Designing Reliable Cyber-Physical Systems

Gadi Aleksandrowicz, Eli Arbel, Roderick Bloem, Timon D. ter Braak, Sergei Devadze, Goerschwin Fey, Maksim Jenihhin, Artur Jutman, Hans G. Kerkhoff, Robert Könighofer, Shlomit Koyfman, Jan Malburg, Shiri Moran, Jaan Raik, Gerard Rauwerda, Heinz Riener, Franz Röck, Konstantin Shibin, Kim Sunesen, Jinbo Wan, and Yong Zhao

Abstract Cyber-physical systems, that consist of a cyber part—a computing system—and a physical part—the system in the physical environment—as well as the respective interfaces between those parts, are omnipresent in our daily lives. The application in the physical environment drives the overall requirements that must be respected when designing the computing system. Here, reliability is a core aspect where some of the most pressing design challenges are:

- monitoring failures throughout the computing system,
- determining the impact of failures on the application constraints, and
- ensuring correctness of the computing system with respect to application-driven requirements rooted in the physical environment.

This chapter gives an overview of the state-of-the-art techniques developed within the Horizon 2020 project IMMORTAL that tackle these challenges throughout the stack of layers of the computing system while tightly coupling the design

G. Aleksandrowicz • E. Arbel • S. Koyfman • S. Moran
IBM Research Lab, Haifa, Israel

R. Bloem • R. Könighofer • F. Röck
Graz University of Technology, Graz, Austria

T.D. ter Braak • G. Rauwerda • K. Sunesen
Recore Systems, Enschede, The Netherlands

S. Devadze • A. Jutman • K. Shibin
Testonica Lab, Tallinn, Estonia

G. Fey • J. Malburg • H. Riener
German Aerospace Center, Bremen, Germany

M. Jenihhin • J. Raik (✉)
Tallinn University of Technology, Tallinn, Estonia
e-mail: jaan.raik@ttu.ee

H.G. Kerkhoff • J. Wan • Y. Zhao
University of Twente, Enschede, The Netherlands

F. Fummi, R. Wille (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 454,
DOI 10.1007/978-3-319-62920-9_2

methodology to the physical requirements. (The chapter is based on the contributions of the special session *Designing Reliable Cyber-Physical Systems* of the *Forum on Specification and Design Languages* (FDL) 2016.)

Keywords Adaptive test strategy generation • Automatic test case generation • Checker minimization • Checker qualification • Concurrent online checkers • Counterexample-guided inductive synthesis • CPS • Cross-layered fault management • Cyber-physical systems • Dependable CPSoC • Embedded systems • Fault classification • Fault management infrastructure • Fault tolerance • Gating-aware error injection • Gradual degradation • Health monitors • Heterogeneous • IDDQ • IEEE 1687 • Many-core • NBTI aging • Parameter synthesis • Reliability analysis • Resource management software • Run-time resource mapping • Satisfiability modulo theories • System-on-chip

1 Introduction

Cyber-physical systems (CPS) [30] are smart systems that integrate computing and communication capabilities with the monitoring and control of entities in the physical world reliably, safely, securely, efficiently, and in real-time. These systems involve a high degree of complexity on numerous scales and demand for methods to guarantee correct and reliable operation. Existing CPS modeling frameworks address several design aspects such as control, security, verification, or validation, but do not deal with reliability or automated debug aspects.

Techniques presented in this chapter are developed in the EU Horizon 2020 project IMMORTAL.[1] These techniques target reliability of CPS throughout several abstraction layers during design and operation, considering fault effects from different error sources ranging from design bugs via wear-outs and soft errors towards environmental uncertainties and measurement errors [3].

We consider the cyber part of CPS at four layers of abstraction shown in Fig. 1. The *analog/mixed-signal (AMS) layer* models the components, especially sensors and actuators, using Matlab/Simulink or VHDL-AMS. In this layer we focus on the aging behavior of the design. Thermal and electrical stress can degenerate sensor quality, actuator quality, or reduce the overall performance characteristics of the design. In Sect. 2 we present a health monitoring approach to warn the system early if functional parts of the system degenerate to such an extent that reliable operation can no longer be ensured and, e.g., redundant components must be activated.

At the *digital hardware layer* the CPS is either described at the *Register Transfer* (RT)-level, e.g., in a synthesizable subset of VHDL or Verilog, or as gate-level netlists. At this layer the analog signals of the analog/mixed-signal layer are abstracted to binary values. During operation of the CPS, even correctly designed

[1]Integrated Modelling, Fault Management, Verification and Reliable Design Environment for Cyber-Physical Systems, http://www.h2020-immortal.eu.

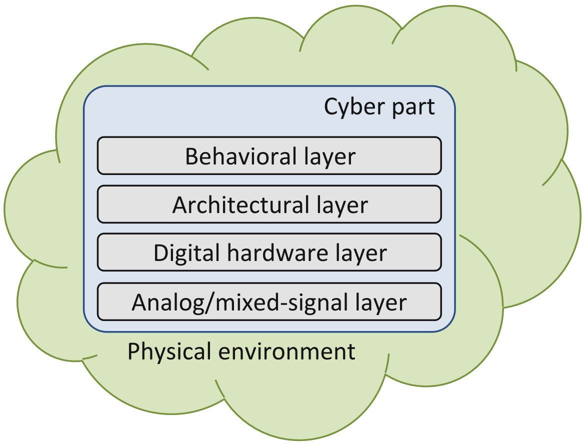

Fig. 1 The stack of layers of a CPS

systems may behave incorrectly, e.g., radiation may change values in latches or change the signal level in wires. Such effects are called soft errors and appear as bit-flips at the digital hardware layer. Error detection codes, e.g., parity bits, or *Error Correction Codes* (ECC), e.g., Hamming codes, are used to mitigate soft errors. In Sect. 3 we present approaches to automatically detect storage elements that are not protected by error detection or error correction codes or prove that storage elements are protected. Moreover, Sect. 4 provides advanced online-checker technology beyond traditional ECC schemes achieving full fault coverage.

At the *architectural layer*, we consider the CPS as a set of computational units with different capabilities, a communication network between those computational units, and a set of tasks that are described at a high level of abstraction, which should be executed on the CPS. Section 5 proposes an infrastructure for reading out the information about occurrences of faults in the lower layers and accumulating this information for preventing errors resulting from those faults. Section 6 explains how to use this infrastructure to (re)allocate and (re)schedule resources and tasks of the CPS if a computational unit can no longer provide reliable operation. As a result the CPS is enabled for fault-tolerant operation.

The *behavioral layer* considers the functional behavior and tasks of the CPS. The elements at this layer are modeled as behavioral descriptions of the system's functionality and can be realized either in software or in hardware. In Sect. 7 we consider the generation of test strategies from a system's specification given as temporal logic formulæ. Here, we focus on specifications which are agnostic of implementations and allow freedom for the implementation. Therefore the generated test cases must be able to adapt to different implementations. In Sect. 8 we present an approach to automatically synthesize parameters for behavioral descriptions of a CPS. The parameter synthesis approach can be used to assist a designer in finding suitable values for important design parameters such that given requirements are met, eliminating the need for manual error prone decisions.

2 Health Monitoring at the Analog/Mixed Signal Layer

CPS have to cope with analog input and provide analog output signals in the physical world, and be able to carry out computational tasks in the digital world. Practice has shown that major problems in terms of failures occur in the analog/mixed-signal part, which includes (on-chip) sensors and actuators. In contrast to the digital world, the (parametric) faults in the analog/mixed-signal parts of a CPS are much more complex to detect and repair.

In the case of wear-out, e.g., resulting from *Negative-Bias Temperature Instability* (NBTI) [51], it has been shown that analog stress signals cause different wear-out results as compared to digital ones, leading to more sophisticated NBTI models. The NBTI aging mechanism usually results in increased delay times (lower clock frequencies) in pure digital systems [54] while in analog/mixed-signal systems several key system-performance parameters will change, like for instance the offset voltage in OpAmps and data converters [50]. Experiments have also shown that drift of sensors [53] and actuators are often key parameters to cause faulty behavior in a CPS as a result of aging.

Stress voltages, stress temperatures, and duration of them (mission profile) are the principal factors of wear-out. Hence, in the case of a real CPS, these stress parameters must be measured during life-time and subsequently handled as mission profiles cannot be predicted accurately in advance. A combination of environmental *Health Monitors* (HMs) [4] and key performance parameters [50], nowadays implemented as embedded instruments, are required for this purpose. Temperature, voltage, and current health monitors, as well as gain, offset, and delay monitors have been developed for this purpose. It is obvious that these embedded instruments should be extremely robust against aging and variability.

In the new generation of CPS, these embedded instruments will be connected by the new IEEE 1687 standard [4]. The embedded instrument will consist in that case of the original *raw* instrument and the IJTAG *wrapper* part. An example of an IJTAG-compatible I_{DDT} health monitor [24] is shown in Fig. 2. It is related to the well-known reliability-sensitive quiescent power supply I_{DDQ} measurements. The embedded instrument consists of a current-to-voltage conversion, remaining as close to V_{DD} for the core under test as it is possible. As the resulting voltages are small, several amplification stages are required after this. The last step is the conversion to a 14-bits digital word, via the frequency. In addition several supporting circuits are required, like controller and samples memory.

In order to obtain highly dependable CPS, which includes reliability, availability, and maintainability [26], more than just health monitors and embedded instruments are required. It also includes software and computational capabilities to extract the correct information from the HMs, and calculate the remaining lifetime of *Intellectual Property* (IP) components being part of a CPS from that [54]. Useful HMs for the digital cores have shown here to be I_{DDQ} and I_{DDT} embedded instruments as well as delay monitors.

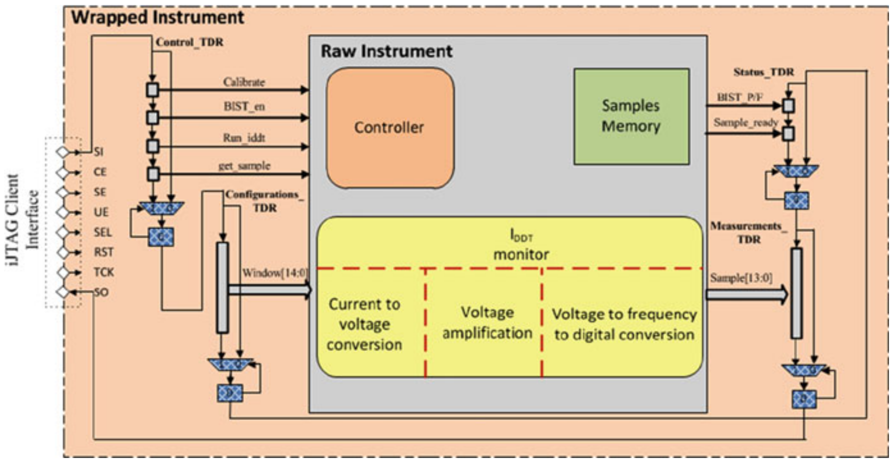

Fig. 2 An IJTAG-compatible I_{DDT} health monitor for lifetime prediction

For digital systems, like multi-core processor *System-on-Chips* (SoCs) this platform is already well on the way. To know the remaining life-time is essential in dependable CPS, as many applications are safety-critical and hence do not allow *any* down-time to ensure high availability. Existing digital systems have already been shown to be capable to react *after* a failure has occurred, mainly by the use of pseudo online *Build-In Self Test* (BIST) of processor cores. In addition, the (on-chip) repair in the case of multi-core processor SoCs has been successfully accomplished by shutting down the faulty core and replace it by a spare processor core, or increase the workload of a partly-idle processor core.

In the case of the analog/mixed-signal part of a *CPS-on-Chip* (CPSoC), the situation is much more difficult. Phenomena like NBTI aging result in this case in changing key system parameters of IPs (OpAmps, filters, ADCs, and DACs), like offset, gain, and changing frequency behavior. Using our new analog/mixed-signal NBTI model in our local designs of 65 and 40 nm TSMC OpAmps and SAR-ADCs, higher-level system key parameters were derived which were used subsequently in a Matlab environment. Figure 3 shows four possible degradation scenarios, as well as the application of our two-stage repair approach. First, key parameters are monitored and digitally tuned if changing; when the maximum tuning range is accomplished, a bypass and spare IP counter action is carried out.

One can see from the figure that the CPSoC remains within its green boundaries (of parameter P) of correct operation. The figure also shows that the different degradation mechanisms trigger tuning and replace counter measures at different times. The dependability improves by several factors at the cost of more sophisticated health monitors, software and embedded computational resources, all translating into more silicon area.

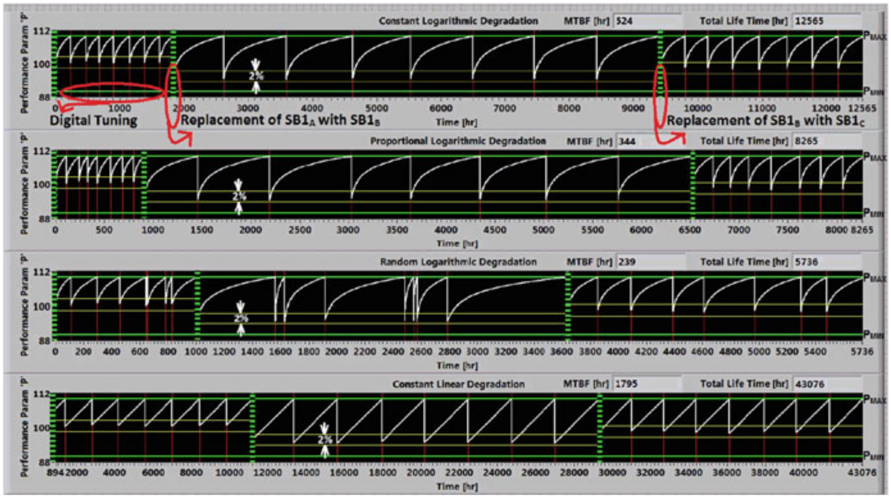

Fig. 3 Simulation of a redundant and digital IP tuning platform for highly dependable mixed-signal CPS system for four different degradation scenarios

3 Comprehensive and Scalable RT-Level Reliability Analysis

The dependability of CPS crucially depends on the reliability and availability of their digital hardware components. Even if all digital hardware components are free of design bugs, they may still fail at run-time due to, e.g., environmental influences such as radiation or aging and wear-out effects that result in occasional misbehavior of individual hardware components. In the following we subsume such transient errors under the term *soft error* [33].

A common approach to achieve resiliency against soft errors adds circuitry to automatically detect or even correct such errors [35]. This can be achieved by including redundancy, e.g., in the form of parity bits or more sophisticated error detection or correction codes [33]. Soft error reporting in the RT level can be done by using *error checkers*. Once a soft error is reported by a checker, it is up to the *Fault Management Infrastructure* (FMI) to decide how to react to this transient fault.

Therefore, the ability to understand the reliability of a given hardware component in a CPS becomes a key aspect during the component design phase. In order to cope with the ever shrinking design cycles it is highly desired that this analysis is performed in pre-silicon. Many methods for pre-silicon resiliency analysis have been proposed. These methods can be roughly classified into two categories: simulation-based methods, e.g., [22, 28, 31, 32], and formal methods, e.g., [16, 27, 43]. At the heart of the simulation-based methods lies the concept of error injection. In this approach the design is simulated and verified for robustness in the presence of transient faults injected deliberately during simulation. This approach is workload-dependent and achieves low state and fault coverage due to the enormous state space size. In an attempt to alleviate the coverage issues of the simulation-based approach formal methods have been suggested. A common practice in this approach is to perform formal verification using a fault model which models single event upsets. Being applied monolithically, this approach suffers from capacity limits inherent to formal verification methods which makes it impractical in many real-life industrial cases.

Many hardware mechanisms used for soft error protection are local in their nature. For example, parity-based protection, Error Correction Code (ECC) logic, and residue checking mechanisms are all examples of design techniques aimed at protecting relatively small parts in the design, referred to as *protected structures*. An *error detection signal* is a Boolean expression that is assigned true when an error has occurred. A *protected structure* consists of an error checker fed by error detection signals, of *protected sequential elements* and of various *gating conditions* on the way to the checker. Gating conditions are required in high performance designs to turn off reliability checks when certain parts of the logic are not used.

Based on the locality of the protected structures, we propose a novel approach for reliability analysis and verification, a basic version of which was presented in [7]. We divide the reliability verification process into an *analysis stage* and a *verification stage*. In the *analysis stage* the local protection structures are identified, and in the *verification stage* it is verified that the protection structures work properly. There are aspects of the verification that can be proved with formal verification, e.g., it can be proved formally that a certain sequential element is protected by a certain checker under certain gating conditions [7]. Since each protection is local in its nature, applying formal techniques is scalable. Other aspects may require dynamic simulation; for example, proving that the gating conditions are not over-gating the protected structure [6]. In the following we provide an overview of our new approach, and give a glimpse at the technical "how."

3.1 Analysis Stage

The analysis stage identifies the protected structures and is divided into two substages. The *identification of error detection signals* stage and the *structural analysis* stage.

Identification of Error Detection Signals In this stage the building blocks of the error detection and correction logic are identified. For this purpose we use the error checkers as anchors and employ formal and dynamic methods to accurately and efficiently identify various error detection constructs. An example for parity checking identification is described in [7]. Other examples of error detection logic that can be identified accurately and locally in this stage are residue and one-hot checking. Error correction code, however, doesn't need to be connected to error checkers. To detect ECC computation we rely on the fact that we are looking for linear ECC, a computation of the form $v = Au$ for vectors v, u over $\mathbb{Z}_2$. To achieve

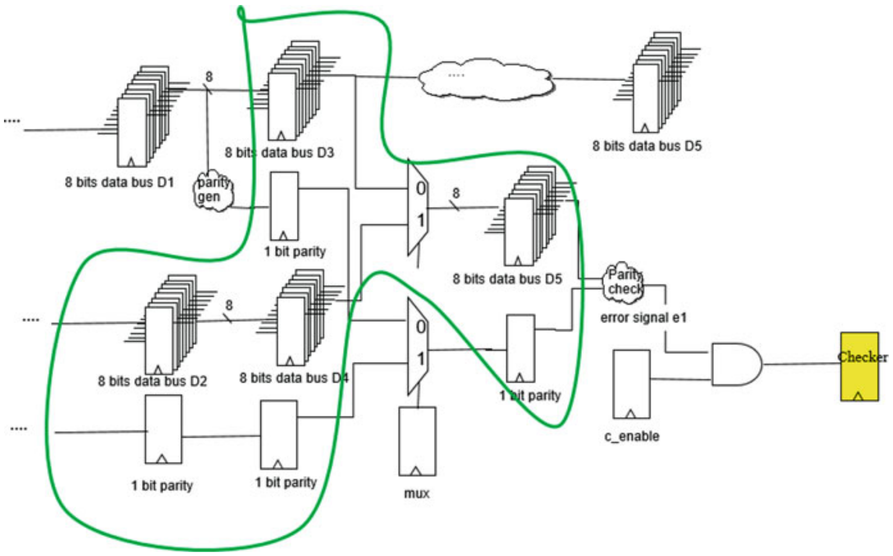

Fig. 4 A parity protected structure

this, we iterate over all the vectors in the design and identify the vectors with bits that are leaves of a XOR-computation tree. After discovering an ECC-like matrix we first purge non-ECC instances, such as the case of the identity matrix, or matrix with too many one-hot columns indicating that most input bits are used only once. We determine whether this is an ECC generation or an ECC check by searching for a unit submatrix with dimensions corresponding to the output size.

Structural Analysis In order to identify the protected structure of each error detection signal, we analyze the topology of the netlist representing the design. The objective is to identify the set of sequential elements protected by an error detection signal. The challenge here is twofold:

- Understating the boundaries of the protection, e.g., if the protection is parity-based, the parity generation logic and the parity checking logic form the boundary of the protected sequential elements.
- Proper identification of the corresponding gating logic.

For example, in Fig. 4 the protected sequential elements are the encircled ones, plus more sequential elements connected to data bus $D2$ and the corresponding parity bit which are out of scope. Specifically, the *c_enable* signal is the gating condition of the error detection signal—an erroneous parity check will make the error checker fire only if the value of the sequential element is 1, and this sequential element is not a part of the protected structure; similarly, the *mux* signal is not protected. Also, the data bus *D1* is located before the parity generation logic and thus is not a part of the protected structure either. Due to lack of space, the full algorithm for detecting the protected structure, will not be provided here. However, we will give a glimpse of the way the algorithm copes with the above challenges.

Consider a parity protected structure. When the parity generation is in the scope of the given netlist, then the boundary of the parity protection can be easily identified by detecting the parity generation. Moreover, in this case the protected data bus and parity bits are the intersection of the input cone of the parity check and the output cone of the parity generation, whereas the gating conditions are not in that intersection. Hence, the boundaries of the protection and the gating condition logic can be identified quite easily.

However, when the parity generation is not in the scope of the given netlist, it is more difficult to distinguish between the protected structure and gating conditions. In industrial systems this situation is quite common. In order to distinguish between data and gating conditions in such cases, the analysis is performed on the *parse tree* which represents more clearly the designer intent than the corresponding Boolean logic representation. Consider the following assignment for a vector bus *data*:

$$\text{data}(0\ldots7) \Leftarrow \text{data}_1(0\ldots7)\text{ when cond}_1\text{ else }00000000$$

It is quite easy to understand from the parse tree that cond_1 is a gating condition, while $\text{data}_1(0\ldots7)$ is the data source, while it is more challenging to infer the same from a set of logical assignments of the form

$$\text{data}(i) \Leftarrow \text{data}_1(i) \wedge \text{cond}_1$$

Moreover, it is impossible to distinguish between data source and the gating condition when a statement

$$\text{bit}_1 \Leftarrow \text{bit}_2\text{ when cond else }0$$

is represented by a logical equivalent

$$\text{bit}_1 \Leftarrow \text{bit}_2 \wedge \text{cond}$$

Therefore, in order to cope with the above challenge we perform the analysis using the parse tree.

3.2 Verification Stage

At this stage we verify that the constructs found at the earlier stage indeed protect the relevant sequential elements. The verification that is required here has two aspects: (a) verifying that under the relevant gating conditions the sequential elements are indeed protected by the corresponding error detection signals or error correction logic; (b) verifying that the gating conditions are not over-gating and will not prevent a checker from firing when it should, causing silent data corruption.

For the former, formal verification can be used, leveraging the locality of protection structures. In [7] we perform it for simple parity protection. More research is still required to expand the approach from [7] to include other protection types and more complex parity structures.

The challenge in the latter verification aspect is that in order to verify that the gating conditions are not over-gating a global scope is required, since the gating conditions can be dependent on various parts of the design. In [6] we present a novel and effective approach to verify that the gating conditions are not over-gating. We use the identification of the analysis stage to synthesize drivers that perform smart *gating aware* error injection. These drivers are then integrated in the standard functional verification environment existing for any industrial system.

4 Qualification and Minimization of Concurrent Online Checkers

Besides standard approaches for fault detection we also consider advanced error detection schemes on the digital hardware layer for CPS. Particularly, the proposed online checkers enable cost-efficient mechanisms for detecting faults during lifetime of the state-of-the-art many-core systems. These mechanisms must detect errors within resources and routers as well as enable reconfiguration of the routing network in order to isolate the problem and provide graceful degradation for the system.

Our approach [41, 42] exceeds the existing state of the art in concurrent online checking by proposing a tool flow for automated evaluation and minimization of the verification checkers. We show that starting from a realistic set of verification assertions a minimal set of checkers are synthesized that provide 100% fault coverage with respect to single stuck-at faults at a low area overhead and the minimum fault detection latency of a single clock-cycle. The latter is especially crucial for enabling rapid fault recovery in reliable real-time systems.

An additional feature of the proposed approach is that it allows formally proving the absence or presence of true misses over all possible valid inputs for a checker, whereas in the case of traditional fault injection only statistical probabilities can be calculated without providing the user with full confidence of fault detection capabilities. The formal proof as well as the minimal fault detection latency is guaranteed by reasoning on a pseudo-combinational version of the circuit and by the application of an exhaustive valid set of input stimuli as the verification environment.

The checker qualification and minimization flow starts with synthesizing the checkers from a set of combinational assertions. Thereafter, a pseudo-combinational circuit is extracted from the circuit of the design under checking. The pseudo-combinational circuit is derived from the original circuit by breaking the flip-flops and converting them to pseudo primary inputs and pseudo primary outputs. Note that, at this point, additional checkers that also describe relations on the pseudo primary inputs/outputs may be added to the checker suite in order to increase the fault coverage.

Subsequently, the checker evaluation environment is created by generating exhaustive test stimuli for the extracted pseudo-combinational circuit. These stimuli are fed through a filtering tool that selects only the stimuli that correspond to functionally valid inputs of the circuit. As a result, the complete valid set of input stimuli that serve as the environment for checker evaluation is obtained. The obtained environment, pseudo-combinational circuit, and synthesized checkers are applied to fault-free simulation. The simulation calculates fault-free values for all the lines within the circuit. Additionally, if any of the checkers fires during fault-free simulation, it means a bug in the checker or an incorrect environment.

If none of the checkers is firing in the fault-free mode, then checker evaluation takes place. The tool injects faults to all the lines within the circuit one-by-one and this step is repeated for each input vector. As a result, the overall fault detection capabilities for the set of checkers in terms of fault coverage metrics are calculated. In addition, each individual checker is weighted by summing up the total number of true detections by the checker. Finally, the weighting information is exploited in minimizing the number of checkers, eventually allowing to outline a trade-off between fault coverage and the area overhead due to the introduction of checker logic.

Experiments carried out on the control part (routing and arbitration) of a *Network-on-Chip* (NoC) router showed on a realistic application the feasibility and efficiency of the framework and the underlying methodology. Experimental results showed that the approach allowed selecting the minimal set of 5 checkers out of 31 verification assertions with the fault coverage of 100% and area overhead of only 35% [41, 42].

5 Managing Faults at SoC Level During In-Field Operation of CPS

When a fault occurs during in-field operation in a complex SoC within a CPS, which is working under the control of the software, it is necessary that the latter becomes aware of the fault and reacts to it as quickly as possible. The SoC management software, e.g., *Operating System* (OS), must then take actions to isolate and mitigate the effects of the fault. These actions include fault localization, classification based on diagnostic information, and proper handling of affected resources and tasks by the OS. This implies a cross-layer *Fault Detection, Isolation, and Recovery* (FDIR) procedure, since the faults can be detected on the hardware layer, and recovery actions can be taken throughout the stack of layers.

5.1 *Fault Management Infrastructure*

In order to deliver the information from the instruments, store health and statistics information, and provide the required inputs to the OS, the SoC contains the FMI which consists of both hardware and software side.

We propose a hierarchical in situ FMI (see Fig. 5) with low resource overhead and high flexibility during operation. IEEE 1687 IJTAG is used as a backbone of FMI to implement a hierarchical instrumentation and monitoring network for efficient and flexible access to the instruments which are attached to the monitored resources. The main benefit of using IEEE 1687 IJTAG infrastructure for in situ fault management is based on considerable reuse of existing test and debug infrastructure and instrumentation later in the field for the new purpose of fault management. In our architecture, traditional IJTAG is extended with asynchronous fault detection signal propagation to significantly improve the fault detection latency.

Fault Manager (FM) is a part of OS (kernel) which is responsible for updating both health and resource maps. If a fault is detected in the system, FM must start a diagnostic procedure to find out the location of the fault as precisely as possible. This location information must be reflected in the *Health Map* (HM) by setting the fault flag for the appropriate resource and updating the fault statistics.

Instrument Manager (IM) is a hardware module which is responsible for the communication with the instruments through IJTAG network. It informs FM about fault detections and provides the read/write access to the instruments.

Health map (HM) is a data structure in a dedicated memory which holds the detailed information about the faults and the fault statistics. HM is the run-time model of CPS including fault monitors and implements a structural view of

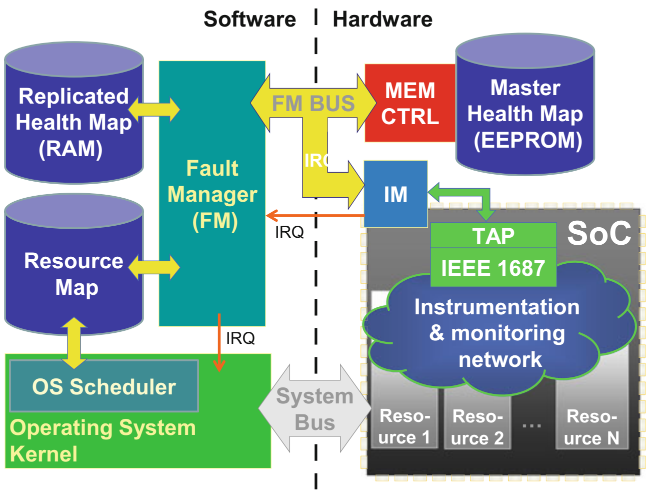

Fig. 5 Overview of the fault management infrastructure

the system's hardware resources and its important parts identified by the static (design time) analysis. To retain the information about the known faults across power cycles, HM should be stored in a reliable non-volatile memory.

Resource map (RM) is a data structure in the system memory which holds the information about the current status of the system's resources. It should be modified on the fly during system's normal operation, should a fault be detected by an instrument or a diagnostic routine.

5.2 Fault Classification and Handling

Ability to classify errors, malfunctions, and faults is an important basis for health map management, effective system recovery, and fault management. We propose to classify the faults according to their severity levels and their contribution to the permanent malfunction of system's components and modules. Such classification has a strong relation to fault management processes and the architecture of the Health Map.

The classification of faults to be used in FM procedures consists of the following categories:

- **Persistence**: This parameter shows what the nature of the fault occurrence is, i.e., whether it is transient, intermittent, or permanent.
- **Severity**: Faults can be different in their influence on the resource. While one fault can be benign (e.g., one of several similar execution units in a superscalar CPU fails), another can make the resource useless (e.g., program counter in a CPU core).
- **Criticality**: Depending on the resource where the fault has occurred, its consequences for operability and stability of the system as a whole can span from none to total system failure.
- **Diagnostic Granularity**: A fault is found by an instrument or deducted by diagnostic procedure. A fault entry in the data structure of fault management system should be assigned with the information about how it was found, e.g., by an instrument, diagnostic procedure, or an OS self-test.
- **Fault location**: The attributed location of the fault is the result of a fault detection or a fault diagnosis procedure.

When a fault occurs in the system and is detected with the help of FMI, the system must react and handle that fault in order to mitigate the current or future effects it can have on the system. The information which the proposed fault classification method offers is used in this process. The complete procedure which allows for quick and efficient fault handling should consist of the following steps: fault detection, fault localization, coarse-grained fault classification (before detailed diagnostic information becomes available), immediate system response (e.g., rescheduling a task affected by the fault), fault diagnosis, and, finally, a conclusive fine-grained fault classification.

6 Many-Core Resource Management for Fault Tolerance

On the architectural layer advanced CPS will rely on heterogeneous many-core SoCs to provide the demanded throughput computing performance within the allowed energy budget. Heterogeneous many-core architectures typically have many redundant and distributed resources for processing, communication, memory, and IO. This inherent redundancy can potentially be used to implement systems that are fault tolerant and degrade gradually. To realize this potential, we combine the FMI with run-time resource management software. First, the many-core architecture is instrumented with FMI and online checkers and health monitors. As we have explained in the previous sections, the online checkers and health monitors report faults and physical degradation at the lower hardware layers through the FMI that makes the information available for system and application software. The proposed instrumentation can thus provide a system wide HM showing the health and the functioning of the hardware resources of the running system. It reports on faulty components and also on health issues warning about fault expectancy. The former allows reacting on and recovering from faults whereas the latter allows anticipating and reconfiguring before faults occur. Second, the health information is lifted and abstracted to augment run-time resource management software [23, 47, 48] with information about hardware resources to be used less or entirely avoided by reconfiguring the way tasks and communications are mapped to resources.

The resource manager partitions computation, communication, and memory resources based on resource reservations of the application [1, 46]. The run-time mapping algorithms of the resource management software relies on abstract representations of task and platform graphs and are optimized for embedded systems [48].

In [2, 49] and [45] it was shown how reconfigurable multi/many-core architectures in combination with run-time resource management software can be used to implement fault-tolerance features. This work depended on ad hoc detection and reporting of faults and did not include health information about physical wear-out or accelerated aging. Here an important next step is taken to combine resource management with detailed health information systematically reported by a cross-layered fault management infrastructure at run-time.

The run-time resource management [23, 47] is made health-aware. Figure 6 illustrates the resource management with integrated HM information. Through the fault manager described in the previous section, measurements of health monitors and checkers provide domain-specific and/or hardware-specific information. For separation of concerns and extensibility, it is desired to hide this domain-specific knowledge from the upper software layers. At the lower layers, the domain-specific knowledge is required to map the sensor/checker data (the domain) onto a fixed range of values. So, the health data stored in the HM is modeled as a health function $\mathsf{health} : R \rightarrow [0, 1]$ that maps each hardware resource (provider) $r \in R$ to a health value $v \in [0, 1]$, where R is the finite set of resources in the target

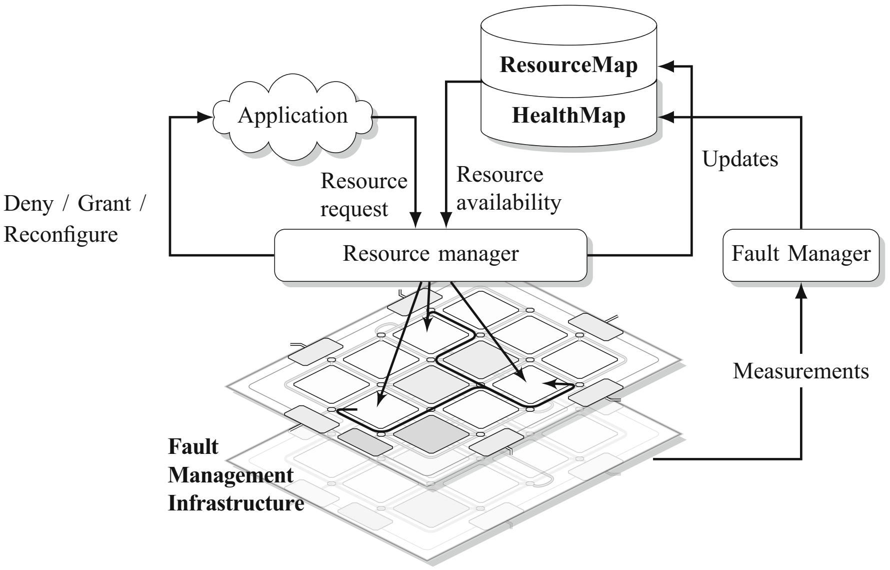

Fig. 6 Health information in resource management

architecture. A high health value $\mathsf{health}(r)$ indicates that a resource $r \in R$ is functioning correctly, whereas a low health value indicates the deterioration of the resource.

The advantages of using a health function with a range in the real numbers, as opposed to a function with a Boolean range, is that degradation can be modeled. The resource manager may circumvent the use of specific resources to reduce aging and hot spots. These resources are assumed to function correct when the health function is still positive, and can, therefore, still be activated when the system utilization increases.

The health function can be further extended to cover more details about the resource providers which could help the resource manager to choose best fitting resources for each task. As Fig. 6 illustrates, the fault manager reads the sensor/checker data out of the FMI, and processes the measurements by mapping the outcome to the range of the health function. Multiple sensors measuring the same hardware component should apply sensor fusion to conform to this HM function. In this fusion, again domain-specific knowledge is leveraged to weight the importance and possible relation between the sensors.

The health function is subsequently used in the selection process of the resource management, in which two use cases are identified:

1. New resource requests are handled according to the information contained in the HM.
2. For a resource in use, if the health indicator exceeds a configurable threshold, the resource manager will isolate the resource and attempt to reconfigure the applications currently using the corresponding resource.

For use case (1), a new request for resources is made by an application and the resource manager consults the RM to find the most suitable resources to fulfill the request. Both the assignment of tasks to processing elements and inter-task communication through the interconnect are taken into account. In this process, the resource manager uses a cost function to determine the best fit of the (partial) application onto the available resources of the platform. The configurable cost function takes the health map into account to define optimization objectives such as wear leveling. The cost function is designed to assign increasingly higher cost to a hardware resource $r \in R$, which should be used less or should not be used according to the HM, such that

$$\left(\lim_{\mathsf{health}(r)\to 0} \mathsf{cost}(r)\right) = \infty$$

For use case (2) whenever the HM is updated with new measurements, the new values are compared with a configurable threshold. When the threshold is exceeded, action needs to be taken to reduce the usage of that resource or completely stop using it. For resources currently in use possibly by several applications, this can require one or multiple granted resource requests to be reassigned to a different resource.

In a system including the proposed FMI and fault management approach, the FDIR procedure is facilitated by the results of the fault classification based on different fault categories determined by monitoring in lower layers, information from the instruments as well as the accumulated fault statistics.

7 Deriving Adaptive Test Strategies from LTL-Specifications

To obtain confidence in the correctness of a CPS system at the behavioral layer, model checking [13, 39] can prove that (a model of) the system satisfies desired properties. However, it cannot always be applied effectively.

This may be due to third-party IP components for which no source code or model is available, or due to high effort for building system models that are precise enough. Since our *System Under Test* (SUT) is safety critical, we desire high confidence in its adherence to specification φ. Nevertheless, even though φ may be simple, the implementation of the SUT can be too complex for model checking. Especially, if it considers further signals to synchronize with other systems. And finally, model checking can only verify an abstracted model and never the final and “live” system.

Testing is a natural approach to complement verification, and automatic test case generation allows to keep the effort at reasonable size. Deriving tests from a system specification instead of the implementation, called black-box testing, is particularly attractive as (1) tests can be generated way before the actual implementation work starts, (2) these tests can be reused on various realizations of the same specification, and (3) the specification is usually way simpler than the actual implementation. In addition, the specification focuses on the most important aspects that require intensive testing. Fault-based techniques [25], in which test cases are generated to detect certain fault classes, are particularly interesting to detect bugs.

Various methods focusing on coverage criteria exist to generate test sets from executable system models (e.g., finite state machines). Methods to derive tests from declarative requirements (see, e.g., [19]) are less common, as the properties still allow implementation freedom and, therefore, cannot be used to fully predict the system behavior under given inputs. Thus, test cases have to be *adaptive*, i.e., able to react to observed behavior at run-time. This is especially true for *reactive systems* that interact with their environment. Existing techniques often get around this by requiring a deterministic model of the system behavior as additional input [18].

In [10] we presented a new approach to synthesize test strategies from temporal logic specification. This approach is also applicable on a CPS if a temporal logic specification is given. The derived adaptive strategies can be used during the development process for system verification as well as after deployment for run-time verification to detect faults that occur only after a certain amount of time, for example due to aging. Figure 7 outlines our proposed testing setup. The user provides a specification φ, expressing requirements for the system under test in *Linear Temporal Logic* (LTL) [37]. The specification can be incomplete. The user also provides a fault model, for which the generated tests shall cause a specification violation, in form of an LTL formula that has to be covered.

Based on hypotheses from fault-based testing [36], we argue that tests that reveal faults as specified by our fault models are also sensitive to more complex bugs. We assume permanent and transient faults by distinguishing various fault occurrence frequencies and computing tests to reveal faults for the lowest frequency for which this is possible. Test strategies are generated using reactive synthesis [38] with partial information [29], providing strong guarantees about all uncertainties: If the synthesis is successful and if the computed tests are executed long enough, then they reveal all faults satisfying the fault model in every system that realizes the specification. Finally, existing techniques from run-time verification [9] can be used

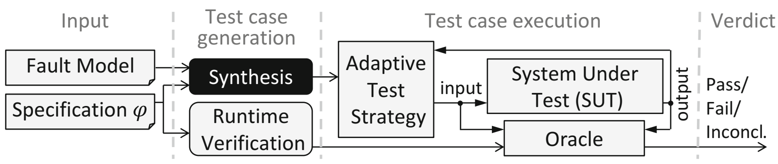

Fig. 7 Synthesis of adaptive test strategies from temporal logic specifications [10]

to construct an oracle that checks the system behavior against the specification while tests are executed.[2]

If the specification is incomplete, tests may have to react to observed behavior at run-time to achieve the desired goals. Such adaptive test cases have been studied by Hierons [21] from a theoretical perspective, however, relying on fairness (every non-deterministic behavior is exhibited when trying often enough) or probabilities.

Testing reactive systems can be seen as a game between two players: the tester providing inputs and trying to reveal faults, and the SUT providing outputs and trying to hide faults, as pointed out by Yannakakis [52]. The tester can only observe outputs and has, therefore, partial information about the SUT. The goal for the game is to find a strategy for the tester that wins against every SUT. The underlying complexities are studied by Alur et al. [5]. Our work builds upon reactive synthesis [38] (with partial information [29]). This can also be seen as a game, however, we go beyond the basic idea. We combine the concept of game theory with fault models defined by the user. Nachmanson et al. [34] synthesize game strategies as tests for non-deterministic software models. Their approach, however, is not fault-based and focuses only on simple reachability goals.

To mitigate scalability issues, we compute test cases directly from the provided specification φ. Our goal is to generate test strategies that *enforce* certain coverage objectives *independent* of any freedom due to incomplete specification. Some uncertainties about the behavior of the SUT may also be rooted in uncontrollable environment aspects like weather conditions. For our proposed testing approach, this makes no difference.

We follow a fault-centered approach. The definition of the fault class is a composition of the fault kind and the fault frequency. While the fault kind expresses the type of the fault, such as a bit flip or a stuck-at fault, the fault frequency describes the frequency of the fault being present in the system. This can be (1) a permanent fault that is present all the time, (2) a fault that occurs from some point onwards, (3) a fault that occurs again and again, or even (4) a fault that occurs only once in the future. A test strategy that is capable of detecting a fault that occurs only at a low frequency, for example only once in the future, is also capable of detecting a fault that occurs at a higher frequency, for example from some point in time onwards. Thus, the goal is to derive a strategy for the lowest fault-frequency possible.

Certain test goals may not be enforceable with a static input sequence. We thus synthesize *adaptive* test strategies that direct the tester based on previous inputs and outputs and, therefore, can take advantage of situational possibilities by exploiting previous system behavior. The derived strategies force the system to enter a state in which it has to violate the specification if the fault is present in the system.

[2]The semantics of LTL are defined over infinite execution traces; however, we can only run the tests for a finite amount of time. This can result in inconclusive verdicts [9]. To overcome this problem, we refer to existing research on interpreting LTL over finite traces [14, 15, 20].

Our generated test strategies reveal all instances of a user-defined fault class for every realization of a given specification and do not rely on any implementation details.

8 Parameter Synthesis for CPS

Many problems in the context of computer-aided design and verification of CPS can be reduced to deciding the satisfiability of logic formulæ modulo background theories [8]. In parameter synthesis, the logic formulæ describe how the CPS evolves over time from a set of initial states, where some parameters are kept open and have to be filled such that none of a given set of bad states is ever reached. Parameter synthesis can be effectively reduced to solving instances of $\exists\forall$-queries. An $\exists\forall$-query asks for the existence of parameter values such that for all possible state sequences, the CPS avoids reaching a bad state.

Solving such $\exists\forall$-queries is especially challenging in the context of CPS, where the variables are quantified over countably infinite or uncountably infinite domains. Different approaches for parameter synthesis for hybrid automata, e.g., [11, 12, 17], have been proposed. The approaches considered the problems of computing one value for the parameters as well as all possible parameter values, but are restricted to hybrid automata with linear and multiaffine dynamics. We propose a *Satisfiability Modulo Theories* (SMT)-based framework for synthesizing one value for open parameters of a CPS modeled as logic formulæ [40] using *Counterexample-Guided Inductive Synthesis* (CEGIS) [44] and introduce the notion of *n-step inductive invariants* to reason about unbounded CPS correctness.

CEGIS CEGIS is an attractive technique from software synthesis to infer parameters in a sketch of a program leveraging the information of a provided correctness specification. In software synthesis, CEGIS was able to infer those parameters in many cases, where existing techniques from quantifier elimination failed.

Suppose that Q, I, and K are the sets of all possible states, inputs, and parameter valuations, respectively. We use the correctness formula $\mathsf{correct} : I' \times K \rightarrow \mathbb{B}, (i', k) \mapsto \mathsf{correct}(\hat{i}', \hat{k})$ that evaluates to true if and only if the CPS with concrete parameter values $\hat{k} \in K$ is correct when executed on the concrete input sequence $\hat{i}' \in I'$, where $I' = Q \times I^n$. The basic idea of CEGIS is to iteratively refine candidate values for parameters based on counterexamples until a correct solution is obtained. The CEGIS loop is depicted in Fig. 8. The loop repeats two steps to compute parameter values $\hat{k} \in K$ such that $\forall i' \in I' : \mathsf{correct}(i', \hat{k})$ holds and maintains a database $D \subseteq I'$ of concrete parameter values, which is initially empty. The database is used to lazily approximate the domain of I' with a small set of values. In the first step, a candidate parameter $\hat{k}$ is computed such that $\bigwedge_{\hat{i}' \in D} \mathsf{correct}(\hat{i}', \hat{k})$ holds, i.e., the parameter values $\hat{k}$ guarantee correctness of the CPS for (at least) all input sequences stored in the database D. The candidate parameters are then verified by checking if a counterexample $\hat{i}'$ exists that refutes $\forall i' \in I' : \mathsf{correct}(i', \hat{k})$

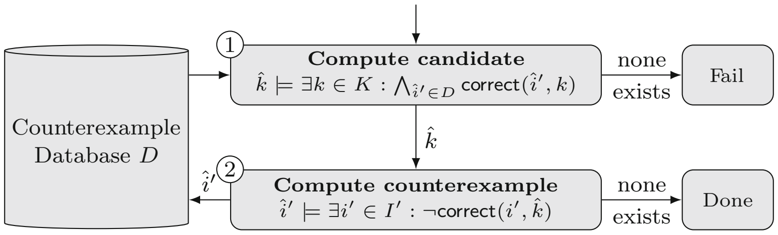
Counterexample Database D
1
Compute candidate
$\hat{k} \models \exists k \in K : \bigwedge_{\hat{i}' \in D} \mathsf{correct}(\hat{i}', k)$
none exists
Fail
$\hat{k}$
2
$\hat{i}'$
Compute counterexample
$\hat{i}' \models \exists i' \in I' : \neg\mathsf{correct}(i', \hat{k})$
none exists
Done

Fig. 8 Counterexample-guided inductive synthesis (CEGIS) [44]

considering the entire domain I' of input sequences. If so, the counterexample $\hat{i}'$ is added to the database D. Otherwise, if no counterexample exists, the approach terminates and returns the parameters $\hat{k}$. In the general case, the CEGIS loop has three possible outcome: (1) parameters $\hat{k} \in K$ can be found such that the formula $\forall i' \in I' : \mathsf{correct}(i', \hat{k})$ becomes true (Done), (2) the unsatisfiability of the formula $\exists k \in K : \forall i' \in I' : \mathsf{correct}(i', k)$ is proven because no new parameters can be computed (Fail), or (3) the CEGIS loop does not terminate but refines the candidate values for the parameters forever. To guarantee termination of the loop, at least one of the two involved domains, K or I', has to be finite. However, even if both domains are infinite, the approach is in many cases able to synthesize parameters.

***n*-Step Inductive Invariants** The correctness of a CPS is defined by using an invariant-based approach. A user symbolically defines the set of all possible initial states $\mathsf{init} : Q \times K \to \mathbb{B}$, the set of all safe states $\mathsf{safe} : Q \times K \to \mathbb{B}$, the sets of all states of an inductive invariant $\mathsf{inv} : Q \times K \to \mathbb{B}$, and a transition function $T : Q \times I \times K \to Q$ of the CPS in the form of logic formulæ modulo theories. By induction, a CPS cannot visit an unsafe state and is correct if:

1. all initial states satisfy the invariant, i.e.,

$$\mathsf{A}(q, k) :\Leftrightarrow \Big(\mathsf{init}(q, k) \to \mathsf{inv}(q, k)\Big),$$

2. all states that satisfy the invariant are also safe, i.e.,

$$\mathsf{B}(q, k) :\Leftrightarrow \Big(\mathsf{inv}(q, k) \to \mathsf{safe}(q, k)\Big),$$

3. from a state that satisfies the invariant, the invariant is again satisfied after at most n steps of the transition relation T and all states that can be reached in the meantime are safe, i.e.,

$$\mathsf{C}(q_0, i_1, \ldots, i_n, k) :\Leftrightarrow \Big(\mathsf{inv}(q_0, k) \to \bigvee_{j=1}^{n} \mathsf{inv}(q_j, k) \wedge \bigwedge_{l=1}^{j-1} \mathsf{safe}(q_l, k)\Big),$$

where q_j is an abbreviation for $T(q_{j-1}, i_j, k)$ for all $j > 0$.

The CPS is correct if concrete parameter values $\hat{k} \in K$ exist such that

$$\forall q_0 \in Q : \forall i_1, \ldots, i_n \in I : \mathsf{correct}(q_0, i_1, \ldots, i_n, \hat{k})$$

holds, where the correctness formula

$$\mathsf{correct}(q_0, i_1, \ldots, i_n, \hat{k}) :\Leftrightarrow \Big(A(q_0, \hat{k}) \wedge B(q_0, \hat{k}) \wedge C(q_0, i_1, \ldots, i_n, \hat{k})\Big)$$

is defined over sequences of inputs $(q_0, i_1, \ldots, i_n) \in I'$ of n steps.

8.1 Heuristics and Implementation

We implemented the CEGIS loop depicted in Fig. 8 as a proof-of-concept tool, ParSyn-CEGIS,[3] based on an SMT-solver. The SMT-solver is used to find concrete parameters and counterexamples. In case of CPS typically infinite domains are considered such that the CEGIS loop may not converge. To improve convergence in practice, we developed three simple heuristics:

1. *Counterexample randomization*: To avoid the generation of too similar counterexamples, the ParSyn-CEGIS attempts to randomize every second counterexample. In an iterative loop, for each value of the counterexample, a random value of the same type is generated and substituted. If the adapted counterexample still violates the correctness check, i.e., is still a counterexample, the randomized value is kept. Otherwise, it is rejected.
2. *Restart strategy*: Inspired by the implementation of today's solvers for Boolean satisfiability, we implemented a restart strategy. The restart strategy aids the SMT-solver to recover from learned information that does not help in deciding the overall $\exists\forall$-query. When a restart happens, all counterexamples are removed from the database and the CEGIS loop starts from the beginning without a priori knowledge. After each restart, the period of the restart is increased.
3. *Demand for progress*: Given two subsequent values $\hat{k}_a$ and $\hat{k}_b$ of the same parameter, we measure their progress by $\mathsf{progress}(\hat{k}_a, \hat{k}_b) = \|\hat{k}_a - \hat{k}_b\|$. This measure is used to restart the synthesis procedure when the CEGIS loop gets stuck by producing similar counterexamples, but counterexample randomization is not effective. In each iteration, for the last pair of parameter values $\hat{k}_{c-1}$ and $\hat{k}_c$, the progress value $\mathsf{progress}(\hat{k}_{c-1}, \hat{k}_c)$ is computed. If the progress value repeatedly falls below a fixed progress threshold δ, e.g., more than 10 times, a restart is initiated.

[3] ParSyn-CEGIS, https://github.com/hriener/parsyn-cegis.

Parameter synthesis automates the task of finding good values for important design parameters in CPS and eliminates the error prone design steps involved in determining those parameter values manually.

9 Conclusions

Within the IMMORTAL project, we have identified several challenging problems in the context of reliability and automated debug considering advanced CPS throughout the stack of layers and the design flow. For each of these problems we presented in this chapter a glimpse on how to solve the issues and how tool automation can improve the overall design process. For further details on each of the solutions we refer to the respective publications.

Overall, reliable CPS design and the corresponding design automation is a vivid and ongoing research topic. CPS design automation links traditional hardware-oriented aspects with software engineering and the large body of work in control theory.

Acknowledgements This work was supported in part by the European Union (Horizon 2020 IMMORTAL project, grant no. 644905).

References

1. Abeni, L., & Buttazzo, G. (2004). Resource reservation in dynamic real-time systems. *Real-Time System, 27*(2), 123–167.
2. Ahonen, T., ter Braak, T. D., Burgess, S. T., Geißler, R., Heysters, P. M., Hurskainen, H., et al. (2011). CRISP: Cutting edge reconfigurable ICs for stream processing. In *Reconfigurable computing: From FPGAs to hardware/software codesign* (pp. 211–237). New York: Springer.
3. Aleksandrowicz, G., Arbel, E., Bloem, R., ter Braak, T. D., Devadze, S., Fey, G., et al. (2016). Designing reliable cyber-physical systems – Overview associated to the special session at FDL'16. In *2016 Forum on Specification and Design Languages, FDL 2016, Bremen, Germany, September 14–16, 2016* (pp. 1–8).
4. Ali, G., Badawy, A., & Kerkhoff, H. G. (2016). On-line management of temperature health monitors using the IEEE 1687 standard. In *TESTA* (pp. 1–4).
5. Alur, R., Courcoubetis, C., & Yannakakis, M. (1995). Distinguishing tests for nondeterministic and probabilistic machines. In *STOC* (pp. 363–372).
6. Arbel, E., Barak, E., Hoppe, B., Koyfman, S., Krautz, U., & Moran, S. (2016). Gating aware error injection. In *HVC* (pp. 34–48).
7. Arbel, E., Koyfman, S., Kudva, P., & Moran, S. (2014). Automated detection and verification of parity-protected memory elements. In *ICCAD* (pp. 1–8).
8. Barrett, C. W., Sebastiani, R., Seshia, S. A., & Tinelli, C. (2009). Satisfiability modulo theories. In *Handbook of satisfiability* (pp. 825–885). Amsterdam: IOS Press.
9. Bauer, A., Leucker, M., & Schallhart, C. (2011). Runtime verification for LTL and TLTL. *TOSEM, 20*(4), 14.
10. Bloem, R., Könighofer, R., Pill, I., & Röck, F. (2016). Synthesizing adaptive test strategies from temporal logic specifications. In *FMCAD* (pp. 17–24).

11. Bogomolov, S., Schilling, C., Bartocci, E., Batt, G., Kong, H., & Grosu, R. (2015). Abstraction-based parameter synthesis for multiaffine systems. In *HVC* (pp. 19–35).
12. Cimatti, A., Griggio, A., Mover, S., & Tonetta, S. (2013). Parameter synthesis with IC3. In *FMCAD* (pp. 165–168).
13. Clarke, E. M., & Emerson, E. A. (1981). Design and synthesis of synchronization skeletons using branching-time temporal logic. In *LOP* (pp. 52–71).
14. De Giacomo, G., De Masellis, R., & Montali, M. (2014). Reasoning on LTL on finite traces: Insensitivity to infiniteness. In *AAAI* (pp. 1027–1033).
15. De Giacomo, G., & Vardi, M. Y. (2013). Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*.
16. Frehse, S., Fey, G., Arbel, E., Yorav, K., & Drechsler, R. (2012). Complete and effective robustness checking by means of interpolation. In *FMCAD* (pp. 82–90).
17. Frehse, G., Jha, S. K., & Krogh, B. H. (2008). A counterexample-guided approach to parameter synthesis for linear hybrid automata. In *HSCC* (pp. 187–200).
18. Fraser, G., Wotawa, F., & Ammann, P. (2009). Issues in using model checkers for test case generation. *Journal of Systems and Software, 82*(9), 1403–1418.
19. Fraser, G., Wotawa, F., & Ammann, P. (2009). Testing with model checkers: A survey. *Software Testing, Verification and Reliability, 19*(3), 215–261.
20. Havelund, K., & Rosu, G. (2001). Monitoring programs using rewriting. In *ASE* (pp. 135–143).
21. Hierons, R. M. (2006). Applying adaptive test cases to nondeterministic implementations. *Information Processing Letters, 98*(2), 56–60.
22. Holcomb, D. E., Li, W., & Seshia, S. A. (2009). Design as you see FIT: System-level soft error analysis of sequential circuits. In *DATE* (pp. 785–790).
23. Hölzenspies, P. K. F., ter Braak, T. D., Kuper, J., Smit, G. J. M., & Hurink, J. (2010). Run-time spatial mapping of streaming applications to heterogeneous multi-processor systems. *International Journal of Parallel Programming, 38*(1), 68–83.
24. Ibrahim, A. M. Y., & Kerkhoff, H. G. (2016). An IJTAG-compatible I_{DDT} embedded instrument for health monitoring and prognostics. In *ITC, Poster Session*.
25. Jia, Y., & Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering, 37*(5), 649–678.
26. Khan, M. A. (2014). *On Improving Dependability of Analog and Mixed-Signal SoCs: A System-Level Approach* Ph.D. thesis, University of Twente.
27. Krautz, U., Pflanz, M., Jacobi, C., Tast, H.-W., Weber, K., & Vierhaus, H. T. (2006). Evaluating coverage of error detection logic for soft errors using formal methods. In *DATE* (pp. 176–181).
28. Krishnaswamy, S., Plaza, S., Markov, I. L., & Hayes, J. P. (2009). Signature-based SER analysis and design of logic circuits. *IEEE Transactions on Computer-Aided Design, 28*(1), 74–86.
29. Kupferman, O., & Vardi, M. Y. (1997). Synthesis with incomplete information. In *ICTL* (pp. 91–106).
30. Lee, E. A., & Seshia, S. A. (2015) *Introduction to embedded systems: A Cyber-physical systems approach* (2nd ed.). LeeSeshia.org
31. Leveugle, R., Calvez, A., Maistri, P., & Vanhauwaert, P. (2009). Statistical fault injection: Quantified error and confidence. In *DATE* (pp. 502–506).
32. Maniatakos, M., & Makris, Y. (2010). Workload-driven selective hardening of control state elements in modern microprocessors. In *VTS* (pp. 159–164).
33. Mukherjee, S. S., Weaver, C. T., Emer, J. S., Reinhardt, S. K., & Austin, T. M. (2003). A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *MICRO* (pp. 29–42).
34. Nachmanson, L., Veanes, M., Schulte, W., Tillmann, N., & Grieskamp, W. (2004). Optimal strategies for testing nondeterministic systems. In *ISSTA* (pp. 55–64).
35. Nicolaidis, M. (2010). Design techniques for soft-error mitigation. In *ICICDT* (pp. 208–214).
36. Offutt, A. J. (1992). Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology, 1*(1), 5–20.
37. Pnueli, A. (1977). The temporal logic of programs. In *FOCS* (pp. 46–57).

38. Pnueli, A., & Rosner, R. (1989). On the synthesis of a reactive module. In *POPL* (pp. 179–190).
39. Queille, J.-P., & Sifakis, J. (1982). Specification and verification of concurrent systems in CESAR. In *PROGRAM* (pp. 337–351).
40. Riener, H., Könighofer, R., Fey, G., & Bloem, R. (2016). SMT-based CPS parameter synthesis. In *ARCH@CPSWeek* (pp. 126–133).
41. Saltarelli, P., Niazmand, B., Hariharan, R., Raik, J., Jervan, G., & Hollstein, T. (2015). Automated minimization of concurrent online checkers for network-on-chips. In *ReCoSoC* (pp. 1–8).
42. Saltarelli, P., Niazmand, B., Raik, J., Govind, V., Hollstein, T., Jervan, G., et al. (2015). A framework for combining concurrent checking and on-line embedded test for low-latency fault detection in NoC routers. In *NOCS* (pp. 6:1–6:8).
43. Seshia, S. A., Li, W., & Mitra, S. (2007). Verification-guided soft error resilience. In *DATE* (pp. 1442–1447).
44. Solar-Lezama, A. (2013). Program sketching. *International Journal on Software Tools for Technology Transfer, 15*(5–6), 475–495.
45. Sourdis, I., Strydis, C., Armato, A., Bouganis, C.-S., Falsafi, B., Gaydadjiev, G. N., et al. (2013). DeSyRe: On-demand system reliability. *Microprocessors and Microsystems, 37*(8-C), 981–1001.
46. Steffens, L., Fohler, G., Lipari, G., & Buttazzo, G. (2003). Resource reservation in real-time operating systems – a joint industrial and academic position. In *ARTOSS* (pp. 25–30).
47. ter Braak, T. D. (2014). Using guided local search for adaptive resource re-servation in large-scale embedded systems. In *DATE* (pp. 1–4).
48. ter Braak, T. D. (2016). *Run-Time Mapping: Dynamic Resource Allocation in Embedded Systems*. Ph.D. thesis, University of Twente.
49. ter Braak, T. D., Toersche, H. A., Kokkeler, A. B. J., & Smit, G. J. M. (2011). Adaptive resource allocation for streaming applications. In *SAMOS* (pp. 388–395).
50. Wan, J., & Kerkhoff, H. G. (2015). Embedded instruments for enhancing dependability of analogue and mixed-signal IPs. In *NEWCAS* (pp. 1–4).
51. Wan, J., Kerkhoff, H. G., & Bisschop, J. (2016). Simulating NBTI degradation in arbitrary stressed analog/mixed-signal environments. *IEEE Transactions on Nanotechnology, 15*(2), 137–148.
52. Yannakakis, M. (2004). Testing, optimizaton, and games. In *LICS* (pp. 78–88).
53. Zambrano, A. C., & Kerkhoff, H. G. (2015). Fault-tolerant system for catastrophic faults in AMR sensors. In *IOLTS* (pp. 65–70).
54. Zhao, Y., & Kerkhoff, H. G. (2016). A genetic algorithm based remaining lifetime prediction for a VLIW processor employing path delay and IDDX testing. In *DTIS* (pp. 1–5).