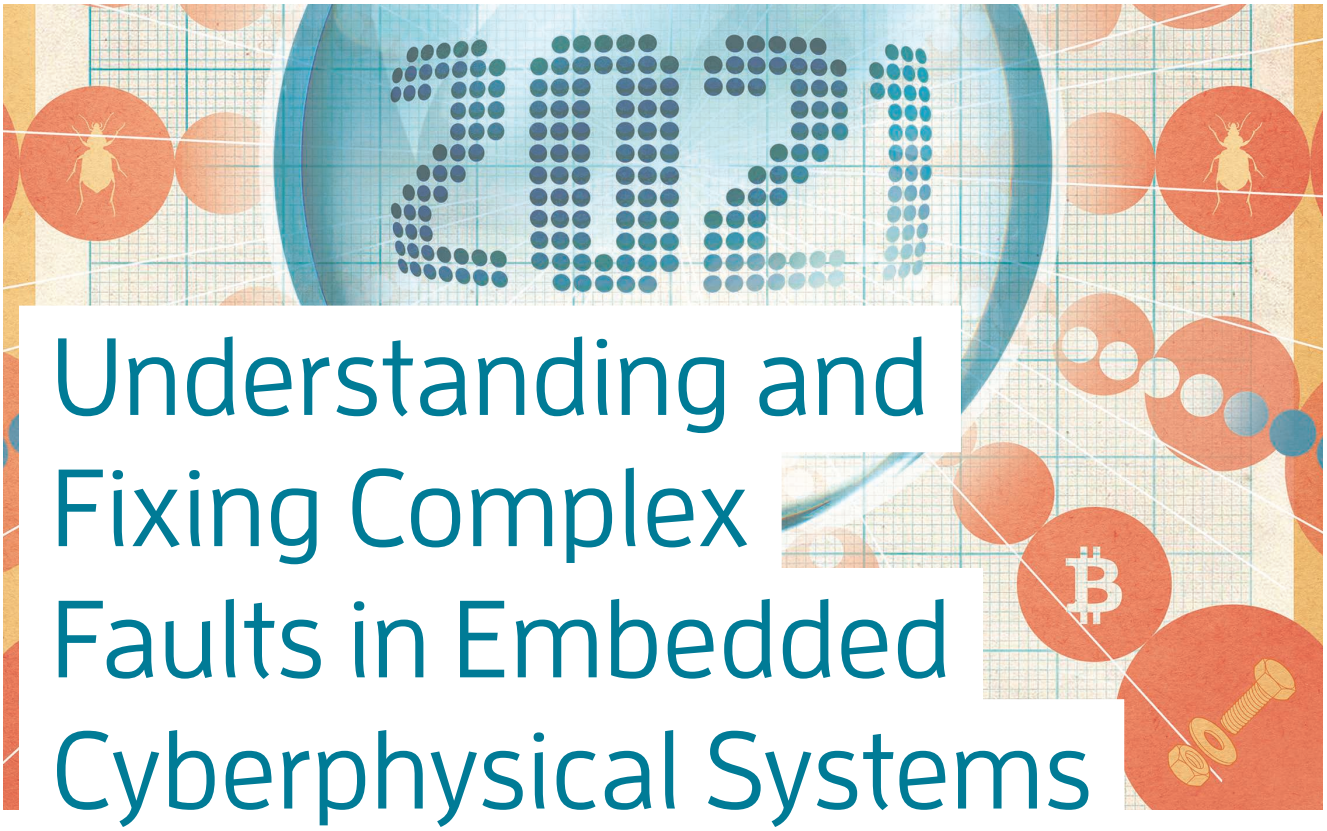

Understanding and Fixing Complex Faults in Embedded Cyberphysical Systems

Alexander Weiss, Accemic Technologies
Smitha Gautham, Athira Varma Jayakumar, and Carl R. Elks, Virginia Commonwealth University
D. Richard Kuhn and Raghu N. Kacker, National Institute of Standards and Technology
Thomas B. Preusser, Accemic Technologies

Embedded systems are becoming ubiquitous companions in all our lives. This article reviews the terminology and modern understanding of complex anomalies and state-of-the-art debugging. It details sophisticated omniscient debugging and runtime verification and describes a novel technique to combine the benefits of those processes.

Understanding fault types can lead to novel approaches to debugging and runtime verification. Dealing with complex faults, particularly in the challenging area of embedded systems, demands more powerful tools, which are now becoming available to engineers.

Digital Object Identifier 10.1109/MC.2020.3029975
Date of current version: 14 January 2021

MISTAKES, ERRORS, DEFECTS, BUGS, FAULTS, AND ANOMALIES

Embedded systems are everywhere, and they present unique challenges in verification and testing. The real-time nature of many embedded systems produces complex failure modes that are especially hard to detect and prevent. To avoid system failures, we need to understand the nature and common types of software anomalies. We also have to think about how mistakes lead to observable anomalies

and how those issues can be differentiated according to their reproducibility. Another key need for efficient fault detection is the comprehensive observability of a system. This kind of understanding leads to the principle of "scientific debugging."

In everyday language, we inconsistently and confusingly use a number of words, such as bug, fault, and error, to describe the malfunctioning of a software-based system. This also happens to the authors of this article unless they pay strict attention to their choice of words. Therefore, we would like to start with a brief clarification based on the terminology used by *IEEE Standard Classification for Software Anomalies*,[1] outlined in the following:

- If coders notice a mistake themselves, the finding is called an *error* ("a human action that produces an incorrect result"[1]).
- If a tester is the first to notice an anomaly ("a product does not meet its requirements"[1]), the discovery is called a *defect*. After confirmation by the developer, a defect becomes known as a *bug*.
- If an end user finds a problem ("manifestation of an error"[1]), we have what is referred to as a *fault*.

Figure 1 illustrates the semantics of anomalies. The term *anomaly* may be used to refer to errors, defects, bugs, and faults. It refers to something that deviates from what is expected or normal with respect to the required behavior, which is ideally defined by a system specification.

REPRODUCIBILITY OF ANOMALIES

For an engineer to eliminate a bug or a fault (debugging), reproducibility is crucial. This property is therefore an essential classification criterion for anomalies. A deterministic manifestation is the repeatable occurrence of an anomaly under a well-defined, but possibly not yet understood, set of conditions. Such a manifestation is also called a Bohrbug, named after Bohr's deterministic atom model. To be consistent with the terminology established in related work, we will use the terms Bohrbug, Mandelbug, and so forth instead of the more consistent but odd sounding wording Bohr anomaly, Mandel anomaly, and so on.

If the underlying causes of an anomaly are so complex and obscure that the anomaly appears to be nondeterministic, we speak of a Mandelbug (named after the chaotic Mandelbrot set). Race conditions seen in concurrent programs are common examples of Mandelbugs. The literature defines 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, and unreleased files and locks. A typical example of an aging-related bug is the software fault in the Patriot missile-defense system (see "Patriot's Fatal Flaw").
- Anomalies that seem to disappear or alter their behavior when they are investigated are called Heisenbugs, after the uncertainty principle described by the physicist Werner Heisenberg; the principle is often informally conflated with the probe effect.

It might seem that the predominant class of faults should be complex Mandelbugs. This is not the case. In practice, there is a surprisingly high proportion of Bohrbugs. An example is given by Grottke et al.,[3] who analyzed the software faults of 18 Jet Propulsion Laboratory/NASA space missions. Out of 520 software faults, 61.4% were clearly identified as Bohrbugs, while 36.5% were Mandelbugs (4.4% were age related). Nonetheless, Mandelbugs increasingly

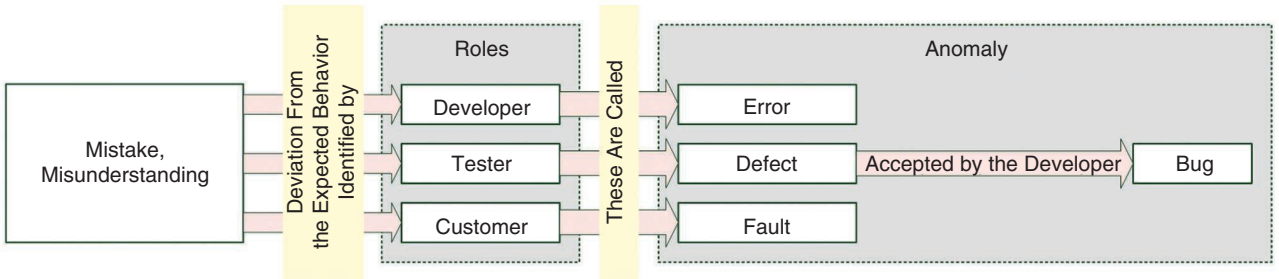

FIGURE 1. The semantics of mistakes, errors, defects, bugs, faults, and anomalies.

PATRIOT'S FATAL FLAW

On 25 February 1991, during the first Gulf War, a Patriot air defense system near Dhahran, Saudi Arabia, failed in its task of detecting and intercepting an Iraqi scud missile, which, unhindered, hit an American barracks, killing 28 soldiers and injuring roughly 100. The reason was an "aging-related bug" in the Patriot system's weapon control computer. An inaccuracy caused by rounding a floating-point variable continued to propagate. Through time, the system's target detection became so inaccurate that the trajectory of attacking missiles was wrongly judged as harmless.[4]

gain importance as more complex systems are developed, and seemingly nondeterministic fault patterns occur more frequently with concurrency and parallelism in multicore systems. For example, Ratliff et al.[5] showed that Mandelbugs tend to involve more interacting factors than the deterministic Bohrbugs, with roughly one additional indirect factor on average.

THE ANATOMY OF AN ANOMALY

The possible effects of a mistake are illustrated in Figure 2. Our example system traverses a sequence of states, $z_1, \ldots, z_6$, which are characterized by the internal variables $i_1, \ldots, i_4$. The system produces the observable outputs e_1 and e_2. Each program execution step computes an update to the internal variables and the outputs to be produced. 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 where an anomaly has manifested. By the transition from state z_1 to z_2, two code segments with mistakes are executed. One of them (code A) computes a wrong state of the internal variable i_3; the other (code B) produces a wrong observable output e_2. The latter is a textbook manifestation of a Bohrbug as long as the anomaly can be reproduced under a well-defined, but possibly not yet understood, set of conditions.

A typical Mandelbug scenario (as caused by code A) is an anomaly that changes only an internal variable. This may not be easily detectable. Worse, the actual root infection can potentially be overwritten and masked regularly. The path of propagating and overwriting infections can cause many headaches. In our example, during the transition 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 anomalies. To avoid this nightmare scenario, comprehensive monitoring capabilities are essential.

THE DEBUGGING PROCESS

The process of understanding the underlying cause of an anomaly—that is, the identification of the mistake—and fixing the problem is called debugging. Starting from an observed anomaly, a hypothesis (that is, a testing theory) that narrows the space of possibilities for the cause is developed. The next step is to develop an experiment to test this hypothesis. If the hypothesis is supported, either the detected mistake can be understood and fixed or the hypothesis can be further refined. If the hypothesis is false, a new one has to be developed. It is obvious that observability is a crucial factor for an efficient debugging process. The technology and tools to cope with complex faults in embedded systems are now becoming available.

OBSERVATION TOOLBOX

Printf() debugging

Named after the `printf()` C function, printf() debugging [Figure 3(a)] is the most basic form of observation in the debugging process. Source code is manually instrumented to output debugging

information. Unfortunately, this approach can have a massive impact on timing behavior, and it can even introduce unintended synchronization in concurrent programs when the same output console is shared. This creates a perfect setup for running into otherwise unrelated Heisenbugs. Additionally, if the information of interest changes, the software must be adapted and recompiled. Even though this approach is archaic, it is still in use. In the worst case,

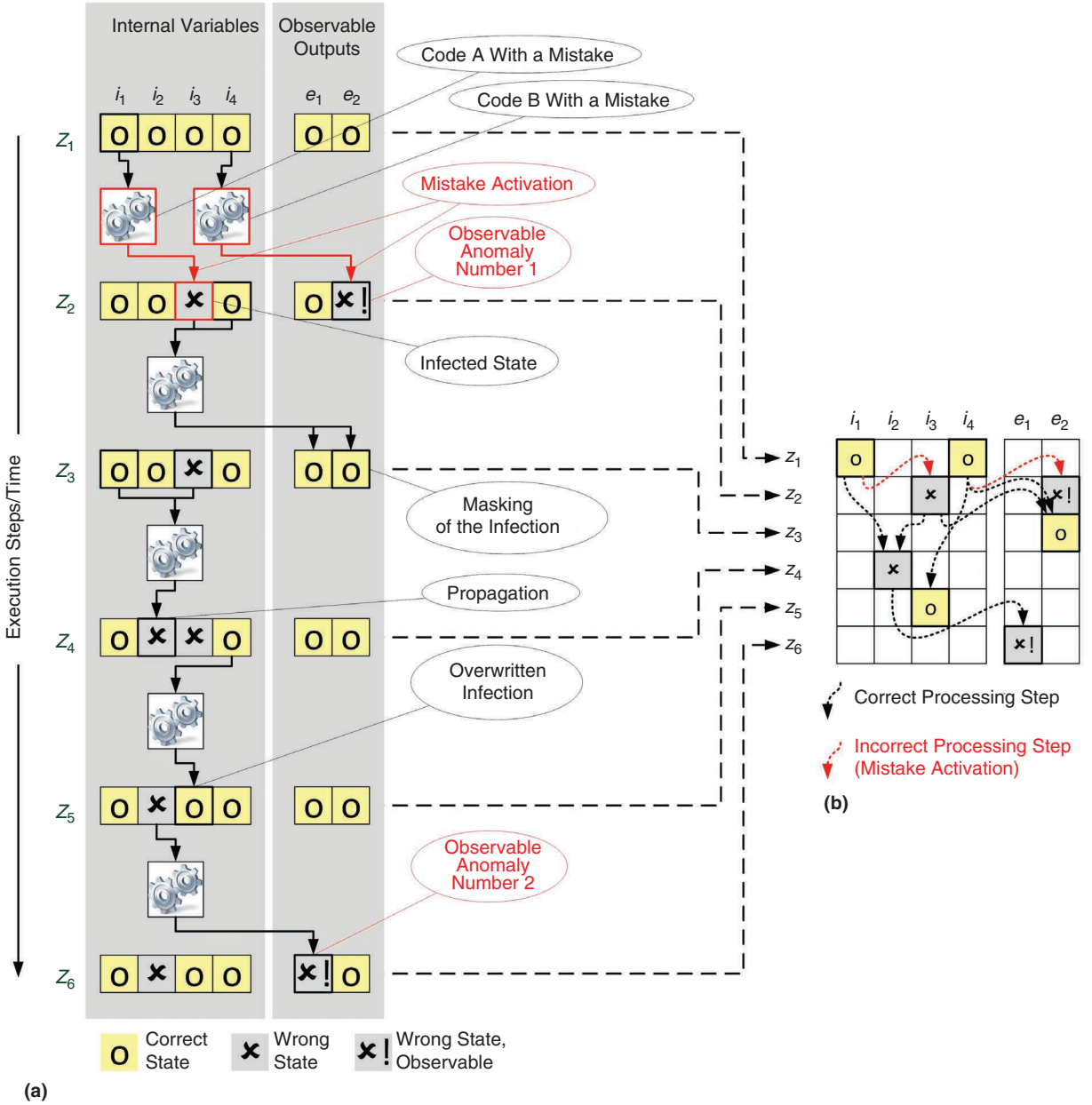

FIGURE 2. A defective program execution as a succession of states (inspired by Zeller et al.[2]). (a) An extended depiction of states and transitions. (b) A compacted depiction of the same scenario.

its tediousness and time-consuming nature may threaten project goals if no other more advanced debugging method is available.

Start/stop debugging

This common approach [see Figure 3(b)] is based on the direct control of a program's execution. When the execution is halted at selected breakpoints, the reached program state may be inspected and analyzed in detail. However, this approach has major drawbacks for

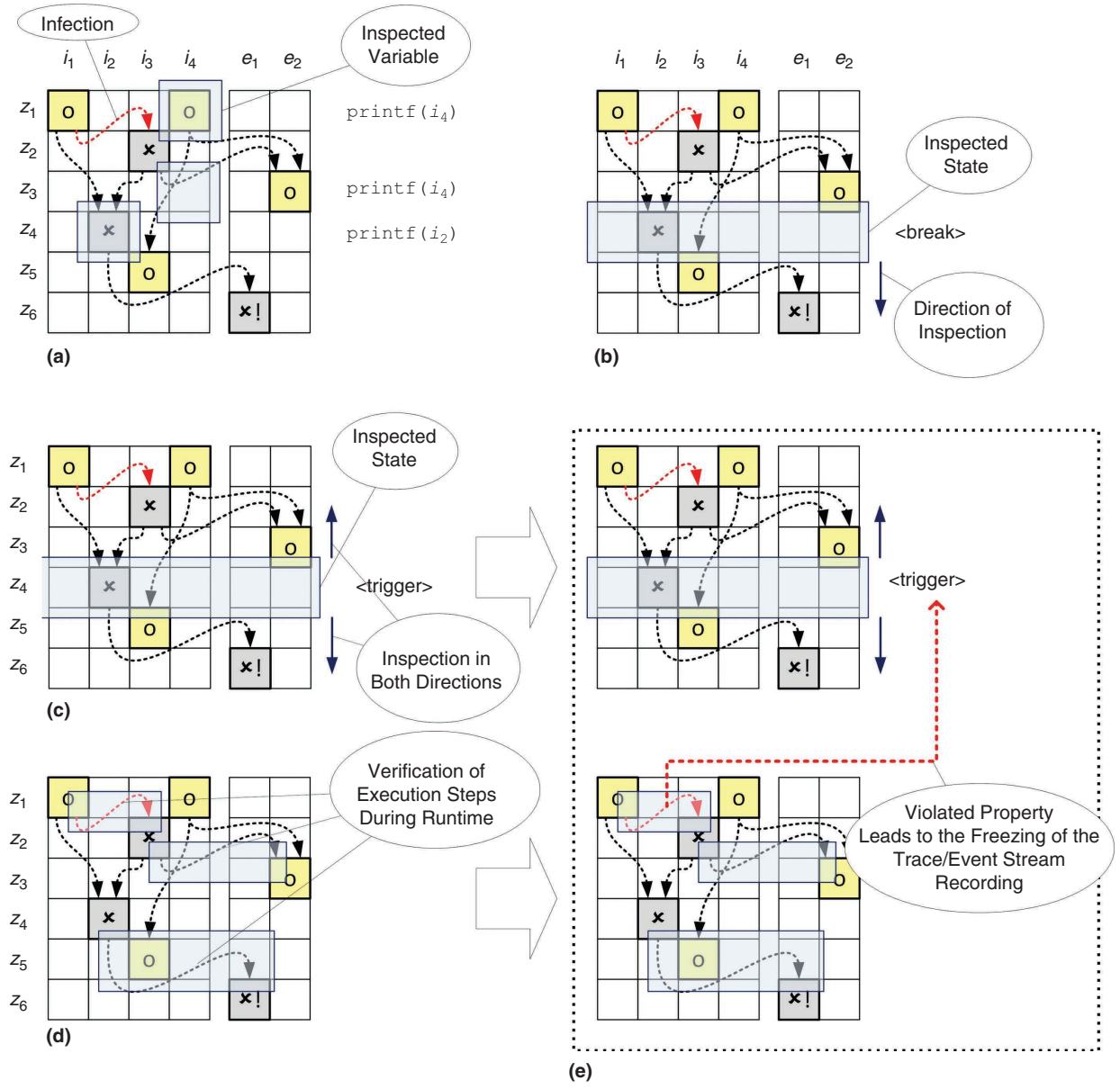

FIGURE 3. (a) Printf() debugging. (b) Start/stop debugging. (c) Omniscient debugging. (d) Runtime verification. (e) A novel approach combining the best of the omniscient debugger and the runtime verification approaches. This illustration follows the representation introduced in Figure 2.

analyzing complex transient anomalies. The drawbacks consist of the following:

1. Directly controlling the execution progress of a program changes the timing behavior. This makes it hard to reproduce anomalies whose manifestation depends on timing. In cyberphysical systems, where the software is controlling physical actuators, halting the control could even cause physical damage.
2. The method typically facilitates only stepping forward. Breakpoints have to be chosen thoughtfully to be early enough to reflect the root cause of an observed anomaly. This typically leads to running software over and over again in an effort to trace back the original 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 and real-time programs with dynamic asynchronous input data.

Omniscient debugging

These debuggers are also known as back-in-time and reversible debuggers [Figure 3(c)]. They record all or parts of a program's execution. This enables going back in time by reconstructing the execution history and corresponding program contexts. This approach most naturally aligns with tracing back the root cause of an anomaly, starting from an unexpected observation. The challenge posed by omniscient debugging is in capturing the execution trace. Software instrumentation is an option. Due to the technique's behavioral feedback into the monitored application, engineers face a delicate dilemma of having to trade off tracing detail against the faithful behavioral representation of a program. Compromises made by performing more coarse-grain tracing, as on the level of function calls, are common. Again, within critical control environments and with hardware in the loop, even such observation compromises are often not tolerable.

The key technology to enable truly nonintrusive yet detailed omniscient debugging is the embedded trace. Dedicated on-chip modules directly integrated into the monitored CPU capture, encode, and emit the execution trace. Arm CoreSight[6] and Intel Processor Trace[7] are prominent implementations of this technology. Embedded trace implementations rely on aggressive, sophisticated trace data compression. A naive encoding of only the addresses of the instructions executed by a 1-GHz CPU coming from a 32-bit address space would produce 32 Gb/s of data. Economically feasible implementations reduce this bandwidth demand to 1 Gb/s and lower. This is achieved by encoding only diversions from the default sequential control flow and pruning all information that can be inferred from the executed application binary. Other, often optional, information channels may add to the actual bandwidth demand. Providing timing information and a data trace[8] is particularly challenging in this respect.

Omniscient debugging, in practice, is severely challenged by the trace data volume and limited by the storage capacity. Two capture paths are common.

- *System memory*: This option competes over memory bandwidth and space within the monitored system, so it impacts the system's performance and timing behavior and is limited to short observation time spans.
- *External capture*: An external tracing device captures trace data from a designated interface for later offline processing. While this avoids possibly disruptive feedback into the monitored system, implementations are limited by the buffer capacity. In practice, trace clips of up to a few seconds are possible.

Since both these state-of-the-art approaches rely on buffering, they impose similar analytic limitations on the back-end trace interpretation. First, the escalation of a root cause into an observed anomaly must fit the trace window to avoid a multirun reconstruction. Also, the anomaly must be reliably and predictably reproducible to facilitate its capture by a quick trace snapshot. The only assistance natively offered by the various embedded trace architectures consists of a few primitive triggers that enable filtering the trace at its source. The triggers are too simple to capture complex interrelated conditions and too few to facilitate a defense against multiple hypothetical escalation paths.

Runtime verification

One promising technique that enables the immediate detection of infected states is runtime verification,[9,10] a dynamic testing technique that is often called lightweight formal verification

[Figure 3(d)]. It sits between the traditional dynamic testing methods discussed in previous sections and formal mathematical proof techniques. Unlike a proof, it is not complete and exhaustive, but it provides more guarantees of correctness than debugging-style testing. In general, runtime verification makes use of a monitor that observes the execution behavior of a target system.

To verify correctness, a monitor uses a specification of acceptable behavior, which is often derived from natural language requirements and translated into temporal logic formulas to enable reasoning across time-sensitive sequences and states. These formulas reside in the monitor. Execution trace information (that is, states, function variables, and decision predicates) is extracted directly from the target system and forwarded to the monitor, where the temporal logic is elaborated with the trace data for an on-the-fly validation of the system behavior, as shown in Figure 3(d). As such, runtime verification can immediately detect violations of predefined properties. Runtime verification extends debugging and testing by ensuring that important system properties are continuously checked during operational phases. It can be used to detect Bohrbugs as well as Mandelbugs that may be masked, as seen in the scenario in Figure 2. Detecting Mandelbugs through runtime verification is especially beneficial, as these bugs can elude development phase testing.

That said, a key concern related to runtime verification is the observability of the system operation at time intervals of interest. This is often attained by software instrumentation, which has limitations as described previously. Recently, new approaches for minimally intrusive runtime verification have produced significant capabilities to confirm and test very complex program behavior, as discussed in the following.

Runtime verification for testing

We often see testing complemented by runtime verification, as the two provide an excellent means to detect complex faults (see Falzon and Pace[11] and Colombo[12]). In particular, model-based designs provide a good framework to exploit synergies between testing and runtime verification. Additionally, runtime monitors can act as test oracles, ensuring that critical properties hold true during evaluation. Runtime verification is supported by testing frameworks, such as MathWorks Simulink and ModelJUnit, a model-based testing framework written in Java.[12]

> **WE OFTEN SEE TESTING COMPLEMENTED BY RUNTIME VERIFICATION, AS THE TWO PROVIDE AN EXCELLENT MEANS TO DETECT COMPLEX FAULTS.**

Runtime verification languages need more expressiveness to specify complex properties that emerge from multithreaded interactions in complex embedded systems. Signal temporal logic, event calculus, and metric temporal logic[13] are some of the runtime verification languages able to express specifications for monitoring complex embedded systems and cyberphysical systems. Another recent approach is stream-based runtime verification, which combines event processing and runtime verification. An example of a runtime verification tool that performs stream-based runtime verification is the Temporal Stream-Based Specification Language (TeSSLa),[14] which is designed for specifying properties where timing and sequencing is critical. It supports time-stamped events and a declarative programming style, which is well suited for expressing specifications. Furthermore, runtime verification tools, such as TeSSLa and Copilot, enable the automatic synthesis of executable monitor code (which is often realized in C or hardware description languages), directly from the runtime verification language.[13, 14]

As we stated in the "Observation Toolbox" section, most microprocessors today provide embedded trace in some form to support either start/stop or omniscient debugging. These embedded trace features (sometimes called on-chip debugging cores) can significantly reduce the intrusiveness of software instrumentation to support runtime verification. For example, the Arm CoreSight architecture[6] features an instrumentation trace macrocell (ITM) that can be used to output custom trace messages with minimal intrusion. Compared to traditional printf() debugging, which takes several milliseconds of CPU time, ITM takes very few clock cycles (hundreds of nanoseconds) to emit

custom trace data. However, these embedded trace cores are not straightforward to interface with and program for software instrumentation purposes. Therefore, they remain a challenge for certain applications.

Novel approaches

The approaches and methods discussed previously are established in practice but largely separate from one another. To make substantial gains in testing efficiency and reliability, we need novel methods that are both nonintrusive and more integrative [Figure 3(e)]. The great practical challenge of omniscient debugging is the limited size of the trace buffer memory combined with the general problem of identifying the manifestation of anomalies within the trace data stream. The trigger logic built into a processor's execution trace infrastructure is too simple to validate meaningful higher-level behavioral execution models. It takes designated runtime verification to deliver precise and complex triggering.

The key enabler of such a capable runtime verification is the instantaneous full behavioral disclosure of the system operation. This demands that captured execution trace data are not stored away but decompressed and interpreted on the fly. The decoded information can be leveraged to keep an in-sync digital twin that models the relevant behavior of the observed system. Violations of behavioral constraints are, then, reliable triggers for freezing the recording of raw execution trace data and refined higher-level events in a ring buffer. The captured trace clip precisely describes the pathway taken by the system toward this violation ["save on trigger" in Figure 3(e)].

Decompressing and interpreting the processor execution trace online and at the rate of its emission is technically challenging. Field-programmable gate arrays (FPGAs), which are user-programmable integrated circuits that enable the implementation of custom circuitry on off-the-shelf chips, are well suited for such applications. For example, the CEDARtools platform[15] utilizes large FPGAs, enabling tremendous parallelism to manage such online decompression and interpretation of traces. Spatially designated processing modules decode the trace data stream into messages, resolve the messages against a model of the executed binary, extract a stream of relevant higher-level events from the reconstructed control flow, and subject this event stream to a constraint's validation. The latter is achieved by an array of dataflow processors that are programmed in the TeSSLa specification language by using temporal logic. Violations of the specified constraints freeze the relevant most recent trace and event history for in-depth analysis by an engineer.

Use case: Smart sensor for nuclear power applications

The authors recently conducted a study of applying a software systematic testing process on an embedded smart sensor device that is representative of the type found in nuclear power generation applications.[16] Nuclear power generation facilities worldwide are steadily trending toward "aging infrastructure," with the average age of a light water reactor in the United States reaching approximately 37 years. The need to modernize these generation facilities with software-based instrumentation and control systems and smart sensors is high priority for the industry. These systems have to be tested and validated to very high levels of safety assurance (International Electrotechnical Commission Standard 61508-SIL 4) to ensure as reasonably possible that no software faults exist.

The Virginia Commonwealth University smart sensor is a real-time embedded device used in a cyberphysical systems context: monitoring the physical states of a nuclear reactor. The smart sensor hosts a real-time, lightweight operating system that executes several concurrent threads associated with sensing the barometric pressure and the temperature and maintaining the device health (see Appendix B in Elks at al.[17]). The systematic testing methodology we employed is called *pseudoexhaustive testing*, which is built around T-way combinatorial testing, partitioning, boundary value analysis, and modified-condition/decision-coverage path analysis.[18] We applied the systematic testing of the software following the well-known unit test, integration test, and system test paradigm.

A testbed architecture was built using state-of-the-art test automation tools that are necessary to conduct real-time systematic testing experiments on the smart sensor.[17] The three major tools employed in our testbed are 1) TESSY, 2) the National Institute of Standards and Technology (NIST) Automated Combinatorial Testing for Software tool, and 3) Keil Interactive Debugger. TESSY, developed by Razorcat, is an automated testing tool for safety-critical embedded systems software.[19]

Among the findings from this study, which reinforce the observation toolbox discussed earlier, is that analyzing test failures to find an underlying cause can be a significant effort, depending on the nature of the bug.

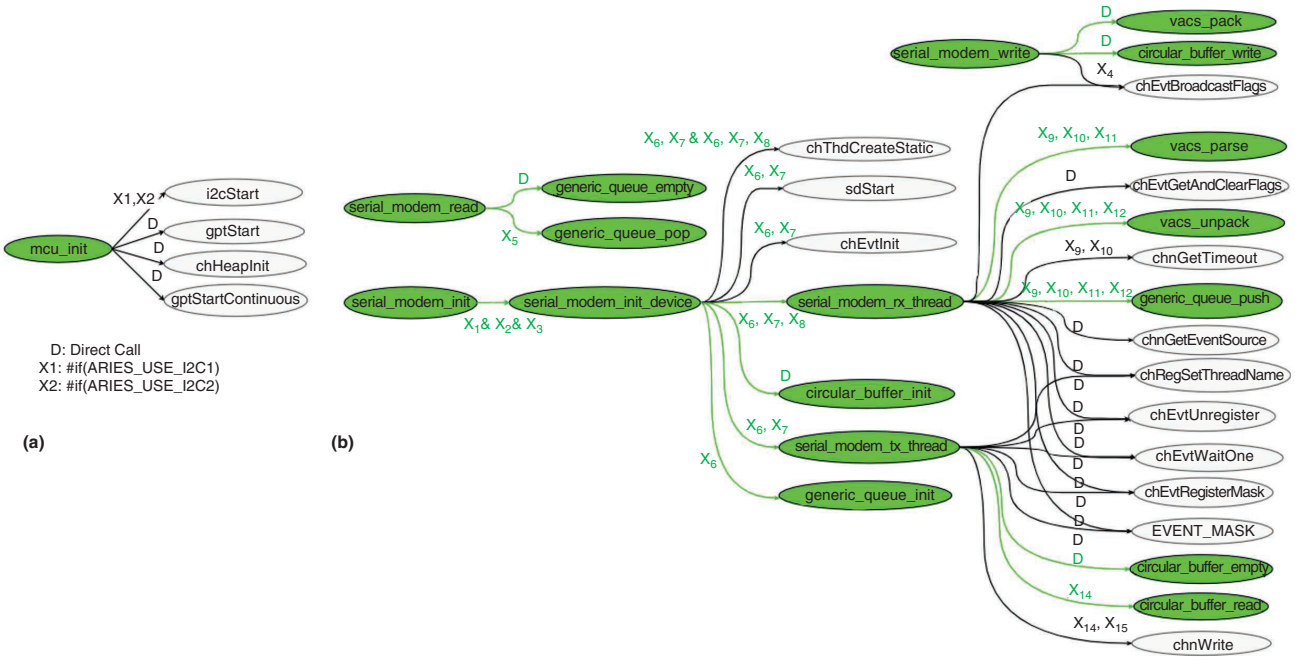

FIGURE 4. (a) A simple function call graph of "mcu_init." (b) A complex function call graph of the "serial_modem" thread.

Detecting failures at the interface level is a preferable starting point, but finding causality and fixing a bug requires the circumspection of the internal software behaviors.[9] Analyzing test failures is contingent on a number of factors, but, in our experience, three factors are significant—observability, and complexity—with respect to target system software execution. Observability is the ability to witness software state and interaction conditions at the temporal resolution needed to perceive anomalous behavior. As software becomes more complex and multithreaded, demands on the observability of executing code become more critical.

Complexity is not a goal of software engineering; it's a consequence of the way processor technology has evolved to become more computationally powerful. Many of the challenges we encountered while testing the smart-sensor-embedded software were directly related to the complexity of software interactions among software threads. This convolution is evident to anyone who has developed or tried to read the labeled control flow and dataflow graphs of multithreaded software artifacts. The number of control and dataflow paths determines the number of evaluations required to get complete test execution coverage of software units.

Figure 4(a) illustrates a simple example of a labeled function call graph from the smart sensor software with the edge labels defined. Each node in this directed graph denotes unique functions within the software, and the edges within the graph are unidirectional in nature, indicating a function at the edge head being called by another function at the edge tail. There are direct/unconditional function calls and function calls that are conditionally made based on single and multiple conditions. The edge between the functions "mcu_init" and "i2cStart" is labeled with conditions "X1, X2," denoting that the function "mcu_init" calls the function "i2cStart" when either the "ARIES_USE_I2C1" or "ARIES_USE_I2C2" macros are defined, that is, when any of the two inter-integrated circuit (I2C) channels' usage is enabled in the software. The graph also indicates that the function "mcu_init" unconditionally calls the functions "gptStart," "chHeapInit," and "gptStartContinuous," as the edges between these functions are labeled "D," which stands for direct call.

As an example of the complexity that can arise with simple multithreaded

applications, Figure 4(b) is a complete call graph of one of the several "serial_modem" threads within the multithreaded smart sensor software. The comprehension, analysis, and testing of software with complex call graphs becomes extremely challenging, as the dataflow and control flow couplings are difficult to track, even with the aid of advanced debuggers, such as Interactive Disassembler Pro.[20]

Real-time embedded systems need to be verified in both the temporal and value domains. Testing these function call graph execution sequences for all valid dataflows ended up being more complex and challenging than combinatorial testing, as it involved understanding and handling the periodicity and order of invoking tasks via the real-time operating system scheduler. To accurately emulate the function call order as per the call graphs, software testers needed to develop a detailed knowledge of the function call and control flow sequencing. Debugging test failures identified during sequence testing is very challenging since it involves stepping across multiple functions; watching internal variables, outputs, and arguments passed across multiple functions; and verifying the order of external function calls.

Our study revealed a number of bugs that had remained latent in the code for years. We found that combinatorial T-way testing helped quickly detect and identify a few Bohrbugs (only two-way interaction was needed). Those Bohrbugs were easily reproducible and repeatable, with multiple tests resulting in "infinity" and "not-a-number" outcomes. A harder bug to detect and identify was a Heisenbug scenario that appears rarely in situations related to hardware faults and cyberattacks that lead to an invalid serial bus (for example, the I2C serial bus) input. This bug was caught during combinatorial testing with corner case inputs. This buffer overflow, which is a Heisenbug, is caused by two colluding factors: the usage of a longer I2C input buffer size and a missing bounds check on received data. Software anomalies that occur due to the combination of multiple root causes in the software are more difficult to track down with traditional testing and debugging methods.

Last, a Mandelbug that was uncovered during integration testing was the most challenging to detect and analyze. In this case, a deadlock situation arose when two software components that interacted with each other were not allowed to proceed, as they were both waiting for inputs from each other. These kinds of anomalies are seen in multithreaded software with complex state machine interactions such that the threads/functions progress their states based on inputs from other components (both hardware and software). The root cause of these kinds of deadlock behaviors can be very difficult to observe, detect, and analyze because it requires carefully tracing back the entire history of state progressions within the software components under the given input conditions.

This use case concerned a modest software-based device—a smart sensor—and it presented challenges for the testers. We reasonably conclude that, with the proliferation of advanced high-performance embedded systems (for example, multicore processors and heterogeneous processors), there is a strong need for powerful approaches and supporting tools to manage the complexity of embedded software testing.

Embedded systems are becoming much more common in daily life, and better ways of finding and preventing failures are essential. The complexity posed by cyberphysical systems presents grand challenges to testing and verification. The state of practice for embedded software is at a point where new methods and novel techniques are needed to adequately test these critical systems. Advancements in understanding the nature of complex faults, and applying this understanding in maturing testing and verification, make it possible to build embedded cyberphysical systems that are safe and secure.

ACKNOWLEDGMENTS

This research was supported, in part, by the U.S. Department of Energy Idaho National Laboratory through the Light Water Reactor Sustainability program, under contract FP 217984; the European Union's Horizon 2020 research and innovation program, under grant 732016; and Germany's Federal Ministry of Education and Research, through KMU Innovative project Cocos (project 01IS19044). Any mention of commercial products in this article is for information only; it does not imply a recommendation or an endorsement by NIST.

REFERENCES

1. *IEEE Standard Classification for Software Anomalies*, IEEE Standard 1044-2009, 2010.
2. A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Burlington, MA: Morgan Kaufmann, 2009.
3. M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," in *Proc. 2010 IEEE/IFIP Int. Conf. Dependable Syst. Netw.*

ABOUT THE AUTHORS

ALEXANDER WEISS is the co-founder and chief executive officer of Accemic Technologies, Dresden, Germany. His research interests include cyberphysical systems runtime analysis, software verification, functional safety, and cyberattack detection. Weiss received his Dr.-Ing. in computer science from TU Dresden. Contact him at aweiss@accemic.com.

SMITHA GAUTHAM is a Ph.D. student in the Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, Virginia, USA. Her research interests include the design and assessment of heterogeneous runtime verification architectures, runtime safety and security monitors for cyberphysical systems, and model-based design assurance and verification. Gautham received her M.S. in electrical engineering from Virginia Commonwealth University. Contact her at gauthamsm@vcu.edu.

ATHIRA VARMA JAYAKUMAR is a research assistant in the Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, Virginia, USA. Her research interests include the dependability and security of cyberphysical systems, fault and cyberattack injection, model-based design assurance and verification, and the validation of embedded software. Jayakumar received her M.S. in computer engineering from Virginia Commonwealth University. Contact her at jayakumarav@vcu.edu.

CARL R. ELKS is an associate professor in the Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, Virginia, USA. His research interests include the analysis, design, and assessment of dependable embedded cyberphysical systems of the type found in critical infrastructure. Elks received his Ph.D. in electrical engineering from the University of Virginia in 2005. He is a Member of IEEE. Contact him at crelks@vcu.edu.

D. RICHARD KUHN is a computer scientist in the National Institute of Standards and Technology Computer Security Division, Gaithersburg, Maryland, USA. His research interests include combinatorial methods for software verification and applications to autonomous systems as well as empirical studies of software failure. Kuhn received his M.S. degree in computer science from the University of Maryland, College Park. He is a Fellow of IEEE. Contact him at kuhn@nist.gov.

RAGHU N. KACKER is a mathematical statistician in the National Institute of Standards and Technology Applied and Computational Mathematics Division, Gaithersburg, Maryland, USA. His research interests include combinatorial methods for software testing, artificial intelligence testing, and autonomous systems. Kacker received his Ph.D. degree in statistics from Iowa State University. Contact him at raghu.kacker@nist.gov.

THOMAS B. PREUSSER leads the research team at Accemic Technologies, Dresden, Germany. His research interests include massively parallel compute acceleration and data processing in heterogeneous FPGA-based systems as well as computer arithmetic. Preusser received his Ph.D. (Dr.-Ing.) in computer engineering from TU Dresden. Contact him at tpreusser@accemic.com.

(DSN), pp. 447–456. doi: 10.1109/DSN.2010.5544284.

4. "Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia," Government Accountability Office, Washington, D.C., Feb. 1992. [Online]. Available: https://www.gao.gov/products/IMTEC-92-26
5. Z. B. Ratliff, D. R. Kuhn, R. N. Kacker, Y. Lei, and K. S. Trivedi, "The relationship between software bug type and number of factors involved in failures," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshop (ISSREW)*, Oct. 2016, pp. 119–124. doi: 10.1109/ISSREW.2016.26.
6. "CoreSightTM architecture specification v2.0," ARM Limited, Cambridge, U.K., ARM IHI 0029B, 2013.
7. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, Santa Clara, CA, 2016.
8. N. Decker et al., "Online analysis of debug trace data for embedded systems," in *Proc. 2018 Des., Automat. Test Eur. Conf. Exhib. (DATE)*, pp. 851–856. doi: 10.23919/DATE.2018.8342124.
9. J. B. Michael, G. W. Dinolt, and D. Drusinsky, "Open questions in formal methods," *Computer*, vol. 53, no. 5, pp. 81–84, May 2020. doi: 10.1109/MC.2020.2978567.
10. M. Leucker and C. Schallhart, "A brief account of runtime verification," *J. Log. Algebr. Program.*, vol. 78, no. 5, pp. 293–303, 2009. doi: 10.1016/j.jlap.2008.08.004.
11. K. Falzon and G. J. Pace, '"Combining testing and runtime verification techniques," in *Model-Based Methodologies for Pervasive and Embedded Software*, vol. 7706, R. J. Machado, R. S. P. Maciel, J. Rubin, and G. Botterweck, Eds. Berlin: Springer-Verlag, 2013, pp. 38–57.

12. C. Colombo, "Combining testing and runtime verification," in *Proc. Comput. Sci. Annu. Workshop (CSAW'12)*, Msida, Malta, Nov. 19–20, 2012. Accessed: Sept. 25, 2020. [Online]. Available: https://www.um.edu.mt/library/oar/handle/123456789/23043
13. L. Pike, A. Goodloe, R. Morisset, and S. Niller, "'Copilot: A hard real-time runtime monitor," in *Runtime Verification*, vol. 6418, O. Sokolsky and N. Tillmann, Eds. Berlin: Springer-Verlag, 2010, pp. 345–359.
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. Mousavi, Eds. Berlin: Springer-Verlag, 2018, pp. 144–162.
15. CEDARtools. Accessed: Sept. 25, 2020. [Online]. Available: https://accemic.com/cedartools/
16. A. V. Jayakumar et al., "Systematic software testing of critical embedded digital devices in nuclear power applications," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. (ISSRE 2020)*, 2020, p. 8.
17. C. Elks et al., "Preliminary results of a bounded exhaustive testing study for software in embedded digital devices in nuclear power applications," Final Rep., U.S. Dept. of Energy, Washington, D.C., 2019. Accessed: Sept. 18, 2020. [Online]. Available: https://lwrs.inl.gov/Advanced%20IIC%20System%20Technologies/Preliminary_Results_Bounded_Exhaustive_Testing_Study_Software_Embedded_Digital_Devices_NP_Applications.pdf
18. D. R. Kuhn and V. Okun, "Pseudo-exhaustive testing for software," in *Proc. 30th Annu. IEEE/NASA Softw. Eng. Workshop (SEW-30 2006)*, 2006, pp. 153–158. doi: 10.1109/SEW.2006.26.
19. *TESSY: Test System*, Razorcat Development GmbH, Berlin. https://www.razorcat.com/en/product-tessy.html (accessed Feb. 29, 2020).
20. C. Eagle, *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. San Francisco, CA: No Starch Press, 2011.

Digital Object Identifier 10.1109/MC.2020.3045249