IMMORTAL

Integrated Modelling, Fault Management, Verification and Reliable Design Environment for Cyber-Physical Systems

Type of Action: Research and Innovation Action (RIA)
Topic: Smart Cyber-Physical Systems
Grant Agreement no: 644905
Website: www.h2020-immortal.eu

Status on Modeling, Fault/Reconfiguration Modeling and Reliability Metrics (Deliverable D1.1)

Start date of project: March 1, 2015
Duration: Three years

Revision 1.3

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
<td>Public, fully open</td>
</tr>
<tr>
<td>CO</td>
<td>Confidential, restricted under conditions set out in Model Grant Agreement</td>
</tr>
<tr>
<td>CL</td>
<td>Classified</td>
</tr>
</tbody>
</table>
Notices
For information, contact e-mail: info@h2020-immortal.eu.
This document is intended to fulfil the contractual obligations of the IMMORTAL project concerning deliverable D1.1 described in contract number 644905.
© Copyright IMMORTAL 2015-2017. All rights reserved.
Table of Revisions

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Description and reason</th>
<th>Author</th>
<th>Affected sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>August 17, 2015</td>
<td>Contents Created</td>
<td>H. Riener</td>
<td>All</td>
</tr>
<tr>
<td>1.1</td>
<td>November 2, 2016</td>
<td>Revised soft error part</td>
<td>S. Moran</td>
<td>Ex. Sum., Section 3.3</td>
</tr>
<tr>
<td>1.2</td>
<td>December 20, 2016</td>
<td>Added information about HM, RM and reconfiguration</td>
<td>T. T. Braak</td>
<td>Ex. Sum., Section 4.3</td>
</tr>
</tbody>
</table>

Authors, Beneficiary

Görschwin Fey, DLR
Heinz Riener, DLR
Siavoosh Payandeh Azad, TUT
Jaan Raik, TUT
Roderick Bloem, TUG
Robert Könighofer, TUG
Hans Kerkoff, UT
Eli Arbel, IBM
Shiri Moran, IBM
Sergei Devadze, TL
Arthur Jutman, TL
Timon ter Braak, RS
Kim Sunesen, RS
Gerard Rauwerda, RS

About

This document is a status report on modeling, fault reconfiguration modeling and reliability metrics.

List of Abbreviations

AMS - Analog and Mixed Signal
CMOS - Complementary Metal-Oxide Semiconductor
CPM - Critical Path Monitor
CPS - Cyber-Physical System
CPU - Central Processing Unit
DSP - Digital Signal Processing
ECU - Electronic Control Units
FIR - Fault Isolation Registers
FIT - Failure in Time
HDL - Hardware Description Language
ICL - Instrument Connectivity Language
IEEE - Institute of Electrical and Electronics Engineers
IJTAG - Internal JTAG
IP - Intellectual Property
JTAG - Joint Test Action Group
MEU - Multiple Event Upset
MTBF - Mean Time Between Failure
NBTI - Negative-Bias Temperature Instability
OBC - On-Board Computer
RAM - Random Access Memory
RTL - Register-Transfer Level
SEU - Single Event Upset
WP - Work Package
# Table of Contents

Table of Revisions ................................................................. iii
Authors, Beneficiary ............................................................... iii
About....................................................................................... iii
List of Abbreviations ............................................................. iii
Table of Contents .................................................................... v
Executive Summary ............................................................... vi
1 Introduction .......................................................................... 1
2 CPS Model ............................................................................ 1
  2.1 Conceptual Overview of CPS ......................................... 2
  2.2 Cross-Layer CPS Model .................................................. 3
3 Fault Model .......................................................................... 11
  3.1 Fault Terminology .......................................................... 11
  3.2 Fault Modeling ............................................................... 12
  3.3 Robustness to Soft Errors ............................................... 13
4 Reconfiguration Modeling ...................................................... 15
  4.1 Fault Tracking and Isolation ............................................. 15
  4.2 IJTAG Network .............................................................. 16
  4.3 Fault Reconfiguration ..................................................... 17
5 References ............................................................................ 21
Executive Summary

This first report of WP1 on Cyber-Physical System (CPS) modelling gives a status of the early work done at M6 including discussions on the baselines and aims beyond the state of the art. The report covers all task of WP1 and the executive summary is organized based on the task break-down: CPS model (Task 1.1), Fault model (Task 1.2), and Reconfiguration modelling (Task 1.3).

CPS Model

Typically, CPSs are modeled on a high level as a hybrid automaton [14] and on a lower more implementation specific level for example as RTL design. In IMMORTAL, we strive to provide a cross-layer modeling framework, that allows to use information from one layer at other layers or even to model different parts of the CPS at different layers. Previous work did not consider such a cross-layer modeling of CPS. Even for pure digital systems, which do not interact with the physical environment, such cross-layer frameworks for reliability estimation and improvement are a new research direction [27]. A cross-layer reliability framework can reduce the product-cost by reducing over-engineering, reducing time-to-market and can increase the overall reliability [6]. In our framework we are using state-of-the-art models at the different layers and provide formalism for their combination. For the CPS model we are considering four layers on which the CPS or parts of the CPS can be modeled. This cross-layer modeling allows to model different parts of the CPS at different levels of detail, and thus to model currently considered parts of design in more detail, while still being able to consider the CPS as a whole. The four layers of our framework are:

Application layer. On this layer the CPS is modeled as a set of tasks, which are described in a high level language, e.g. C/C++.

Operating System/Firmware Layer. On this level the system is modeled by two graphs. One is the architecture graph that contains the different computation units of the system and their connection. The other is the task graph that describes the inter-dependencies of the different tasks.

Digital Hardware Layer. At this level individual IP cores or small chips are modeled. The description is done using a HDL, e.g., Verilog, or netlists.

Analog/mixed Signal Layer. On this layer, the system’s interfaces to the physical environment are described. In contrast to the digital hardware layer, this includes analog and mixed signal components used in sensors and actuators.

Fault Model

One of the dominant factors affecting hardware reliability are soft errors [23] induced by radiation [29]. Several approaches have been proposed to quantify and analyze the vulnerability of a circuit to soft errors, e.g. [5, 12, 13, 15]. A common reliability metric is failure in time (FIT) [23]; One FIT represents an error in a billion (10^9) hours. There are several approaches for FIT estimation, some pre-silicon and some post-silicon e.g. [5, 15].

The FIT of a circuit is determined by the raw FIT of each memory element specified by the manufacturer and by masking effects [2, 23]. A soft error is masked
when an erroneous voltage transient does not result in a system error. These masking effects are quantified as derating factors. A common way to decrease the derating factors – or increase the reliability – of a given hardware is to add protections against soft errors [23].

In IMMORTAL we will generalize the FIT metrics and derating factors to the entire CPS. In addition, we will propose a methodology to fine grain and tighten the FIT estimation by taking into account knowledge about protection schemes against soft errors that exist in the hardware.

In the area of integrated modeling, verification and fault management for the analogue and mixed-signal parts of cyber-physical systems as such, very little research has been carried out, especially with regard to fault management. The latter has to do with cumulative fault propagation. At the higher hierarchy level, there have been developments in modeling, enabling co-simulation for verification of analogue and digital parts and their interaction. Co-simulators of VHDL parts and VHDL-AMS parts are available (e.g. ANSYS). The modeling of the analog/mixed-signal parts however, is difficult and rare; for aging and variation, no (fault) simulators are available. Models for aging are only available for old digital-only technology. We are investigating new aging models for NBTI for analog/mixed-signal to be incorporated in transistors. Especially the analog nature of stress voltages is important. Based on this, also higher-level (performance) models can be extracted from lower-level simulations. The path of Matlab/Simulink simulations will be followed in first instance, and their link to digitally oriented tools (e.g. Modelsim) or even integrated within Matlab/Simulink.

Reconfiguration Modeling

Rapid emergence of embedded instrumentation as an industrial paradigm and adoption of respective IEEE 1687 standard [16] by key players of semiconductor industry opens up new horizons in developing efficient test, debug and health monitoring frameworks. The IEEE Std 1687 also shortly called IJTAG was initially started as an initiative to standardize access to on-chip embedded instrumentation, like monitors, sensors and checkers as well as DFT (Design-for-Testability) infrastructure, various BIST (Built-In Self-Test) and trace data collection solutions for system and software debug [26]. The IJTAG concept embraces also the paradigm of Reconfigurable Scan Networks (RSN) [3] and has become a very attractive industrial solution for both scan-based manufacturing test and system debug [26, 24, 19]. Operation of IJTAG networks is defined by Capture-Shift-Update (CSU) cycles. Capture, Shift and Update, as well as TCK (test clock) are global signals controlling behavior of all scan registers in IJTAG. The first IJTAG modeling approach proposed in [3] is based on graph representations of RSNs and a so-called CSU-accurate model (CAM), which can be seen as a temporally abstract FSM, where transitions correspond to complete CSU operations. The CAM is directly constructed from the ICL [16]. Most of follow up works are based on the principles described in [3] or similar ones.

In IMMORTAL we both extend the architectural model to include Fault Management specific elements (like FCX bits) and reduce the behavioral complexity from CSU to simple shift operations, which is sufficient for given tasks.

Fault management consists of both prevention and recovery. Faults may be pre-
vented by adding a fault management infrastructure to the architecture to monitor the health of the hardware components. System-wide wear leveling may then increase the lifespan of the system. Both wear leveling and fault recovery require a reconfiguration infrastructure that is able to reason about resources on the scale of complete applications and architectures in order to (re)compute configurations at run-time. In IMMORTAL we are introducing the paradigm of reservation based resource partitioning to regulate the access of the application to the hardware resources. Having a model of both the hardware and the application available in the system is required to enable the mapping, routing and scheduling algorithms.

Task scheduling on dual-core processors at OS level with deterministic communication behavior is addressed in [1]. The work does not consider many-core systems based on Network-on-Chip targeted in the scope of IMMORTAL project. Moreover, the work looks into faults at tasks instead of being detected at hardware level. This will result in having faults that will not manifest themselves in the application layer in any of the considered task failure scenarios (i.e. deadline misses, execution time delays or task sequence errors). Having less information about the faults at the hardware level will make it more difficult for the system to make proper scheduling decisions. Also the above mentioned fault detection at application layer, will result in propagation of un-noticed corrupt data and huge latency in the fault detection which in turn will result in more costly recovery in the system (i.e. recovery after propagation of the fault to other parts of the system). PharOS is considering the hardware memory protection mechanisms to provide error correction. However, the faults within data and control path of processors are not considered.

In contrast to the above, in IMMORTAL we target a multi-layer fault resilience approach to target the non-deterministic nature of the network-on-chips as a communication medium. The proposed approach gathers information from hardware, using checkers and monitors and performs online fault classification. The classified fault information would later on be abstracted to be signaled to the application layer. The hardware approach will be used in a hybrid approach along with software based testing on the processors and will provide a complete overview of system’s health (System Health Map) which will be in turn utilized in task scheduling and mapping. Furthermore, in the IMMORTAL project, we aim to isolate and repair the effects of faults in the system instantly once they are discovered by the checkers and monitors in order to prevent their propagation to other system components and reduce the chance of a full system failure.
1 Introduction

*Cyber-physical systems* (CPS) [20] are smart systems that integrate computing and communication capabilities with the monitoring and control of entities in the physical world dependably, 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. The main aim of IMMORTAL is to close this gap by introducing reliable design and automated system debug for CPS. This is achieved by developing a cross-layer CPS model in addition with a holistic fault model that conjointly allow for the detection, isolation and recovery from faults for many-core based CPS networked architectures. IMMORTAL targets reliability on multiple layers including device, circuit, network, architecture, firmware and software and focuses on fault effects from different error sources ranging from design bugs via wear-outs and soft errors towards environmental uncertainties and measurement errors.

This status report describes the work-in-progress state of the cross-layer CPS model and the holistic fault model towards reliability and reconfiguration. The two models are developed within WP1 and can be seen as the “backbone” of the IMMORTAL infrastructure. The models serve as the basis for other work packages and are, thus, eligible for enabling effective reasoning (WP2), successful reliability design (WP3), and fast fault management and reconfiguration (WP4). In particular, this report aims at three objectives:

1. Description of the current state of the cross-layer CPS models (time-triggered application and hardware models) within IMMORTAL with an emphasis on a conceptual overview of the IMMORTAL infrastructure.
   (Associated to T1.1, completion: M12, final report: D1.2)

2. Description of the current state of the fault model with an emphasis on reliability metrics to quantify the vulnerability of individual parts of the system to failure with respect to the cross-layer CPS model.
   (Associated to T1.2, completion: M18, final report: D1.3)

3. Description of the fault management and recovery infrastructure with focus on fault detection, isolation and recovery.
   (Associated to T1.3, completion: M12, final report: D1.4)

The remainder of this report is structured as follows: in the sections 2, 3 and 4, the current state of the cross-layer CPS model, the holistic fault model and the reconfiguration modeling are presented, respectively.
2 CPS Model

This section describes the current state of the cross-layer CPS model that is under development in IMMORTAL. Firstly, Section 2.1 gives a conceptual overview of CPSs and describes them from application perspective in the context of closed-feedback loop control systems. Secondly, in Section 2.2 the cross-layered CPS model is presented consisting of four layers. A simplified version of a temperature control application of an on-board computer (OBC) system is introduced which serves as an example.

2.1 Conceptual Overview of CPS

The concept of a CPS has recently emerged as a model that tightly integrates digital computation with its physical environment. A discrete system continuously interacts with its physical environment via sensors and actuators such that CPS describe closed-feedback loop systems. From the application perspective, this reflects the view of traditional control theory, where a feedback or closed-loop control system consisting of a digital controller is interfaced with its physical environment (plant). Such systems are typically described with box diagrams as shown in Figure 1. The objective of the controller is to control a physical signal $y$ produced by the plant with an actuator signal $u$ in such a way that it follows a reference signal $r$. The controller monitors $y$ and compares the signal to the reference to adapt $u$ based on the differences.

![Figure 1: Digital controller interfaced with its physical environment](image)

Control system theory deals with the design and analysis of control systems with an emphasis on the stability and the robustness of the system considering uncertainties of parameters and the measurement. Classical control system theory, however, abstracts from the execution environment and the communication interfaces of the system, assuming that the digital controller runs as a single task on a central processing unit (CPU) that instantaneously interacts with its physical environment.

For today’s computer systems, this abstraction is typically too coarse. For instance, consider an OBC of a satellite system, where multiple concurrent control tasks are simultaneously scheduled by an operating system on a multi-core CPU. The tasks may include thermal control for keeping the temperature in the satellite at a level
that does not destroy any component, guidance and navigation control for tracking and adapting the position as well as orientation of the satellite, and space-to-ground communication, i.e., executing telecommands coming from the ground and sending telemetry data containing the status of the satellite including the scientific experiments. Moreover, the OBC may be interfaced with sensors and actuators via complex network topology and involves non-trivial protocols and communication overhead. The correctness of such a control system does not only depend on the controller interfaced with its physical plant, but on technical details of the execution environment like the scheduling imposed by the operating system and the timing of the underlying hardware, not modeled in traditional control system theory.

The design and development of such complex systems, furthermore, becomes more cumbersome when individual components are developed by separate teams and exact timing constraints are only available when all components are integrated, such that system level verification and validation is highly important.

CPS extend existing embedded systems considering computation deeply integrated into a physical environment and, thus, allows to model control application beyond the traditional perspective of control system theory. Existing CPS frameworks, however, do not deal with reliability or system level debug aspects.

2.2 Cross-Layer CPS Model

In IMMORTAL, a cross-layer CPS model is developed with an emphasis on reliable design and automated system debug of CPS. The work-in-progress CPS model consists of four (modeling) layers, where each layer offers different capabilities to detect, isolate and recover from faults. Within IMMORTAL, a CPS may be completely described on one layer or parts of the CPS are described in detailed on different layers, depending on the actual application and fault detection, isolation and recovery mechanism used.

In the following, the four layers are shortly described:

1. The top most layer, the application layer, describes the behavior of the CPS algorithmically on a high abstraction level without providing details on the actual technical realization. This abstraction level is suitable for early design stages, where the system specification is still vague and a multitude of different design choices is possible. For instance, on this layer, the separation of software and hardware functionality, the exact clock-level timing of the hardware, or the communication infrastructure and channels may not be reflected in detail. Thus, many high level programming languages, e.g., ANSI-C, C++, SystemC or Matlab/Simulink are suitable to provide a system level model.

2. On the operating system/firmware layer, the CPS is described in terms of tasks that are executed periodically on an abstract hardware platform. A
Figure 2: Control task (in SW) interfaced with its physical environment

task is an arbitrary chunk of application functionality that may be realized as special-purpose hardware or implemented as software executed on a CPU. This layer is used to describe system-wide reconfiguration after faulty computing elements or broken links have been detected; and requires that the overall hardware architecture is known.

3. The digital hardware layer concentrates on the description of individual intellectual property (IP) cores or smaller standalone chips. On this layer, a hardware building block may be described as a gate level circuit or as register-transfer level (RTL) code utilizing hardware description languages (HDLs), e.g., Verilog or VHDL.

4. The analog/mixed signal layer focuses on the description of interfaces to the physical environment. This includes analog components, e.g., sensors, actuators, amplifiers or filter, as well as mixed signal components, e.g., data converters or phase-locked loops. On this layer, the analog and mixed signal (AMS) extensions of HDLs are used, e.g., Verilog-AMS or VHDL-AMS.

**Temperature Control as CPS.** A temperature control application of an emergency heating system as a part of an OBC serves as a simplified version of a real-world use case that tightly integrates a safety-critical digital control system with its physical environment. The objective of the system is to maintain the temperature of the battery of a satellite in an operational range such that power supply of the overall system can be guaranteed. The control application is implemented in software as a set of interacting tasks and executed periodically by a real-time operating system with a general-purpose CPU on a single core. The interface to the physical environment is realized via a temperature actuator, e.g., a $10\Omega/35W$ resistor used as heater, and a temperature sensor, e.g., AD590, interfaced to the OBC with point-to-point communication links. The links are realized as SpaceWire (SpW) [8] communication networks and the communication follows the Remote Memory Access Protocol (RMAP) [9, 10, 11], which defines a standard for reading and writing to registers and memories within a SpW unit. Figure 2 shows the hardware architecture of the described CPS.

The environment of the CPS describes the physics of the analog plant controlled by the digital system. Many formalisms are suitable to describe the physical behavior depending on the application and the physical entity actually modeled. For this example, the behavior of the analog plant is described in terms of the hybrid automaton depicted in Figure 3. The automaton describes the physics of the environment as differential equations in two different discrete modes (heating and
heating $\dot{\tau} = K$
cooling $\dot{\tau} = -k$

Figure 3: Hybrid automaton that models the physics of analog temperature.

cooling) assuming that the initial temperature $\tau \in [m, M]$ with constants $m \leq M$. The temperature $\tau$ increases with the constant rate $K$ when the heater is turned on, whereas the temperature $\tau$ decreases with the constant rate $k$ when the heater is turned off. The exact rates $K$ and $k$ are determined by characteristics of the temperature sensor and temperature actuator. The transitions between the two modes are modeled as non-deterministic choices, which are externally controlled by the digital system by a discrete signal that enables or disables the heater.

The simple architecture from Figure 2 can be easily extended to the scalable CPS platform in Figure 4, which can be found in many automotive and aerospace applications. Figure 4 shows two computing elements connected via a common bus and interfaced with their physical environment via sensors and actuators. The design of the individual interfaces depends on the exact communication protocol in use and may contain random access memory (RAM) blocks, protocol-specific transmitter/receiver IP cores as well as data converters to communicate with the sensors and actuators. The kernel of a computing element is a CPU, which takes care of control tasks, but may also perform digital signal processing (DSP) calculations, often in real-time. In real-world applications, many more computing elements and sensors/actuators may be employed, which are often termed electronic control units (ECU) or OBCs.

Application Layer

On the application layer, the CPS is understood by its functionality as a control application, i.e., a hardware/software system that controls a physical entity of its environment through sensors and actuators. On this layer, there is no explicit separation between functionality realized in hardware and software. The behavior of the CPS is described uniformly in a high level programming language and thus enables modeling of the CPS on a high abstraction level in early design stages.

Figure 5 shows a C++-like pseudo code implementation that models the cybernetics of such a CPS and its interfaces to the physical environment via classes TemperatureSensor and TemperatureActuator. The classes may also be implemented in pseudo code or modeled externally utilizing suitable libraries or a specific modeling environment, e.g., Simulink, interfaced with the program. In contrast to standard purpose software, a time- or event-based synchronization mechanism is necessary to exchange data between the cybernetic and the physical system part. In the example, a common and global clock is used that is accessed and manipulated with a specific API, e.g., the TIME_LOOP macro. Switching between the environment modes is signaled by the method calls setHeatBatteryOn and setHeatBatteryOff and could be either kept abstract or modeled using the
respective API. Figure 6 describes the \texttt{setHeatBatteryOn} method utilizing the SpaceWire Light API [28].

**Operating System/Firmware Layer**

On the operating system/firmware layer, a graph-based modeling framework that is widely used in the design of networked multiprocessor systems, e.g., in [21, 4], is adapted. The framework allows to reason about the configuration and reconfiguration of an application in a CPS and consists of two main components. An architecture graph models the hardware architecture of the CPS platform with its computing elements and communication links. A task graph models the application that is supposed to run periodically on the CPS platform, where a task is a chunk of application functionality either realized as hardware or implemented in software and executed by a CPU.

**Architecture graphs.** The architecture graph is a directed graph $AG = (P, L)$, where $P$ are the nodes of the graph and $L \subseteq P \times P$ are directed edges between the nodes. The architecture graph can be cyclic. Every node $p \in P$ represents a...
#include <common.hpp>

class TemperatureSensor { /* ... */ }

class TemperatureActuator { /* ... */ }

class ThermalControl : public Thread
{
    ThermalControl(): Thread("ThermalControl") {}
    void init() { /* ... */ }
    void run() {
        int t = TIME_LOOP(Common::THERMAL_CONTROL_START, 20*SECONDS)
        temperature_t temperature = sensor.getCalibratedValue();
        if (temperature > temperature_to_turn_heater_off)
            actuator.setHeatBatteryOff();
            continue;
        if (temperature < temperature_to_turn_heater_on)
            actuator.setHeatBatteryOn();
            continue;
    }

    TemperatureSensor sensor;
    TemperatureActuator actuator;
};

class TemperatureActuator
{ /* ... */
    spw_handle handle;
    rmap_command set_heat_battery_on_command;
    void setHeatBatteryOn()
    { struct spwl_txbuf *buffer;
        auto status = spwl_reclaim_txbuf(handle, &buffer, SPWL_WAIT);
        if (status != RTEMS_SUCCESSFUL) goto error;
        buffer.data = set_heat_battery_on_command.data;
        buffer.nbytes = set_heat_battery_on_command.length;
        buffer.eop = SPWL_EOP;
        status = spwl_send_txbuf(handle, &buffer, SPWL_WAIT);
        if (status != RTEMS_SUCCESSFUL) goto error;
        return;
    }
    /* error handling */
};

Figure 5: Application layer model of a temperature controller implemented in C++-like pseudo code.

Figure 6: The method setHeatBatteryOn implemented in C++-like pseudo code utilizing the SpaceWire Light API.

computing element and every edge \( l = (p_1, p_2) \in L \) represents a communication link between two computing elements, meaning that \( p_1 \) can send data to \( p_2 \). Both the computing elements and the communication links can be equipped with additional information. We will refer to such additional information as attributes of the architecture graph elements. For example:

- The attribute \( \text{speed}(p) \) may define the computation speed of computing element \( p \in P \).
• The attribute bandwidth(l) may define the bandwidth of a communication link l = (p1, p2) ∈ L between the computing elements p1 and p2.

• Etc.

We do not fix the attributes of the architecture graph elements but leave them open at this point in the project. This makes our modeling framework for task deployment and application-level reconfiguration flexible. Attributes can be added as their need arises in the use cases.

Task graphs. Similar to the architecture graph, a task graph is a tuple TG = (T, C), where T are the graph nodes and C ⊆ T × T are directed edges between them. A task graph is always acyclic. Intuitively, every node t ∈ T represents a time-triggered task of the application that is to be deployed on the CPS platform. Every edge c = (t1, t2) ∈ C represents a data dependency between the tasks t1 and t2. The intuitive meaning of a data dependency c = (t1, t2) is that task t2 needs data from task t1 in order to perform its job. In this sense, the data dependencies impose two requirements. First, they define requirements for the communication network in case the tasks t1 and t2 are run on two different computing elements p1 and p2. Second, a data dependency also imposes task scheduling constraints because task t2 cannot perform its job before the data from t1 is available. As for the architecture graph, we will also equip task graph elements with additional attributes. For example:

• The attribute wcet(t) may define the worst-case execution time of task t ∈ T.

• The attribute info(c) may define the amount of information to be exchanged between the tasks t1 and t2 of the data dependency c = (t1, t2) ∈ C.

• Etc.

Again, we do not fix the attributes but leave them open at this point in the project. The task graph and the architecture graph can be used to compute a desirable task deployment (with respect to some desirability metric) of the application on the CPS platform. This task deployment typically involves the following operations.

• Mapping: answers the following questions: Which task is supposed to run on which computing element? Which data dependency between tasks is realized using which communication link of the platform?

• Routing: computes a configuration of the routing elements in the communication network in order to realize the communication links as required by the mapping.

• Scheduling: defines the starting time for each task and each communication.

Each data transaction on the network would also be considered as a time-triggered task for each network component (link, router components) during mapping and scheduling process. This means that considerations should be made for the possible nondeterminism in the communication medium.

Figure 7a shows an architecture graph for the CPS platform of the temperature controller from Figure 2 and a corresponding scheduled task graph in Figure 7b that is periodically executed, where each node ti, 1 ≤ i ≤ 11, refers to a task.
Digital Hardware Layer

On the digital hardware layer, individual IP cores or small chips are described either as gate level circuits or RTL blocks written in a HDL. The considered designs are synchronous and modeled using state-transition systems, which are widely used, e.g., in [12, 18], to analyze robustness or to reason about reconfiguration of hardware systems. The states are defined by the configurations of the memory elements (latches or flip-flops) of the circuit and the transitions are defined by the combinational logic.

Sequential Circuit. A sequential circuit is defined as $C = (V, E, X, Y, S, N, F, P)$ where

- $G = (V, E)$ is an acyclic directed graph that defines the connections,
- $X = \{x_0, \ldots, x_{n-1}\} \subseteq V$ is the set of primary inputs,
- $Y = \{y_0, \ldots, y_{m-1}\} \subseteq V$ is the set of primary outputs,
- $S = \{s_0, \ldots, s_{l-1}\} \subseteq V$ is the set of present state nodes,
- $N = \{n_0, \ldots, n_{l-1}\} \subseteq V$ is the set of next state nodes,
- $F : V \rightarrow (\{0, 1\}^* \rightarrow \{0, 1\})$ associates a Boolean function $f_v = F(v)$ to a node $v$, i.e., in particular projection functions of variables are assigns to input nodes and present state nodes,
- $P : (V \setminus (X \cup S)) \rightarrow (V \setminus (Y \cup N))^*$ is an ordered tuple of predecessors of $v$ such that $P(v) = \{w_0, \ldots, w_{p-1}\}$ describes the input variables of $f_v$.

The sequential circuit model can be either used to describe a circuit on the gate level, where each component refers to a gate or a flip-flop, or on the architectural level, where each component refers to a complex module or a block of memory elements. By considering complex modules as components, multiple faults can be modeled.
State-Transition System. A state-transition system $M = (I, S, T)$ is a triple, where $S$ describes the state space of the system, $I \subseteq S$ its initial states and $T \subseteq S \times S$ the transition relation, such that $T(s, s')$ evaluates to true if state $s'$ is reachable in one clock step from state $s$.

The reliability of a digital component can be analyzed based on the behavior of its state-transition system when a fault is injected.

Analog/mixed Signal Layer

On the analog/mixed signal layer, the systems’ interfaces to the physical environment are described in terms of their hardware building blocks. In contrast to the digital hardware layer, this includes analog and mixed signal components used in sensors and actuators as well as operational amplifiers to amplify analog signals from small and noisy sensors and filters to get rid of other disturbance. On this layer, gate level circuits or HDLs together with their AMS extensions, e.g., VHDL-AMS, are used to define the structure and the behavior of the interfaces.
3 Fault Model

In this section, the current state of the holistic fault model is described. The fault model is designed with respect to the cross-layer CPS model. In Section 3.1, a fault terminology is presented to clarify terms. Section 3.2 is dedicated to fault models and describes which faults are addressed on which layer of the CPS model. Lastly, in Section 3.3 robustness metrics are presented to quantify the vulnerability of the system or individual parts of the system to failure.

3.1 Fault Terminology

The fault vocabulary based on the ISO-2626-1:2011 standard [17] is used which distinguishes the terms fault, failure and error.

- **Fault/Defect**: Abnormal condition that can cause an element, i.e., a system, a part of a system, hardware component, or software unit, to fail (note: permanent, intermittent and transient faults are considered).

- **Failure**: Termination of the ability of an element to perform a function as required (note: incorrect specification is a source of failure).

- **Error**: Discrepancy between a computed, observed or measured value or condition, and the true, specified or theoretically correct value or condition.

A fault may occur due to different physical reasons:

- **Design defects** summarize all faults that may occur during specifying, designing, developing, or fabricating a system including hardware and software.

- **Physical influences**, e.g., mechanical or thermal influences, refer to external impact that permanently harm or damage a system and may lead to permanent malfunctioning, e.g., a broken link or computing element.

- **Ionizing or cosmic radiation** refers to the impact of energetic particles that either temporarily cause glitches and soft errors in the system or permanently change the arrangement of atoms in the crystal lattice. This phenomenon could be increasingly encountered in outer space and during high-altitude satellite or flight missions. A taxonomy of fault types and countermeasures in case of cosmic radiation is outlined in Figure 8.
3.2 Fault Modeling

In IMMORTAL, a holistic fault model is developed that addresses faults from different error sources on the layers of the CPS model described in Section 2. We restrict ourselves to representative faults for each level.

On the application layer, we consider permanent design faults in hardware and software components as well as permanent and transient faults in sensors and actuators. Faults on this layer are either modeled as syntactic changes (mutations) of the source code of a high level description or as additive error signals that add noise to the monitored physical entities of the environment.

On the operating system/firmware layer, we consider broken elements (CPUs or communication links) of the CPS platform as well as aging modeled by degrading performance parameters (such as computation or communication speeds) of specific elements of the CPS platform. Permanent as well as transient faults may occur. Design errors in the software that is running on the CPS platform are not considered.

On the digital hardware layer, we consider permanent faults due to aging and wearouts and transient faults due to soft errors, e.g., caused by cosmic radiation. Permanent faults are modeled on the gate or the register transfer level utilizing, e.g.,
IMMORTAL Status on Modeling, Fault/Reconfiguration Modeling and Reliability Metrics

the stuck-at fault model. For transient faults, the single event upsets (SEU) model for flip-flop/latches as well as the multiple event upset (MEU) model for memories are considered. The SEU model deals only with single bit flips in memory elements. This simplification is typically justified by the fact that soft errors caused by radiation are rare and local events and, thus, a design that is resilient against single bit flips is assumed resilient (to a certain extent) against multiple simultaneous bit flips, too. However, to complement the considerations, we also consider the MEU model.

On the AMS layer, we consider aging of transistors caused by negative-bias temperature instability (NBTI), which leads to an increase in threshold voltage and decrease of drain current depending on temperature and voltage stress.

3.3 Robustness to Soft Errors

One of the main factors affecting hardware reliability is its robustness to soft error effects. In IMMORTAL, we will develop reasoning engines for offline assessment of hardware robustness and susceptibility to soft and hard errors. The analysis engines will consider the SEU as well as MEU models.

In addition, error checkers realized in hardware will be used to dynamically monitor the health of the hardware logic and detect transient, intermittent and permanent faults. We also suggest to incorporate means for monitoring chip performance and the effect of aging thereof using critical path monitors (CPM) [7]. The error checkers, along with the CPM and any other digital/analog mechanism will provide information for a health tracking entity which will be responsible for updating a health map described in Section 4 in detail.

Hardware Reliability Metrics. A common reliability metric is failure in time (FIT) [23]. One FIT represents an error in a billion (10^9) hours. Another common metric for reliability is mean time between failures (MTBF) which can be derived from the FIT.

The FIT metric is additive and the overall failure rate of a system can be obtained by summing the FIT rates of its components, assuming these are independent of each other. Each elementary memory element (e.g. latch) comes with a raw FIT, specified by the manufacturer. However, the actual system FIT is significantly lower than the one predicted from summing the raw FITs of the systems’ different components. This is due to masking effects [23, 2]. A soft error is masked when an erroneous voltage transient does not result in a system error. The following masking effects are considered:

- Electrical masking. [2]. Occurs when the bandwidth of the transient is higher than the cutoff frequency of the CMOS circuit; these transient will be attenuated and sometimes eventually have no effect.
Temporal masking. [2]. Occurs when the result of the transient fault reaches a latch but not at the clock transition where the input value of the latch is captured.

Logical masking [23, 2]. Occurs when the value of a sequential element of the circuit is affected but this element does not affect the output due to a subsequent gate whose output is completely determined by its other input; e.g., where the gate is an `and` and its other input is zero.

Application masking. Occurs when the erroneous value does affect an architectural state or an output of the relevant sub-system unit but eventually does not affect the output of the system; e.g., when the floating point unit output is affected but the application does not use this floating point output.

The effect of these masking effects on the actual system FIT rate is quantified as derating factors. Formally, the derating factor of a certain component is a number in the range [0, 1] representing the probability of it affecting the system in case of a fault in the component. Each of the masking above implies a derating factor and the final derating is the multiplication of all the individual derating factors.

A common way to decrease the derating factors – or increase the reliability – of a given hardware is to add protections against soft errors [23].

In IMMORTAL, we will generalize these metrics and derating factors to the entire CPS, aggregating the unique metrics of the different components comprising the CPS and accounting for reliability of the links between those components. E.g., we will analyze the architectural graph and the relations between different components when computing the application masking. In addition, we will propose a methodology to fine grain and tighten the FIT estimation by taking into account knowledge about protections schemes against soft errors that are exist in the hardware.
4 Reconfiguration Modeling

The fault management and reconfiguration infrastructure is an important element of a dependable CPS platform that enables reliable operation of the system under pressure of transient and permanent faults. The infrastructure developed within IMMORTAL relies on three ingredients: a local fault isolation mechanism to detect faults and relate them to their error sources. A scalable IEEE1687-based (IJTAG) network to collect information from various types of fault sensors and error checkers located in different parts of the system. This network serves as a transport layer which operates independently of the rest of the design and constantly collects and transfers fault and health status information across various levels of the system. The health status is collected and stored in a global health map that can be used to reconfigure the system in case of faults.

This section is organized as follows: in Section 4.1, we describe the fault tracking and isolation mechanism and the basic building blocks of its hardware implementation. Section 4.2 is dedicated to modeling the IJTAG network. Finally, in Section 4.3, the fault reconfiguration based and health map are described.

4.1 Fault Tracking and Isolation

In IMMORTAL, we will construct and support a fault isolation model, starting in a single hardware component up to system level, that supports multiple and distributed hardware components. Our fault isolation model will carefully track and analyze the source of each fault as well as its frequency, thus enabling, e.g. distinguishing between transient soft errors (for which at the most part reconfiguration should suffice) and broken hardware (hard errors) which could require rendering some parts of the system (or in extreme cases the entire system) inoperable and taking appropriate actions. We will extend and befit components and concepts from IBM’s fault mechanism e.g. [22]. The basic hardware building blocks of our fault isolation will be error checkers as well as fault isolation registers (FIRs) and a status register described below in more detail.

• **Error checker.** The basic component indicating that an error occurred somewhere in the system is an error checker; it is a dedicated latch that is being fed from error detection logic (e.g. parity) or sensors and indicates whether some error condition occurred, in which it will turn on and output the logical value 1. Otherwise, the error checker’s default value is 0, indicating no error has occurred. To the most part error checkers are used to indicate some local error condition — e.g. a bit flip in one of the latches in a certain area of a
hardware component. Since error checkers are the most basic components indicating that an error had occurred, and for the most part distinguishing in debugging session between different sources of the same error checkers could be impossible, the location and granularity of error checkers should be chosen with care. Also, each error checker will be associated and configured with the type of action it should trigger. The triggered actions could be one of the following, ordered in increased severity:

- **Informational.** The error does not cause any data integrity or system damaging effects, but assists in debugging.
- **Recoverable.** The error may cause temporary data integrity or system damage, but should be fully recoverable by retrying recent operations.
- **Attention.** The error may cause temporary data integrity or system damage and may or may not be recoverable by retrying recent operations. Each such attention will have an associated followup procedure defined through its respective FIR bits.
- **Checkstop.** The error may cause data integrity and/or system damage and there is no known recovery action. Therefore, errors in this class will drive a request for an appropriate checkstop.

- **Fault isolation registers (FIR).** FIRs are dedicated registers for tracking the faults and determining the required actions. In order to keep track of the various sources of the faults in the system and their roots, a hierarchical FIR structure is maintained; the FIR in the lowest level collect information from the error checkers and FIRs in higher levels collect information from other FIR registers. Each error checker will activate a corresponding FIR bit, and each FIR bit corresponds to the area of the fault; in addition, each FIR bit has a related action (e.g. checkstop). Also, each FIR bit has an additional *whose on first* structure that enables keeping track on the initial cause of each fault — namely, the first error checkers that turned on. The FIR bits are being reset only by the firmware.

- **Status register.** The status register comprises a general status indication, and contains a summary of hardware recovery action.

As said, we will extend these basic building blocks to support an entire CPS. We will integrate the basic hardware building blocks into the health map and enable fault tracking and self healing in the presence of distributed hardware components.

### 4.2 IJTAG Network

The structure of an arbitrary IJTAG network can be precisely described using the *instrument connectivity language* (ICL) which is a part of IEEE1687 standard. The language describes IJTAG scan registers inside a design and their interconnection as well as specifies activation conditions of every scan segment. However, ICL descriptions contain too much structural information and need to be thoroughly
analyzed in order to extract actual network topology and function. The latter complicates functional simulation of the network and the analysis of its performance.

To overcome this problem and effectively model the IJTAG-based fault management network, a graph-based representation is developed. The graph model provides a functional description of the network in a simplified representation and at the same time hides/discards implementation-specific structural details not needed for system modeling and simulation. Moreover, the graph representation could also serve as the source for automated generation of the respective description in ICL language, which could be further used for generating VHDL descriptions of the infrastructure (Figure 9).

We model the structure of a scan network with a directed acyclic graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges. The set of vertices $V$ consists of scan segment nodes $V_S$ and two special nodes $V_{SI}$ and $V_{SO}$. The set $V_S$ embraces all scan cells in the network, while elements of $V_S$ are addressable objects of the scan chain such as scan registers of segment insertion bit registers consisting of one of many scan cells each. Essentially, every node $v_i \in V_S$ has different weight $w_i$ equal to the number of scan cells in that node. In addition, each graph contains two special auxiliary nodes $V_{SI}$ and $V_{SO}$ that denote primary scan input $SI$ and primary scan output $SO$.

The set of edges $E \subseteq V \times V$ represents connections between scan network elements such that an edge $e = (v_i, v_j) \in E$ if and only if the two elements corresponding to $v_i, v_j \in V$ are either directly connected together or can be concatenated through a multiplexer to form a single segment of a scan-path. If a node has more than one successor or in other words subsequent elements, each outgoing edge $e_k$ is labeled with a set of activation conditions $c_k$, which also corresponds to the select signal $sel(v)$ of the corresponding scan segment, which in its turn is issued in agreement with the current state of the corresponding multiplexer. A particular outgoing edge $e_k$ is activated in the current state of the network if and only if each condition $c_l \in C_k$ is satisfied.

An example of a small IJTAG-based fault management scan network and the resulting graph are shown in Figure 10.
4.3 Fault Reconfiguration

Configuring a CPS application is a challenging problem because it involves choosing between many degrees of freedom (e.g., which tasks run where and when using which communication links) under a multitude of constraints and cost metrics. In IMMORTAL, also the problem of reconfiguration is considered, which asks to compute changes of the configuration of a CPS application based on the health status of the underlying CPS platform and the active set of resources, stored in the resource map of the application.

In order to enable fault management and reconfiguration of CPSs, the graph-based modeling framework on the operating system/firmware layer is extended with a third component, which is the health map. Intuitively, the health map stores the health status of the different components of the CPS platform. Reconfiguration algorithms can then react to changes in the health status.

**Health map.** Intuitively, the health map stores the current health status of the different elements of the architecture graph $AG = (P, L)$. For instance, processing elements $p \in P$ may be broken, processing elements $p \in P$ may have become slower due to aging, communication links $l \in L$ may be broken, etc. We will model the health map using additional attributes of the architecture graph elements. For example:

- The attribute $\text{health}(p)$ or $\text{health}(l)$ can define a health factor of a processing element $p$ or link $l$.
- The attribute $\text{broken}(p)$ or $\text{broken}(l)$ can define if a processing element $p$ or link $l$ is currently defective or usable; that is, a defective processor has a health$(p)$ of 0, a defective link has a health$(l)$ of 0, whereas a processor in perfect condition has a health$(p)$ of 1 and a link in perfect condition has a health$(l)$ of 1.
- The attribute $\text{slowdown}(p)$ can define a slowdown factor for the usual speed $\text{speed}(p)$ of a processing element $p$ (e.g., because 1 out of 4 cores of $p$ are broken, or half of the second-level cache was disabled because of a failure).
- Etc.
The health map can change over time. This allows us to model both permanent faults and transient faults. In case of a permanent fault, the health status changes for the worse but does not change back. In case of a transient fault, the health status can also change back to better values. The impact of a change in health status on the application can be determined by analyzing the resource map.

**Resource map.** Upon start-up of an application TG, a set of resources is requested to accommodate its execution on architecture AG. Whenever this request is granted, the application is guaranteed to function correctly. Faulty or degraded hardware thus have to be considered in the allocation of the resources.

- The attribute reserved($t, p$) or reserved($c, l$) can define the amount of resources reserved by the task $t$ and channel $c$ of an application TG on the specified processor $p$ and link $l$ of an architecture AG, respectively.

After a successful resource allocation, the set of resources is dedicated to application TG. No other application is able to prohibit the guaranteed access (duration) to those resources. This paradigm is known as reservation based resource partitioning [25]. The active resource partitioning on the system AG can only be changed with an explicit reconfiguration procedure. This procedure can, for example, be started when the health status of a component exceeds a configurable threshold.

**Reconfiguration.** An application has an initial configuration on the target architecture. Any changes required afterwards demands for a reconfiguration of the application. Two scenarios are to be considered that may cause a reconfiguration:

1. The health status of a processor or link in the architecture is changed, and thereby exceeds a configurable threshold. This implies that the component is no longer usable. Any application that has reserved this component to execute its functionality needs to be reconfigured and/or recovered.

2. The health status of the system as a whole has changed significantly over time. While not urgent, a reconfiguration of one or more applications may improve the efficiency of the system, or may reduce the burden on hardware components with reduced health.

The first scenario demands for a quick reconfiguration to (partially) restore the functionality of the application. In the previously valid resource map, some assignments have become invalid. A new configuration needs to be computed to fill in these gaps, potentially changing other parts of the resource map as well to meet the application constraints, or to work with the actual resource availability. The second scenario can be considered as an in-system optimization procedure. Such a procedure can recompute the active resource reservations with respect to the changes in the health map. The application can then be reconfigured when it has become beneficial to adopt the new configuration.

A deployment under perfect health assumptions may be suboptimal or even impossible if the health status of the platform changes for the worse. In IMMORTAL, we will thus research and develop methods to detect faults online (Task T4.1) and to update the health map accordingly (Task T4.3). Triggered by changes of the health
map, we will then compute reconfigurations of the deployment (i.e., changes of the mapping, routing and scheduling). Algorithms to compute such reconfigurations will be developed in the frame of Task T4.2.
5 References


