H2020-ICT-2014/H2020-ICT-2014-1

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

Integrated demonstrator on modelling, verification and reliability analysis
(Deliverable D5.1)

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

Revision 1.0

Project funded by the European Commission within the Horizon 2020 programme for research, technological development and demonstration (2015-2018)

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th>PU</th>
<th>CO</th>
<th>CL</th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
<td>Public, fully open</td>
<td>☒</td>
<td></td>
</tr>
<tr>
<td>CO</td>
<td>Confidential, restricted under conditions set out in Model Grant Agreement</td>
<td></td>
<td>☐</td>
</tr>
<tr>
<td>CL</td>
<td>Classified</td>
<td></td>
<td>☐</td>
</tr>
</tbody>
</table>
Notices

For information, contact info@h2020-immortal.eu.

This document is intended to fulfil the contractual obligations of the IMMORTAL project concerning deliverable D5.1 described in contract number 644905.

© Copyright IMMORTAL 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>December 15, 2017</td>
<td>Final Version</td>
<td>All</td>
<td>All</td>
</tr>
</tbody>
</table>

Authors, Beneficiary

Jan Malburg, DLR
Annika Ofenloch, DLR
Jaan Raik, TUT
Franz Roeck, TUG
Eli Arbel, IBM
Shiri Moran, IBM
Hans Kerkhoff, UT

About

This document describes the integrated demonstration of the developed software tools. We present the example use case of fault-resilient many-core based satellite. Then, tools are presented based on an example development flow. Finally, the relation of the tools to achieving the measurable objectives is described.
List of Abbreviations

AIG  And-Inverter Graph
AMS  Analogue Mixed Signal
ASIC Application-Specific Integrated Circuit
CEI  Checkers Efficiency Index
CPS  Cyber-Physical System
CPU  Central Processing Unit
DDG  Dynamic Dependency Graph
DLR  Deutsches Zentrum für Luft- und Raumfahrt
ECC  Error Checking and Correction
EP   Electronic Power
ESL  Electronic System Level
FC   Fault Coverage
FDIR Fault Detection, Isolation and Recovery
FIFO First In - First Out
FIT  Failures in Time
FPR  False Positive Ratio
FSM  Finite State Machine
GUI  Graphical User Interface
HAL  Hardware Abstraction Layer
HCI  Hot Carrier Injection
IP   Internet Protocol
LBDR Logic-Based Distributed Routing
LTL  Linear Temporal Logic
NBTI Negative-Bias Temperature Instability
NoC  Network on Chip
OBC  On-Board Computer
OBSW On-Board Software
PBTI Positive-Bias Temperature Instability
PCDU Power Conditioning and Distribution Unit
RTL  Register Transfer Level
SEU  Single Event Upset
TCP  Transmission Control Protocol
TSMC Taiwan Semiconductor Manufacturing Company
UDP  User Datagram Protocol
# Table of Contents

Table of Revisions ........................................................................................................ iii
Authors, Beneficiary ........................................................................................................ iii
About................................................................................................................................ iii
List of Abbreviations ........................................................................................................ iv
Table of Contents ............................................................................................................ v
Executive Summary .......................................................................................................... vi
1 Introduction ................................................................................................................ 1
2 Development Flow ...................................................................................................... 3
3 Tool Flow ................................................................................................................... 5
   3.1 CPS simulation and reasoning engine .................................................................. 5
   3.2 Network analysis framework ............................................................................. 10
   3.3 Network testing ................................................................................................ 12
   3.4 Property Generation with DDG-Miner ............................................................ 15
   3.5 Checker synthesis and minimization ............................................................... 17
   3.6 Reliability Analysis ......................................................................................... 21
   3.7 Reliability verification ..................................................................................... 22
   3.8 PARTYStrategy ............................................................................................ 23
4 Status on Measurable Objectives ............................................................................ 33
5 Conclusion ................................................................................................................. 37
6 References ................................................................................................................ 39
Executive Summary

This document describes the integrated demonstration of the software tools developed in IMMORTAL. As an example use case a fault-resilient many-core based satellite is used. For the application software we have chosen the Fault Detection, Isolation and Recovery (FDIR) application of the DLR’s Eu:CROPIS satellite. The usage of software tools developed in the scope of IMMORTAL is also provided. The tools described in this deliverable are:

**CPS simulation environment**: The CPS simulation environment is used to simulate the power control application as part of the On-Board Computer (OBC) of the Eu:CROPIS-satellite for testing the flight software and for analyzing the behavior of the system in case of errors or state changes. The simulation is carried out with software models, which simulate the functional behavior of the system.

**Linking AMS and reasoning engines**: Starting from low-level (aged transistor) analog/mixed-signal IPs, crucial higher-level functional performance descriptions are Booleanized, and subsequently introduced in reasoning engines to analyze complex system reliability behaviour.

**Network analysis framework**: The network analysis framework is a C++ library to measure the performance parameters (latency and bandwidth) of networks. One focus of this library is the usage at embedded systems and non-TCP/IP networks. Another focus is the support of a broad variety of different targets, such it not only supports real hardware, but also Register Transfer Level (RTL) or Electronic System Level (ESL) simulations. The framework is used to created a model of the network.

**Network testing**: Using the model created with the network analysis framework, network testing a simulator is used to check the behavior of the network under fault conditions. For this links or routers are marked as broken and the effect on the network’s performance parameters are measured. This allows to identify critical parts in the network as well as to learn at which point the network can no longer provide the performance required for reliable function of the system.

**DDG-Miner**: The DDG-Miner is a tool to generate properties for RTL designs. For this a set of use cases of the design is utilized and Dynamic Dependency Graphs (DDGs) for those use cases are created. From those DDGs control- and data-dependency is extracted and a set of initial properties is created. In several refinement steps those initial properties are abstracted and model-checked to create a final set of abstract properties from.

**Checker qualification and minimization**: This tool qualifies and ranks concurrent-online checkers according to their fault-detection capabilities and allows minimization of the set of checkers for a circuit under given fault coverage constraints. The tool can read the input in the form of checkers synthesized from properties derived by the DDG-miner. The checker minimization output can be further analyzed by the Reliability Analysis tool.

**Reliability Analysis**: A tool-set for analyzing the reliability of a given CPS is described. This tool-set is applied during the design stage on the RT-level description and is aimed at analyzing and validating that the design meets its pre-defined reliability targets and assist with identifying areas in the design which lack proper
protection against soft errors. The tool-set is comprised of a vulnerability estimation tool and a protection analysis tool. The vulnerability analysis tool computes and outputs a ranked list of flip-flops based on their susceptibility to soft-error hits. The protection analysis tool identifies existing error correction mechanisms in the design.

**Reliability verification:** This tool is applied at the integration test and verification stage of an *System on Chip* (SoC) project life cycle. The goal of this tool is to verify the gating conditions of existing error protection mechanisms, such as parity protection. In general, gating conditions are used to both save power as well as for preventing error checkers from firing (and subsequently initiate costly recovery actions) when the data they protect is not used. Gating the error checking logic too much however is undesired as it can lead to silent data corruption. The reliability verification tool identifies the gating conditions and is used to drive error injection such that errors are injected in cycles when the error protection is gated, hence maximizing the probability of uncovering gating-related bugs.

**PARTYStrategy:** The tool PARTYStrategy is evaluated on the FDIR component of DLRs satellite Eu:CROPIS that is currently under development. The test strategies are applied to the real C++ implementation and to mutated versions of it to evaluate the fault finding capabilities of the computed test suite. Being derived from the generic *Linear Temporal Logic* (LTL) specification, the computed (implementation independent) strategies kill 71.23% of the mutants.

From those tool, following tools are planned to be shown at the final presentation in Brussels:

- CPS simulation environment
- Checker synthesis and minimization
- Reliability Analysis

The tools presented in this deliverable helps in achieving the following measurable objective.

- MO1: Verification effort minimized 2x
- MO4: Up to 40% reduction in reliability related tasks
- MO5: Up to 10% area savings by optimizing hardware protection logic

We show that the first two of those measurable objectives have been archived with even better results than initially expected. The last one is still being evaluated, but early results look highly promising.
1 Introduction

This deliverable presents a set of tools developed within the IMMORTAL project. The tools target different goals:

- Allowing decreased development time and cost due to the use of simulation,
- helping in generating adequate protection mechanisms for the designs,
- verifying the reliability of the system, or
- assisting in testing the design.

The IMMORTAL software demonstrator is integrated around a satellite use-case we base on DLR’s current satellite project Eu:CROPIS expected to be launched mid 2018. As a software application example, the FDIR of Eu:CROPIS has been selected. Data obtained from the development process of this real-world application is used as a baseline to evaluate the impact of software testing and verification tools of IMMORTAL. For the hardware platform we utilize the fault-resilient Bonfire many-core architecture developed in the IMMORTAL project. The architecture is based on a communication infrastructure consisting of Network on Chip (NoC) routers with network interfaces for connecting processors to the network. It includes reliability structures such as concurrent online checkers, monitors of physical parameters, packet dropping mechanisms and IJTAG based infrastructure for accessing reliability information. We envision this to be a scalable prototype architecture to future space missions, where the on-board computers will be based on many-core based, networked approaches.

Not all of the tools presented in this deliverable will be able to be shown in the final presentation, primarily because either they require too long execution time (several hours of) or because they utilize commercial tools, which need access to a license servers. The tools planned to be shown during the final review are:

- CPS simulation environment
- Checker qualification and minimization
- Reliability analysis

The rest of the deliverable is organized as follows: In Section 2 we present the IMMORTAL development model. The different tools are presented in Section 3. In Section 4 we show the relation between the presented tools and the measurable objectives. Section 5 concludes the deliverable.
Figure 1: The IMMORTAL development as a V-model. Tools with white text are presented in this deliverable.

Figure 1 shows a V-model flow including different tools and techniques developed within IMMORTAL. On the left side of the V-model we see the steps related to the development of the system and on the right side, steps related to testing, verification and operation. Tools presented in this deliverable have white text color. In the following we will describe the development flow on the example of a research satellite, further we show how the different IMMORTAL tools and technique fits into this development flow.

The first step is the concept of operation. Here the mission goal is decided and the main mission parameters are defined. This leads to the high-level specification and requirements. Some example decisions which are fixed at this stage which are relevant for IMMORTAL related tasks: The computation power required for the mission, as this may result in the need for many-core architecture or even high performance but non-radiation-hardened components. The mission duration, as this influences the vulnerability to aging. Whether it is a deep-space mission or one in an orbit where the earth’s magnetic field still provide some protection from cosmic radiation.

At this stage the CPS simulation environment can be used to evaluate different choices using models of the satellite and its subcomponents. For the further development of the simulator, models are to be replaced by concrete hardware or more refined models to enable early testing of the satellite and its components in all phases.
In the requirements and architecture stage, the main architectural decisions are made, such as the number and types of cores, connection infrastructure between the cores, and used (third party) IP. With respect to IMMORTAL tools, first we use the network analysis tool to decide whether the designed network infrastructure provides the required bandwidth and latency. The network analysis tool is closely coupled with the network testing tool from the Integration Test, and Verification stage, as the network testing evaluates the performance under failure conditions. Also, in the requirements and architecture step we use property generation to create properties for (third party) IP cores.

In the detailed design stage, checker synthesis and minimization is applied to create a minimal set of checkers from the properties generated with the property generation approach, in addition to other manually created checkers. The reliability analysis finalizes the design-for-reliability so that the pre-defined reliability targets are being met. To that end, relevant analysis tools are applied and required means (mainly adding protection) are applied, until the reliability targets are met. The analysis mainly concerns detecting those memory elements which are already protected by checkers against soft errors and providing an estimation how likely it is that a soft-error in unprotected elements will propagate. Later in the detailed design stage (analogue) monitors are integrated and checkers and monitors are connected to the IJTAG scan-chain.

After this the hardware is ready for production in the implementation stage and the post-silicon integration test and verification can start. With respect to the IMMORTAL tools and techniques this primarily consists of software. Reliability verification is applied to verify that the protection mechanisms function correctly, and specifically that checkers are correctly gated; gating on the checkers are put to prevent redundant recovery - in cases where the relevant data is not used - and to save power.

In the system verification and validation stage, the test strategy generation generates adaptive test strategies based on LTL formulas describing the expected behavior. In our case we use it to test whether the software correctly handles failure case. As the specification of those cases includes high implementation freedom, adaptive tests are required. In addition, network testing in order to validate various parameters of the entire many-core system is carried out.

During operation of the satellite, maintenance by external interaction is not possible for satellites, therefore in case of failing components they must be replaced by redundant parts, or failure must be prevented in advance. Sensor fusion gathers information from the different checkers and monitors and provide information of the different components health and aging state. By scheduling and mapping, the use of defect components is prevented and also is used to lift load from aged parts to prevent complete failure of the component. The fault management component is used to detect faults and recover from them, for example by recomputation.
3 Tool Flow

In this section we will present the different tools. The tools are sorted according to the order they are applied during the development.

3.1 CPS simulation and reasoning engine

The CPS simulation environment presented in Section 3 of D2.2 can be used during the entire system development cycle to reduce the costs and development time. We use this tool to simulate the power control application of the Eu:CROPIS satellite.

We have also addressed the difficulty to formally verify large Analogue-Mixed Signal (AMS) system in the presence of continuous variables as highlighted in the deliverable D2.3. It remains computationally very challenging, even using state-of-art techniques. In order to close this gap, we are developing a tool, in which it will be possible to include aging-related degradation for a particular system in a specific technology to see if, for a given time, the system will hold the given property.

CPS simulation environment

The developed event-oriented CPS simulation environment allows a distributed simulation to test and validate complex CPS. It is very extensible and has hardly any restrictions in the modeling of a CPS. Models can be executed on distributed computer systems, while the communication takes place over TCP. Due to the publisher-subscriber pattern, the models are loosely coupled. This allows adding new software models to the simulation or replacing them by physical hardware components with minimal effort during the development process.

As an example we are using a power control application as part of an On-Board Computer (OBC) of the Eu:CROPIS satellite from DLR [9]. This application tightly integrates a safety-critical digital control system with its physical environment. The objective of the system is to manage the energy sources by controlling and distributing the power to the satellite’s units. The OBC, on which the flight software (On-Board Software (OBSW)) runs, is located on the satellite bus and connected to all subcomponents - providing data handling services, receiving commands and generating telemetry. In this example the Power Conditioning and Distribution Unit (PCDU) is connected to the OBC via a serial interface. The PCDU performs power control tasks, which includes the acquisition of telemetry like current and voltage. In order to test them, system models for the OBC and PCDU are developed and integrated into the simulation environment.
An implementation of the simulation environment can be found in Figure 2, referring to the Eu:CROPIS satellite [12]. A flight software simulation is used and integrated into the simulation environment to test the OBSW. The OBSW uses a Hardware Abstraction Layer (HAL) called Device Driver Factory to access the connected physical or simulated hardware. To integrate the simulation into the HAL, it requires a clock model ($M_{Clock}$) which specifies the time since the OBC started. The OBC starts at the same time as the simulation. Thus, the clock model must get the current simulation time, which is converted by the OBSW to its internal representation and passed on to the other devices. Furthermore, a serial model ($M_{Serial}$) is needed to provide an interface to the device model ($M_{PCDU}$), which performs the power control tasks. During operation, PCDU commands are generated and sent to the device model via the serial model. The device model includes a simulated PCDU that has the functionality to process the received command and generates the corresponding telemetry data. This data is sent back to and temporarily stored in the serial model. When the flight software simulation evaluates the PCDU telemetry data, the buffered data is read out and returned from the serial model. Additional events can be triggered through the event queue ($M_{Queue}$) to modify the model states. This is used to simulate error scenarios and to analyze the reactions of software and hardware components in case of errors or state changes.

To run the simulation, all user-defined models and the simulation environment models must be started. For testing purposes, all models are started locally via a bash script, as it can be seen in Listing 1. The models can be distributed from a central computer via an automated deployment process.

Listing 2 gives an excerpt of the simulation output. First, events are exchanged between the models for the configuration and synchronization process. Then the events containing PCDU commands and PCDU telemetries are exchanged between the serial model and the device model. The listing shows that the flight software simulation requested successfully telemetry data from the virtual PCDU.
#!/bin/bash ./configuration_server/bin/config_server &
./simulation_model/bin/sim_model &
./subcomponents/pcdu_model/bin/pcdu_model &
./queues/event_queue/bin/event_queue &
./subcomponents/eucropis_flightsw/bin/flightsw_sim

Listing 1: Bash script to start the simulation.

```bash
# event_queue subscribes to SimTimeChanged
# event_queue subscribes to End
# event_queue subscribes to Configure
# event_queue subscribes to SyncMsg
# simulation_model publishes SyncMsg at 0 ms
# pcdu_model subscribes to Configure
# pcdu_model subscribes to End
# pcdu_model subscribes to PCDUCommand
# pcdu_model subscribes to SyncMsg
# clock_model subscribes to End
# clock_model subscribes to SimTimeChanged
# clock_model subscribes to SyncMsg
# simulation_model publishes SyncMsg at 0 ms
# clock_model receives [SyncMsg] at 0 ms
# event_queue receives [SyncMsg] at 0 ms
# simulation_model receives ConfirmMsg
# simulation_model receives ConfirmMsg
# pcdu_model receives [SyncMsg] at 0 ms
# simulation_model receives ConfirmMsg
# pcdu_serial_model subscribes to End
# pcdu_serial_model subscribes to SimTimeChanged
# pcdu_serial_model subscribes to PCDUTelemetry
# pcdu_serial_model subscribes to SyncMsg
# simulation_model publishes SyncMsg at 0 ms
# pcdu_serial_model receives [SyncMsg] at 0 ms
# simulation_model receives ConfirmMsg
# synchronization complete
# Start Flightsoftware Simulation
# Start Simulation
# Simulation time: 0 ms
# simulation_model publishes SimTimeChanged at 0 ms
# event_queue receives [SimTimeChanged] at 0 ms
# clock_model receives [SimTimeChanged] at 0 ms
...;
# TC: 11 194 0 1 0 0
# PCDUSerialModel::write
# pcdu_serial_model publishes PCDUCommand at 0 ms
# pcdu_model receives [PCDUCommand] at 0 ms
# Updated telemetry to output buffer
# TM: ...0 0 0 0 0 1 0 1 0 2 0 2 0 0 0 0 0 0 0
# pcdu_model publishes PCDUTelemetry at 0 ms
# pcdu_serial_model receives [PCDUTelemetry] at 0 ms
# PCDUSerialModel::read
...;
# Simulation time: 1000 ms
# simulation_model publishes SimTimeChanged at 1000 ms
...;
# TC: 11 194 0 1 0 0
# PCDUSerialModel::write
```

| # pcduserial_model publishes PCDUCommand at 1000 ms |
| # pcdu_model receives [PCDUCommand] at 1000 ms |
| # Updated telemetry to output buffer |
| # TM: ...0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 0 0 0 0 0 0 |
| # pcdu_model publishes PCDUTelemetry at 1000 ms |
| # pcduserial_model receives [PCDUTelemetry] at 1000 ms |
| # PCDUSerialModel::read |
| ... |
| # Simulation time: 2000 ms |
| # simulation_model publishes SimTimeChanged at 2000 ms |
| ... |
| # simulation_model publishes End at 11000 ms |
| ... |
| # Stop models |

Linking AMS and reasoning engines

Conventional tools for mixed-signal system evaluation show a very limited capability of handling the AMS parts. To resolve this bottleneck, AMS systems are over-simplified which helps to meet computational capabilities but leads to a less-efficient formal verification. As integrated circuit technology is scaling down, aging-related effects like NBTI, PBTI & HCI are becoming dominant and are the major reason for the system’s failure; it is therefore highly desirable to include this reliability concern (aging) during the formal verification process. To accomplish this, it is important to form a scalable link between the actual AMS design world and the formal verification domain.

Aging affects a system subjectively as it depends on the device properties, switching activities and its environmental conditions (for instance, a high environmental temperature can accelerate the device’s aging phenomena). On the other hand, aging is very subjective to different technology nodes e.g. technology nodes like 40nm, 22nm and 7nm, all have different degradation (e.g. $V_{th}$) curves over the time scale. Therefore inclusion of aging information for a given AMS system at the formal verification level holds two major challenges; first, it is mathematically very challenging to model a system with its aging behavior, and second it is difficult to simplify (while conveying all the information) for the verification tools.

We are developing a tool that will provide a link between these two different worlds. With this link, it will be possible to include aging-related degradation for a particular system in a specific technology to see if, for a given time, the system will hold the given property. With this type of formal verification analysis, one can determine the required specifications of the embedded instruments, which can ensure reliable operation of the system over a given time. Figure 3 shows the flow for our methodology. Details will be provided in deliverable D3.3.

Circuit level: we started with the development of AMS circuits using the TSMC 40nm technology node. These circuits have been thoroughly investigated initially at the transistor level (in Cadence) to extract their degradation, by using our developed model, primarily due NBTI. At circuit level, all the quantities involved are
Development of a Boolean Model: To form a link between circuit world and formal verification tools, our AMS design is converted to a Boolean model that offers SPICE level accuracy. For this model development, we have extended the approach presented in [10] for our AMS circuits with their degradation information using our tool (MATLAB-platform based). This tool converts the continuous domain into the Boolean state-space. This Boolean model is a representation of an analog circuit in a Finite-State Machine (FSM). Each discretized input combination is associated with a DC state. For example, if the circuit has 2-inputs, and each input is discretized using 2-bits (i.e. 4-levels), the Boolean model would have 16 DC states that capture the DC behavior of the circuit. To capture the transient behavior of the AMS circuit, additional transient FSM states are introduced between every pair of DC states.

**Formal Verification:** To demonstrate the working link between circuit-level design and formal verification tools, the developed Boolean model is converted into an And Invertor Graph (AIG), that will be used as an input by a third-party open access tool ABC [7] for the formal verification purpose.

For demonstration purposes, we are continuing our case study of the thermal control loop for the satellite computer as discussed in D2.3. This case study will be presented for the following two scenarios:

1. An ideal temperature-sensor module will be converted into its equivalent Boolean model. This Boolean model will be formally verified with the ABC tool.

2. The temperature-sensor module with its aging information will be converted to its equivalent Boolean model. This developed Boolean model will be further used for the formal verification with ABC.
In the first case, since the temperature sensor is ideal, the thermal-loop system will hold the constraints (against which it is formally verified) but in the second case a counterexample is expected because of aging in the AMS part.

3.2 Network analysis framework

The network analysis framework is used to measure the bandwidth and the latency of a given network under different load scenarios. The framework is a library that can be used on the real hardware as well as within RTL or ESL, e.g., SystemC, simulation. A technical description of the network analysis and the network testing is given in D2.2, in this deliverable we focus more on the usage scenario of the tools.

The network analysis is intended to check whether a network provides the necessary performance parameters in the nominal case. In a real-time system, which most CPS are, the communication latency is a critical parameter. Based on the application also the available bandwidth may be a concern. In modern designs we have network based, e.g., NoC, communication topologies in which communications can interfere with each other. Therefore, our tool tests the network under different load scenarios to find such interference effects. The overall goal of the network analysis is to create a model of the network, based on which we can decide whether the performance parameters of the network fulfill the requirements of the usage scenario. The measurement is conducted in a peer to peer approach between each pair of endpoints and also in both direction, as there may be unsymmetrical links.

The tool is based on a cooperative approach. For this on each node a measurement process is deployed. One of those processes is set as master the rest as slaves. The master controls the other processes and gathers the results. At first each node pair measures the latency and bandwidth between each other while all other nodes remain silent. Based on this measurement we say for each node its bandwidth is the maximum from all measurements initiated by that node. Then the measurement of bandwidth and latency is repeated, however this time while all other nodes create random traffic with a certain fraction of their bandwidth. The measurements are repeated with increasingly amount of cross-traffic until the networked gets saturated.

In the end a model of the network is created, for this the known topology of the network can be used for better results. Currently, the network analysis process is ported to the hardware from the EU-project MaMMoTH-Up, on the RTL-model of the NoC-design used in the ASIC demonstrator and for the simulation environment presented in the next section as part of the network testing.

Usage: As a tool that is not executed on a desktop computer but on the, possibly simulated, CPS, there is no command line or GUI to run the approach directly. Instead it is a library written in C++11 which provide an interface to implement. This interface encapsulates hardware specific functionality: time measurement,
Figure 4: The structure of the NoC used in the IMMORTAL ASIC.

Figure 4: The structure of the NoC used in the IMMORTAL ASIC.

sending and receiving messages, node identification, and a logging mechanism that allows the master node to output the results. For porting the library this interface must be implemented for the target hardware. To be embedded system friendly the library uses neither floating point nor threading. The deployment is then done by the regular methods for the corresponding target, e.g., via JTAG debugging interface.

We also applied the tool to the RTL model of the NoC used in the IMMORTAL ASIC. In this case we combined the VHDL model of the NoC, with the library by putting the network testing library into a SystemC module that then is instantiated instead of the CPU cores. The basic structure of this NoC is shown in Figure 4. The module replacing Node 0 is used as master node.

Listing 3 gives an excerpt of the measurement conducted on the NoC of the IMMORTAL ASIC. At first the latency is measured without cross-traffic. Note that the latency is measured for different amounts of bytes, but those are mostly omitted for space reason. After that the bandwidth is measure. Then each measurement is repeated under different amount of cross-traffic. As we can see by 10% cross-traffic the latency is not effected, but the bandwidth is reduced, in case with more than 1 hop even more then 10%.

**Connection to other IMMORTAL tools:** The network analysis generates or refines a network model. This model is used by the network testing approach to introduce faults and check the effect of faults using simulation.
# Info: Server online
# Info: All clients online
# Info: All clients time synced
# Result: Latency from 0 to 1 with 24 bytes is 0.00000028 sec
# Result: Latency from 1 to 0 with 24 bytes is 0.00000028 sec
# Result: Latency from 1 to 2 with 24 bytes is 0.00000034 sec
# Result: Latency from 1 to 3 with 24 bytes is 0.00000028 sec
# Result: Latency from 0 to 2 with 24 bytes is 0.00000028 sec
# Result: Latency from 2 to 0 with 24 bytes is 0.00000028 sec
# Result: Latency from 2 to 1 with 24 bytes is 0.00000034 sec
# Result: Latency from 2 to 3 with 24 bytes is 0.00000028 sec
# Result: Latency from 0 to 3 with 24 bytes is 0.00000028 sec
# Result: Latency from 3 to 0 with 24 bytes is 0.00000034 sec
# Result: Latency from 3 to 1 with 24 bytes is 0.00000028 sec
# Result: Latency from 3 to 2 with 24 bytes is 0.00000034 sec
# Result: Latency from 3 to 0 with 25 bytes is 0.00000034 sec
# Result: Latency from 1 to 0 with 25 bytes is 0.0000003 sec

# Result: Bandwidth from 0 to 1 is 343624161 bytes/sec
# Result: Bandwidth from 1 to 0 is 343624161 bytes/sec
# Result: Bandwidth from 1 to 2 is 341333333 bytes/sec
# Result: Bandwidth from 1 to 3 is 343624161 bytes/sec
# Result: Bandwidth from 0 to 2 is 343624161 bytes/sec
# Result: Bandwidth from 2 to 0 is 343624161 bytes/sec
# Result: Bandwidth from 2 to 1 is 341333333 bytes/sec
# Result: Bandwidth from 2 to 3 is 343624161 bytes/sec
# Result: Bandwidth from 0 to 3 is 341333333 bytes/sec
# Result: Bandwidth from 3 to 0 is 341333333 bytes/sec
# Result: Bandwidth from 3 to 1 is 343624161 bytes/sec
# Result: Bandwidth from 3 to 2 is 343624161 bytes/sec

... # Result: Under traffic of 10%
# Result: Latency from 0 to 1 with 24 bytes is 0.00000028 sec
# Result: Bandwidth from 0 to 1 is 318181819 bytes/sec
# Result: Latency from 1 to 0 with 24 bytes is 0.00000028 sec
# Result: Bandwidth from 1 to 0 is 318181819 bytes/sec
# Result: Latency from 1 to 2 with 24 bytes is 0.00000034 sec
# Result: Bandwidth from 1 to 2 is 309093747 bytes/sec
# Result: Latency from 1 to 3 with 24 bytes is 0.00000028 sec
# Result: Bandwidth from 1 to 3 is 322147565 bytes/sec
# Result: Latency from 0 to 2 with 24 bytes is 0.00000028 sec
# Result: Bandwidth from 0 to 2 is 318172581 bytes/sec
# Result: Latency from 2 to 0 with 24 bytes is 0.00000028 sec
# Result: Bandwidth from 2 to 0 is 318288589 bytes/sec
# Result: Latency from 2 to 1 with 24 bytes is 0.00000028 sec
# Result: Bandwidth from 2 to 1 is 309093747 bytes/sec
# Result: Latency from 2 to 3 with 24 bytes is 0.00000028 sec
# Result: Bandwidth from 2 to 3 is 300000001 bytes/sec
# Result: Latency from 0 to 3 with 24 bytes is 0.00000028 sec
# Result: Bandwidth from 0 to 3 is 300000001 bytes/sec
# Result: Latency from 3 to 0 with 24 bytes is 0.00000028 sec
# Result: Bandwidth from 3 to 0 is 309166665 bytes/sec
# Result: Latency from 3 to 1 with 24 bytes is 0.00000028 sec
# Result: Bandwidth from 3 to 1 is 318173236 bytes/sec
# Result: Latency from 3 to 2 with 24 bytes is 0.00000028 sec
# Result: Bandwidth from 3 to 2 is 311409395 bytes/sec
# Result: Latency from 3 to 0 with 24 bytes is 0.00000034 sec
# Result: Bandwidth from 3 to 0 is 309166665 bytes/sec

... Listing 3: An excerpt of the measurement results for the IMMORTAL NoC.
3.3 Network testing

The network testing is closely coupled with the network analysis. Network testing utilizes the network model created by the network analysis. The idea hereby is that we have a simulator, which can simulate the network based on the network model created by the network analysis. Then, the measurement processes from the network analysis approach can be executed on the simulated network. The network simulator uses a static routing table created whenever the model changes, i.e., faults are introduced or removed. The routing algorithm uses a shortest-path algorithm, where the model can assign cost to router and links.

The idea is now that we introduce errors in the network model, e.g., remove links or routers, or decrease the bandwidth of links. On this erroneous network models we can again measure the latency and bandwidth between each node using the network analysis tool. With this we can test which faults in the network will cause its performance to degenerate in such way that the required performance parameters are no longer provided. Likewise, it also provides information about critical components of the network.

This information can also be used to harden the network at critical points.

**Usage:** The network testing consists of two applications. First, the network simulator:

```
> NetworkSimulatorServer 6666 models/noc
```

Its first parameter is the port on which it listens for connections of client applications and the second parameter is the files that includes the network model. The clients connect to the simulated network via UDP, initial the server waits till enough clients have connected where the number of clients is the number of nodes in the network model. Then each clients gets a node assigned and the simulators waits for packages from the clients.

The clients are implementations of the network analysis library that implement the interface of the network simulator. To start they require IP and port of the simulator:

```
> NetworkTesting 127.0.0.1 6666
```

Figure 5 shows the model created for the NoC, with parameters as bandwidth and latency. Note that the routers do wormhole routing.

Listing 4 shows a part of the measurement where the link between router 2 and router 3 is broken. What we can see is that in the case without cross-traffic the effect of the broken link only causes higher latency between node 2 and node 3, but otherwise not effect the network. However, as we see in the bottom part, under cross-traffic not only the connection between node 2 and node 3 is affected, but the performance parameter are affected regardless whether the broken links is on the original path between the nodes or not. Another effect is that even small cross-traffic affects the latency, where in the nominal case small cross-traffic does not affect the latency.
Figure 5: The generated model of the IMMORTAL ASIC NoC.

**Connection to other IMMORTAL tools:** The network testing get the model from the network analysis approach. It provides information of the behavior of the network under failure condition, this information is currently not used by another tool, but provided to the designers.

<table>
<thead>
<tr>
<th>Info: Server online</th>
</tr>
</thead>
<tbody>
<tr>
<td>Info: All clients online</td>
</tr>
<tr>
<td>Info: All clients time synced</td>
</tr>
<tr>
<td>Result: Latency from 0 to 1 with 24 bytes is 0.00000028 sec</td>
</tr>
<tr>
<td>Result: Latency from 1 to 0 with 24 bytes is 0.00000028 sec</td>
</tr>
<tr>
<td>Result: Latency from 1 to 2 with 24 bytes is 0.00000034 sec</td>
</tr>
<tr>
<td>Result: Latency from 1 to 3 with 24 bytes is 0.00000028 sec</td>
</tr>
<tr>
<td>Result: Latency from 0 to 2 with 24 bytes is 0.00000028 sec</td>
</tr>
<tr>
<td>Result: Latency from 2 to 0 with 24 bytes is 0.00000028 sec</td>
</tr>
<tr>
<td>Result: Latency from 2 to 1 with 24 bytes is 0.00000034 sec</td>
</tr>
<tr>
<td>Result: Latency from 2 to 3 with 24 bytes is 0.00000028 sec</td>
</tr>
<tr>
<td>Result: Latency from 0 to 3 with 24 bytes is 0.00000034 sec</td>
</tr>
<tr>
<td>Result: Latency from 3 to 0 with 24 bytes is 0.00000034 sec</td>
</tr>
<tr>
<td>Result: Latency from 3 to 1 with 24 bytes is 0.00000028 sec</td>
</tr>
<tr>
<td>Result: Latency from 3 to 2 with 24 bytes is 0.00000041 sec</td>
</tr>
</tbody>
</table>

... Result: Bandwidth from 0 to 1 is 343605706 bytes/sec
Result: Bandwidth from 1 to 0 is 343605787 bytes/sec
Result: Bandwidth from 1 to 2 is 343603498 bytes/sec
Result: Bandwidth from 1 to 3 is 343605901 bytes/sec
Result: Bandwidth from 0 to 2 is 343602597 bytes/sec
Result: Bandwidth from 2 to 0 is 343605830 bytes/sec
Result: Bandwidth from 2 to 1 is 343605793 bytes/sec
Result: Bandwidth from 2 to 3 is 343605814 bytes/sec
Result: Bandwidth from 0 to 3 is 343605832 bytes/sec
Result: Bandwidth from 3 to 0 is 343601351 bytes/sec
Result: Bandwidth from 3 to 1 is 343605713 bytes/sec
Result: Bandwidth from 3 to 2 is 343605832 bytes/sec
Result: Under traffic of 10%
Result: Latency from 0 to 1 with 24 bytes is 0.00000033 sec
Result: Bandwidth from 0 to 1 is 243527186 bytes/sec
Result: Latency from 1 to 0 with 24 bytes is 0.00000037 sec
Result: Bandwidth from 1 to 0 is 243275580 bytes/sec
Result: Latency from 1 to 2 with 24 bytes is 0.00000043 sec
Result: Bandwidth from 1 to 2 is 236227404 bytes/sec
Result: Latency from 1 to 3 with 24 bytes is 0.00000039 sec
Result: Bandwidth from 1 to 3 is 257704425 bytes/sec
Result: Latency from 0 to 2 with 24 bytes is 0.00000029 sec
Result: Bandwidth from 0 to 2 is 249732647 bytes/sec
Result: Latency from 2 to 1 with 24 bytes is 0.00000044 sec
Result: Bandwidth from 2 to 1 is 236228982 bytes/sec
Result: Latency from 2 to 0 with 24 bytes is 0.00000036 sec
Result: Bandwidth from 2 to 0 is 252713026 bytes/sec
Result: Latency from 2 to 3 with 24 bytes is 0.00000049 sec
Result: Bandwidth from 2 to 3 is 238745639 bytes/sec
Result: Latency from 0 to 3 with 24 bytes is 0.00000043 sec
Result: Bandwidth from 0 to 3 is 204015973 bytes/sec
Result: Latency from 3 to 1 with 24 bytes is 0.00000038 sec
Result: Bandwidth from 3 to 1 is 236228927 bytes/sec
Result: Latency from 3 to 2 with 24 bytes is 0.00000055 sec
Result: Bandwidth from 3 to 2 is 236124148 bytes/sec
Result: Latency from 3 to 0 with 24 bytes is 0.00000046 sec
Result: Bandwidth from 3 to 0 is 230857157 bytes/sec

Listing 4: An excerpt of the measurement on the model where the link between router 2 and router 3 is broken.

### 3.4 Property Generation with DDG-Miner

D2.3 presented the property generation approach used by the DDG-Miner in detail. Here we give only a brief summary. The property generation approach is a semi-formal approach that uses simulation-data to generate symbolic expressions describing behavior of the design. In the first step the use case is used to generate a Dynamic Dependency Graph (DDG) of the corresponding use case. From this a set of initial properties is generated, where control-dependency yield the antecedence and the data-dependency the consequence. A set of abstraction rules is applied to those initial properties to generate more general ones. In the end the generated properties are model-checked and only the most abstract ones are kept.

Automatically, generated properties can be used for a variate of goals:

- As they encode behavior of the design, they can be used to better understand the design.
- Comparing the generated properties with the specification can be used to uncover faults in the design.
The properties can be transformed into checkers which can uncover soft errors.

The first two points can be used to decrease the time for developing, and thus save effort. With respect to the presented development flow the last point is of special interest. For the transformation of such properties in RTL-checkers a set of open and commercial tools are already available, for example FoCs from IBM.

**Usage:** To run DDG-Miner following three things must be provided:

- The design for which the properties should be created.
- One or more use cases for the design.
- The information which primary input is the clock signal.

An example call of DDG-Miner looks as follows:

```
> ddgminer -a -o y86 -z clk -t Test.v -N Y86.sva Y86_seq.v
```

The "-a" is optional and indicates that primary outputs are allowed to be used as variables in the consequence; this often cause easier to read and smaller properties. The "-o y86" indicate the folder which should be used for output and intermediate files. With "-z clk" we indicate which primary input is the clock. There can be several "-t" followed by some file name to indicate the use cases. Each use case must be fully contained in a single file, i.e., including the test-harness. With "-N" we can define a file to which the properties should be written, otherwise they will be written to command line only. In the end, the list of the files of the design is given.

In this example we are generating properties of a CPU core which implements a sub-set of the x86-32 instruction set architecture. Listing 5 show an excerpt of the generated properties. The shown properties are the final set for the write enable primary output of the design. DDG-Miner generated a property related to each pipeline-stage of the design. This output is exactly high when a value should be written from a register to the memory (property `bus_WE_2`) otherwise it is low. Note that we also create comments to make the repetition amount more human readable.

**Connection to other IMMORTAL tools:** A problem with turning automatically generated properties into checkers is that they are likely not an optimal set, as several checkers may check the same error. Also some properties may be over constrained and therefore cause sub-optimal and too complex checkers. In the next section we will present approaches to optimize the amount and size of checkers.

```
property bus_WE_0;
@ (posedge clk)
(
  (rst)
  ##0 (##1(!rst)[*5][*1:5])or 1)||(rst)[n*5], a>=0
)
|->##1 bus_WE == 'b0;
endproperty

property bus_WE_1;
@ (posedge clk)
```
3.5 Checker synthesis and minimization

Generating the environment for checker qualification. The many-core based satellite use case of IMMORTAL software demonstrator utilizes the Bonfire NoC architecture developed by TUT in the IMMORTAL project\(^1\). Concurrent checkers in the control part of Bonfire NoC router are generated by the following methodology [14, 4]. First, a set of combinational assertions for the control modules of the router (i.e. FIFO control unit, LBDR, allocator/arbiter) are synthesized and evaluated as checker candidates. In order to allow checker evaluation, also the control modules are converted to their pseudo-combinational counterparts by creating pseudo-primary inputs and pseudo-primary outputs from the outputs and inputs of

\(^1\)https://github.com/Project-Bonfire/Bonfire

Listing 5: The final properties for the bus_WE of the Y86 design.
the state preserving flipflops respectively. The pseudo-combinational versions of the LBDR and arbiter modules is presented in Figure 6.

Figure 6: Pseudo-combinational versions of the LBDR and arbiter modules

To derive the environment for the evaluation, a set of exhaustive valid stimuli is derived for the control module. This is feasible due to the fact that the number of primary and pseudo-primary inputs for control modules is relatively small. The stimuli set is restricted via filtering it by input and state constraints derived for the module. As a result, exhaustive valid stimuli for the control module is generated constituting the environment for checker validation. As proof for feasibility of this approach is the fact that the exhaustive valid set of stimuli consisted of 33792 stimuli for FIFO control unit, 3250944 stimuli for LBDR and 1149380 stimuli for the allocator, respectively.2

Checker qualification. Then, fault simulation of the checkers combined with the control module under test is performed using the derived exhaustive valid stimuli. We have implemented a tool that simulates faults within the control module under test only and observes the output of the checker. The fault simulation allows performing analysis of which Single Event Upsets (SEUs) are covered by each of the checkers. Due to the fact that we are considering the entire exhaustive valid stimuli set, it is guaranteed that faults missed by the fault simulation are going to cause true misses. The fault tables obtained by the fault simulation procedure allow minimization of the set of the checkers while keeping the full coverage guarantees for the SEUs.

Let \( D \) be the number of true detections, \( X \) be the number of benign misses, \( W \) be the number of true misses and \( F \) be the number of false positives over all the simulation stimuli, respectively. Then we define the metrics of Fault Coverage (FC), Checkers Efficiency Index (CEI) and False Positive Ratio (FPR) as follows:

\[
FC = \frac{D + X}{D + X + W + F} \tag{1}
\]

\[
CEI = \frac{D}{D + W + F} \tag{2}
\]

2The complete set of checkers for the control part of Bonfire NoC are available at: https://github.com/Project-Bonfire/Bonfire
Here, FC shows the overall probability of the checkers detecting faults while CEI highlights the probability of checkers detecting critical faults. Finally, FPR indicates the probability of having a false positive over all the error flaggings of the checker. In the Bonfire set, none of the checkers resulted in false positives. Figure 7 shows the console output (FC, CEI and FPR metrics) along with execution time for the checker qualification tool on the example of LBDR and allocator circuits.

The proposed methodology is superior with respect to traditional fault injection, since the latter is extremely time consuming as faults have to be injected and simulated at each signal and each clock cycle separately. Furthermore, fault injection experiments provide only probabilistic results without any proof of the absence of true misses by the checkers. Moreover, previous works on applying concurrent checkers to network-on-chips result in detection latency of tens [8] or thousands [13] of clock cycles and are based on ad hoc selection of checker sets. The methodology presented here guarantees the lowest possible detection latency while maintaining low area overhead of the checking infrastructure.

Note, that the qualification framework supports also temporal checkers and/or sequential circuits (See e.g. experiments with a sequential FIFO module [15]). However, under these conditions generation of valid exhaustive qualification environment is not possible. Moreover, the absence of true misses cannot be proven and in the case of temporal checkers the latency of fault detection is exceeding the minimum.

Checker minimization.

The minimization is based on a greedy algorithm. Checkers are ranked according to their weights in terms of CEI values. Iteratively, a checker with the highest
weight is selected and moved to the resulting set. Fault information is recalculated and the faults covered by the selected checker are removed from the fault table.

In the LBDR and arbiter example, 5 checkers out of the initial 31 were proven to cover all the SEUs within the two circuits by the minimization tool. Figure 8 presents the console output of the minimization tool, showing the values for input, output variables, and checkers nodes, along with fault detection metrics (CEI, FC, and FPR). In addition, the minimized set of checkers is listed along with checkers’ detection information (fault table) and checkers’ detection absolute weights (the number of true detections) are represented.

![Figure 8: Console output of the checker minimization](image)

**Usage:** The environment of exhaustive valid stimuli is generated by an open source tool tpg.py created in IMMORTAL which can be accessed at https://github.com/Project-Bonfire/Bonfire/blob/master/Tools/TPG/tpg.py. Example of its use scenario is given below.

```plaintext
> python tpg.py −IF 2 oh 1 z 3 oh 5 we −o all_test_patterns.inp
```

The command generates a set of 11-bit input stimuli with a 2+1+3+5 bit pattern, where the first two bits are one-hot (oh), the next 1 bit is constantly zero (z), 3 next bits are also one-hot and next 5 bits will have all the possible combinations (we).

The qualification and minimization tool is an extension of TUT’s in-house fault simulator `analyze` from the Turbo Tester system [1]. In order to execute the program one needs to install Turbo Tester and run:

```plaintext
> analyze −no_drop −chk_eval −chk_module_name CHECKERS design
```

Here, `−no_drop` tells the fault simulator to generate a full fault table and not to drop faults. `−chk_eval` switches on the checker qualification mode, the name of the Verilog/VHDL module (CHECKERS) containing the checkers follows the `chk_module_name` option. `design` is the name of the design combined with the checkers.

**Connection to other IMMORTAL tools:** TUT’s Checker Qualification and Minimization framework is linked to DLR’s Property Generation tool presented in
Subsection 3.4 and IBM’s Reliability Analysis tool presented in Subsection 3.6 as follows. DLR’s automatically generated SystemVerilog properties are synthesized to register-transfer level VHDL or Verilog checkers using the well-known FoCs software by IBM [2]. These checkers, in turn, are used as an input to checker qualification and minimization. Subsequently, the resulting circuit with a minimized set of checkers is analyzed by IBM’s reliability analysis tool in order to automatically identify the protected areas within the circuit.

### 3.6 Reliability Analysis

The **Reliability Analysis** is part of the **Detailed Design** stage. Its objective is to finalize the design for reliability so that the pre-defined reliability targets are meet. It is applied after the **Checkers synthesis and minimization** stage and in addition after the design team has potentially added manually protection mechanism and corresponding checkers; such protections could be put, for example, in specific areas identified as highly susceptible to soft errors.

The reliability targets are defined in the Requirements and Architecture phase and considers several aspects - e.g. the specific application relevant to the specific hardware (CPS) component. These targets varies from one case to another. Such a target could be a specific FIT target - as described in Section 2.5 of D3.2; it could be a requirement that all memory elements with *soft error vulnerability* - the likelihood that a soft error in that memory element will propagate and affect the application - above certain threshold are protected; and it could be something else.

To finalize the design for reliability as above, the following two tools - the vulnerability estimation and the protection analysis - are applied iteratively. Both are executed on a netlist representation of the design

- **Vulnerability estimation.** This tool provides an estimation of how vulnerable each memory element is for soft error. Namely, how likely it is that a bit flip in each memory element will propagate and affect the application. The tool is based on an extension of the work from [11] to sequential elements.

- **Protection analysis.** This tool identifies and analyzes error detection and correction mechanisms. These mechanisms usually follow regular structures, making their identification feasible. The identification is performed using a combination of static analysis methods (based on formal and structural analysis) as well as dynamic methods. The identified detection mechanisms include parity protection, residue protection, orthogonality checking and ECC error detection and correction mechanism. This is all described in D3.1 and D3.2.

After each application of the tools the reliability targets are checked against what is defined in the Requirements and Architecture phase. If the reliability targets are not met certain means would be taken; the most common mean would be to add
immortal

Integrated demonstrator on modelling, verification and reliability analysis

... protection; but other means are also possible - e.g. specifying that certain areas of the logic are *spare* and should not be considered in the reliability analysis, as described in Section 2.5 of D3.2. The process is terminated and the design for reliability is completed once the reliability targets are met.

### 3.7 Reliability verification

The **Reliability verification** is part of the *Integration Test and Verification* stage. Its objective is to verify that the design for reliability defined in the Reliability Analysis stage is indeed in place and that the derived protection mechanisms function correctly. It is applied after the implementation, when a complete functional testbench is already established and in place. While the design analysis stage is performed by the logic design team, this stage is performed by the verification team.

The correct functionality of the protection mechanisms is highly related to correct *gating* between the error detection logic and the checkers which initiate the recovery logic. Each recovery action is very expansive; in many cases memory elements hold invalid data which is not being used; and saving power becomes more and more essential. All these factors result in the essential need to introduce *gating* between the checks and the actual checkers which initiate the machine recovery. The gating will prevent the checker from firing and as a result from initiating a recovery process when the data is invalid and is not being used. However, this also introduces an additional verification challenge: if the gating are *over gating* then a bit flip in a memory element which is protected could still result in silent data corruption. An exhaustive verification process must verify that we are not over gating.

The **Reliability Verification** is composed of the following two sub-stages

- **Protection analysis.** Identifies and analyzes error detection and correction mechanisms, as described above in Subsection 3.6.

- **Protection verification.** This stage takes advantage of the data gathered by the protection analysis and of the fact that any industrial design has to be functionally verified, and as such has a simulation (and in many cases also a formal) testbench. Data gathered by the protection analysis is being processed and then synthesized into the standard functional verification testbench, to verify the protection mechanisms function as designed and that there is no over gating. This results in enhanced verification testbench that also verifies the correctness of the protection structures and specifically of the gating. This is described in details in Section 2.4 of D3.2 and was published in [3].

Thus, in the reliability verification stage we first apply, again, the protection analysis. The objective here is twofold: first, to verify the required protection mechanisms designed in the analysis stage are indeed in place. And second, to gather the
required data for applying the protection verification. Then, the verification of the protection mechanisms is performed as part of the standard verification flow. At the end of this stage it is verified that the reliability design is in place and operates correctly.

We conclude this subsection and the previous one (Subsection 3.6 and Subsection 3.7), with an example of actual invocation of the tool that performs the analysis stage. Figure 9 presents the actual command line that invokes the tool for a sub-component of the chip named `AA_ZTOP` (a real name is not presented due to confidentiality). For most sub-components in the next generation of the IBM’s mainframe, the execution terminates after a few minutes, and a few reports are produced, as described in Section 2.6 of D3.3.

For example, a report with the summary and the details of the finding of the protection analysis is produced. The summary section contains the number of memory elements and the number of protected memory elements in each hierarchy level of the sub-components (the hierarchical level are defined by the design team and identified by the tool). In addition, a summary of the number of memory elements protected by each protection type is also presented. Figure 10 is an example of such a list. Again, the full names were disguised due to confidentiality reasons.

**Figure 9: invocation of the analysis engine**

```
run_insighter --proto_name AA_ZTOP --eng protected_se_detector
```

**Figure 10: Summary of protected memory elements**

<table>
<thead>
<tr>
<th>Block</th>
<th>#memory element</th>
<th>#protected</th>
<th>%protected</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA.BB</td>
<td>36</td>
<td>36</td>
<td>100.00</td>
</tr>
<tr>
<td>DD.III</td>
<td>5561</td>
<td>4993</td>
<td>89.79</td>
</tr>
<tr>
<td>DD.II</td>
<td>378</td>
<td>227</td>
<td>60.05</td>
</tr>
<tr>
<td>II</td>
<td>831</td>
<td>657</td>
<td>79.06</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protection Type</th>
<th>#protected</th>
<th>%protected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity</td>
<td>x1</td>
<td>y1</td>
</tr>
<tr>
<td>One hot</td>
<td>x2</td>
<td>y2</td>
</tr>
<tr>
<td>DMR</td>
<td>x3</td>
<td>y3</td>
</tr>
<tr>
<td>ECC</td>
<td>x4</td>
<td>y4</td>
</tr>
</tbody>
</table>
3.8 PARTYStrategy

We apply the method we have presented in [6] on the FDIR component of the Eu:CROPIS satellite. The following results are part of a paper that is yet to be submitted.

Eu:CROPIS FDIR Specification

An important task of each space and satellite system is to maintain its health state and react on failure. In modern space systems this task is encapsulated in the FDIR component, which collects the information from all relevant sensors and on-board computers, analyzes and assesses the data in terms of correctness and health, and initiates recovery actions if necessary. The FDIR component is organized hierarchically [16] with the overall objective of maximizing the system life-time and correct operation.

On the system-level, the FDIR mechanism deals with coarse-granular anomalies of the system behavior like erroneous sensor data or impossible combinations of signals. Likewise the recovery actions are limited to restarting certain sub-systems, switching between redundant sub-systems if available, or switching into the satellite’s safe mode.

Eu:CROPIS FDIR. In Figure 12 we illustrate where the FDIR component of the Eu:CROPIS on-board computing system is placed in practice and in Figure 13, we give a high-level overview of the FDIR component and its environment. The FDIR
component regularly obtains system information from two redundantly-designed control units, $S_1$ and $S_2$, and interacts with them via the electronic power system, EP. The control units $S_1$ and $S_2$ have the same functionality, but only one of them is active at any time. The other control unit serves as a backup that can be activated if necessary. The FDIR component requests, when necessary, the activation (or deactivation) of a control unit by the EP which regulates the power supply.

We distinguish two types of errors the FDIR component observes, called non-critical error and severe error. In case of a non-critical error, two recovery actions are allowed. Either the erroneous control unit is disabled for a short time and enabled afterwards again or the erroneous control unit is disabled and the redundant control unit is activated to take over its task. In case of the severe error, however, only the latter recovery action is allowed, i.e., the erroneous control unit has to be disabled and the redundant control unit has to be activated. If the redundant control unit as well shows erroneous behavior, the FDIR component initiates a switch of the satellite mode into safe mode. The safe mode is a fall-back satellite mode designed to give the operators on ground the maximum amount of time to analyze and fix the problem. It is only invoked once a problem cannot be solved on-board and requires input from the operators to restore nominal operations.

**LTL specification.** We model the specification of the FDIR component in LTL. Let $I_{FDIR} = \{ \text{mode}_1, \text{mode}_2, \text{err}_{nc}, \text{errs}, \text{reset} \}$ and $O_{FDIR} = \{ \text{on}_1, \text{off}_1, \text{on}_2, \text{off}_2, \text{safemode} \}$ be the Boolean variables corresponding to the input signals and the output signals of the FDIR component, respectively.

These Boolean variables are abstractions of the real hardware/software implementation. The values of the Boolean variables are automatically extracted from the information which is periodically collected from EP (mode$_1$, mode$_2$) and $S_1$ or $S_2$.
Table 2: Descriptions of input, observable output and hidden output signals of the FDIR component.

<table>
<thead>
<tr>
<th>Boolean variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode₁</td>
<td>true iff S₁ is activated</td>
</tr>
<tr>
<td>mode₂</td>
<td>true iff S₂ is activated</td>
</tr>
<tr>
<td>err_nc</td>
<td>true iff a non-critical error is signaled by S₁ or S₂</td>
</tr>
<tr>
<td>errₘ</td>
<td>true iff a severe error is signaled by S₁ or S₂</td>
</tr>
<tr>
<td>reset</td>
<td>true iff the FDIR component is reset</td>
</tr>
<tr>
<td>on₁</td>
<td>true iff S₁ shall be switched on</td>
</tr>
<tr>
<td>off₁</td>
<td>true iff S₁ shall be switched off</td>
</tr>
<tr>
<td>on₂</td>
<td>true iff S₂ shall be switched on</td>
</tr>
<tr>
<td>off₂</td>
<td>true iff S₂ shall be switched off</td>
</tr>
<tr>
<td>safemode</td>
<td>true iff the FDIR component initiates the safemode of the satellite</td>
</tr>
<tr>
<td>lastup</td>
<td>true if the last active system was S₁ and false if the last active system was S₂</td>
</tr>
<tr>
<td>allowswitch</td>
<td>true iff a switch of S₁ to S₂ or S₂ to S₁ is allowed</td>
</tr>
</tbody>
</table>

(err_nc, errₘ). The two error variables encompass multiple error conditions (e.g. communication timeouts, invalid responses, electrical errors like over-current or under-voltage, etc.) which are detected by the sub-system. The reset variable corresponds to a telecommand sent from ground to the FDIR component. For the output direction the values of the variables are used to generate commands which are sent to the EP or the satellite mode handling component. Additionally, we use the auxiliary Boolean variables \( O' = \{\text{lastup, allowswitch}\} \) to model state information on specification level which does not correspond to any real signals in the system. These auxiliary variables serve as unobservable outputs of the FDIR component. In Table 2, we summarize the Boolean variables involved in the specification and their meaning.

The complete LTL specification of the FDIR component consists of the assumptions A1-A6 and the guarantees G1-G13. All properties are listed in Table 3, expressing the following intentions:

A1 Whenever both systems are off, then there is no running system that can have an error. Thus, the error signals have to be low as well.

A2 The error signals are mutual exclusive. If the environment enforces a reset then both error signals have to be low, because we assume that ground control has taken care of the errors.

A3 After a reset enforced by the environment, one of the two systems has to be running and the other has to be off.

A4 Whenever the FDIR component sends \(\text{on₁}\), we assume that in the next time step system number one is running \(\text{mode₁}\) and the state of the second system \(\text{mode₂}\) does not change. The same assumption applies analogously for \(\text{on₂}\).

A5 Whenever the FDIR component sends \(\text{off₁}\), we assume that in the next time step system number one is off \(\neg\text{mode₁}\) and the state of the second system \(\text{mode₂}\) does not change. The same assumption applies analogously for \(\text{off₂}\).
A6 We assume that the environment, more specifically the electronic power unit, is not immediately free to change the state of the systems when there is no message from the FDIR component. It has to wait for one more time step (with no messages of the FDIR component).

G1 This guarantee stores which system was last activated by the FDIR component.

G2 We require the signals $on_1, off_1, on_2$, and $off_2$ to be mutually exclusively set to high.

G3 Whenever both systems are off, then the FDIR component eventually requests to switch on one of the systems ($on_1, on_2$) or activates safemode or observes a reset.

G4 We restrict the FDIR component to not enter safemode as long as the component can switch to the backup system.

G5 The FDIR component must not request to switch on one of the systems ($on_1, on_2$) as long as one of the systems is running.

G6 Whenever the FDIR component is not allowed anymore to switch to the backup system, then it must not request to switch the backup system on.

G7 Once the FDIR component switches to the backup system it is not allowed anymore to switch again (unless the environment performs a reset, see G9).

G8 As long as the FDIR component only restarts the same system it is still allowed to switch in the future.

G9 A reset by the environment allows the FDIR component again to switch to the backup system if required.

G10 Whenever the FDIR component is in safemode it must not request to switch-on one of the systems ($on_1, on_2$).

G11 Once a switch is not allowed anymore and the environment does not perform a reset, then the switch is also not allowed in the next time step.

G12 Whenever the FDIR component observes a server error ($err_s$), it must eventually switch to the backup system or activate safemode unless the environment performs a reset or the error disappears by itself (without restarting the system).

G13 Whenever the FDIR component observes a non-critical error ($err_{nc}$), it must eventually switch to the backup system or activate safemode or the error disappears (restarting the currently running system is allowed).

**Experimental Results**

The test strategy computation from the specification is independent of the implementation and was presented in D2.2.

We have also presented first results of mutation testing already in D2.2. We repeat them here and give a more detailed analysis by (1) visualizing a faulty execution,
| A1 | $G(\neg \text{mode}_2 \land \neg \text{mode}_1 \rightarrow \neg \text{err}_{nc} \land \neg \text{err}_s)$ |
| A2 | $G(\neg \text{err}_{nc} \lor \neg \text{err}_s) \land G(\text{reset} \rightarrow \neg \text{err}_{nc} \land \neg \text{err}_s)$ |
| A3 | $G(\text{reset} \rightarrow X(\text{mode}_2 \oplus \text{mode}_1))$ |
| A4 | $G(\neg \text{mode}_1 \land \text{on}_1 \land \neg \text{off}_1 \land \neg \text{on}_2 \land \neg \text{off}_2 \land \neg \text{reset} \land \neg \text{safemode} \rightarrow \text{X}(\text{mode}_1) \land (\text{mode}_2 \leftrightarrow X(\text{mode}_2)))$ |
| A5 | $G(\neg \text{mode}_1 \land \neg \text{on}_1 \land \neg \text{off}_1 \land \neg \text{off}_2 \land \neg \text{reset} \land \neg \text{safemode} \rightarrow X(\neg \text{mode}_1) \land (\text{mode}_2 \leftrightarrow X(\text{mode}_2)))$ |
| A6 | $G(\neg \text{mode}_2 \land \neg \text{mode}_1 \land \neg \text{off}_1 \land \neg \text{on}_1 \land \neg \text{off}_2 \land \neg \text{reset} \land \neg \text{safemode} \land X(\neg \text{reset}) \land X(\neg \text{safemode}) \land X(\text{mode}_1))$ |
| G1 | $G((\text{on}_1 \land \neg \text{off}_2) \rightarrow (X(\text{lastup})))$ |
| G2 | $G((\neg \text{off}_1 \land \text{on}_2 \land \neg \text{off}_2))$ |
| G3 | $G((\neg \text{off}_2 \land \text{on}_1 \land \neg \text{off}_1))$ |
| G4 | $G(\text{allowswitch} \rightarrow \neg \text{safemode})$ |
| G5 | $G((\text{mode}_2 \land \text{mode}_1) \rightarrow \neg \text{on}_1 \land \neg \text{on}_2)$ |
| G6 | $G((\neg \text{allowswitch} \land \text{lastup} \rightarrow \neg \text{on}_2)$ |
| G7 | $G((\neg \text{reset} \land \text{allowswitch} \land \text{lastup} \land \text{on}_1) \rightarrow (\neg \text{allowswitch}))$ |
| G8 | $G((\text{allowswitch} \land \neg ((\text{lastup} \land \text{on}_1) \lor (\neg \text{lastup} \land \text{on}_1)))) \rightarrow X(\text{allowswitch}))$ |
| G9 | $G(\text{reset} \rightarrow X(\text{allowswitch}))$ |
| G10 | $G(\text{safemode} \rightarrow (\neg \text{on}_1 \land \neg \text{on}_2))$ |
| G11 | $G((\neg \text{allowswitch} \land \neg \text{reset}) \rightarrow X(\neg \text{allowswitch}))$ |
| G12 | $G(\neg \text{mode}_1 \land \neg \text{reset} \rightarrow F(\text{reset} \lor \text{safemode} \lor \text{mode}_2 \lor (\text{mode}_1 \lor \neg \text{err}_{nc})))$ |
| G13 | $G(\neg \text{err}_{nc} \land \neg \text{mode}_1 \land \neg \text{reset} \rightarrow F(\text{reset} \lor \text{safemode} \lor \text{mode}_2 \lor (\text{mode}_1 \lor \neg \text{err}_{nc})))$ |
(2) removing equivalent mutants from the analysis, (3) presenting the individual fault detecting capabilities of the different fault models, (4) comparing the results to random testing and (5) providing a code coverage analysis.

Test setting. In the Eu:CROPIS satellite the FDIR component is implemented in C++. The implementation is not an exact realization of the specification in Table 3 but extends it by allowing commands to the EP to be lost (e.g. due to electrical faults). This is accommodated by adding timeouts for the execution of the switch-on/off commands and reissuing the commands if the timeout is triggered.

The implementation is designed with testability and portability in mind and uses an abstract interface to access other sub-systems of the satellite. This allows to exchange the used interface with a set of test adapters which connect to the signals generated by the test strategies. As we are only interested in the functional properties of the implementation, we can run the code on a normal Linux system, instead of the microprocessor which is used in the satellite. This gives access to all Linux based debugging and test tools and allows us to use gcov to measure the line and branch coverage of the source code.

A time step of a test run consists of the following operations: request values for the input variables $I_{FDIR}$ from the test strategy; feed the values to the test adapter from which they are read by the FDIR implementation; run the FDIR implementation for one cycle; extract the output values $O_{FDIR}$ from the test adapter and feed them back to the test strategy to get new input values. For each time step the execution trace is recorded, i.e., the values assigned to the inputs $I_{FDIR}$ and outputs $O_{FDIR}$ of the FDIR component.

Mutation testing. We apply mutation analysis to assess the effectiveness, i.e., fault finding abilities, of a test suite. A test suite kills a mutant program $M$ if it contains at least one test strategy that, when executed on $M$ and the original program $P$, produces a trace where at least one output of $M$ differs in at least one time step from the respective output of $P$ (for the same input sequence). A mutant program $M$ is equivalent to the original program $P$ if $M$ does not violate the specification.

For our evaluation we manually identify and remove equivalent mutants. We generate mutant programs of the C++ implementation of the FDIR component by systematically introducing the following four mutations in each line: 1) deletion of the line, 2) replacement of $\text{true}$ with $\text{false}$ or $\text{false}$ with $\text{true}$, 3) replacement of $==\text{ with }!=\text{ or }!=\text{ with }==\text{, and 4) replacement of }&&\text{ with }||\text{ or }||\text{ with }&&\text{.}$

In total, 198 mutant programs are generated. We use the GNU compiler gcc to remove all mutant programs which do not compile and thus do not conform to the C++ programming language. Also all mutant programs which fail during runtime e.g. by raising a segmentation fault are removed. We analyzed the remaining 96 mutants manually and identified 23 mutants that are correct with respect to the specification, i.e., equivalent mutants. Thus, 73 mutants violate the specification. Moreover, 11 of these 73 mutants can only violate the specification if the $\text{off}_1$ and $\text{off}_2$ commands can fail, which contradicts our assumptions on the EP unit. We keep those mutants to check whether the strategies can kill them nevertheless.

Next, we executed all test strategies on the mutant programs for 80 time steps each and log the corresponding execution traces.

From the 73 mutants that violate the specification, our strategies all together are
Figure 14: Test strategy that tests for a stuck-at-0 fault of signal safemode.

Figure 15: Execution trace from a faulty system under the strategy that tests for a stuck-at-0 fault of signal safemode. Bold signals are controlled by the strategy.

able to kill 52, i.e., we achieve a mutation score of 71.23%. If we do not take the 11 mutants into account that violate our assumptions for the test strategy generation, then the mutation score increases to 80.65%.

We illustrate in Figure 15 the execution of the test strategy from Figure 14 (this strategy was already presented in D2.2) on a mutant. This strategy aims for revealing a stuck-at-0 fault of signal safemode. The test strategy first forces the FDIR component to eventually switch to the backup system. The switch happens in time step 14 after several restarts of the system. Then the strategy forces the FDIR component to eventually activate safemode. However, this mutant is faulty and instead of activating safemode the system remains silent from time step 26 onwards. Thus, violating guarantee G3\(^3\).

As the strategies are only derived from requirements, without any implementation-

\(^3\)Given that the user has decided that we have waited long enough for safemode to become true.
Table 4: Mutation coverage by fault models and signals when executing all four derived strategies.

<table>
<thead>
<tr>
<th>Output</th>
<th>Fault Model</th>
<th>S-a-0 [%]</th>
<th>S-a-1 [%]</th>
<th>Bit-Flip [%]</th>
<th>All [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>on₁</td>
<td></td>
<td>65.75</td>
<td>39.73</td>
<td>5.48</td>
<td>65.75</td>
</tr>
<tr>
<td>off₁</td>
<td></td>
<td>5.48</td>
<td>4.11</td>
<td>9.59</td>
<td>9.59</td>
</tr>
<tr>
<td>safemode</td>
<td></td>
<td>61.64</td>
<td>6.85</td>
<td>6.85</td>
<td>61.64</td>
</tr>
<tr>
<td>All</td>
<td></td>
<td>71.23</td>
<td>39.73</td>
<td>9.59</td>
<td>71.23</td>
</tr>
</tbody>
</table>

specific knowledge, they are applicable on any system that claims to implement the given specification. The mutation score of 71.23% illustrates that our strategies, although computed for only three different faults that are assumed to only affect a single output signal, are also sensitive to many other faults.

If we only apply one of the four strategies we computed per fault model and output signal, then the resulting test suite can kill (1) 51 mutants, (2) 51 mutants, (3) 49 mutants and (4) 49 mutants. While one strategy per fault and output already achieves a high mutation score, these numbers illustrate the advantage of computing multiple strategies per fault model and output signal.

In Table 4 we present the mutation score of the individual combinations of signals and fault models. From all the mutants killed, there were 9 mutants only killed by a single signal / fault model combination, namely on₁ with stuck-at-0 assumption exclusively killing 7 mutants and safemode with stuck-at-0 assumption exclusively killing 2 mutants.

Random testing. We compared the fault finding abilities of the generated test strategies and random testing executed for 100, 10’000, and 100’000 time steps, respectively. For random testing we use a similar test setup to the test strategy setup, but instead of requesting the input values $I_{FDIR}$ from a test strategy we use uniformly distributed random values. For each time step, the input and output values are recorded. For each mutant the same input sequence is supplied and the output sequence of the mutant is compared to the output sequence of the actual implementation.

Random testing for 100 time steps killed 46 mutants (mutation score of 63%), while random testing for 10’000 time steps killed 69 mutants (mutation score of 94.5%). With increased time steps the results stayed the same. Random testing for 100’000 time steps killed 69 mutants as well.

Our strategies are able to kill three mutants that are missed by all of the three random test sequences. These mutants can only be killed when executing certain input/output sequences and it is very unlikely for random testing to hit one of the required sequences. The corresponding sequence requires that a sequence of $err_{nc}, mode₁$ going low and $mode₁$ going high is executed multiple times before either $err_s$ or reset is triggered.

One mutant is neither covered by the test strategies nor by the random sequences. This mutant requires a longer sequence as well in order to be executed. The mutant is not covered by the test strategies because the sequence is about the timeout of
Table 5: Code coverage.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Time steps</th>
<th>Line coverage [%]</th>
<th>Branch coverage [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>100</td>
<td>80.5</td>
<td>64.8</td>
</tr>
<tr>
<td>Random</td>
<td>10'000</td>
<td>96.3</td>
<td>85.2</td>
</tr>
<tr>
<td>Random</td>
<td>100'000</td>
<td>96.3</td>
<td>85.2</td>
</tr>
<tr>
<td>Test strategy</td>
<td>80</td>
<td>76.8</td>
<td>64.8</td>
</tr>
<tr>
<td>Together</td>
<td></td>
<td>97.6</td>
<td>87.0</td>
</tr>
</tbody>
</table>

an EP command, which is not covered by the specification from which the test strategies are derived.

**Code coverage.** Table 5 lists the line coverage and branch coverage measured with `gcov` for the different testing approaches. The table is built as follows: each line belongs to one testing approach. The first column names the approach, the second column lists the number of time steps, and the third and the fourth column present the line and branch coverage. Overall, the random testing approaches achieve a higher code coverage than the generated adaptive test strategies when executed on the source code of the FDIR component. The test strategies are directly derived from the specification and independent from a concrete implementation. Parts of the implementation which refine the specification or which are not specified at all are not necessarily covered. As mentioned above, the implementation adds timeouts for operations of the EP. Manual analysis revealed that removing the corresponding instructions would increase the line coverage to 87.3% and the branch coverage to 74.5%. In combination random tests and our strategies together achieve a line coverage of 97.6% and a branch coverage of 87%.
4 Status on Measurable Objectives

In this section we consider the presented tools with respect to the measurable objectives. Figure 16 shows the V-Model development flow with the IMMORTAL tools as presented in Section 2. Additionally, we included the numbers of the related measurable objectives. Red numbers indicate measurable objectives that the corresponding tool helps to achieve, and blue numbers indicate measurable objectives which the tool not helps to achieve but to validate.

Note that not all the tools are directly linked to a specific measurable objective, this may have several reasons: The tool provides an overall improvement, and thus cost and development time reduction, to the whole design flow (the CPS simulation environment is such a case), or the tool merely provides input for other tools (e.g. the property generation).

In the following we provide information how the different tools help to achieve the measurable objectives.
**MO1: Verification effort minimized 2x**

Verification of CPSs is typically performed by manually created stimuli. This is notoriously incomplete and time-consuming. Therefore, automatic test case generation methods were examined and the Automated Test Strategy Generation Tool called PARTYStrategy, described in Section 4 of D2.2, was developed. The output of this tool contains test strategies valid for every implementation of the specification.

The approach was applied in a case study of the Eu:CROPIS satellite for the evaluation, as described in Section 3.8. This case study includes the verification of system-level FDIR in the context of a satellite.

We compared the verification effort of manual testing with the verification effort to test the component with the automated generated test strategies from PARTYStrategy. The results can be seen in Table 6. If we test the FDIR component manually, which includes the definition of the FDIR rules, we have a moderate specification effort. However, we have a much higher verification effort afterwards, since we have to define, implement and run the tests manually. In comparison, if we use PARTYStrategy, we have a high specification effort, because a formal specification of the FDIR component is required for PARTYStrategy. Once the specification has been created, test strategies are generated automatically using PARTYStrategy. Since these have a uniform format, the execution of the tests can be automated, minimizing the verification effort. In addition, parts of the specification can be reused for future satellite projects, reducing the specification effort.

<table>
<thead>
<tr>
<th>Table 6: Evaluation of verification effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual testing</td>
</tr>
<tr>
<td>specification effort</td>
</tr>
<tr>
<td>verification effort</td>
</tr>
</tbody>
</table>

The results of the evaluation have shown that overall, the verification effort for a single project was minimized by a factor of 1.8. This factor increases for future satellite projects when parts of the specification are reused. The verification effort can therefore be minimized with a factor greater than or equal to two.

**MO2: Speeding up FDIR 4x**

The tools presented in this demonstration are not related to this measurable objective.

**MO3: Up to 15% of network resources can fail**

The tools presented in this demonstration are not helps to reaching this goal, however the network analysis and network testing can be used to validate that this goal
has been reached. For this minimal performance parameters of the network has to be defined and then the network testing can show which combination of failed network resources leads to a violation of those performance parameters.

**MO4: Up to 40% reduction in reliability related tasks**

In the following we show that using our tools the resources invested in reliability analysis and verification as per IBM’s methodology can be reduced by 40% or even more. Our analysis is based on experimental results on IBM’s chips and specifically on the next generation of the IBM’s mainframe, also known as system z. Due to confidentiality we can not provide explicit numbers; rather we will provide lower bounds of the finding of our tools, with which we can show that in the relevant tasks the required manual effort is indeed reduced by 40%.

The process of reliability analysis and verification (prior to the development of our tools within the IMMORTAL project) are as follows:

1. Verifying the correctness of the error detection and recovery logic
   (a) Manual analysis and review by designer
   (b) Reliability verification (error injection) by verification engineers
2. Analyzing the overall chip- and system-level reliability
   (a) Manual preparation of flipflops annotation
   (b) FIT calculation, potentially adding protections/signaling logic as *spare* - logic that should have no affect on the FIT, e.g. debug logic.

   This iterative process is repeated until the desired reliability targets are reached.

Our tools, as described in D3.1 and D3.2 and in Subsections 3.6 and 3.7 above, identifies error detection and correction mechanisms and the flipflops that are protected by these; the tools produce detailed and hierarchical reports listing the protected and not protected flipflops in each hierarchical level of the design. Analysis (performed by highly trained logic designers) of the results of our tools on IBM’s chips and specifically IBM’s next generation of the mainframe reveals that our tool identifies a vast majority of the protected flipflops. Thus, the required manual effort by the relevant tasks performed by the design team is reduced significantly. This effort concerns Task 1a which is performed in the early design stage and Task 2a which is performed in the late stages of the design stage. The FIT calculation in stage 2b relies on the flipflops annotation of stage 2a; as such it is now completely automated.

Specifically, Task 1a is now reduced to comparing the lists provided by the tool to the lists of the flipflops that were planned to be protected, and closing these gaps or providing explanations as to their existence. The gaps can be the result of a bug of the designer who forgot to put protection, or of a protection which is highly irregular that is thus not identified by the tool; it can also be a result of a bug in the tool. But all these cases put together should form a small portion of the cases, and as such the manual effort in this task should be significantly reduced, by even
more than 40%. Task 1a is now reduced to annotation of these flipflops that are protected but the tool did not identify as such; as mentioned this is a small portion of the flipflops, far below 50%.

Now consider Task 1b. As described in D3.1 and D3.2 and in Subsection 3.7 above, the flipflops that were detected as protected are now verified as part of the standard verification flow, and a separate and stand alone effort is no longer required. It is still required to verify the recovery process and perhaps error inject into flipflops that the design team claim are protected but that were not identified by our tool as such. Given that our tools identifies a vast majority of the protected flipflops, the reduction in effort here is significant and specifically reaches the desired 40%. Moreover, while in former error injection strategies only a small portion of the flipflops were error injected into, and so the coverage was low, we are now able to perform a much more exhaustive verification, reaching much higher coverage of flipflops resulting in a more reliable design.

MO5: Up to 10% area savings by optimizing hardware protection logic

Highly-dependable systems, such as CPS for automotive or aerospace applications, employ various hardware mechanisms to ensure reliable operation in the presence of soft-errors. Such mechanisms include parity protection, double or triple redundancy mechanisms and so on. While important for maintaining high reliability those hardware protection mechanisms add to the total area and power consumption of the chips and hence needs to be used carefully to ensure proper reliability while keeping area and power consumption as low as possible.

The reliability analysis and verification tools developed as part of IMMORTAL will be used to assess the effectiveness of existing hardware protection mechanisms and optimize it accordingly. In particular, the following tools and methods will be used for this purpose:

**Checker minimization** Preliminary experiments [5] with this method show 3.9-23% minimization in terms of logic area, and hence, proportionally in the power consumption of the related checking infrastructure. On top of that, additional gains are expected to be achieved by the vulnerability estimation explained below.

**Vulnerability estimation** This method, developed in WP3, is aimed at providing a ranked list of latches, based on latch vulnerability to soft errors. As this method is based on realistic workloads, it offers an accurate view of the vulnerability of latches to soft errors. For optimizing a given design, this method is used while ignoring existing error protection structures. This way we can identify latches which are protected but have low vulnerability in the given workloads. In such cases the protection of those latches can be relaxed or even removed without sacrificing the overall reliability of the chip.
5 Conclusion

In this deliverable we presented a set of tools developed within the IMMORTAL project and explained how they are used within the development flow. The presented tools work on different stages in the development and include tools to handle the software as well as the hardware part of the CPS.

Furthermore, we give numbers to show that the related measurable objectives have been reached.

For the Final review meeting we plan to show a demonstration of the following tools:

- CPS simulation environment
- Checker synthesis and minimization
- Reliability analysis
6 References


