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

System-level reliability analysis tools (Deliverable D3.3)

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

Revision 1.0

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

For information, contact Prof. Dr. Jaan Raik, e-mail: jaan@pld.ttu.ee.

This document is intended to fulfil the contractual obligations of the IMMORTAL project concerning deliverable D3.3 described in contract number 644905.

© Copyright IMMORTAL 2018. 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>0.7</td>
<td>August 31, 2017</td>
<td>Preliminary version</td>
<td>all</td>
<td>all sections</td>
</tr>
<tr>
<td>1.0</td>
<td>February 28, 2018</td>
<td>Final version</td>
<td>all</td>
<td>all sections</td>
</tr>
</tbody>
</table>

Authors, Beneficiary

Jan Malburg, DLR  
Annika Ofenloch, DLR  
Eli Arbel, IBM  
Shiri Moran, IBM  
Franz Roeck, TUG  
Hans Kerkhoff, UT  
Rinat Iusupov, TUG

About

This document presents the methods to obtain a system-wide view of reliability developed in task T3.3.
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMS</td>
<td>Analog/Mixed-Signal</td>
</tr>
<tr>
<td>CMS</td>
<td>Consolidated Masking Scheme</td>
</tr>
<tr>
<td>CPS</td>
<td>Cyber-Physical System</td>
</tr>
<tr>
<td>DDG</td>
<td>Dynamic Dependency Graph</td>
</tr>
<tr>
<td>FXU</td>
<td>Floating Point Unit</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>ISU</td>
<td>Instruction Store Unit</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>NI</td>
<td>Non-interference</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SAT</td>
<td>Boolean Satisfiability</td>
</tr>
<tr>
<td>SCA</td>
<td>Side-Channel-Analysis</td>
</tr>
<tr>
<td>SMT</td>
<td>Satisfiability Modulos Theory</td>
</tr>
<tr>
<td>SNI</td>
<td>Strong Non-interference</td>
</tr>
<tr>
<td>SVA</td>
<td>SystemVerilog Assertions</td>
</tr>
<tr>
<td>TI</td>
<td>Threshold analysis</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst Case Execution Time</td>
</tr>
</tbody>
</table>
Executive Summary

This deliverable describes tools and methods to improve and verify system wide reliability.

The tools presented in this deliverable can be divided into two groups: First, the tools that are used during development to help to reduce the effort of reliability related task or help to improve the reliability of the system. The tools in this group are the CPS-Simulation-Environment and the property generation for latency requirements. The second group are the tools that are used to verify or measure the reliability. This group contains the tools: Analysis and Verification of Cross Component communication, Runtime Verification for LTL, Hardware verification for security, and Reliability Metrics of AMS Components in Integrated CPS.

CPS-Simulation-Environment:
The simulation environment for CPS has been extended to detect functional errors in the early development phase of the system. This includes the detection of causality errors and livelocks during simulation.

Property generation for latency requirements:
In this deliverable we describe and evaluate our approach of generating conditions under which a designs fulfills a given latency requirement. This approach is similar to worst case execution time (WCET) analysis in the sense that its targets to ensure that latency requirement is fulfilled. However, in contrast to WCET our approach does not just says whether the latency is fulfilled always or not. Instead it provides the designer with the information under which inputs the latency is fulfilled. One advantage is that there are designs which do not have a finite WCET or the WCET may not fulfill the latency requirement and the design cannot be changes, for example when it is some third party Intellectual Property IP. In such a case the information provided by our approach can help the developer to only use the IPD in a way that the latency requirement is still ensured.

The approach is based on Dynamic Dependency Graphs (DDGs). First, we extract data-transfers from DDGs created by use cases of the design. From those transfers we extract the conditions for those transfers. Next, we divide those conditions into progress conditions, which help the transfer, and idle conditions, which stall the transfer. Based on this we compute for the different starting states of the transfer the amount of progress conditions required and generate the corresponding formula. The approach was evaluated on two designs showing it functionality as well as low run time requirement.

Analysis and Verification of Cross Component communication:
For CPS system level reliability analysis we focus in this part at the reliability of data that is being transferred between different components of the CPS. To this end, we took as a case study the communication between microprocessor units in a modern, large-scale microprocessor design. The underlying assumption of our methodology is that every data that is being communicated from one unit to the other must be communicated along with redundancy bits for providing reliability protection. Based on this we built a methodology that can validate the proper protection of data which crosses unit boundaries via unit interfaces.

The analysis is comprised of 4 steps: 1) Interface Analysis, validating that all data bits in the interface have corresponding protection bits 2) Logic Analysis, in
which we detect cases where the sender or the receiver are not handling the interface protection bits properly, 3) Error Injection, by which we verify that the errors are indeed being detected in the receiving units and 4) Error Detection Analysis in which error injection results are compared to interface and logic analysis results.

The proposed methodology is based on various capabilities developed during IMMORTAL in WP3 and was experimented with on a real IBM microprocessor design.

**Runtime Verification for LTL:**
We developed a counting semantics for LTL that allows the user to interpret finite traces with respect to previous observed behavior. In addition to the verdicts of "satisfaction", "violation" and "inconclusive", the evaluation with the counting semantics also provides the verdicts "presumable satisfaction" and "presumable violations" to refine an inconclusive evaluation.

**Hardware verification for security:**
In the deliverable, we describe a method to formally verify if a given circuit is protected against Differential Power Analysis attacks. The method also allows to pin point the parts of HW which leads to the information leakage. We use the conservative estimation of Fourier expansion of Boolean functions in order to check statistical dependency between probes and secret variables. Further, we harness an SMT solver to efficiently make a resolution on a circuit security.

**Reliability Metrics of AMS Components in Integrated CPS:**
Approaches that has been adopted to model and verify Analog/Mixed-Signal (AMS) systems mainly suffer from two interrelated drawbacks i.e. accuracy and scalability; mainly because of the computational capabilities of the verification tools available. In this deliverable, we presented an approach to communicate spice level accuracy of AMS system to the higher level verification tools for the reliability analysis of the whole CPS system. For this purpose, we took an example of open-loop Op-amp circuit which is highly non-linear. The Op-amp is designed using TSMC 40nm process (in Cadence) and thoroughly investigated for its NBTI related degradation. To meet the computational capabilities of state-of-art verification tools (e.g. NuSMV), the I/O behavior of Op-amp is discretized using MATLAB with a 12-bit encoding to preserve its non-linear dynamics.

From the discretized behavior of the Op-amp, an equivalent FSM based Boolean model is developed using Python3. This developed Boolean model can be used for the system level verification of the CPS. With this methodology, we can consider aging related degradation effects at system level verification which is highly desirable.
# Table of Contents

Table of Revisions ................................................................. iii  
Authors, Beneficiary ........................................................... iii  
About .................................................................................. iii  
List of Abbreviations ........................................................... iv  
Executive Summary ............................................................... v  
Table of Contents .................................................................. vii  
1 Introduction ......................................................................... 1  
2 Runtime Causality Error and Livelock Detection .................... 1  
3 Computation of conditions to fulfill latency requirements .......... 5  
   3.1 Mining Latency Guarantees ........................................ 5  
   3.2 Approach .................................................................... 7  
   3.3 Case Studies ............................................................ 12  
   3.4 Conclusion ................................................................... 16  
4 Analysis and Verification of Cross Component Reliability .......... 19  
   4.1 Interface Analysis ..................................................... 20  
   4.2 Logic Analysis .......................................................... 21  
   4.3 Error Injection ......................................................... 22  
   4.4 Error Detection Analysis .......................................... 22  
5 Runtime Verification for LTL .................................................. 25  
   5.1 Motivation and Related Work ..................................... 25  
   5.2 Counting Semantics .................................................. 26  
6 Hardware verification for security ......................................... 35  
   6.1 Motivation ............................................................... 35  
   6.2 Related Work .......................................................... 36  
   6.3 Contribution ............................................................ 38  
7 Reliability Metrics of AMS Components in Integrated CPS ....... 41  
   7.1 Op-amp case study ................................................... 41  
   7.2 Verification approach ................................................. 42  
8 Conclusion .......................................................................... 47  
9 References .......................................................................... 49
1 Introduction

This deliverable, presents the methods and tools developed for Cyber-Physical System (CPS) system-level reliability analysis. The goal is to obtain a system-wide view of reliability and to ensure that the entire system meets its reliability requirements.

As CPSs often are used in safety critical environments where even human lives are effected, e.g., automotive, their reliability is of up-most importance. The tools presented in this deliverable can be divided into two groups: those tools that are used during the design in order to improve the reliability of the design or reduce the cost of the reliability related tasks and those tool that are used to verify the reliability of the CPS. For the later, not only reliability against soft-errors are considered but intentional attacks, i.e., security. This deliverable is the final deliverable for work package 3. Based on the decision from review meeting 2 this deliverable is the extension of the preliminary version submitted for M30.

The rest of the deliverable is organized as follows: First, we consider the tools which helps during design of the CPS to improve the reliability. Hence, in Section 2 we consider the identification of faults in a simulation and in Section 3 we generate properties which ensure that a given latency requirement is fulfilled. Then we consider the tools that are used to verify the reliability. Section 4 gives the attention to the communication between different sub components of a CPS to provide methods for analyzing and verifying the reliability of the communication. In Section 5 we describe our approach of verifying LTL properties on finite traces, for this we mark whether a given trace is likely to fulfill the property or not when he would continue similar the the already observed behavior. The formal verification of hardware against side channel attacks is presented in Section 6. In Section 7 we consider the verification of AMS components of a CPS. Section 8 concludes the deliverable.

2 Runtime Causality Error and Livelock Detection

In order to detect functional errors in the early development phase of the system, the simulation environment has been extended to detect causality errors and livelocks during simulation.

Causality errors occur if events are executed in the wrong order [24]. This happens when a new simulation cycle starts at $t_i$ and the previous cycle at $t_{i-1}$ has not yet
completed all delta cycles. Thus, the newly started cycle may access parameters of a model which have not yet been updated in the previous time step. This leads to misinterpretations and error propagation during the simulation. Such a critical cycle shown in Fig. 1 must be detected and intercepted. Since the simulation model and the other models do not know when the last delta cycle within a simulation cycle is completed, critical sections must be identified by checking the time stamps of the received events. Each model must check whether the time stamp from the currently received event is smaller than the time stamp of the previously received events. If this is not the case, an event from the new cycle has already been executed and the simulation must be terminated. Thereafter, the cycle time can be increased to provide more time to complete all delta cycles and the simulation must be restarted.

LVelocks occur when two or more processes are blocked. But unlike a deadlock, they do not remain in one state. Instead, they are constantly switching back and forth between several states [17]. As shown in Fig. 2, a livelock occurs when a cyclic dependency between two models in the data exchange is present and endlessly many delta cycles are executed. They will not necessarily lead to errors in other models, but indicate a general error in the defined system or models. If no events from the next simulation cycle are sent to the affected models, livelocks cannot be identified by checking the time stamps. Instead, it must be ensured that the simulation is terminated after a specific number of delta cycles. According to that, the event gets as additional information the current number of delta cycles. Another approach is to stop the execution of delta cycles as soon as no relevant state changes occur. Therefore, the simulation will be carried out completely and not be terminated prematurely.
MQueue requires a clock model (MClock) which specifies the time hardware. To integrate the simulation into the HAL, it needs a simulation environment to test the OBSW. The OBSW uses flight software simulation is used and integrated into the HAL, as shown in Fig. 5, referring to the Eu:CROPIS satellite [23]. A simulation environment is its simplicity.

An implementation of the simulation environment can be configured and loaded with events. As part of an example, we use a power control application. The power control application tightly integrates a safety-critical digital control system with its physical environment. The objective is to manage the energy sources by controlling power flows from power conditioning and distribution units (PCDUs) to satellite units. The on-board software (OBSW) of the Eu:CROPIS satellite from the German Aerospace Center (DLR) [22] shows the power of the simulation environment.

In the future, the simulation environment will be used to analyze the behavior of the system in case of errors or state transitions. Error scenarios can be simulated to test the OBSW, and livelocks can be identified, without comparing the time stamps of the events. Present and endlessly many delta cycles are executed. They will not necessarily lead to errors in other models, but indicate a general error in the defined system or models. If livelocks can be identified, they will not necessarily lead to errors in other models, but indicate a general error in the defined system or models. If livelocks are not identified, the simulation will be carried out completely and not be retried as soon as no relevant state changes occur. Therefore, the simulation environment needs to be able to detect livelocks and to handle them appropriately.

Another approach is to stop the execution of delta cycles based on the livelock detection. However, this is not always feasible, as it can lead to a loss of important data. Instead, it must be ensured that the livelocks are detected and intercepted when they occur. This can be done by analyzing the reactions of software and hardware components, and by identifying livelocks in the event queue to modify the model states. This is used to generate the corresponding telemetry data. This data is sent back to and temporarily stored in the serial model. When the livelock is detected, the corresponding data is sent to the device model via the serial interface. The device model includes a virtual PCDU that has the functionality to process the received command and generate the corresponding telemetry data. This data is sent back to the device model via the serial interface. The serial model (MSerial) is needed to provide an interface to the device model (MPDU), which performs power control tasks, which includes the acquisition of power data, the buffering of data, the reading out and returning of data, the processing of received commands, and the generation and sending of telemetry data.

This application tightly integrates a safety-critical digital control system with its physical environment. The objective is to manage the energy sources by controlling power flows from power conditioning and distribution units (PCDUs) to satellite units. The on-board software (OBSW) of the Eu:CROPIS satellite from the German Aerospace Center (DLR) [22] shows the power of the simulation environment.

The use of FRASER in the Eu:CROPIS mission of the satellite demonstrates the potential of the simulation environment.

Figure 2: Simulation process with a livelock

```
Figure 2: Simulation process with a livelock
```

Public

Runtime Causality Error and Livelock Detection • 3
3 Computation of conditions to fulfill latency requirements

Typically, CPS are real-time systems for which the specification defines time-limits for the system to react on external events, e.g., sensor measurements, or internal events, i.e., timer ticks. In order to react on an event, information about the event has to be transmitted between different block of the design, e.g., read sensor data has to be transmitted to a CPU to compute the correct reaction. In this section we are focusing on the communication delay. For communicating elements like router, buses, or intermittent storage queues are used. Each of these elements introduces some latency to the communication. However, in most cases this latency is not constant but depends on the current state of the element and ongoing cross traffic. Some elements may include some randomness, for example the arbitration scheme of Ethernet. The optimal case is that we can prove that a communication never exceed a given time-limit under any condition. This is, however, not always possible for example due to inherent randomness of the underlying protocol.

Therefore, we are computing conditions under which a certain time-limit will not fail. A first version of this approach already presented in D3.2. In this deliverable we provide in-depth technical detail about the approach and present evaluation results. In this section we additionally present the main extensions and improvements of the approach which are:

- Support of communication elements that transform the send data.
- Two optimizations to reduce the required simulation data and computation time.
- Simplification and removal of redundant parts of the conditions.

3.1 Mining Latency Guarantees

To mine latency guarantees from an RTL design we need a formal notion for conditions of successful data transfer. We consider a data transfer successful if the data supplied to a predefined input becomes observable at a designated output. The mined conditions guarantee the data transfer under any possible data value. The data may potentially be transformed according to a user-specified function. The number of clock cycles required to transfer the data from input to output is called latency. We define this concept more formally in the following.
Suppose that $D$ is a synchronous RTL design described in a synthesiseable subset of a hardware description language, e.g., Verilog or VHDL, with $n$ (word-level) primary inputs $I = i_1, \ldots, i_n$, $m$ (word-level) primary outputs $O = o_1, \ldots, o_m$, $k$ (word-level) storage elements $S = s_1, \ldots, s_k$. We write $|w|$ to denote the bit-width of signal $w \in I \cup O \cup S$. Let $\mathbb{B} = \{\text{true, false}\}$ be the set of truth-values. We construct a Kripke model $\mathcal{M}_D$ of $D$ over atomic propositions $\mathcal{AP} := \{w = v \mid w \in I \cup O \cup S, v \in \mathbb{B}^{|w|}\}$ that describes the behavior of $D$.

**The $t$-latency guarantee mining problem.** Given a designated input $\hat{i} \in I$, a designated output $\hat{o} \in O$ of $D$, and a maximal latency bound $t \in \mathbb{N}$, the $t$-latency guarantee mining problem asks for finding a temporal formula $\varphi$ over variables $I \cup S$ such that

$$\mathcal{M}_D \models G(\hat{i} \rightarrow \bigwedge_{c=0}^{t} \bigwedge_{v \in \mathbb{B}^{|\hat{o}|}} (\hat{i} = v \rightarrow X^c(\hat{o} = g(v))))$$  \hspace{1cm} (1)$$

holds when $\varphi$ substitutes the place-holder $\hat{o}$. The mapping $g : \mathbb{B}^{|\hat{i}|} \rightarrow \mathbb{B}^{|\hat{o}|}$ defines the correct, expected output value $g(v) \in \mathbb{B}^{|\hat{o}|}$ for input value $v \in \mathbb{B}^{|\hat{i}|}$. The temporal logic operators $G$ and $X$ have the usual meaning, additionally we define $X^c, c \in \mathbb{N}$, inductively as $X^c(\zeta) = X(X^{c-1})(\zeta)$ with base case $X^0(\zeta) = \zeta$ for all temporal formulas $\zeta$. The typical syntax of temporal logic requires enumerating all possible values of $\hat{i}$. Our algorithm does not unfold this formula. We call a formula $\varphi$ that satisfies (1), a *solution*.

The formula $\varphi = \text{false}$ is a vacuous solution. An optimal solution for our problem specifies the maximal number of successful data-transfers.

An assignment to the state for the first cycle and to the primary inputs for $t$ clock cycles relates to a successful data-transfer if the assignment yields a solution to the problem query. Formally, let $a : I \cup S \times \mathbb{N} \rightarrow \mathbb{B}^*$ be an assignment function that maps each state variable $s \in S$ for time step 0 to the value $a(s, 0)$ and each primary input $i \in I$ for clock cycle $c \in \{0, \ldots, t\}$ to the value $a(i, c)$. While $a$ is never accessed for a clock cycle other than 0 for state variables $s$, we do not reflect this in our definition for simplicity. There is only a single consistent sequence of state values in a deterministic design, once the first state and all primary inputs are fixed. The function $r(a)$ assigns a value to each state variable at clock cycle 0 and to each input for up to $t$ clock cycles with the exception of $\hat{i}$ in the first clock cycle:

$$r(a) := \left( \bigwedge_{i \in I \setminus \hat{i}} (i = a(i, 0)) \right) \land \left( \bigwedge_{s \in S} (s = a(s, 0)) \right) \land \left( \bigwedge_{c=1}^{t} \bigwedge_{i \in I \setminus \hat{i}} X^c(i = a(i, c)) \right)$$

**Quality of solutions.** Let $\mathcal{A}$ be the set of all possible $a$. A solution $\varphi$ is *optimal* if $\varphi$ maximizes the quality metric

$$|\{a \in \mathcal{A} \mid r(a) \rightarrow \varphi\}|.$$  \hspace{1cm} (2)$$
Correspondingly, given two solutions $\phi_1, \phi_2$. We say $\phi_1$ is better than $\phi_2$ if

$$|\{a_1 \in A \mid r(a_1) \rightarrow \phi_1\}| > |\{a_2 \in A \mid r(a_2) \rightarrow \phi_2\}|.$$

Let $\phi_1, \phi_2$ be solutions. Then $\phi_3 = \phi_1 \lor \phi_2$ is a solution, additionally $\phi_3$ is at least as good as the better of the two solutions $\phi_1, \phi_2$. Consequently, if we consider semantic equality there is only one single optimal solution.

A naïve way to construct an optimal $\phi$ is to build the disjunction of all $a \in A$ for which $r(a)$ holds. This is similar to finding an exact solution for a logic query as stated in [18]. While solutions for logic queries are only Boolean formulas, we are looking for a temporal formula. However, the temporal formula refers only to a fixed number of up to $t + 1$ time steps. Thus our problem may be reduced to logic query checking. Due to combinatorial explosion enumerating all solutions is infeasible for all but the most trivial designs. The approach we present in the section solves this problem by combining simulation, symbolic representation, and heuristics.

### 3.2 Approach

Next, we present an approach that computes a good solution for $\phi$ guided by simulation and heuristics. The provided RTL design is first instrumented and then simulated for a fixed set of use cases. During the simulation, control- and data-flow information is collected in form of Dynamic Dependency Graphs (DDGs). The DDGs are analyzed to determine successful transfers and to generate the temporal logic formula $\phi$. In detail, the approach works in five steps:

1. Generate DDGs;
2. Compute traces of successful transfers;
3. Extract conditions from transfers;
4. Categorize input conditions;
5. Generate a temporal logic formula.

During the computation we assume that (partial) assignments to primary inputs always have the same impact on data transfer. A specific assignment always either helps to transport data towards $\hat{o}$ or stops data from progressing. This assumption holds for a large class of designs and simplifies the procedures to extract generalized conditions from simulation data.

**Example 1.** Consider a 1-word data buffer which can either be read or written. Before a new element can be written, the previous element has to be read. Synchronous sequential circuits can be modeled as Moore machines with state transitions at rising clock edges. Fig. 3 presents the example design as a Moore ma-
chinate. In our example the latency requirement $t$ is 2 and the transformation function $\varepsilon$ is the identity. Input data at $i$ is stored if $\varepsilon$ is set to $R$ and the stored value is written to the output if $\varepsilon$ is set to $W$.

\[
\begin{align*}
\text{Inputs} & \quad \text{State} & \quad \text{Output} \\
\text{Transform}_1 & \quad \text{Transform}_2 & \\
\text{Output} & \\
\end{align*}
\]

\[
\begin{align*}
\text{Transform}_1 & : i \mapsto f \\
\text{Transform}_2 & : s_0 \mapsto s_1 \\
\text{Output} & : o := s_1 \\
\end{align*}
\]

\[
\begin{align*}
\text{State} & : s_0, s_1, s_2 \\
\text{Inputs} & : i \in \{R \text{ead}, W \text{rite}\} \\
\end{align*}
\]

\[
\begin{align*}
\text{Outputs} & : o \in \{E mpty, F ull\} \\
\end{align*}
\]

\[
\begin{align*}
\text{Start} & :	ext{Initial state } s_0 \in \{E mpty, F ull\} \\
\end{align*}
\]

\[
\begin{align*}
\text{Initial state} & : s_0 = E & \text{if } e = W & \text{and } s_2 = E \\
\text{Initial state} & : s_0 = F & \text{otherwise} \\
\text{Output} & : o := s_1 \text{ if } s_2 = W \\
\text{Output} & : o = 0 \text{ otherwise} \\
\end{align*}
\]

\[
\begin{align*}
\text{Transition} & : s_0 := E & \text{if } f = R & \text{and } s_0 = E \\
\text{Transition} & : s_1 := i & \text{if } f = R & \text{and } s_0 = E \\
\text{Transition} & : s_2 := W & \text{if } f = W & \text{and } s_2 = E \\
\end{align*}
\]

\[
\begin{align*}
\text{Generating the DDGs.} \\
\end{align*}
\]

DDGs are computed while simulating use cases on an instrumented version of the design. DDGs symbolically describe the data- and control-flow during the simulation of a use case. DDGs are a well-known data structure for program analysis, a description of how we compute DDG can be found in [32].

\[
\begin{align*}
\text{Figure 3: The example design represented as a Moore machine.} \\
\end{align*}
\]

\[
\begin{align*}
\text{Figure 4: A DDG for our example shown as unrolled machine. The DDG contains two successful transfers (blue and red). Bold arrows show the data-transfer. Dotted arrows show the conditions under which the transfer occurs.} \\
\end{align*}
\]

\[
\begin{align*}
1\text{Here we switch from the LTL operator ‘=’ to the corresponding SVA operator ‘==’.} \\
\end{align*}
\]
Example 2. A DDG can also be seen as an unrolling of the design under a given input sequence. Fig. 4 shows a DDG of our example in form of the unrolled Moore machine. The symbols in the figure are as defined in Fig. 3. \( i_0, i_1, \ldots \) are the values assigned to input \( i \) for the clock cycles indicated by the subscripts.

Computing traces of successful transfers.

We compute backtraces encoded by subgraphs of the DDG which include successful transfers, i.e., cases where data read at the designated primary input \( \hat{i} \) reached the output port \( \hat{o} \).

First, starting at \( \hat{i} \) having value \( v \) at clock cycle \( c_i \) we traverse the DDG to find whether output \( \hat{o} \) has value \( \epsilon(v) \) at the same clock cycle or any later clock cycle denoted by \( c_o \). Next, we compute the backtrace for \( \hat{o} \) at clock cycle \( c_o \) towards the inputs. If we hit a storage element at clock cycle \( c_i \) or earlier, we stop backtracing.

Example 3. In our example the transfer is successful for clock cycles 0 and 3, the corresponding values of \( c_o \) are 2 and 6, respectively. In Fig. 4 the data-paths of those two transfers are shown in bold, the transfer starting at clock cycle 0 has a latency of 2 and is colored red; the transfer starting in clock cycle 3 has a latency of 3 and is colored blue.

Extracting trace conditions from backtraces.

Per backtrace we extract trace conditions \( C \) sufficient to transfer the input value of \( \hat{i} \) to the output \( \hat{o} \) along the path in the DDG. Conditions result from control-flow-statements (if, switch, loops), conditional operators (?:), and array accesses. These conditions are of the form symbolic.value == concrete.value. Where concrete.value is the concrete value observed during simulation and symbolic.value is a formula over state variables of the design at clock cycle \( c_i \), primary inputs of the design from clock cycle \( c_i \) up to clock cycle \( c_o \), and constants, extracted from the DDG. We refine the conditions in such a way that a single condition only contains variables from a single clock cycle.

The set of conditions sorted into the trace conditions \( C = (C_0, \ldots, C_t) \) where \( t = c_o - c_i \) such that \( C_x \) only contains conditions over variables from the clock cycle \( c_{i+x} \). We call \( t \) the latency of the data transfer specified by \( C \). Conditions in \( C_0 \) are start conditions, they are conditions over the state variables and primary input variables except for \( \hat{i} \). Conditions in \( C_c \) where \( c \in \{1, \ldots, t\} \) are input conditions over primary input variables.
**Example 4.** Fig. 4 shows the parts of the DDG which contribute conditions with dotted lines resulting in the following trace conditions:

\[
\begin{align*}
C_{\text{red}} &= (C_{0,\text{red}}, C_{1,\text{red}}, C_{2,\text{red}}), \quad \text{where} \\
C_{0,\text{red}} &= \{(s_0 == E), (\varepsilon == R)\} \\
C_{1,\text{red}} &= \{(\varepsilon == R)\} \\
C_{2,\text{red}} &= \{(\varepsilon == W)\} \\
C_{\text{blue}} &= (C_{0,\text{blue}}, C_{1,\text{blue}}, C_{2,\text{blue}}, C_{3,\text{blue}}), \quad \text{where} \\
C_{0,\text{blue}} &= \{(s_0 == E), (\varepsilon == R)\} \\
C_{1,\text{blue}} &= \{(\varepsilon == R)\} \\
C_{2,\text{blue}} &= \{(\varepsilon == R)\} \\
C_{3,\text{blue}} &= \{(\varepsilon == W)\}
\end{align*}
\]

We simplify sets of conditions by removing simple redundancies. A typical example compares a variable once to a fixed value and once to other variables, e.g., any two of the conditions \(g==0\), \(h==0\), and \((g==h)==\text{true}\) (\(g\) equals \(h\)) suffice. Our simplification preferably keeps conditions, which assign variables to a fixed value, i.e., would remove \((g==h)==\text{true}\).

**Categorizing input conditions into progress and idle.**

According to the assumption at the beginning of Section 3.2, a partial input assignment always either supports data in progressing to the output or halts the transfer. We call input conditions supporting the transfer *progress conditions* \(\mathcal{N}_p\) and we call input conditions halting the transfer *idle conditions* \(\mathcal{N}_i\). Let \(\mathcal{N}\) be the set of all input conditions in all computed trace conditions.

**Idle conditions:**
First, we compute the set of idle conditions \(\mathcal{N}_i\). The idea is to search for traces that:

- share the same start condition,
- have different length,
- the amount of input conditions differ for exactly one input condition, and
- the difference equals the difference in the latency of the traces.

The difference in the input conditions of two such traces indicate idle conditions. The input conditions which appear more often in the longer trace are added to the set of idle conditions \(\mathcal{N}_i\). This basic approach is extended by two optimizations.

**Optimization 1:**
Once an input condition is found to be an idle condition, we consider the sequences without this idle condition. This allows to extract information about idle conditions from more traces. For example, consider traces \(T = (T_0, T_1, T_2, T_3)\) and
\( T' = (T_0, T_3) \) where \( T_1 \) and \( T_2 \) are idle conditions. Without this optimization the pair yields no information. However, using the optimization after identifying idle condition \( T_1 \), the pair yields \( T_2 \). As a result, less simulation data is required to identify all idle conditions.

**Optimization 2:**
The second optimization considers superset relations between input conditions to extrapolate idle conditions. We say an input condition \( N_1 \) is a true superset of \( N_2 \) if \( N_1 \) contains all conditions of \( N_2 \) and \( N_1 \) contains conditions that are not included in \( N_2 \), i.e., \( N_1 \) is more general than \( N_2 \). The idea of this optimization is that, if for some input condition \( N_1 \) all input conditions that are supersets of \( N_1 \) are found to be idle conditions, we also consider \( N_1 \) an idle condition. This optimization is only exact if all possible supersets of conditions are considered in the use cases. Otherwise the optimization may find spurious idle conditions, because there may be unobserved input conditions that may cause progress.

**Progress conditions:**
In the next step of the categorization, we compute the set of progress conditions \( N_p \). Our initial results showed that traces without idle conditions having a different start state than traces including idle conditions often relate to a special case. Therefore, we do not use them for the computation of progress conditions. From the remaining traces we gather all input conditions that are not idle conditions and consider them as progress conditions. Consequently, in case no idle condition has been found, the set of progress conditions is empty and all traces are considered special cases.

We apply simple containment and algebraic simplifications to the final sets \( N_p \) and \( N_i \).

**Example 5.** For our example we have:

\[
N_i = \{\{(\varepsilon = \text{R})\}\}, \{\{(\varepsilon = \text{W})\}\}
\]

Additionally, we count the number of input conditions in each trace.

\[
C_{\text{red}} : \{(\varepsilon = \text{R})\} \rightarrow 1, \{(\varepsilon = \text{W})\} \rightarrow 1
\]
\[
C_{\text{blue}} : \{(\varepsilon = \text{R})\} \rightarrow 2, \{(\varepsilon = \text{W})\} \rightarrow 1
\]

The start conditions of both traces are identical, so they are compared to identify idle conditions. The latency of the blue transfer is exactly one higher than the latency of the red transfer, which is the same difference as for the number of \( \{(\varepsilon = \text{R})\} \) in the traces. There is no other difference between both traces. Thus, \( N_i \) is \( \{\{(\varepsilon = \text{R})\}\} \) and \( N_p \) is \( \{\{(\varepsilon = \text{W})\}\} \).

**Generating the formula \( \varphi \).**

In the last step we create the final formula. First, we compute the set of special cases which are those traces without input conditions, i.e., the transfer is finished.
within the first clock cycle, or with neither progress conditions nor idle conditions. The disjunction of all special cases which fulfill the latency requirement is a solution of the problem.

For all other traces we compute the effective latency as the length of the trace ignoring clock cycles with idle conditions. Similarly, the effective input length is the number of progress conditions in \( C \).

All traces with effective latency higher than the latency requirement are removed. Traces with start conditions with identical effective input length and identical effective latency are grouped. For each such group the disjunction of all combinations of progress and idle conditions are generated that fulfill the latency requirement:

- The number of progress conditions must be equal to the effective input length.
- The number of idle conditions may at most be latency requirement minus number of progress conditions.
- The last input must be a progress condition.

The conjunctions of the start conditions in the corresponding group with the generation disjunction of input conditions are solutions for the problem. In the last step compute the disjunction of the solutions created by special cases and from the grouping and apply basic simplifications to create the final solution.

**Example 6.** In our example no special cases exists. For both of our traces the effective length and the number of progress conditions is 1. Both partitions have the same start condition \( b = \{ (s_0 = E), (f = R) \} \). Remember the latency requirement \( t \) of our example is 2, thus the two combinations are:

\[
\{(f = W), (f = R) \land X(f = W)\}
\]

### 3.3 Case Studies

We evaluate our approach using two case studies, a bus bridge and an SPI-connection written in Verilog. All experiments run in a virtual machine having access to 4 cores of an i7-4712MQ and 12GB of RAM on a 64-Bit Debian 8. For the simulation of the instrumented design a commercial simulator is utilized. The runtime given is CPU-time measured using the `getusage` system call including user and system CPU-time for the program and its child processes, i.e., the execution of the Verilog compiler and simulator.
(a) Number of clock cycles required to identify all idle conditions for different buffer sizes and optimizations.

(b) Relation between latency requirement, number of simulated clock cycles, and quality of the formula.

(c) Relation between the number of clock cycles simulated and total runtime.

Figure 5: Evaluation result for the bus bridge.
**Bus bridge.**

The bus bridge contains a buffer of configurable length implemented as ring-buffer with a default size of 8 words. The user may request to write into the internal buffer of the bridge and to read from the internal buffer. Besides its internal buffer the design can store one pending read request and one pending write request. For simulation, we apply a pseudo-random use case with a parameterizable length and fixed random seed. The use case initially resets the design and simulates random values for read- and write-requests afterwards.

**Optimized input categorization:**
In Section 3.2 we proposed two optimizations for input categorization. Optimization 1 considers already known idle conditions and optimization 2 uses superset relations between input conditions. Fig. 5a shows the number of simulated clock cycles required until correctly identifying all idle conditions for different buffer sizes, i.e., different sizes of the design’s state space. Without optimizations 50,000 clock cycles were not sufficient for buffer length of 16 and above.

In all cases considered, the combination of both optimizations only yields the same result as the better one of the two optimizations alone. Optimization 1 yields better results than Optimization 2 except for buffer size two. In conclusion both optimizations largely reduce the amount of simulation data required. Additionally, the combination Optimization 1+2 is the fastest with respect to runtime under the same amount of simulation data. However, with only an average difference of 1% between the fastest combination and the slowest (Optimization 2), the effect of the optimizations on the runtime is very small.

**Example:**
Listing 1 shows a part of the formula created for the design with a buffer size of two and latency requirement of three. The first six lines include all the start conditions with effective latency of two. Variables without period are primary inputs. Variables with periods are state variables of the top module or of a submodule with instantiation path. The fact that at the first clock cycle the data should be read is included in the non-synchronously written variable .w_state_next, therefore the read request primary input does not occur in the start condition. The last line combines idle and progress input, once only the progress input and once the idle input followed by the progress input.

```
Listing 1: An excerpt of the generated formula.
```

**Completeness of the formula:**
One reason we chosen this design was that we know the optimal formula. This allows us to assess the quality the generated formula. For buffer size 8 we manually derived all possible start conditions for a given latency requirement and assessed
the percentage found by our approach. This is a slightly different quality metric than (2). The reason for this is that for our case study the used quality metric better reflects the hard cases that have a large effective latency. However, those cases have only less freedom for the inputs yielding only a minor influence on (2). The optimal formula is identical for both metrics. Start conditions are restricted to state variables and those inputs affecting the control-flow. Our approach further reduces the start conditions to conditions over only a few state-machine variables and read-/write pointers.

Fig. 5b shows the effect of the latency parameter and amount of simulation data on the quality of the generated formula. The minimal latency for the bus bridge is two clock cycles. For buffer size 8, all successful data transfers have a latency lower than 11. For a use case of 1,750 clock cycles an optimal formula including all possible start conditions and input conditions is created for latency requirements $t = 2$ and $t = 3$. To create an optimal formula for $t = 8$ and $t = 10$, 4,000 clock cycles are required. Unexpectedly, increasing the latency requirement does not necessarily reduce the quality of the formula. Increasing the latency requirement processes more start conditions used as starting point for transfers which fulfill the latency requirement. Thus, finding relatively many new start conditions increases the relative quality. Not finding new start conditions in the traces may decrease the quality of the formula. Overall a small number of simulation cycles suffices to create good formulas for latency guarantees.

**Runtime:**

The relation between the runtime and number of simulated clock cycles is shown in Fig. 5c. While the latency requirement $t = 10$ is used, simulation, parsing the DDG, and categorization of the inputs take by far most of the runtime. The difference between different latency requirements is below the measurement precision. The runtime increases linearly with the number of clock cycles at a rate of roughly 6 seconds per 1,000 clock cycles.

**SPI-connection.**

In this case study we consider an SPI-connection from OpenCores. The SPI-clock signal is generated by the master using a configurable clock divider. The slave runs at SPI-clock-speed. The design allows two transfer-directions: from master to slave and from slave to master where the second is optional, i.e., each request transfers data from master to slave, while data from slave to master is only sent when the corresponding primary input is high. The minimal latency of the design is 31 clock cycles and the maximal latency is 241 clock cycles.

Again, we consider a random use case of configurable length that first resets the design and then applies random requests. The following constraints are applied: the primary inputs (a) that decide whether the slave should also send data to the master and (b) that decide whether the low bit or the high bit should be sent first only change when a new transfer request is issued. Changing these signals during a transfer would cause the transfer to fail. Consequently, a fully random use case would execute to few successful transfers.
In this case study our approach abstracts the state of the design to such an extend that each abstracted state defines the length of the corresponding transfer. Therefore, no idle conditions are found and all transfers are considered special cases. In contrast to the previous case study, this design has a finite worst case latency for successful transfers. We use this as latency requirement to analyze the largest relevant state space which is the hardest case for our approach. Also for this case study the inherent abstraction that our approach applies to the result allows us to manually inspect the quality of the formula, as the resulting optimal formula consists of eight abstract special cases.

Fig. 6 shows the resulting quality of the formula for different numbers of simulated clock cycles. We considered both transfer directions: from master to slave (master → slave) and from slave to master (slave → master). For data transfer from slave to master more simulation data is required as these transfers only occur when the respective input pin is high. The computation of the optimal formula for the master to slave direction needs less than 3,250 clock cycles of simulation data and 43 seconds runtime. The slave to master direction needs less than 5,000 clock cycles of simulation data and 53 seconds runtime.

3.4 Conclusion

Deriving a formal guarantee to enforce a given latency requirement may be formulated as a logic query. While solving logic queries is hard, our semi-formal
approach efficiently solves the problem. The simulation data provided guides the mining process towards wanted specifications of data transfer.

We evaluated our approach using a bus bridge and an SPI-connection. In both cases, our approach finds a formula completely describing all possibilities to fulfill the latency requirement in less than a minute.
Computation of conditions to fulfill latency requirements
4 Analysis and Verification of Cross Component Reliability

In this deliverable we approach the CPS system level reliability analysis by addressing reliability analysis of data that is being passed between different components of the CPS. Under the framework of IMMORTAL we have developed methodology and tools for analyzing and verifying reliable communication between sub-components of a microprocessor. This methodology was explored on the next generation of IBM’s mainframe. In many aspects, the communication between a microprocessor sub-components emulate communication between different components in a CPS system. In the following we describe this methodology which, with some adjustments, can also be used in the context of communication between different CPS components.

Before going forward with the description, let us recall some basic related terms mentioned in previous deliverables. An hardware error checker (or simply a checker) is a sequential element which fires, or outputs the logical value 1, in case of an error (hard or soft). Otherwise, the value of the checker is 0, indicating that no error has occurred. When an error checker fires, a recovery action is being initiated. Each error checker has a corresponding gating condition. A blocking gating condition will prevent the checker from firing. Gating conditions are widely used in industry since in many cases data is not valid or is not being used [2]. Thus, since a recovery process is very expensive and also to save power it is essential to use gating conditions. A typical hardware chip is usually divided into sub components, referred to as units, each of which defining a particular hardware functionality - e.g. load store unit, memory control unit, floating point unit. In many cases and specifically in IBM each such sub unit is owned by a dedicated design team and a dedicated verification team.

The basic underlying assumption in cross units reliability design is that every data that is being communicated from one unit to another must be communicated along with redundancy bits for providing reliability protection, referred to as interface protection bits. To the most part, this is done by adding a parity bit for chunks of data being communicated. See e.g. Figure 7; in the Figure, two vectors of eight bits each are being communicated from the Floating Point Unit - the FXU - to the Instruction Store Unit - the ISU. The two vectors are then being muxed in the ISU, and the vector that is chosen is checked against the relevant interface protection bit and is further proceeded by the ISU.

The analysis and verification of cross-units reliability is divided into four sub-stages where each stage relies on the former one; namely, if certain data failed
sub-stage 1 then sub-stage 2 is not even applicable for it. The first two stages are static while the last two relies on dynamic simulation. The sub-stages are as follows.

1. **Interface Analysis.** Analysis of the interface between units and identification of pairs of data bits and their interface protection bits. Specifically, detection of data that is being communicated without interface protection bits.

2. **Logic Analysis.** Analysis of the logic handling the interface protection bits and relevant data in the receiving end and in the sender end. Specifically, detection of cases in which the sender or the receiver are not handling the interface protection bits properly.

3. **Error Injection.** Error injection into interface protection bits, to verify that the error is indeed being detected.

4. **Error Detection Analysis.** Comparison of the error detection driven by the error injection in item [3] to the analysis described in item [2], and detection of cases in which the results of the error injection is different than expected by the analysis.

Following is a detailed description of what is being done in each of the above sub-stages.
4.1 Interface Analysis

Since the interface protection bits are being communicated from one unit to another, and since each unit has its own designated design and verification teams, the interface protection bits and relevant protected data must be documented and follow some naming conventions - e.g. based on regular expressions, so that the receiving end can identify and handle it. Figure 8 shows an example of parity protection bits along with the relevant chunks of data they protect; the interface protection bit is the bit that ends with \( p \).

In this stage analysis is done to verify that the interface contains the required protection bits and that it is coherent. Namely:

1. Validation that all data bits in the interface have corresponding protection bits, and issue a warning in cases this fails to hold. This is done based on the naming convention used.

2. Validate that the interfaces in the sender and the receiver end are coherent; namely, verify that any protection bit being sent by the sender is indeed received by the receiver.

The above was implemented for IBM methodology which is based on a regular expression based naming convention, and can be easily extend to other conventions.

4.2 Logic Analysis

The communication of interface protection bits can be seen as a contract between the sender and the receiver; the sender has to properly generate the interface protection bits for the relevant data, and the receiver must check that the data and the
We verify that the sender indeed implemented the relevant parity calculation for feeding the interface protection bits. For the other end, we verify that the receiver indeed implemented the relevant check that is connected properly to an hardware checker. More specifically, it is being verified that each pair of interface data and relevant protection bit are part of a protected structure as described in D3.2 and D3.1. This is done with the algorithm described in Section 5 of Deliverable D3.1 for identifying structures of protected sequential elements, while treating each chunk of data and relevant interface protecting bits in the sender side and receiver side as connected. Note that the analysis in this stage is static - we verify that the relevant logic exists and is in place, but we do not verify that it actually works as intended in run time.

4.3 Error Injection

In this sub-stage error injection is performed into the interface protection bits and it is verified that the errors are detected. Specifically, the simulation team injects errors into the interface protection bits and monitors the behavior, while expecting an error checker to fire. Note that this is not trivial as one may assume since in many cases there are muxes as well as gating conditions on the way from relevant sequential elements and its protection bit to the error checker. These cases in which an error injection of an interface protection bit did not result in an error checker firing could be result of one of the following:

1. A bug in the protection mechanism that the design team should fix.
2. An indication that the testbench with which the error injection was performed failed to simulate an interesting scenario. This is since if no error checker fired (due to gating conditions) and the design is behaving properly then the data corresponding to the error injected protection bit is not actually being used in the scenario derived by the testbench. In this case the simulation team should repeat the error injection with another, more appropriate, testbench.

4.4 Error Detection Analysis

In this sub-stage the results of the previous sub-stage are compared with the analysis results of the first two sub-stages. That is, if an error is injected into an interface protection bits and take relevant actions in case this fails to hold. In this sub-stage we verify that both ends have implemented the relevant logic for keeping the contract.
IMMORTAL

System-level reliability analysis tools

To summarize, we address the reliability of data that is being passed between different components of the CPS by developing methodology for analyzing and verifying reliable communication between different units of a microprocessor. The methodology is composed of four sub-stages built on top of each other such that each sub-stage validates a deeper aspect of correct communication. While the first sub-stage analyzes syntactic aspect of the communication’s interface, the last sub-stage verifies that the behavior of the error detection logic is as expected. The methodology was explored and tested on the next generation of IBM’s mainframe and can be used, with some adjustments, for analysis and verification of communication between different components of the CPS.
System-level reliability analysis tools
5 Runtime Verification for LTL

In this section we consider the verification of LTL properties on finite simulation traces.

5.1 Motivation and Related Work

While LTL is defined on infinite paths we only observe a finite trace in practice. Thus, we somehow need to interpret this finite trace and evaluate whether it satisfies given LTL properties or not.

Assume the following unbounded response requirement is given:

$$\psi \equiv G(request \rightarrow F grant)$$

that requires for an implementation to eventually provide a grant whenever a request is set. We have two systems that implement this property and produce the traces presented in Table 2, where trace $\tau_1$ was produced by one system and $\tau_2$ by the other. In both traces we have an open request, i.e., to not violate the property the system has to to eventually provide a grant.

When manually investigating trace $\tau_2$, the tester might assume that a continuation of the trace would have provided in grant in two time steps, because in the past it always took for every request two time steps to be granted. In trace $\tau_1$, however, the grant for the previous request came within one time step and the last request did not get granted for the last three additional time steps. This looks at least suspicious and the respective system needs further analysis by the tester.

While manual investigation reveals a difference between the two given traces, existing literature on finite LTL interpretation would always evaluate both traces to the same outcome.

One approach in literature is to introduce weak and strong temporal operators [33] (or views [20]), where a weak operator (or view) only requires that the subformula
Another approach is a 3-valued finitary LTL interpretation of LTL [8], where (1) the verdict is true if the trace is a good prefix, i.e., for any continuation of the trace the property cannot be violated anymore, (2) the verdict is false if the trace is a bad prefix, i.e., for any continuation of the trace the property cannot be satisfied anymore, and (3) the verdict is inconclusive in all other case. For this approach, the inconclusive evaluation dominates as a globally property does not have a good prefix and an eventually property does not have a bad prefix. Thus, the this approach would evaluate both our traces to inconclusive. The authors did then come up with an updated version [9], where the inconclusive evaluation is refined by using weak and strong operators. Still, the updated approach cannot distinguish the two given traces.

The authors in [34] propose a finitary semantics for each of the LTL hierarchy classes (safety, liveness, persistence and recurrence) that asymptotically converges to the infinite traces semantics of the logic. However, the specification $\psi$ also evaluates to the same verdict for both the traces $\tau_1$ and $\tau_2$.

In our approach we take the (observed) behavior of the system into consideration when evaluating a finite trace. We evaluate a trace to presumably true if we expect a continuation of the trace to satisfy the property with respect to the previously observed behavior and we evaluate a trace to presumably false if we expect a continuation of the trace to violate the property with respect to previously observed behavior.

5.2 Counting Semantics

We extend the set of natural numbers (incl. 0) with the two special symbols $\infty$ (infinite) and $-$ (impossible) and refer to it as $\mathbb{N}_+ = \mathbb{N}_0 \cup \{\infty, -\}$. We define an order on it such that for all $n \in \mathbb{N}_0$, we have $n < \infty < -$. To add two elements $a, b \in \mathbb{N}_+$ we define the addition-operator $\oplus$ as follows:

$$a \oplus b = \begin{cases} a + b & \text{if } a, b \in \mathbb{N}_0 \\ \max\{a, b\} & \text{otherwise} \end{cases}$$

To express our counting finite semantics we use pairs $(s, f)$ with $s, f \in \mathbb{N}_+$ and define the following operations on the pairs:
Definition 2 (Operations $\sim$, $\oplus 1$, $\sqcup$, $\sqcap$). Given two pairs $(s, f) \in \mathbb{N}_+ \times \mathbb{N}_+$ and $(s', f') \in \mathbb{N}_+ \times \mathbb{N}_+$, we have:

$$\sim(s, f) = (f, s)$$

$$s, f \oplus 1 = (s \oplus 1, f \oplus 1)$$

$$(s, f) \sqcup (s', f') = (\min(s, s'), \max(f, f'))$$

$$(s, f) \sqcap (s', f') = (\max(s, s'), \min(f, f'))$$

In Equation 3 (operator $\sim$) we define the swap between the two values of a pair. The operator $\oplus 1$ in Equation 4 defines the increment of both values in the pair by the value 1. The binary operator $\sqcup$ (we refer to it also as minmax) in Equation 5, computes a new pair with the minimum of the first values as the new first value and the maximum of the second values as the new second value. The binary operator $\sqcap$ (we refer to it also as maxmin) in Equation 6 is symmetric to the minmax operator, i.e., it computes the maximum of the first values as the new first value and the minimum of the second values as the new second value. We now give some examples to illustrate some operations on pairs:

Given the pairs $(0, 0)$, $(\infty, 1)$ and $(7, -)$ we have:

$$\sim(0, 0) = (0, 0) \quad \sim(\infty, 1) = (1, \infty)$$

$$(0, 0) \oplus 1 = (1, 1) \quad (\infty, 1) \oplus 1 = (\infty, 2)$$

$$(0, 0) \sqcup (\infty, 1) = (0, 1) \quad (\infty, 1) \sqcup (7, -) = (7, -)$$

$$(0, 0) \sqcap (\infty, 1) = (\infty, 0) \quad (\infty, 1) \sqcap (7, -) = (\infty, 1)$$

Remark. Note that $\mathbb{N}_+ \times \mathbb{N}_+$ forms a lattice where $(s, f) \leq (s', f')$ when $s \geq s'$ and $f \leq f'$ with join $\sqcup$ and meet $\sqcap$. Intuitively, larger values are closer to true.

With operations on the pairs defined, we now introduce our counting semantics for LTL. For an arbitrary position $i$ of a given finite trace $\pi \in \Sigma^*$ and a given LTL formula $\phi$ we give a pair $(s, f) \in \mathbb{N}_+ \times \mathbb{N}_+$. We refer to $s$ as satisfaction witness count and to $f$ as violation witness count. Intuitively, the value $s$ denotes the number of additional steps needed to witness the satisfaction (violation) of the formula. The value $\infty$ is used to denote that the property is satisfied (violated) only with an infinite number of steps and $-$ denotes that the property cannot be satisfied (violated) by any continuation of the trace.

Definition 3 (Counting finitary semantics). Let $\pi \in \Sigma^*$ be a finite trace, $i \in \mathbb{N}_{>0}$ be a position in or outside the trace and $\phi \in \Phi$ be an LTL formula. We define the counting finitary semantics of LTL as the function $d_\pi : \Phi \times \Sigma^* \times \mathbb{N}_{>0} \rightarrow \mathcal{P}(\mathbb{N}_+ \times \mathbb{N}_+)$ such that:
\[ d_\pi(p, i) = \begin{cases} 
(0, -) & \text{if } i \leq |\pi| \land p \in \pi_i \\
(-, 0) & \text{if } i \leq |\pi| \land p \notin \pi_i \\
(0, 0) & \text{if } i > |\pi|, 
\end{cases} \]

\[ d_\pi(\neg \phi, i) = \sim d_\pi(\phi, i), \]

\[ d_\pi(\phi_1 \lor \phi_2, i) = d_\pi(\phi_1, i) \sqcup d_\pi(\phi_2, i), \]

\[ d_\pi(X\phi, i) = d_\pi(\phi, i+1) \oplus 1. \]

\[ d_\pi(\phi \lor \psi, i) = \begin{cases} 
\phi & \text{if } i \leq |\pi| \\
\phi \lor \psi & \text{if } i > |\pi|, 
\end{cases} \]

\[ d_\pi(F\phi, i) = \begin{cases} 
\phi & \text{if } i \leq |\pi| \\
\phi \lor \psi & \text{if } i > |\pi|, 
\end{cases} \]

**Proposition** The evaluation of a proposition for a position inside the trace is trivial, as the proposition either holds or not. If the proposition holds, then we do not need any additional steps to observe satisfaction, i.e., \( s \) becomes 0, and it is impossible to violate it, i.e., \( f \) becomes \(-\). In case the proposition does not hold, we have the symmetric witness counts. For the evaluation of the empty word, we take an optimistic view and assume that we can both satisfy and violate the proposition right away, i.e., with 0 additional steps.

**Negation** Negating a formula simply swaps the witness counts. If we witness the satisfaction of \( \phi \) in \( n \) steps, then we witness the violation of \( \neg \phi \) in \( n \) steps, and vice versa.

**Disjunction** For the disjunction we take the shorter satisfaction witness count, because the satisfaction of one subformula is enough to satisfy the property. And we take the longer violation witness count, because both subformulas need to be violated to violate the property.

**Next** The next operator naturally increases the witness counts by one step.

**Eventually** We use the rewriting rule \( F\phi \equiv \phi \lor XF\phi \) to define the semantics of the eventually operator. When evaluating the formula after the end of the trace, we replace the remaining obligation \( (XF\phi) \) by \((-\infty,)\). Thus, \( F\phi \) evaluated on the empty word is satisfied by a suffix that satisfies \( \phi \), and it is violated only by infinite suffixes.

**Until** We use the same principle for defining the until semantics that we used for the eventually operator. We use the rewriting rule \( \phi U \psi \equiv \psi \lor (\phi \land X(\phi U \psi)) \).

To illustrate the use of our counting semantics, we take the motivating example from Table 2 and evaluate the trace \( \pi_1 \) with respect to the specification \( \psi \). We present the outcome in Table 3.

The rules in Definition 3 restrict the resulting pairs to certain combinations of values in the pair.
Table 3: Request/Acknowledge motivating example with $π_1$.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>ΕΟΕ</th>
</tr>
</thead>
<tbody>
<tr>
<td>$π_0$</td>
<td>T</td>
<td>–</td>
<td>–</td>
<td>T</td>
<td>–</td>
<td>–</td>
<td>T</td>
<td>–</td>
</tr>
<tr>
<td>$π_1$</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$d_{π}(r)$</td>
<td>0, –</td>
<td>(–, 0)</td>
<td>(–, 0)</td>
<td>0, –</td>
<td>(–, 0)</td>
<td>(–, 0)</td>
<td>0, –</td>
<td>(–, 0)</td>
</tr>
<tr>
<td>$d_{π}(r)$</td>
<td>(–, 0)</td>
<td>0, 0</td>
<td>0, 0</td>
<td>–</td>
<td>(–, 0)</td>
<td>(–, 0)</td>
<td>0, –</td>
<td>(–, 0)</td>
</tr>
<tr>
<td>$d_{π}(r)$</td>
<td>(–, 0)</td>
<td>0, 0</td>
<td>0, 0</td>
<td>(–, 0)</td>
<td>0, –</td>
<td>(–, 0)</td>
<td>0, 0</td>
<td>(–, 0)</td>
</tr>
<tr>
<td>$d_{π}(s)$</td>
<td>(–, 0)</td>
<td>0, 0</td>
<td>0, 0</td>
<td>(–, 0)</td>
<td>0, –</td>
<td>(–, 0)</td>
<td>0, 0</td>
<td>(–, 0)</td>
</tr>
<tr>
<td>$d_{π}(r → Fg, s_ξ)$</td>
<td>(–, 0)</td>
<td>0, 0</td>
<td>0, 0</td>
<td>(–, 0)</td>
<td>0, –</td>
<td>(–, 0)</td>
<td>0, 0</td>
<td>(–, 0)</td>
</tr>
</tbody>
</table>

Lemma 1. Let $π \in \Sigma^*$ be a finite trace, $φ$ an LTL formula and $i \in \mathbb{N}_{>0}$ an index. We have that $d_{π}(φ, i)$ is of the following form:

<table>
<thead>
<tr>
<th>(s,f)</th>
<th>$a$</th>
<th>$b$</th>
<th>$\infty$</th>
<th>$-$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td></td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\infty$</td>
<td></td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$-$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where $a \leq |π| - i$ and $b > |π| - i$. 

Proof. The proof is obtained using structural induction on the LTL formula. □

Finally, we relate our counting semantics to the three valued semantics in Lemma 2.

Lemma 2. Given an LTL formula and a trace $π \in \Sigma^*$ where $i \in \mathbb{N}_{>0}$ is an index in or outside the trace and $φ$ is an LTL formula, we have that

\[
\mu(φ, i) = \top \iff d_{π}(φ, i) = (a, –), \\
\mu(φ, i) = \bot \iff d_{π}(φ, i) = (–, a), \\
\mu(φ, i) = ? \iff d_{π}(φ, i) = (b_1, b_2),
\]

where $a \leq |π| - i$ and $b_j$ is either $\infty$ or $b_j > |π| - i$ for $j \in \{1, 2\}$.

Lemma 2 holds because we only introduce the symbol “–” within the trace when a satisfaction (violation) is observed. And the values of a pair only propagate into the past, i.e., to positions smaller than or equal to the current position.

We now present our evaluation function that assigns a truth value to every pair.

We use a 5-valued set of truth values consisting of true ($\top$), presumably true ($\top_p$), inconclusive ($?$), presumably false ($\bot_p$) and false ($\bot$) verdicts. We define the following order over these five values:

\[
\bot < \bot_p < ? < \top_p < \top.
\]

We equip this 5-valued domain with the negation ($\neg$) and disjunction ($\lor$) operations, letting $\neg \top = \bot_p$, $\neg \bot_p = \bot_p$, $\neg ? = ?$, $\neg \bot = \top$, and $\phi_1 \lor \phi_2 = \max \{\phi_1, \phi_2\}$. We define other Boolean operators such as conjunction by the usual logical equivalences ($\phi_1 \land \phi_2 = \neg (\neg \phi_1 \lor \neg \phi_2)$, etc.).
We evaluate a property on a trace to $\top (\bot)$ when the satisfaction (violation) can be fully determined from the trace, following the definition of the three-valued semantics $\mu$. Intuitively, this takes care of the case in which the safety (co-safety) part of a formula has been violated (satisfied), at least for properties that are intentionally safe (intentionally co-safe, resp.) [31].

Whenever the truth value is not determined, we distinguish whether $d_\pi(\phi, i)$ indicates the possibility for a satisfaction, respective violation, in finite time or not. For possible satisfactions, respective violations, in finite time we make a prediction on whether past observations support the belief that the trace is going to satisfy or violate the property. If the predictions are not inconclusive and not contradicting, then we evaluate the trace to the (presumable) truth value $\top_\pi$ or $\bot_\pi$. If we cannot make a prediction to a truth value, we compute the truth value recursively based on the operator in the formula and the truth values of the subformulas (with temporal operators unrolled).

We use the predicate $\text{pred}_\pi$ to give the prediction based on the observed witnesses for satisfaction. The predicate $\text{pred}_\pi(\phi, i)$ becomes $?$ when no witness for satisfaction exists in the past. When there exists a witness that requires at least the same amount of additional steps as the trace under evaluation then the predicate evaluates to $\top$. If all the existing witnesses (and at least one exists) are shorter than the current trace, then the predicate evaluates to $\bot$. For a prediction on the violation we make a prediction on the satisfaction of $d_\pi(\neg \phi, i)$, i.e., we compute $\text{pred}_\pi(\neg \phi, i)$.

**Definition 4** (Prediction predicate). Let $s, f$ denote natural numbers and let $s_\pi(\phi, i), f_\pi(\phi, i) \in \mathbb{N}_+$ such that $d_\pi(\phi, i) = (s_\pi(\phi, i), f_\pi(\phi, i))$. We define the 3-valued predicate $\text{pred}_\pi$ as

$$\text{pred}_\pi(\phi, i) = \begin{cases} \top & \text{if } \exists j < i. d_\pi(\phi, j) = (s', -) \text{ and } s_\pi(\phi, i) \leq s', \\ ? & \text{if } \nexists j < i. d_\pi(\phi, j) = (s', -), \\ \bot & \text{if } \exists j < i. d_\pi(\phi, j) = (s', -) \text{ and } s_\pi(\phi, i) > \max_{0 \leq j < i} \{s' | d_\pi(\phi, j) = (s', -)\}. \end{cases}$$

For the evaluation we consider a case split among the possible combinations of values in the pairs as presented in Lemma 1.

**Definition 5** (Predictive evaluation). We define the predictive evaluation function $e_\pi(\phi, i)$, with $a \leq |\pi| - i$ and $b_j > |\pi| - i$ for $j \in \{1, 2\}$ and $a, b_j \in \mathbb{N}_0$, for the different cases of $d_\pi(\phi, i)$:
where \( r_\pi(\phi, i) \) is an auxiliary function defined inductively as follows:

\[
\begin{align*}
    r_\pi(p, i) &= \top \\
    r_\pi(\neg \phi, i) &= -e_\pi(\phi, i) \\
    r_\pi(\phi_1 \lor \phi_2, i) &= e_\pi(\phi_1, i) \lor e_\pi(\phi_2, i) \\
    r_\pi(X^n \phi, i) &= e_\pi(\phi, i + n) \\
    r_\pi(\exists \phi, i) &= \begin{cases} 
        e_\pi(\phi, i) \lor r_\pi(\exists \phi, i) & \text{if } i \leq |\pi| \\
        e_\pi(\phi, i) & \text{if } i > |\pi| 
    \end{cases} \\
    r_\pi(\phi_1 \lor \phi_2, i) &= \begin{cases} 
        e_\pi(\phi_2, i) \lor (e_\pi(\phi_2, i) \land e_\pi(\exists \phi_1 \lor \phi_2, i)) & \text{if } i \leq |\pi| \\
        e_\pi(\phi_2, i) & \text{if } i > |\pi| 
    \end{cases}
\end{align*}
\]

The predictive evaluation function is symmetric. Hence, \( e_\pi(\phi, i) = -e_\pi(\neg \phi, i) \) holds.

We refer again to our motivating example from Table 2 and evaluate the trace \( \pi_1 \) with respect to the specification \( \psi \). We present the outcome in Table 4. Subformula \( r \rightarrow F_g \) is predicted to be \( \top_p \) at \( i = 7 \) because there exists a longer witness for satisfaction in the past (e.g., at \( i = 1 \)). Thus, as we do not expected the globally property to be violated, the trace evaluates to \( \top_p \), as expected.

In Figure 9 we visualize the evaluation of a pair \( d_\pi(\phi, i) = (s, f) \) for a fixed \( \phi \) and a fixed position \( i \). On the x-axis is the witness count \( s \) for a satisfaction and on the y-axis is the witness count \( f \) for a violation. For a value \( s \), respectively \( f \), that is smaller than the length of the suffix starting at position \( i \) (with the other value of the pair always being \( - \)), the evaluation is either \( \top \) or \( \bot \). Otherwise the evaluation depends on the values \( s_{\text{max}} \) and \( f_{\text{max}} \). These two values represent the largest witness counts for a satisfaction and a violation in the past, i.e., for positions smaller than \( i \) in the trace. Based on the prediction function \( \text{pred}_\pi(\phi, i) \) the evaluation becomes \( \top_p \) or \( \bot_p \), where \( ? \) indicates that the auxiliary function \( r_\pi(\phi, i) \) has to be applied. Starting at an arbitrary point in the diagram and moving to the right increases the witness count for a satisfaction while the witness count for a violation remains constant. Thus, moving to the right makes the pair “more false”. The same holds
when keeping the witness count for a satisfaction constant and moving up in the diagram as this decrease the witness count for a violation. Analogously, moving down and/or left makes the pair “more true” as the witness count for a violation gets larger and/or the witness count for a satisfaction gets smaller.

Our 5-valued predictive evaluation refines the 3-valued LTL semantics.

**Theorem 3.** Let $\phi$ be an LTL formula, $\pi \in \Sigma^*$ and $i \in \mathbb{N}_{>0}$. We have

$$
\mu_\pi(\phi, i) = \top \Leftrightarrow e_\pi(\phi, i) = \top,
\mu_\pi(\phi, i) = \bot \Leftrightarrow e_\pi(\phi, i) = \bot,
\mu_\pi(\phi, i) = \top \wedge e_\pi(\phi, i) \in \{\top \wedge p, \bot \wedge p, ?\}.
$$

Theorem 3 holds, because the evaluation to $\top$ and $\bot$ is simply the mapping of a pair that contains the symbol “$-$”, which we have shown in Lemma 2.

Remember that $\mathbb{N}_+ \times \mathbb{N}_+$ is partially ordered by $\preceq$. We now show that having a

### Table 4: Request/Acknowledge motivating example with $\pi_1$.

<table>
<thead>
<tr>
<th>$r$</th>
<th>$g$</th>
<th>$d_r(x,r)$</th>
<th>$e_r(x,r)$</th>
<th>$d_g(y,r)$</th>
<th>$e_g(y,r)$</th>
<th>$d_g(x,F_x)$</th>
<th>$e_g(x,F_x)$</th>
<th>$d_g(y,F_y)$</th>
<th>$e_g(y,F_y)$</th>
<th>$d_g(F_r,y)$</th>
<th>$e_g(F_r,y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$-$</td>
<td>$(0,-)$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$(0,-)$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$(0,-)$</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$</td>
<td>$-$</td>
<td>$(0,-)$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$(0,-)$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$(0,-)$</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$</td>
<td>$-$</td>
<td>$(0,-)$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$(0,-)$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$(0,-)$</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$</td>
<td>$-$</td>
<td>$(0,-)$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$(0,-)$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$(0,-)$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

Figure 9: Lattice for $(s,f)$ with $\phi$ and $i < |\pi|$ fixed.
trace that is “more true” than another is correctly reflected in our finitary semantics. To define “more true”, we first need the polarity of a proposition in an LTL formula.

**Definition 6** (Polarity). Let \( \#\neg \) be the number of negation operators on a specific path in the parse tree of \( \phi \) starting at the root. We define the polarity as the function \( \text{pol}(p) \) with proposition \( p \) in an LTL formula \( \phi \) as follows:

\[
\text{pol}(p) = \begin{cases} 
  \text{pos}, & \text{if } \#\neg \text{ on all paths to a leaf with proposition } p \text{ is even}, \\
  \text{neg}, & \text{if } \#\neg \text{ on all paths to a leaf with proposition } p \text{ is odd}, \\
  \text{mixed}, & \text{otherwise}.
\end{cases}
\]

With the polarity defined, we now define the constraints for a trace to be “more true” with respect to an LTL formula \( \phi \).

**Definition 7** (\( \pi \sqsubseteq_\phi \pi' \)). Given two traces \( \pi \) and \( \pi' \) of equal length and an LTL formula \( \phi \) over proposition \( p \), we define that \( \pi \sqsubseteq_\phi \pi' \) iff

\[
\forall i \forall p. \quad \text{pol}(p) = \text{mixed} \Rightarrow p \in \pi_i \leftrightarrow p \in \pi'_i \quad \text{and} \\
\text{pol}(p) = \text{pos} \Rightarrow p \in \pi_i \rightarrow p \in \pi'_i \quad \text{and} \\
\text{pol}(p) = \text{neg} \Rightarrow p \in \pi_i \leftarrow p \in \pi'_i.
\]

Whenever one trace is “more true” than another, this is correctly reflected in our finitary semantics.

**Theorem 4.** For two traces \( \pi \) and \( \pi' \) of equal length and an LTL formula \( \phi \) over proposition \( p \), we have that

\[
\pi \sqsubseteq_\phi \pi' \Rightarrow d_{\pi'}(\phi, 1) \sqsubseteq d_{\pi}(\phi, 1).
\]

Therefore, we have for \( \pi \sqsubseteq_\phi \pi' \) that

\[
e_{\pi}(\phi, 1) = \top \Rightarrow e_{\pi'}(\phi, 1) = \top, \quad \text{and} \\
e_{\pi}(\phi, 1) = \bot \Leftarrow e_{\pi'}(\phi, 1) = \bot.
\]

Theorem 4 holds, because we have that replacing an arbitrary observed value in \( \pi \) by one with positive polarity in \( \pi' \) always results with \( d_{\pi'}(\phi, 1) = (s', f') \) and \( d_{\pi}(\phi, 1) = (s, f) \) in \( s' \leq s \) and \( f' \geq f \), as with \( \pi \sqsubseteq_\phi \pi' \) we have that \( \pi' \) witnesses a satisfaction of \( \phi \) not later than \( \pi \) and \( \pi' \) also witness a violation of \( \phi \) not earlier than \( \pi \).

In Table 5 we give examples to illustrate the transition of one evaluation to another one. Note that it is possible to change from \( \top \) to \( \bot \). However, this is only the predicated truth value that becomes “worse”, because we have strengthened the prefix on which the prediction is based on, the values of \( d_{\pi}(\phi, i) \) don’t change and remain the same is such a case.
Table 5: Making a system “more true”.

<table>
<thead>
<tr>
<th>φ</th>
<th>π</th>
<th>$d_{π}(φ, 1)$</th>
<th>$e_{π}(φ, 1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>−</td>
<td>(−, 0)</td>
<td>⊥</td>
</tr>
<tr>
<td>$p \land XFp$</td>
<td>− − − −</td>
<td>(−, 0)</td>
<td>⊥</td>
</tr>
<tr>
<td>$Gp$</td>
<td>− − − −</td>
<td>(3, ∞)</td>
<td>⊥</td>
</tr>
<tr>
<td>$Fp$</td>
<td>T − − −</td>
<td>(3, ∞)</td>
<td>⊥</td>
</tr>
<tr>
<td>$FGp$</td>
<td>T − T − −</td>
<td>(∞, ∞)</td>
<td>⊥</td>
</tr>
<tr>
<td>$p \lor XGp$</td>
<td>− − − T − T − −</td>
<td>(∞, 3)</td>
<td>⊤</td>
</tr>
</tbody>
</table>

34 • Runtime Verification for LTL
6 Hardware verification for security

One of the requirements for a reliable system is to provide security guarantee against external exception cases resulting from lacks in design, failures, or attacks. For a robust system, security have to be included during the design of the architecture. It becomes even more important when the same infrastructure is shared between multiple system parts in case of CPS.

Masking provides a high level of resistance against side-channel analysis. However, in practice there are many possible pitfalls when masking schemes are applied, and implementation flaws are easily overlooked. Over the recent years, the formal verification of masked software implementations has made substantial progress. In contrast to software implementations, hardware implementations are inherently susceptible to glitches. Therefore, the same methods tailored for software implementations are not readily applicable.

We designed a method to formally verify the security of masked hardware implementations that takes glitches into account. Our approach does not require any intermediate modeling steps of the targeted implementation. The verification is performed directly on the circuit’s netlist in the probing model with glitches and covers also higher-order flaws. For this purpose, a sound but conservative estimation of the Fourier coefficients of each gate in the netlist is calculated, which characterize statistical dependence of the gates on the inputs and thus allow to predict possible leakages. In contrast to existing practical evaluations, like t-tests, this formal verification approach makes security statements beyond specific measurement methods, the number of evaluated leakage traces, and the evaluated devices. Furthermore, flaws detected by the verifier are automatically localized.

6.1 Motivation

Security critical embedded systems rely on the protection of sensitive information against exposure. While the transmission of sensitive information over conventional communication channels can be protected by means of strong cryptography, the protection against unintentionally created side channels, like power consumption [30] or electromagnetic emanation [37], requires additional countermeasures. Since the risk of side-channel analysis (SCA) is inevitable in many applications, different countermeasures have been proposed in the past. One of the best researched and most effective countermeasures against SCA is masking. Many dif-
different masking schemes have been introduced [27, 28, 36, 39, 40] over the years. The history of masking, however, is also a history of failure and learning. Some of the first masking schemes were shown to be insecure in practical implementations because glitches in the combinatorial logic (temporary logic states caused by propagation time differences of the driving signals) were not considered in the formal models [28, 40]. The first provably secure masking scheme with inherent resistance to glitches was the threshold implementation (TI) scheme of Nikova et al. [36]. Over the last years the TI scheme has been extended further by Bilgin et al. [15], and other schemes have been proposed like the consolidated masking scheme (CMS) of Reparaz et al. [39], and the domain-oriented masking scheme (DOM and the low-randomness variant UMA) of Gross et al. [26, 27].

Even if the used masking scheme is secure, this is not automatically true for the implementations that rely on this scheme. One problem that thus still remains is the verification of masked implementations. There are basically two approaches that are used in practice to verify the resistance against SCA, namely formal verification and empirical leakage assessment. The predominant approach for the verification of masked hardware implementations is still the empirical leakage assessment in form of statistical significance tests [25] or by attacking the devices by using state-of-the-art side-channel analysis techniques. However, such practical evaluations are never complete, in a sense that if no flaw is found it remains uncertain whether or not the implementation could be broken with a better measurement setup or more leakage traces.

Recently there has been some substantial development towards the formal verification for masked software implementations [5, 11, 22]. However, these verification methods are tailored to software and do not take glitches into account. Therefore, they cannot readily be applied to hardware implementations. In terms of non-empirical verification of hardware implementations, there exist tools to test for leakage by either modeling of the circuit in software [38] or approaches that simulate possible leakages by assuming a specific power model [13]. To the best of our knowledge there exist no formal tools that take glitches into account and directly prove the security of masked hardware implementations on its netlist.

6.2 Related Work

Automated verification of masked implementations has been intensively researched over the last years and recently many works targeting this topic have been published [3, 10, 11, 13, 21, 22, 23, 35]. Most of the existing work, however, targets software based masking which does not include the effects of glitches.

Verification of masked software. One of the most groundbreaking works towards the efficient verification of masked software implementations is the work of Barthe et al. [5]. Instead of proving the security of a whole implementation at once, this work introduces the notion of strong non-interference (SNI). SNI is an
extension to the more general non-interference (NI) notion introduced in [4]. The SNI notion allows to prove the security of smaller code sequences (called gadgets) in terms of composability with other code parts. Gadgets fulfilling this SNI notion can be freely composed with other gadgets without interfering with their SCA resistance.

Verification of algorithms that fulfill this notion scale much better than other approaches but, on the other hand, not all masking algorithms that are secure are also directly composable. As a matter of fact the most efficient software masking algorithms in terms of randomness of Belaid et al. [11, 12], Barthe et al. [6], and Gross et al. [26], for example, do not achieve SNI directly.

In contrast to Barthe et al.’s work on SNI [5], our approach does not check for composability and is therefore less restrictive to the circuits and masking schemes that can be proven (similar to the NI approach of Barthe et al.’ [4]). Since Barthe et al.‘s work is designed to prove masked software implementations it does not cover glitches. In our work we introduce the necessary formal groundwork for the verification of masked circuits and in particular the propagation of glitches. Our approach is thereby not bound to our SAT realization but is also compatible with existing tools like easyCrypt which is developed by Barthe et al. [7].

Most recently another formal verification approach by Coron [19] was introduced that builds on the work of Barthe et al.. Essentially two approaches are discussed in this work. The first approach is basically the same as the approach in [4] but written in Common Lisp language. The second approach is quite different and works by using elementary transformations in order to make the targeted program verifiable using the NI and SNI properties. The work of Coron targets again only software based masking and does not take glitches into account.

Eldib et al. [22] present an approach to verify masked software implementations. Similar to our approach, the verification problem is encoded into SMT and verified by checking the constraints for individual nodes (operations) inside the program. This approach allows direct localization of the vulnerable code parts. However, their approach targets software and therefore does not cover glitches. It also produces SMT formulas that are exponential in the number of secret variables, whereas the formulas that are produced by our approach are only linear.

Bhasin et al. [14] also use Fourier transforms to estimate the side channel attack resistance of circuits. Their approach uses a SAT solver to construct low-weight functions of a certain resistance order. They have not used their approach to evaluate existing implementations of cryptographic functions, and they do not take glitching behavior into account.

**Verification of masked hardware.** Similar to our approach, Bertoni et al. [13] address verification of masked hardware implementations in the presence of glitches. In this work all possible transients at the input of a circuit are considered and all resulting glitches that could occur at the gates are modeled. However, this approach focuses on first-order masking of purely combinatorial logic and uses a rather simple power model to measure the impact (transitions from 0 to 1 result in the same power consumption as transitions from 1 to 0). We note that focusing on combinatorial logic only, leaves out most of the existing hardware-based masking.
schemes such as [26, 27, 36, 39]. Bertoni et al. demonstrated their approach on a masked implementation of Keccak based on a masking scheme that is known to be insecure in the presence of glitches.

In contrast to Bertoni et al.’s work, our approach considers combinatorial logic as well as sequential gates (registers), covers also higher-order leakages, and is not restricted to circuits with only one output bit.

In the work of Reparaz [38], a leakage assessment approach is introduced that works by simulating leakages of a targeted hardware implementation in software. At first, a high-level model of the hardware implementation is created, and the verification then works by simulating the model with different inputs and extracting leakage traces. The verification result is gathered by applying statistical significance tests (t-tests) to the simulated leakage traces. Compared to our approach, the leakage detection approach of Reparaz does not perform a formal verification but an empirical leakage assessment. Furthermore, the verification is not directly performed on the targeted hardware implementation but requires to model its (leakage) behavior in software.

6.3 Contribution

We introduced a method to formally prove the security of masked hardware implementations in the presence of glitches. In contrast to existing formal or non-empirical verification approaches for hardware designs, the introduced approach does not require any additional modeling of the circuit or the leakage source and proves the security of a circuit directly on its netlist. Compared to empirical verification methods based on the statistical analysis of leakage traces, our formal approach allows direct localization of the detected flaws, and gives conclusive security statements that are independent of device- or measurement-specific conditions, or the amount of gathered leakage information.

We base our approach on the probing model of Ishai et al. [28] and take the effects of glitches into account. We introduce a circuit verification method that performs a conservative estimate of the data that an attacker can learn by probing different gates and wires. The verification works directly on the gate-level representation of the circuit. It uses the Fourier expansion (or Walsh expansion) of the functions that are computed and uses the fact that a non-zero Fourier coefficient for a linear combination of variables indicates a correlation between the function and these variables (cf. [14]). A correlation with a linear combination of variables that contains secrets but no uniformly distributed masking variables corresponds to an information leak. By only keeping track of whether coefficients are zero or not, we circumvent the complexity of a full Fourier representation of all functions computed by all gates of the circuit, at the cost of a loss of precision that may lead to false alarms. The information of whether a coefficient is zero or not can be encoded as a propositional logic formula whose size is linear in the size of the circuit and vulnerability can be computed efficiently by a SAT solver.
To show the practicality of this approach, we check a variety of masked circuits that originate from different masking schemes. We focus on acyclic (feedback free) pipelined masked circuits, like the S-boxes of symmetric primitives which are usually the parts of a circuit that are the hardest to protect in practice and therefore most susceptible to flaws. The security of the linear circuits parts, on the other hand, can be established and verified more easily in practice, for instance by ensuring that only one masked value or mask of one variable is used inside each of the linear circuit parts [27]. For the same reason multiple cipher rounds and S-box lookups can be analyzed separately, as long as it is ensured that the masked outputs of the nonlinear parts are always independently and freshly masked (which is the case for most masking schemes).

We ran our tool on a set of example circuits including the S-boxes of Keccak, Fides and AES. Our verifier is efficient enough to formally prove the resistance of a full first-order masked AES S-box. Because of the circuit size of the AES S-box, which consumes about 40% of the entire AES area [27], the parallelized evaluation takes about 10 hours. We also prove a Keccak S-box up to order three, and the $GF(2)$ multipliers of DOM up to order four. Furthermore, we show that our approach correctly detects flaws in masked circuits that are known to be flawed in the presence of glitches e.g. [28, 40].

For more detailed description refer to [16].
System-level reliability analysis tools
7 Reliability Metrics of AMS Components in Integrated CPS

Nowadays, it is almost impossible to realize electronics solely with either analog or digital circuitry. It is usually the case that a mixture of analog and digital circuitry is used to realize electronic system; for which, we call them analog/mixed-signal (AMS) systems. Analog components like filters, equalizers, attenuators, and amplifiers are featuring continuously time varying signals. On the other hand, digital components like combinatorial and sequential logic and any circuit that is built using them feature discrete values (i.e., false, true).

For an AMS system to function correctly not only the building blocks of such system need to be bug-free, but also the overall design should also guarantee correct functionality and adequate performance. And since AMS systems are playing an ever-increasing role in modern chip design, it is very important that they do not become performance bottle-necks or the origin of design bugs and system failures. Unfortunately, formal verification and debugging of AMS systems could not keep up with the rapid growth in demand and complexity of these designs; thus, these components now account for large amount of design bugs in modern circuitry.

In this deliverable, the research on CPS reliability metrics is presented for integrated analogue/MS CMOS systems. Two approaches have been and are being investigated. The reliability-related physical phenomenon we use is aging, resulting from NBTI and HCI failure mechanisms. Reliability in our case is specified over time, were key performance parameters like gain or frequency of the analogue/MS parts will degrade over time due to aging. If one of the key parameters exceeds its limitation, the lifetime has been determined, and hence the reliability of the system can be calculated, e.g. in FIT.

One approach uses low-level simulations at transistor level of IPs/units, as well as the physical description of the time-dependent failure mechanisms. The key parameters are subsequently extracted from the simulations and a higher level model together with these parameters is developed in Mat lab. A complete front end has been simulated and provides clues about the behaviour with respect to reliability. The system also provides a C code of the complete behaviour, which has shown to be compiled in Microsoft Visual Studio. The developed system can be seen as another approach to pure digital reason engines, although less fast, but much more detailed and easy to modify at low levels.
7.1 Op-amp case study

A two-stage miller Op-amp operating in open-loop configuration is taken as a case-study. The Op-amp is designed in TSMC 40nm process. It has a PMOS input stage and is designed to be operated as a buffer for driving off-chip loads. Due to the low operating frequency of the analog sensors, low bandwidth is sufficient for this particular application. The Op-amp has a DC gain of 61dB and a 3-db bandwidth of 6KHz. The frequency response of the Op-amp which shows the gain and bandwidth is given in Fig. 10. The Op-amp is simulated with small-signal step input and the corresponding output response is observed. To enable the generation of the Boolean model, discretization of the input and output signals are required. The input and output signals are discretized using a 12-bit quantizer at the software level and the corresponding digital values are given as input to generate the Boolean model.

![Figure 10: Op-amp gain and bandwidth](image)

The particular case of step response results in two DC states to model the operation. DC1 state corresponds to the minimum value of the input signal and DC2 state corresponds to the maximum value of the input signal. Since the responses from DC1 to DC2 is not identical as DC2 to DC1, both transitions has to be simulated and modelled. Fig. 11 and Fig. 12 shows the DC1 to DC2 transition and DC2 to DC1 transition respectively with both the actual voltages and 12-bit discretized voltages superimposed. The sampling time is 1ns. To induce the effects of aging the design could be simulated with our aging models [41] and the same procedure as above could be repeated.
7.2 Verification approach

In the remaining part of the work, an alternative approach is being investigated which is potentially closer to pure digital reasoning engines [1]. Converting AMS Public Reliability Metrics of AMS Components in Integrated CPS...
systems to digitized form would enable us to use incredible verification techniques to formally prove a given system always fulfills specified properties.

Here is the draft of the method to verify AMS systems:

1. Analogue circuit is approximated by simulating transitions between DC states.
2. Given traces from the simulation we can construct Finite State Machine (e.g. Mealy machine) that captures behavior of an AMS system by utilizing method similar to [29].
3. FSM and the properties to verify are encoded into symbolic model (SMV) and a model checker (e.g. NuSMV) is harnessed to figure out if the system satisfies the properties.

**Implementation.** The described approach is implemented in Python3. First, the tool parses traces from Matlab simulation and constructs a Mealy machine that captures behavior of the corresponding analogous circuit. The Mealy machine as well as the property to check are encoded into SMV format to enable using NuSMV symbolic model checker which is employed to figure out if the property holds on the specified FSM.

As a running example let us consider simple amplifier with two DC stable states. From the matlab simulation one can get the following traces:


Each pair in the trace corresponds to input and output of a particular state. Having such traces as an input, the tool constructs the Mealy machine shown in Fig 13.

![Figure 13: Example of Mealy machine](image)

In the Fig. 13 green and white nodes match up with stable and transient states respectively. Input/output pairs are on the arcs, meaning that if a system is in some state and the input on arc is applied, then the system goes to the corresponding state and yields output stated on the arc. The FSM is then encoded into SMV format in listing 2. For this model we can specify properties to check, e.g. $F(gain = 522)$, where $F$ is the temporal operator meaning “eventually”. Although, for this particular property we can verify the system based only the traces, the general approach described here enables to check also more complicated properties, e.g. bandwidth of a channel.

Having traces from the current state of the system, and the aged system, we can model check both with the same property, and make conclusion on whether the specified property holds before and after aging.

**Listing 2: Example of SMV encoding**

```python
1 MODULE main
```
VAR
s : 0 .. 8;
gain : 0 .. 522;
i : 0 .. 10;
o : 0 .. 10;
ASSIGN
init(s) := 0;
gain := 522;
next(s) :=
case
(s = 0) & (i = 501) : 1;
(s = 1) & (i = 501) : 2;
(s = 5) & (i = 500) : 6;
(s = 2) & (i = 500) : 3;
(s = 4) & (i = 500) : 5;
(s = 6) & (i = 500) : 7;
(s = 7) & (i = 500) : 0;
(s = 4) & (i = 501) : 4;
(s = 3) & (i = 501) : 4;
TRUE : s;
esac;
next(o) :=
case
(s = 0) & (i = 500) : 499;
(s = 2) & (i = 501) : 1020;
(s = 6) & (i = 500) : 500;
(s = 7) & (i = 500) : 499;
(s = 5) & (i = 500) : 517;
(s = 1) & (i = 501) : 1019;
(s = 4) & (i = 501) : 1021;
(s = 4) & (i = 500) : 618;
(s = 3) & (i = 501) : 1021;
(s = 0) & (i = 501) : 903;
TRUE : o;
esac;
8 Conclusion

This deliverable is the final deliverable for work package 3. It presents the tools developed to improve and analyze the reliability of CPSs. The presented tools cover a wide variety of methodologies and targets for the reliability analysis. First, a simulation environment is introduced that allows to verify functionality of the system already in early development stages. Next, in section 3 we introduce a new approach to ensure that a design fulfills its latency requirement. Then, we described a tool that verifies that cross unit communication is correctly protected against soft-errors and we presented an approach to decide the validity of of LTL-formulas (from specification) on finite execution traces of the design. Further, we technique is presented to formally prove the protection of a design against side-channel-attacks, and in section 7 we introduce techniques that allows to formally verify AMS components.
Conclusion Public
9 References


