Designing Reliable Cyber-Physical Systems

Overview associated to the Special Session at FDL'16

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

[*]German Aerospace Center, Germany [†]Graz University of Technology, Austria [§]IBM Research Lab, Israel
[‡]Recore Systems, The Netherlands [††]Tallinn University of Technology, Estonia [**]Testonica Lab, Estonia
[‖]University of Twente, The Netherlands

***Abstract*—CPS, 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 paper provides an overview of techniques discussed in the special session to tackle these challenges throughout the stack of layers of the computing system while tightly coupling the design methodology to the physical requirements.

I. Introduction

Cyber-physical systems (CPS) [1] 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 overview paper for the related special session 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.

We consider CPS at four layers of abstraction shown in Figure 1. The *analogue/mixed-signal 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 Section II 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

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

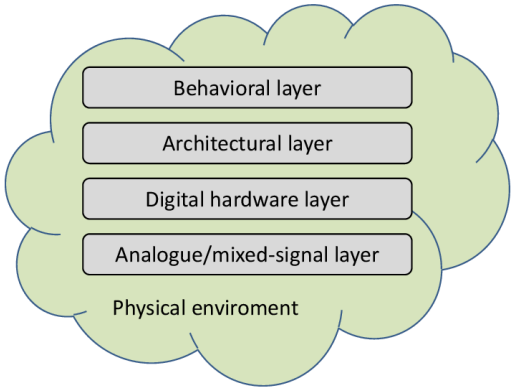

Figure 1. The stack of layers of a CPS.

of the analogue/mixed-signal layer are abstracted to binary values. During operation of the CPS, even correctly designed 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 Section III 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, Section IV 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 V 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 VI 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 modelled as behavioral descriptions of the system's functionality and can be either realized in software or hardware. In Section VII 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 adept to different implementations. In Section VIII we present an approach to auto-

matically 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 meet, eliminating the need for manual error prone decisions.

II. Health Monitoring at the Analog/Mixed Signal Layer

CPSs 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) [2], 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 [3] while in analog/mixed-signal systems several key system-performance parameters will change, like for instance the offset voltage in OpAmps and data converters [4]. Experiments have also shown that drift of sensors [5] and actuators are often key parameters to cause faulty behaviour 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) [6] and key performance parameters [4], 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 order to obtain highly dependable CPS, which includes reliability, availability and maintainability [7], 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 [3]. Useful HMs for the digital cores have shown here to be I_{DDQ} and I_{DDT} embedded instruments as well as delay monitors.

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 on-line *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 behaviour. Using our new analogue/mixed-signal NBTI model in our local designs of 65nm and 40nm TSMC OpAmps and SAR-ADCs, higher-level system key parameters were derived which were used subsequently in a Matlab environment. Figure 2 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.

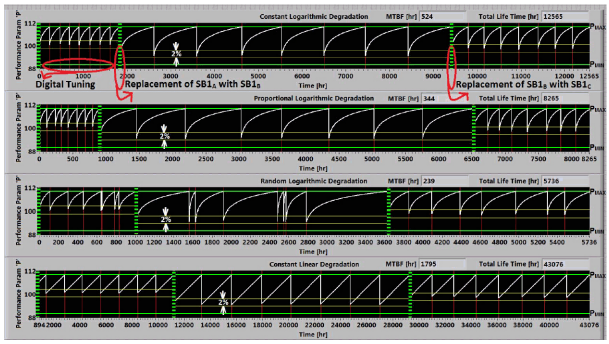

Figure 2. Simulation of a redundant and digital IP tuning platform for highly dependable mixed-signal CPS system for four different degradation scenarios.

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.

III. Comprehensive and Scalable RT-level Reliability Analysis

The dependability of CPSs 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 runtime 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* [8].

A common approach to achieve resiliency against soft errors adds circuitry to automatically detect or even correct such errors [9]. This can be achieved by including redundancy, e.g., in the form of parity bits or more sophisticated error detection or correction codes [8]. Soft error reporting in the RT level can be done, e.g., by using *error checkers*. Once a soft error is reported via a checker, it is up to the *Fault Management Infrastructure* (FMI) to decide how to react to this transient fault.

The ability to understand the reliability of a given hardware component in a CPS thus becomes a key aspect during the design phase of those components. 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. Those methods can be roughly classified into two categories: simulation-based methods, e.g., [10], [11], [12], [13], and formal methods, e.g., [14], [15], [16]. At the heart of the simulation-based methods lies the concept of error injection. In this approach the design is simulated and verified whether it is robust enough 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 pra in this approach is to perform formal verification using a fault m which models single event upsets. Being applied monolithically approach suffers from capacity limits inherent to formal verific methods which makes it impractical in many real-life industrial c

Many hardware mechanisms used for soft error protection local in nature. For example, parity-based protection, Error Corre Code (ECC) logic and residue checking mechanisms are all exam of design techniques aimed at protecting relatively small parts i design, referred to as *protected structures*. An *error detection sign* a Boolean expression that is True when an error occurred. A *prote structure* consists of an error checker fed by error detection sig of *protected sequential elements* and of various *gating condi* 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.

We take advantage of the locality of the protected structures and propose a novel approach for reliability analysis and verification, a basic version of which presented in [17]. We separate the reliability verification process to 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 latch is protected by a certain checker under certain gating conditions [17]. Since each protection is local in nature, applying formal techniques is scalable. Other aspects could require dynamic simulation; for example, proving that the gating conditions are not over gating the protected structure. In the following we provide an overview of our new approach, and give a glimpse as to the technical "how".

A. Analysis stage

The analysis stage identifies the protected structures and is divided into two sub-stages. The *error detecting signals identification* stage and the *structural analysis* stage.

Error detecting signals identification. In this stage the building blocks of the error detection and correction logic are identified. To this end, we use the error checkers as anchors and employ formal and dynamic methods to accurately and efficiently identify various error detection logic constructs. An example for parity checking identification is described in [17]. Other checking logic that can be identified accurately and locally in this stage is, e.g., residue and one-hot checking logic. Error correction code, however, need not be connected to error checkers. To detect ECC computations we rely on the fact that we are looking for linear ECC, hence a computation of the form $v = Au$ for vectors v, u and computations over $\mathbb{Z}/2$. To that end, we iterate over all the vectors in the design and determine whether the bits in that vector are the roots of a XOR-computation tree as described in [17]. After discovering an ECC-like matrix we first purge non ECC instances (such as the case of the identify matrix, or matrix with too many one-hot columns indicating 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 for each error detecting signal what is the set of sequential elements it protects, either by reporting to an error checker or by performing ECC. The challenge here is twofold:

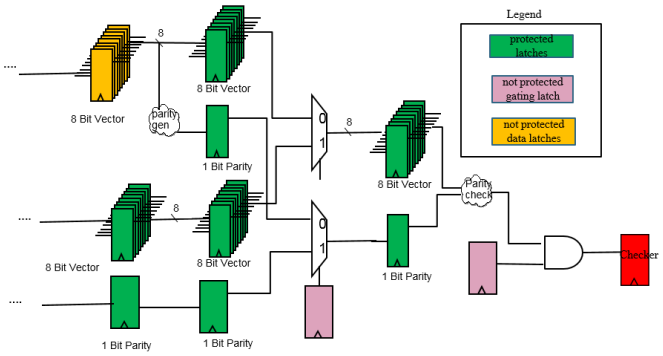

Figure 3. A parity protected structure.

- 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 latches.
- Properly identifying the corresponding gating logic.

For example, in Figure 3 the rightmost pink latch is the gating condition of the error detecting signal - an erroneous parity check will make the error checker fire only if the value of the latch is 1 - and this latch is not part of the protected structure. Also, the yellow bus is located before the parity generation logic and thus is not part of the protected structure either. In order to cope with these challenges, the analysis is performed in word level. In the word level it is easier to keep track of the high level picture and distinguish, e.g., between the data and the gating than in the more common bit level computation model. Due to lack of space, the full algorithm for detecting the protected structure will not be provided here.

B. Verification stage

This stage is based on the analysis stage; namely, the classical problem of reliability verification is reduced to a much easier problem of verifying that the given protection constructs indeed protect the latches they are supposed to protect. The verification that is required here has two aspects (a) verifying that under the relevant gating conditions the relevant latches are indeed protected by the relevant error detections signals or error correction logic; (b) verifying that the gating conditions are not overgating, and will not prevent a checker from firing when it should, causing silent data corruption.

For the former, formal verification can be used, taking advantage of the locality of each protection structure. In [17] we perform it for simple parity protection. Some research is still required to expand the approach from [17] to more protection types and more complex parity structures. In addition, some research has to be done for finding an efficient method to verify that the gating conditions are not over gating. However, clearly the problem of verifying that given gating conditions are not overgating a given protection is dramatically easier than the classical reliability verification problem.

IV. Qualification and Minimization of Concurrent On-Line Checkers

Besides standard approaches for fault detection we also consider advanced error detection schemes on the digital hardware layer for CPS. Particularly, the proposed on-line checkers enable cost-efficient mechanisms for detecting faults during life-time of 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 [18], [19] 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% [18], [19].

V. 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 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. 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.

In the following we present an infrastructure and related data structures to store information about the health status of the system.

A. 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 (see Figure 4).

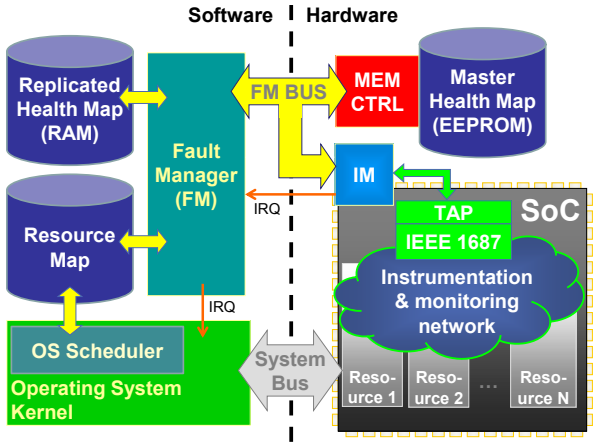

Figure 4. Overview of the fault management infrastructure.

We propose a hierarchical in-situ FMI 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.

The *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. In case if the resource usability is affected, FM must also update the resource map.

The *Instrument Manager* (IM) is a hardware module which is responsible for the communication with the instruments through IJTAG network. Whenever FM needs to access the instruments to get the diagnostic information, it gives a read/write command to IM which in turn opens the path to the instrument through the hierarchical IJTAG network and performs the requested operation.

The HM is a data structure in a dedicated memory which holds the detailed information about the faults and the fault statistics. The HM is the runtime model of CPS including fault monitors and implements

a structural view of the system's hardware resources and its important parts identified by the static (design time) analysis. The HM augments the information about detected faults with the statistical information (e.g. number and frequency of occurrences). To retain the information about the known faults in the system when the system is powered off, the HM should be stored in a reliable non-volatile memory to maintain the prediction capability across the power cycles.

The *Resource map* (RM) is a data structure in the system memory which holds the information about the currently available (healthy) resources of the system. It should be modified on the fly during system's normal operation, should a fault be detected by an instrument or a diagnostic routine. The initial state of the RM (from factory, with all resources operational) can be stored in ROM and serve as a starting point for the initialization during system start-up. During the initialization, information from the HM about the known resource faults is transferred to the RM. The RM is used by the OS scheduler to check which resources are available.

VI. 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 later allows anticipating and reconfiguring before faults occur. Second, the health information is lifted and abstracted to augment run-time resource management software [20], [21] with information about hardware resources to be used less or entirely avoided by reconfiguring the way tasks and communications are mapped to resources.

In [22], [23] and [24] 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 [20], [21] is made health-aware. Figure 5 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

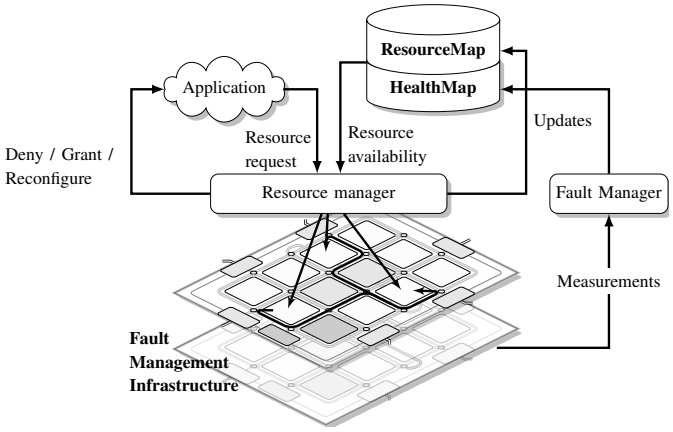

Figure 5. Health information in resource management.

(the domain) onto a fixed range of values. So, the health data stored in the HM is modelled 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 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 Figure 5 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 case 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 levelling. The cost function is designed to assign a hardware resource $r \in R$, which should less or

no longer be used according to the HM, increasingly high costs, such that

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

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. A resource may be in used by several applications requiring one or multiple granted resource request 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.

VII. Deriving Adaptive Test Strategies from LTL-Specifications

To obtain confidence in the correctness of a CPS system at the behavioral layer, model checking [25], [26] 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 addtion, the specification focuses on the most important aspects that require intesive testing. Fault-based techniques [27], 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., [28]) 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 runtime. 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 [29].

Figure 6 outlines our proposed testing setup. The user provides a specification φ, expressing requirements for the system under test in *Linear Temporal Logic* (LTL) [30]. 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 [31], 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

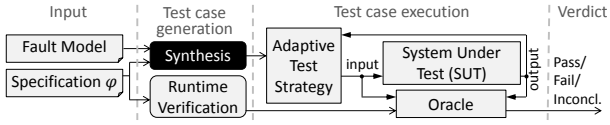

Figure 6. Our testing setup using our approach for test strategy synthesis.

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 [32] with partial information [33], 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 runtime verification [34] can be used 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 runtime to achieve the desired goals. Such adaptive test cases have been studied by Hierons [38] 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 [39]. 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. [40]. Our work builds upon reactive synthesis [32] (with partial information [33]). 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. [41] 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 user defines a class of faults, which can be permanent or transient. 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. 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.

VIII. 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. In parameter synthesis, the logic formulæ describe how the CPS evolves over time from a set

[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 [34]. To overcome this problem, we refere to existing research on interpreting LTL over finite traces [35], [36], [37].

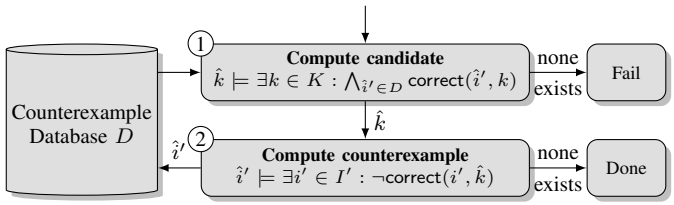

Figure 7. Counterexample-Guided Inductive Synthesis (CEGIS) [42].

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. In the context of CPS, parameter synthesis can be effectively reduced to solving instances of $\exists\forall$-queries over mixed domains. 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 when the variables are quantified over countably infinite or uncountably infinite domains. Different approaches for parameter synthesis for hybrid automata, e.g., [43], [44], [45], 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. In [46], we presented a *Satifiability Modulo Theories* (SMT)-based framework for synthesizing one value for open parameters of a CPS using *Counterexample-Guided Inductive Synthesis* (CEGIS) [42] and introduced the notion of *n-step inductive invariants* to reason about unbounded CPS correctness.

A. 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 $\mathsf{correct} : I' \times K \to \mathbb{B}$, $(i', k) \mapsto \mathsf{correct}(\hat{i}', \hat{k})$, is a correctness formula 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 $\hat{i}' \in I'$. The CEGIS approach repeats two steps to compute parameter values $\hat{k} \in K$ such that $\forall i' \in I' : \mathsf{correct}(i', \hat{k})$ holds. 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 Figure 7. CEGIS 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 $\forall i' \in I : \mathsf{correct}(i', \hat{k})$ considering the whole 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, three outcomes are possible: parameters $\hat{k} \in K$ can be found such that the formula $\forall i' \in I' : \mathsf{correct}(i', \hat{k})$ becomes true (Done), 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 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 typically able to synthesize parameters.

B. n-Inductive invariants

To define the correctness of a CPS, an invariant-based approach is used. By induction, a CPS is safe and correct if:

1) all initial states satisfy the invariant, i.e.,
$$\mathsf{A}(q, k) = \Big(\mathsf{init}(q, k) \to \mathsf{inv}(q, k)\Big),$$
2) all states that satisfy the invariant are also safe, i.e.,
$$\mathsf{B}(q, k) = \Big(\mathsf{inv}(q, k) \to \mathsf{safe}(q, k)\Big),$$
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.,
$$\mathsf{C}(q_0, i_1, \ldots, i_n, k) = \Big(\mathsf{inv}(q_0, k) \to \bigvee_{j=1}^{n} \mathsf{inv}(q_j, k) \wedge \bigwedge_{l=1}^{j-1} \mathsf{safe}(q_l, k)\Big),$$
where q_j is an abbreviation for $T(q_{j-1}, i_j, k)$ for all $j > 0$.

The CPS is correct if parameter values $k \in K$ exist such that
$$\forall q_0 \in Q : \forall i_1, \ldots, i_n \in I : A(q_0, k) \wedge B(q_0, k) \wedge C(q_0, i_1, \ldots, i_n, k)$$
holds. We define the correctness formula $\mathsf{correct}$ by $I' = Q \times I^n$.

C. Heuristics

We implemented to the approach as a proof-of-concept tool based on an SMT solver and developed three simple heuristics to improve convergence of the CEGIS loop presented in Figure 7 applied to CPS, where typically infinite domains are considered.

1) *Counterexample randomization*: To avoid the generation of too similar counterexamples, the proof-of-concept tool 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 serves a simple heuristic to aid 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 synthesis 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 $\mathsf{progress}(\hat{k}_a, \hat{k}_b) = \left\|\hat{k}_a - \hat{k}_b\right\|$. 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.

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.

IX. Conclusions

Within the IMMORTAL project, we have identified several challenging problems in the context of reliabiliy and automated debug considering advanced CPS' throughout the stack of layers and the design flow. For each of these problems we presented in this paper 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 on-going 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] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems — A Cyber-Physical Systems Approach*, 2nd ed. LeeSeshia.org, 2015.

[2] J. Wan, H. G. Kerkhoff, and J. Bisschop, "Simulating nbti degradation in arbitrary stressed analog/mixed-signal environments," *TNANO*, vol. 15, no. 2, pp. 137–148, 2016.

[3] Y. Zhao and H. G. Kerkhoff, "A genetic algorithm based remaining lifetime prediction for a VLIW processor employing path delay and IDDX testing," in *DTIS*, 2016, pp. 1–5.

[4] J. Wan and H. G. Kerkhoff, "Embedded instruments for enhancing dependability of analogue and mixed-signal IPs," in *NEWCAS*, 2015, pp. 1–4.

[5] A. C. Zambrano and H. G. Kerkhoff, "Fault-tolerant system for catastrophic faults in AMR sensors," in *IOLTS*, 2015, pp. 65–70.

[6] G. Ali, A. Badawy, and H. Kerkhoff, "On-line management of temperature health monitors using the IEEE 1687 standard," in *TESTA*, 2016, pp. 1–4.

[7] M. A. Khan, "On improving dependability of analog and mixed-signal SoCs: A system-level approach," Ph.D. dissertation, University of Twente, 2014.

[8] S. S. Mukherjee, C. T. Weaver, J. S. Emer, S. K. Reinhardt, and T. M. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *MICRO*, 2003, pp. 29–42.

[9] M. Nicolaidis, "Design techniques for soft-error mitigation," in *ICICDT*, 2010, pp. 208–214.

[10] M. Maniatakos and Y. Makris, "Workload-driven selective hardening of control state elements in modern microprocessors." in *VTS*, 2010, pp. 159–164.

[11] S. Krishnaswamy, S. Plaza, I. L. Markov, and J. P. Hayes, "Signature-based SER analysis and design of logic circuits." *TCAD*, vol. 28, no. 1, pp. 74–86, 2009.

[12] D. E. Holcomb, W. Li, and S. A. Seshia, "Design as you see FIT: System-level soft error analysis of sequential circuits," in *DATE*, 2009, pp. 785–790.

[13] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence." in *DATE*, 2009, pp. 502–506.

[14] U. Krautz, M. Pflanz, C. Jacobi, H. W. Tast, K. Weber, and H. T. Vierhaus, "Evaluating coverage of error detection logic for soft errors using formal methods," in *DATE*, 2006, pp. 176–181.

[15] S. Frehse, G. Fey, E. Arbel, K. Yorav, and R. Drechsler, "Complete and effective robustness checking by means of interpolation," in *FMCAD*, 2012, pp. 82–90.

[16] S. A. Seshia, W. Li, and S. Mitra, "Verification-guided soft error resilience," in *DATE*, 2007, pp. 1442–1447.

[17] E. Arbel, S. Koyfman, P. Kudva, and S. Moran, "Automated detection and verification of parity-protected memory elements," in *ICCAD*, 2014, pp. 1–8.

[18] P. Saltarelli, B. Niazmand, R. Hariharan, J. Raik, G. Jervan, and T. Hollstein, "Automated minimization of concurrent online checkers for network-on-chips," in *ReCoSoC*, 2015, pp. 1–8.

[19] P. Saltarelli, B. Niazmand, J. Raik, R. Hariharan, V. Govind, T. Hollstein, and G. Jervan, "A framework for combining concurrent checking and on-line embedded test for low-latency fault detection in NoC routers," in *NOCS*, 2015, pp. 6:1–6:8.

[20] P. K. F. Hölzenspies, T. D. ter Braak, J. Kuper, G. J. M. Smit, and J. Hurink, "Run-time spatial mapping of streaming applications to heterogeneous multi-processor systems," *IJPP*, vol. 38, no. 1, pp. 68–83, 2010.

[21] T. D. ter Braak, "Using guided local search for adaptive resource reservation in large-scale embedded systems," in *DATE*, 2014, pp. 1–4.

[22] T. Ahonen, T. D. Braak, S. T. Burgess, R. Geißler, P. M. Heysters, H. Hurskainen, H. G. Kerkhoff, A. B. J. Kokkeler, J. Nurmi, J. Raasakka, G. K. Rauwerda, G. J. M. Smit, K. Sunesen, H. Zonneveld, B. Vermeulen, and X. Zhang, *Reconfigurable Computing: From FPGAs to Hardware/Software Codesign*. Springer New York, 2011, ch. CRISP: Cutting Edge Reconfigurable ICs for Stream Processing, pp. 211–237.

[23] T. D. ter Braak, H. A. Toersche, A. B. J. Kokkeler, and G. J. M. Smit, "Adaptive resource allocation for streaming applications," in *SAMOS*, 2011, pp. 388–395.

[24] I. Sourdis, C. Strydis, A. Armato, C. Bouganis, B. Falsafi, G. N. Gaydadjiev, S. Isaza, A. Malek, R. Mariani, D. N. Pnevmatikatos, D. K. Pradhan, G. K. Rauwerda, R. M. Seepers, R. A. Shafik, K. Sunesen, D. Theodoropoulos, S. Tzilis, and M. Vavouras, "DeSyRe: On-demand system reliability," *MICPRO*, vol. 37, no. 8-C, pp. 981–1001, 2013.

[25] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *LOP*, 1981, pp. 52–71.

[26] J. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *PROGRAM*, 1982, pp. 337–351.

[27] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *TOSE*, vol. 37, no. 5, pp. 649–678, 2011.

[28] G. Fraser, F. Wotawa, and P. Ammann, "Testing with model checkers: A survey," *STVR*, vol. 19, no. 3, pp. 215–261, 2009.

[29] ——, "Issues in using model checkers for test case generation," *JSS*, vol. 82, no. 9, pp. 1403–1418, 2009.

[30] A. Pnueli, "The temporal logic of programs," in *FOCS*, 1977, pp. 46–57.

[31] A. J. Offutt, "Investigations of the software testing coupling effect," *TOSEM*, vol. 1, no. 1, pp. 5–20, 1992.

[32] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *POPL*, 1989, pp. 179–190.

[33] O. Kupferman and M. Y. Vardi, "Synthesis with incomplete informatio," in *ICTL*, 1997, pp. 91–106.

[34] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *TOSEM*, vol. 20, no. 4, p. 14, 2011.

[35] K. Havelund and G. Rosu, "Monitoring programs using rewriting," in *ASE*, 2001, pp. 135–143.

[36] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *IJCAI*, 2013.

[37] G. De Giacomo, R. D. Masellis, and M. Montali, "Reasoning on LTL on finite traces: Insensitivity to infiniteness," in *AAAI*, 2014, pp. 1027–1033.

[38] R. M. Hierons, "Applying adaptive test cases to nondeterministic implementations," *IPL*, vol. 98, no. 2, pp. 56–60, 2006.

[39] M. Yannakakis, "Testing, optimizaton, and games," in *LICS*, 2004, pp. 78–88.

[40] R. Alur, C. Courcoubetis, and M. Yannakakis, "Distinguishing tests for nondeterministic and probabilistic machines," in *STOC*, 1995, pp. 363–372.

[41] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp, "Optimal strategies for testing nondeterministic systems," in *ISSTA*, 2004, pp. 55–64.

[42] A. Solar-Lezama, "Program sketching," *STTT*, vol. 15, no. 5-6, pp. 475–495, 2013.

[43] G. Frehse, S. K. Jha, and B. H. Krogh, "A counterexample-guided approach to parameter synthesis for linear hybrid automata," in *HSCC*, 2008, pp. 187–200.

[44] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "Parameter synthesis with IC3," in *FMCAD*, 2013, pp. 165–168.

[45] S. Bogomolov, C. Schilling, E. Bartocci, G. Batt, H. Kong, and R. Grosu, "Abstraction-based parameter synthesis for multiaffine systems," in *HVC*, 2015, pp. 19–35.

[46] H. Riener, R. Könighofer, G. Fey, and R. Bloem, "SMT-based CPS parameter synthesis," in *ARCH*, 2016.