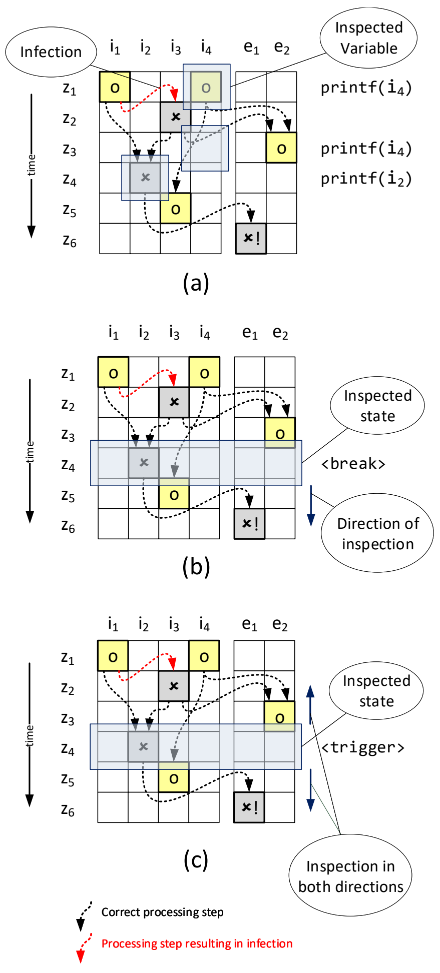


Fig. 5. Comparison of (a) `printf()` (b) start/stop and (c) omniscient debugging approaches for exemplary program execution in Fig. 4

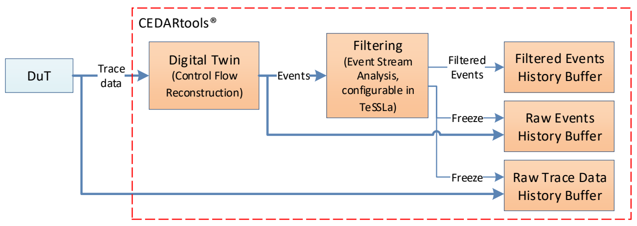

CEDARtools®
DuT
Trace data
Digital Twin (Control Flow Reconstruction)
Events
Filtering (Event Stream Analysis, configurable in TeSSLa)
Filtered Events
Filtered Events History Buffer
Freeze
Raw Events History Buffer
Freeze
Raw Trace Data History Buffer

Fig. 6. Components of the CEDARtools® hardware connected to the DuT

Omniscient Debuggers: These debuggers, also known as *back-in-time* or *reversible debuggers*, record the whole or parts of the program execution for reconstructing the execution history and corresponding program contexts. This allows the engineer going back in time, which conforms to the natural way of searching causes of observed failures. The records may be generated by code instrumentation or by using hardware trace data, generated by dedicated on-chip debug modules [7], [8].

For illustration purposes, we apply the presented approaches to the example shown in Fig. 4. As shown in Fig. 5, *printf()-debugging* partially unveils the internal variables for various states. In comparison, the *start/stop debugging* and *omniscient debugging* approaches provide full insight of the systems state at the time of inspection. Through the ability of the latter approach to go back in time once a trigger condition has been hit, the cause effect chain can be efficiently studied without having to rerun the program, which is especially beneficial for sporadic, hard-to-reproduce failures.

Additionally, both former approaches suffer from the fact that they are highly intrusive whether through direct source code modifications or execution control.

Only omniscient debuggers relying on hardware trace information can greatly avoid an interfere with the program execution. However, the hardware trace is an extremely high-bandwidth data source encoding the program control flow, execution context information and, potentially, data trace values. They easily add up to several hundreds of megabytes per second [9]. In practice, the available memory space limits the feasible recording times and, hence, inspection periods of this approach. This limitation makes it hard to use in production and leaves space for new approaches.

IV. Real-time Hardware-based Trace Analysis

The CEDARtools® system is based on a live synchronized digital twin representing the relevant activities within the Device under Test (DuT). A basic block diagram is shown in Fig. 6. For the interested reader, a more detailed insight is given in Fig. 8, illustrating the whole debugging process using a brief code example.

The synchronization is based on hardware trace data as provided by almost all embedded processor architectures. Many processor vendors facilitate their products with dedicated trace modules providing data about the programs execution without interfering with the execution. Examples are ARM CoreSight™ [10], Intel® Processor Trace [11], Infineon ED [12] and NEXUS (NXP QorIQ®) [13]. The data provided typically comprise control flow information and memory accesses including data for multiple cores. This facilitates a very deep insight into the system.

The trace is filtered and analyzed on-the-fly during execution as it is received. It allows detecting infections as soon as possible during run time by using the filtered events to derive complex triggers. In parallel, various history buffers of the raw trace data as well as the filtered event streams are recorded. The triggers allow freezing these buffers for a later analysis. The digital twin and the processing engines are implemented on a high-performance FPGA in the form of special-purpose *control flow reconstruction units* and *event processing units*, which are optimized for high-throughput data-flow event processing.

All of the event processing, especially the computation of complex triggers – the key feature for the detection of infections – can be described in the high-level language TeSSLa [14].

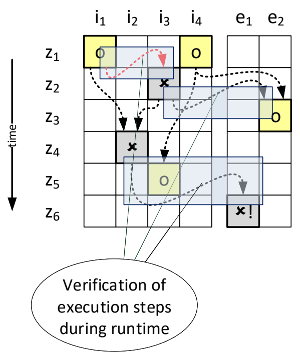


Fig. 7. Observation of execution steps using live hardware trace analysis

This is a temporal stream-based specification language designed for specifying and analyzing the behavior of cyber-physical systems (see http://tessla.io). It acts on the basis of events derived from the raw hardware trace, such as:

- executed instructions,
- function calls,
- task switches, and
- accesses to global variables.

Examples for specifications are checks for invalid program sequences, data range checks, exceeded task timings, response time checks, and others. A fundamental principle of the CEDARtools® workflow is to keep the declaration of such specifications simple so as to make the tool accessible for test engineers of different skill levels. Therefore, TeSSLa supports the implementation of functions to encapsulate more complex logic, and the ability to import existing timing constraint specifications, e.g. from the AUTOSAR™ TIMEX [15] or AMALTHEA [16] formats. In summary, the tool supports a large number of various complex triggers, the combination of synchronized traces from multiple cores and the ability to observe them for an arbitrary amount of time. Moreover, the CEDARtools® hardware has been designed to combine and process trace data from multi-processor systems, but which is out of the scope of this paper.

Coming back to the execution of our example program: Using the proposed approach, a verification of multiple execution steps can be done live at run time for a long observation period as illustrated in Fig. 7. Once armed, it can run autonomously during the tests. The history buffers are frozen on any misbehavior. They can be read after the debug run serving as a basis for further investigation. In order to narrow down the root cause of the observed failure, the observation focus can be easily changed by reconfiguring the system with a new TeSSLa specification.

The shown approach can be seen as a mitigation of the limitations that common omniscient debuggers suffer in terms of observation time. This greatly increases the observability of the DuT and also provides complex triggers to enable a faster successful capture of traces containing the relevant information of root infections.

V. Summary and Conclusions

The rising complexity of software projects leveraging multi-core processors requires new debugging methods in accordance with the ones presented in this paper. Due to its nature and the fulfillment of the stated requirements, the proposed CEDARtools® workflow particularly enables the tackling of complex transient failures in integrated multi-core systems in a new fashion.

Reducing post-release defects, of course, requires the full toolbox of debugging tools and methodology to reduce different types of malfunctions. Therefore, it is highly recommendable to plan early for mastering later-evolving software issues. For hardware trace based tools, the processors capability to produce hardware trace as well as access to the physical trace interface are the key factors. These requirements have to be considered during early project planning phase in order to deliver high-quality safety-critical systems.

References

[1] O. Burkacky, J. Deichmann, G. Doll, and C. Knochenhauer, "Rethinking car software and electronics architecture," Feb. 2018. [Online]. Available: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/rethinking-car-software-and-electronics-Architecture

[2] C. Jones and O. Bonsignour, *The Economics of Software Quality*. Addison-Wesley, 2011.

[3] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009. [Online]. Available: http://books.google.de/books?id=E5uIKu9Y8UEC

[4] F. Münz, "Sporadic failures Implications for Multicore," München, Nov. 2013. [Online]. Available: http://www.mad-workshop.de/slides/1_1.pdf

[5] J. Gray, "Why Do Computers Stop and What Can Be Done About It?" in *Symposium on Reliability in Distributed Software and Database Systems*, 1986, pp. 3–12.

[6] B. Hanke and F. Schulz, "Master Thesis: Assessment of multi-core integration infrastructure," Ph.D. dissertation, University of German Armed Forces Munich, Munich, Aug. 2014.

[7] G. Pothier and E. Tanter, "Back to the Future: Omniscient Debugging," *Software, IEEE*, vol. 26, no. 6, pp. 78–85, 2009.

[8] J. Engblom, "A review of reverse debugging," 2012, pp. 1–6.

[9] J. Thalheim, P. Bhatotia, and C. Fetzer, "INSPECTOR: Data Provenance Using Intel Processor Trace (PT)," 2016, pp. 25–34.

[10] "CoreSight™ architecture specification v2.0 ARM IHI 0029B," ARM Ltd., 2013.

[11] "Intel® 64 and IA-32 architectures software developer's manual," Intel Corporation, 2016.

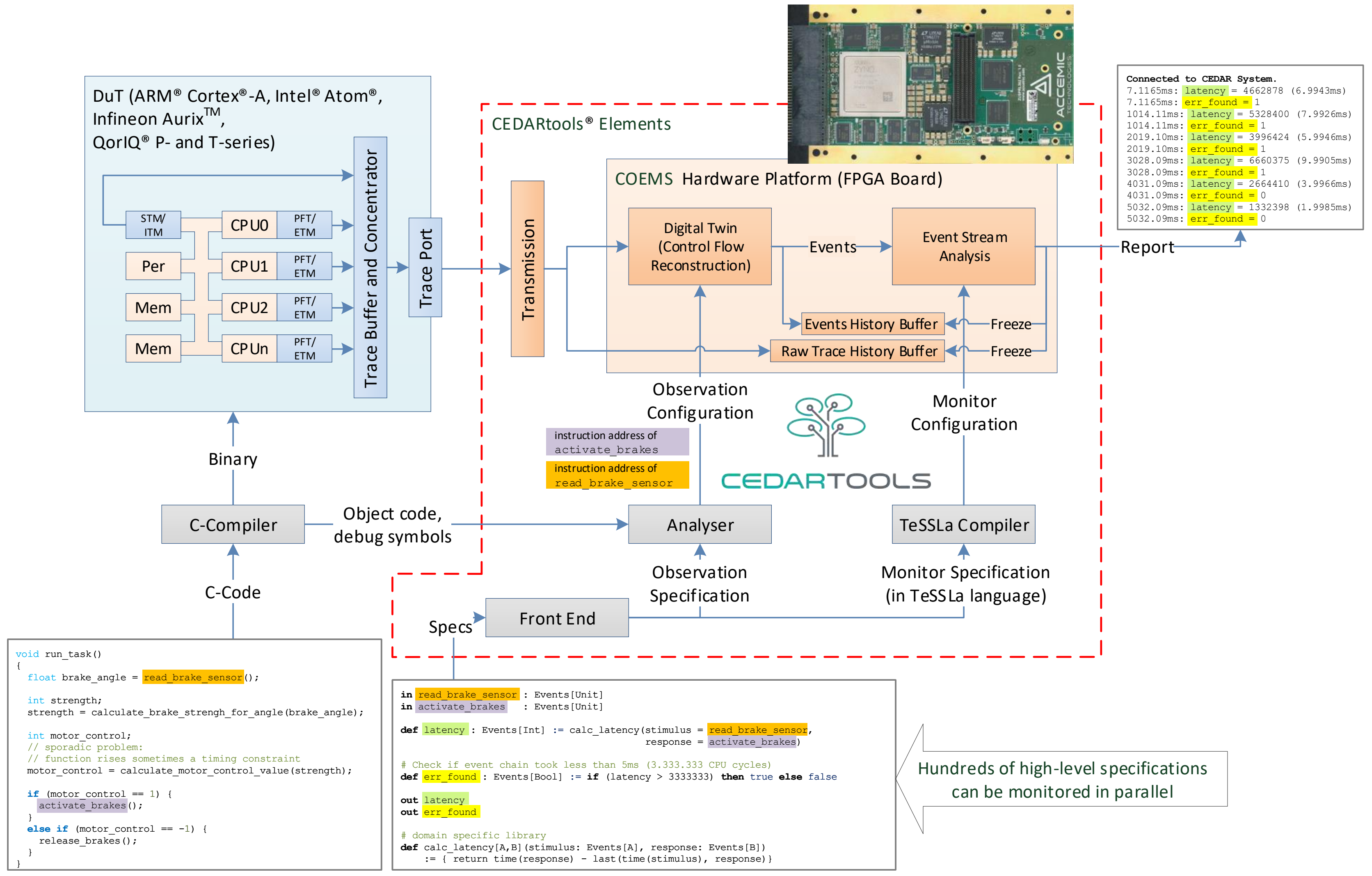
[12] "TC29/7/6/3x ED Emulation devices target specification," Infineon Technologies AG, 2013.

[13] "IEEE-ISTO 5001TM-2012: The Nexus 5001 forum - standard for a global embedded processor debug interface," IEEE-ISTO, Jun. 2012.

[14] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, and D. Thoma, "TeSSLa: Temporal Stream-Based Specification Language," in *Formal Methods: Foundations and Applications*, T. Massoni and M. R. Mousavi, Eds. Cham: Springer International Publishing, 2018, pp. 144–162.

[15] "AUTOSAR-TIMEX: Specification of timing extensions," AUTOSAR. [Online]. Available: http://www.autosar.org/

[16] "AMALTHEA - An open platform project for embedded multicore systems." [Online]. Available: http://www.amalthea-project.org/


DuT (ARM® Cortex®-A, Intel® Atom®, Infineon Aurix™, QorIQ® P- and T-series)
STM/ ITM
Per
Mem
Mem
CPU0
CPU1
CPU2
CPUn
PFT/ ETM
Trace Buffer and Concentrator
Trace Port
CEDARtools® Elements
Transmission
COEMS Hardware Platform (FPGA Board)
Digital Twin (Control Flow Reconstruction)
Events
Event Stream Analysis
Events History Buffer
Raw Trace History Buffer
Freeze
Freeze
Report
Observation Configuration
Monitor Configuration
instruction address of activate_brakes
instruction address of read_brake_sensor
CEDARTOOLS
Binary
C-Compiler
Object code, debug symbols
Analyser
TeSSLa Compiler
C-Code
Observation Specification
Monitor Specification (in TeSSLa language)
Front End
Specs
Hundreds of high-level specifications can be monitored in parallel

```
Connected to CEDAR System.
7.1165ms: latency = 4662878 (6.9943ms)
7.1165ms: err_found = 1
1014.11ms: latency = 5328400 (7.9926ms)
1014.11ms: err_found = 1
2019.10ms: latency = 3996424 (5.9946ms)
2019.10ms: err_found = 1
3028.09ms: latency = 6660375 (9.9905ms)
3028.09ms: err_found = 1
4031.09ms: latency = 2664410 (3.9966ms)
4031.09ms: err_found = 0
5032.09ms: latency = 1332398 (1.9985ms)
5032.09ms: err_found = 0
```

```
void run_task()
{
  float brake_angle = read_brake_sensor();

  int strength;
  strength = calculate_brake_strengh_for_angle(brake_angle);

  int motor_control;
  // sporadic problem:
  // function rises sometimes a timing constraint
  motor_control = calculate_motor_control_value(strength);

  if (motor_control == 1) {
    activate_brakes();
  }
  else if (motor_control == -1) {
    release_brakes();
  }
}
```

```
in read_brake_sensor : Events[Unit]
in activate_brakes   : Events[Unit]

def latency : Events[Int] := calc_latency(stimulus = read_brake_sensor,
                                          response = activate_brakes)

# Check if event chain took less than 5ms (3.333.333 CPU cycles)
def err_found : Events[Bool] := if (latency > 3333333) then true else false

out latency
out err_found

# domain specific library
def calc_latency[A,B](stimulus: Events[A], response: Events[B])
    := { return time(response) - last(time(stimulus), response)}
```

Fig. 8. CEDARtools® debugging approach on an exemplary C-Program illustrating the involved components on software and hardware side