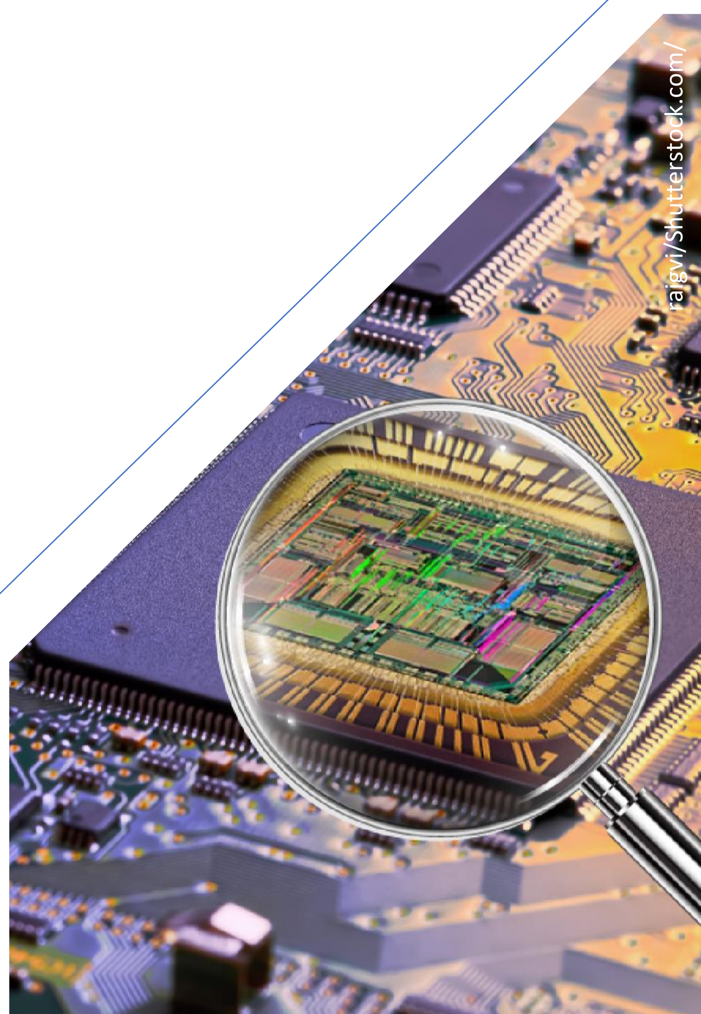
# Understanding and Fixing Complex Faults in Embedded Systems

## WHITEPAPER

Rev. 1.0 / 28 AUGUST 2020

This whitepaper briefly examines common types and the nature of software anomalies. It explains how mistakes lead to observable **anomalies** and how these are differentiated into **Bohrbugs** and **Mandelbugs** according to their reproducibility. The principle of "scientific debugging" is explained. It is shown that the comprehensive observability of a system is a key capability for **efficient debugging**. Subsequently, the advantages and limitations of various existing and novel monitoring solutions such as printf()-debugging, start/stop-debugging, omniscient debugging, runtime verification and the novel CEDARtools® approach are presented and discussed.




ACCEMIC
TECHNOLOGIES

## Something about Mistakes, Errors, Defects, Bugs, Faults and Anomalies

In everyday language (and in many publications), we use a number of words such as bug, fault, error, etc. inconsistently and confusingly to describe the malfunctioning of a software-based system. This also happens to the authors of this paper in their normal life unless they pay strict attention to their choice of words.

Therefore, we would like to start with a brief clarification. The IEEE have done a great job for this. The following terminology is based on the "IEEE Standard Classification for Software Anomalies" [1].

The source of all our problems is human imperfection. Everyone makes mistakes, even the most brilliant developer. If developers notice a mistake (or misunderstanding) themselves, it is called an **error**[1].

If the tester is the first to notice the anomaly, it is called a **defect**[2]. After confirmation by the developer, it becomes a **bug**.

Once the product is deployed and the end user is the first to find the system not working as expected, we have a **fault**[3].

The term "anomaly" may be used to refer to errors, defects, bugs as well as faults.

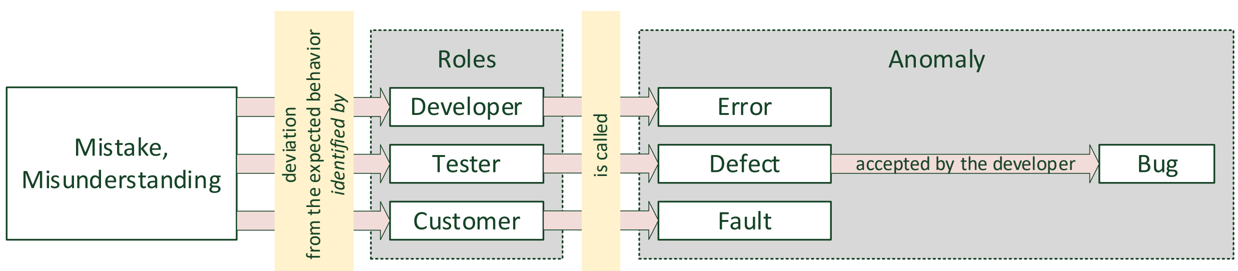



*Figure 1: Semantics of Mistakes, Errors, Defects, Bugs, Faults, Anomalies etc.*

## Reproducibility of Anomalies

For the engineer's ability to eliminate a bug or a fault (= debugging), the reproducibility of the anomaly is crucial. Therefore, this property is an essential classification criterion for anomalies.

A **deterministic** manifestation is the repeatable occurrence of an anomaly under a well-defined, but possibly unknown, set of conditions. Such a manifestation is also called a **Bohrbug**[4] (named after Bohr's deterministic atom model).

---

[1] Error: A human action that produces an incorrect result. [1]

[2] Defect: An imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced. [1]

[3] Fault: A manifestation of an error in software. [1]

[4] Note: The term "bug" is not in line with the terminology introduced above. In order to be consistent with the established terminology used in related work, we will also use the terms "Bohrbug", "Mandelbug" etc. instead of the more consequential terms "Bohr anomaly", "Mandel anomaly" etc.





ACCEMIC TECHNOLOGIES

If the underlying causes for an anomaly are so complex and obscure that it appears to be **non-deterministic**, we also speak of a **Mandelbug** (named after the chaotic Mandelbrot set). An example of a Mandelbug is the well-documented "unintended acceleration problem" [2].

The literature defines some subclasses of Mandelbugs:

- **Aging-Related Bugs** occur in long-running systems due to error conditions caused by the accumulation of problems such as memory leakage, propagating rounding errors or unreleased files and locks. A typical example for an aging-related bug is the software fault in the Patriot missile-defense system [3] .
- Anomalies that seem to disappear or alter their behavior when looked into are called **Heisenbugs** (named after the uncertainty principle described by the physicist Werner Heisenberg, which is often informally conflated with the probe effect).

It seems that the predominant class of faults should be complex Mandelbugs. Surprisingly, this is not the case: In practice, there is a surprisingly high proportion of Bohrbugs. An impressive example is given by Grottke et al. [4] who analyzed the software faults for 18 JPL/NASA space missions. Out of the detected 520 software faults, 61.4% were Bohrbugs, and 36.5% were Mandelbugs (4.4% of those were aging-related bugs).

Nonetheless, the complex Mandelbugs increasingly gain importance as more and more complex systems are being developed and non-deterministic fault patterns will occur more frequently due to parallelism and concurrency in multicore systems.

## The Anatomy of an Anomality

An illustration of the effects of a mistake is shown in Figure 2.
Our example system traverses through a sequence of states, $z_1$.. $z_6$, which are characterized by the internal variables, $i_1$ .. $i_4$, and the observable outputs, $e_1$ .. $e_2$. Each program execution step computes an update to the internal variables and the outputs to produce.
If the executed program contains a mistake, the resulting state might not be as expected. In this case, we speak of an "activated mistake" and a resulting "infected state" – an anomaly has manifested.

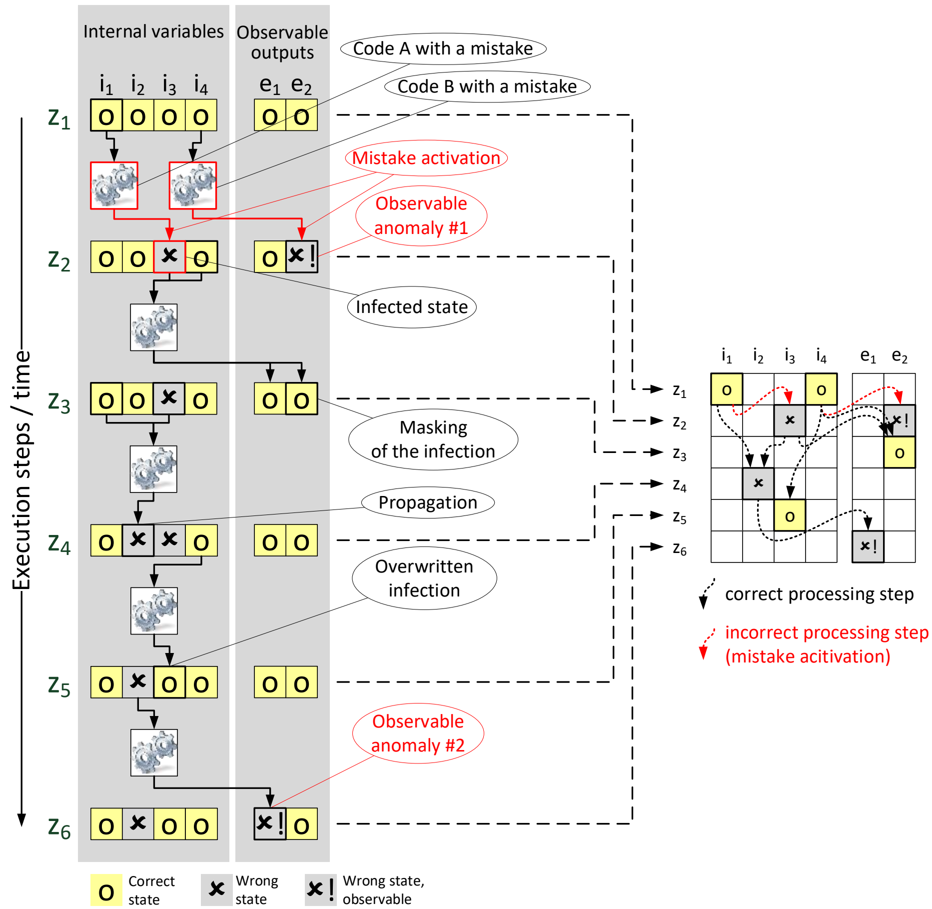



*Figure 2: A defective program execution as a succession of states (inspired by [4]).*
*Left-hand-side: extended depiction of states and transitions.*
*Right-hand-side: simplified depiction of the same scenario*

By the transition from state $z_1$ to $z_2$, two code segments with mistakes are executed. One of them (code A) causes a wrong state of the internal variable $i_3$, the other (code B) causes a wrong state of the observable output $e_2$. The latter is a textbook-like manifestation of a Bohrbug as long as the anomaly can be reproduced under a well-defined, but possibly unknown, set of conditions.

A typical Mandelbug scenario (caused by code A) is a malfunction that only changes an internal variable ($i_1$ .. $i_4$). This may not be detectable easily. Worse, the actual root infection can be regularly overwritten and masked, such as in state $z_5$, before it finally leads to an observable failure in state $z_6$. The path of propagating and overwriting infections can cause many headaches. In our example, when switching to state $z_4$, the wrong internal variable $i_3$ causes a wrong assignment of $i_2$. When $i_3$ is overwritten in $z_5$, the track record of this originally wrong variable is lost before the error is exposed in $z_6$. By this time, no indication remains to point to the software defect in switching from $z_1$ to $z_2$.

If the transitions are processed in a multicore system, there is an increased chance for a more chaotic manifestation of the resulting anomaly.

To avoid this nightmare scenario, comprehensive monitoring capabilities are essential.

## The Debugging Process

The process of understanding the underlying cause of an anomaly, i.e. the identification of the mistake, and fixing the problem is called debugging. Often carried out intuitively, this process always follows the same procedure depicted in Figure 3.

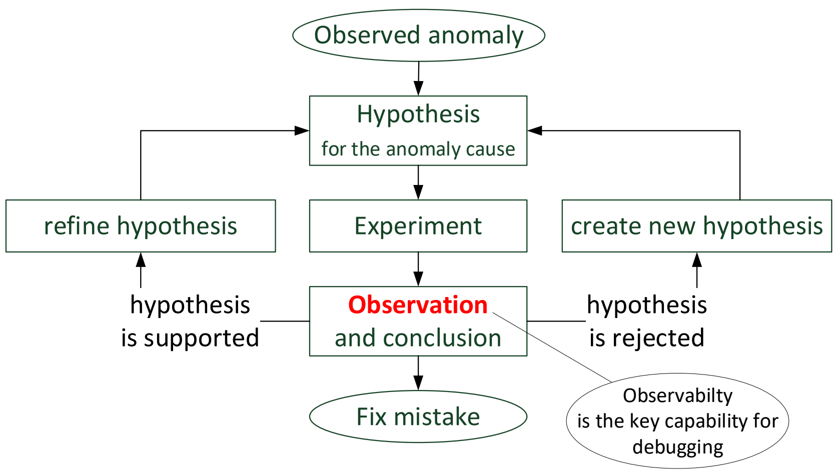



*Figure 3: "Scientific" Debug Process*

Starting from an observed anomaly, a testable theory (hypothesis) that narrows the space of possibilities for its cause is developed. The next step is to develop an experiment to test the hypothesis.

If the hypothesis is supported, either the detected mistake can be fixed or the hypothesis can be further refined. If the hypothesis was false, a new hypothesis has to be developed.

It is obvious that observability is a crucial factor for an efficient debugging process. In the following, we will discuss the currently used and novel observation methods that are so essential for the debugging process.

## The Observation Toolbox

### PRINTF() DEBUGGING (FIGURE 4.A)

Named after the printf() C function, this is the most basic form of observation in the debugging process. The source code is manually instrumented simply to write debug information to a console output. Unfortunately, this approach can have a massive impact on the time behavior of an application. It may even introduce unintended synchronization in concurrent programs when the same output console is shared. This is a perfect setting for Heisenbugs.

Besides, it is the least dynamic approach for obtaining an understanding of what is happening inside a program. If the information of interest changes, the software must be adapted and recompiled. With many iterations, this process can become very tedious and time-consuming. Even though this approach is archaic, it is still in use. In the worst case, it may threaten project goals if no other more advanced debug method is available.

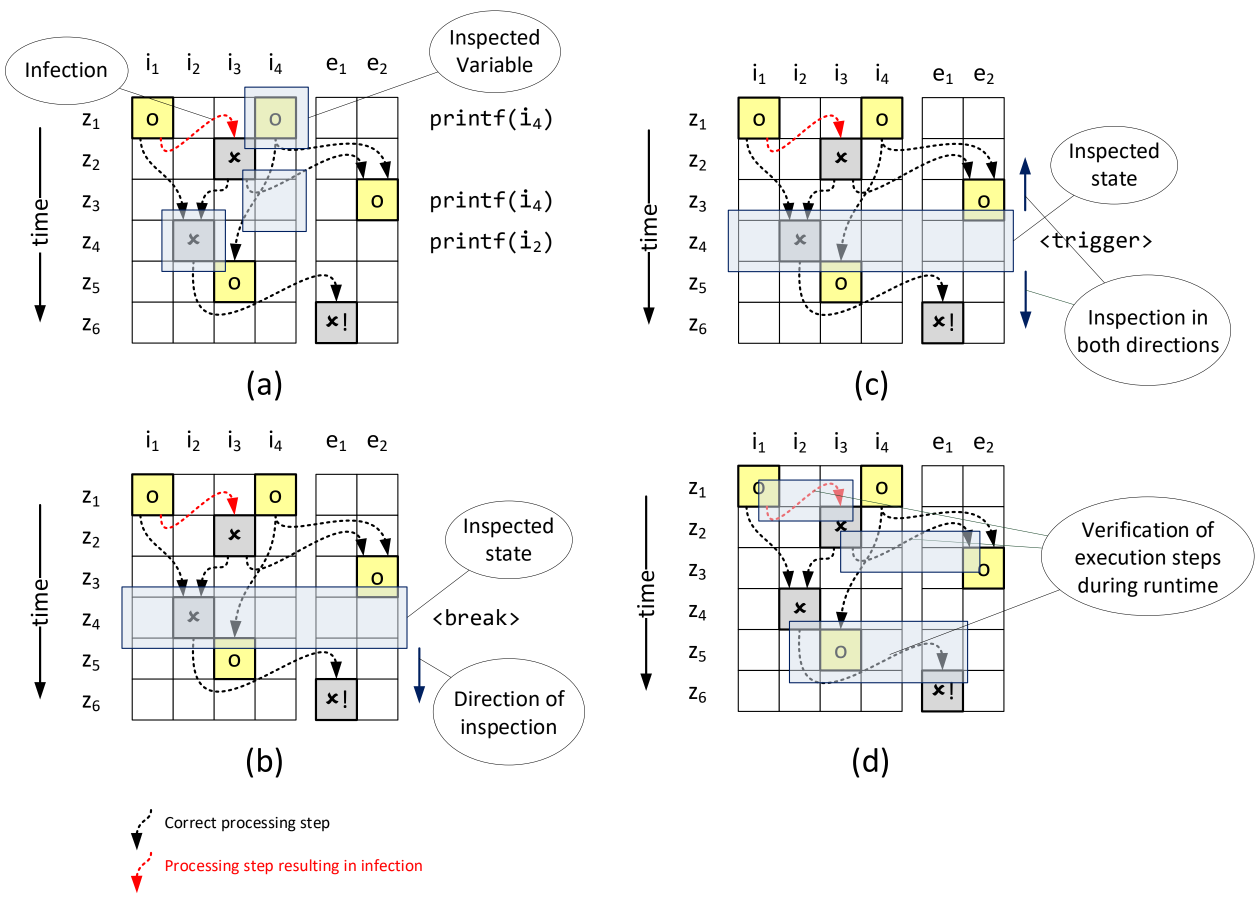



*Figure 4: Todays observation toolbox:*
*(a) printf() Debugging, (b) Start/Stop Debugging, (c) Omniscient Debugging, (d) Runtime Verification.*
*This illustration follows the representation introduced in Figure 2.*

## Start /Stop Debugging (Figure 4.b)

This very common observation 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 analyzing complex transient anomalies:

1) Directly controlling the execution progress of a program changes its timing behavior. This makes it hard to reproduce anomalies, for which timing matters. Besides, in cyber-physical systems where the software is controlling physical actuators, such as an engine, this approach might not even be applicable since halting the execution would cause physical damage.
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 anomaly. This typically leads to a need for rerunning the software over and over again in an effort to trace back the manifestation of an anomaly.
3) Due to the cyclic debugging fashion, the system behavior is required to be fundamentally deterministic to enable the observation and investigation of an anomaly. This is hard or impossible to achieve in parallel or in real-time programs with dynamic asynchronous input data.

## Omniscient Debugging (Figure 4.c)

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 to go back in time, which conforms to the natural way of searching the causes of observed anomalies. The recordings may be generated by code instrumentation or by using hardware trace data. The latter is generated by dedicated on-chip debug modules, e.g. Arm® CoreSight™ [5] or Intel® Processor Trace [6]. This trace recording can be controlled by a few simple triggers, which are usually not suitable for describing really complex conditions.

Omniscient debuggers without any limitations are an unreachable dream. In reality, there is not enough memory to store all the state required to inspect all system states retrospectively for any length of time. For embedded processors, there are systems available with a few Gigabytes of trace buffer. This results in a clip of a few seconds. If the anomaly occurs outside of it, it is bad luck.

## Runtime Verification (Figure 4.d)

Another strategy to immediately detect infected states is the runtime verification approach. Information from a running system is extracted and used to validate the system behavior and to detect the violation of pre-defined properties. If the transition from one state to another violates such a property, this violation is detected immediately. Precondition for this dynamic monitoring principle is the ability to observe the related state transitions. Usually, this is done by software instrumentation with its known limitations.

In summary, it can be said that none of the methods discussed above allows to find non-deterministically occurring anomalies reliably.

Fortunately, there is a bright spot: In the CEDARtools® solution, we combine the advantages of the Omniscient Debugger and the Runtime Verification approach (see Table 1).

Using a digital twin representation of the relevant behavior of the system-under-test, its state transitions are permanently monitored. In case of a violation / trigger, a ring buffer of raw trace data or a refined event stream is frozen -the clip exactly around the violation is thus available ("save on trigger”, Figure 5).

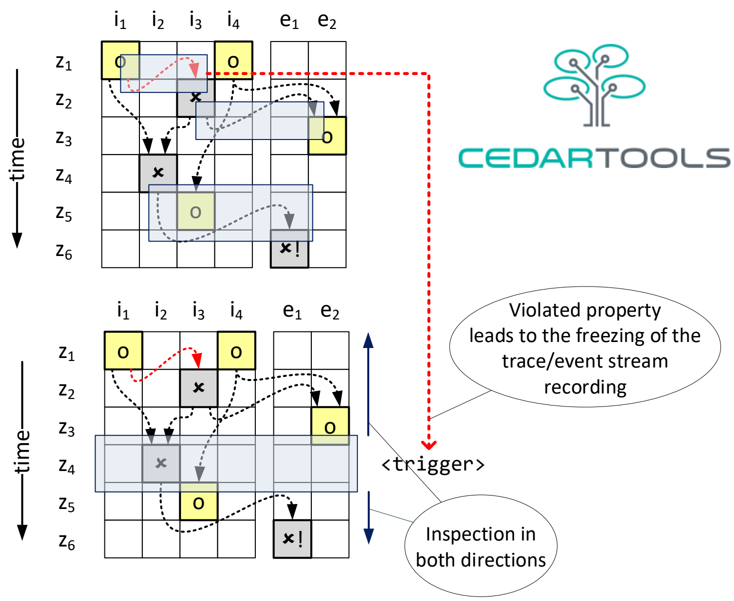



*Figure 5: The CEDARtools® approach, combing the best of the Omniscient Debugger and the Runtime Verification approach*

An in-detail explanation of the CEDARtools® solution and its technical background is available at [https://accemic.com/cedartools/](https://accemic.com/cedartools/).

| | printf() Debugging (Figure 4.a) | Start/Stop Debugging (Figure 4.b) | Runtime Verification (live, with software instrumentation) (Figure 4.d) | Omniscient Debugging (embedded trace based, Figure 4.c) | CEDARTOOLS |
|---|---|---|---|---|---|
| Non-instrusiveness<br>*(No change of the behavior of the system under test)* | No | | | Yes | |
| Decoupling from software<br>*(No software instrumentation)* | No | Yes | No | Yes | |
| Long observation period<br>*(Observation period should not be restricted)* | Limited | n.a. | Yes | No | Yes |
| Multiple focuses<br>*(Analyze multiple independent failures in parallel)* | Yes | | | No | Yes |
| Multi-core support<br>*(Tool should be able to debug parallel software running on multi-core processors)* | Limited | | | | Yes |
| Autonomous operation<br>*(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)* | Yes | No | Yes | No | Yes |
| Bandwidth<br>*(Monitorable events / s)* | some 1000 | some 0.01 | some 1000 | > 100 Mio | |

*Table 1: Comparison of debug techniques (green: good, n.a.: not applicable).*
*A further discussion of the comparison meanings can be found in Schulz et al. [7]*

# References


[1] 'IEEE Standard Classification for Software Anomalies', *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pp. 1–23, 2010.

[2] M. Barr, 'BOOKOUT V. TOYOTA - 2005 Camry L4 Software Analysis', 2013. http://www.safetyresearch.net/Library/BarrSlides_FINAL_SCRUBBED.pdf (accessed Jul. 02, 2014).

[3] 'Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia', Washington, D.C., Feb. 1992, [Online]. Available: https://www.gao.gov/products/IMTEC-92-26.

[4] M. Grottke, A. P. Nikora, and K. S. Trivedi, 'An empirical investigation of fault types in space mission system software', in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, 2010, pp. 447–456, doi: 10.1109/DSN.2010.5544284.

[5] *CoreSight™ Components - Technical Reference Manual - DDI 0314H*. ARM Limited, 2009.

[6] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2016.

[7] A. Schulz, F. Muenz, and Weiss, Alexander, 'Debugging Complex Failures of Real-Time Multi-Core Systems', presented at the Embedded World Conference, Nuremberg, Germany, Feb. 2020, [Online]. Available: https://accemic.com/publications/.

[8] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.

## About Accemic Technologies

Accemic Technologies, a German company with offices in Kiefersfelden (Munich Metropolitan Region) and Dresden (one of the most beautiful cities in the world), has developed CEDARtools® – a patented breakthrough technology for the dynamic analysis of dependable embedded systems.

We make software tests for embedded systems more effective and efficient. We simplify the debugging process and provide the necessary leverage to pin down the root causes of sporadic, non-deterministic anomalies.

Our new analysis method leverages the trace capabilities embedded into virtually all modern processors. Their trace units expose the details of the operation of the CPU and its peripherals to the outside. However, they easily produce a few GBit of trace data per second. This quickly renders approaches combining storage and offline analysis as infeasible options.

The live analysis of the execution trace at run time is a quantum leap enhancement over the offline analysis of recorded trace data as it effectively eliminates the bottlenecks imposed by the need for the intermediate buffering. CEDARtools® enables (a) the measurement of the control flow coverage during the execution of integration tests and system tests, as well as (b) the dynamic constraints monitoring.

We provide development and test engineers with the powerful tool that boosts their productivity by enabling them to monitor a system over large time frames and to pin down even sporadic errors quickly.

Talk to one of our experts today.

**Accemic Technologies GmbH**
Franz-Huber-Str. 39
83088 Kiefersfelden
Germany

+49 8033 6039790
cedartools@accemic.com
www.accemic.com