DIPLOMA THESIS

Automated verification of a System-on-Chip for radiation protection fulfilling Safety Integrity Level 2

Submitted at the Faculty of Electrical Engineering and Information Technology,
Vienna University of Technology
in partial fulfillment of the requirements for the degree of
Diplom-Ingenieur (equals Master of Science)

under supervision of

Univ.Prov. Dipl.-Ing. Dr.techn. Axel Jantsch
Univ.Ass. Dipl.-Ing. Dr.techn. Michael Rathmair

Institute of Computer Technology (E384)
TU Wien

and

Dr. Hamza Boukabache

CERN
European Organization for Nuclear Research

by

Katharina Ceesay-Seitz, BSc
Matr.Nr. 0925147
[redacted]

Vienna, 15.02.2019

Abstract

The new CERN Radiation MOnitoring Electronics (CROME) system is currently being developed at CERN. It consists of hundreds of units, which measure ionizing radiation produced by CERN's particle accelerators. They autonomously interlock machines if dangerous conditions are detected, for example if defined radiation limits are exceeded. The topic of this thesis was the verification of the safety-critical System-on-Chip (SoC) at the heart of these units. The system has been allocated the Safety Integrity Level 2 (SIL 2) of the IEC 61508 standard for functional verification.
The SoC has several characteristics that are challenging for its verification. It is highly configurable with parameters of wide ranges. It will operate continuously for up to 10 years. Measurement outputs are dependent on previous measurements over the complete operating time.
The goal of this thesis was the definition and demonstration of a SIL 2 compliant functional verification methodology for the mentioned SoC. An automated verification software framework should be developed that is reusable on system-level and allows the reverification of future versions of the system.
A methodology for independent functional black-box verification has been defined. Its workflow starts with the specification of semi-formal verification requirements. Natural language properties were introduced and their translation into SystemVerilog assertions was defined. Formal Property Verification was combined with constrained-random simulation. For the latter the Universal Verification Methodology (UVM) was used. A software framework has been developed that automatically creates reproducible results, which are backward-traceable to the functional and safety requirements. Verification completeness was measured with functional, structural and formal coverage metrics. SystemVerilog covergroups have been used to measure the coverage of values spread over thousands of clock cycles, supported by assertions. Regression coverage has been added to the workflow for discovered implementation faults. Through applying the methodology, several faults in the implementation were found and several properties could be formally proven.

Kurzfassung

Ein neues System zur Überwachung von Radioaktivität, genannt CERN Radiation MOnitoring Electronics (CROME), wird derzeit am CERN entwickelt. Es besteht aus hunderten von autonomen Einheiten, welche die radioaktive Strahlung messen, die durch die Teilchenbeschleuniger am CERN produziert wird. Diese Einheiten erkennen gefährliche Bedingungen, wie zum Beispiel Strahlung, die definierte Grenzwerte überschreitet, und unterbrechen den Betrieb der verantwortlichen Maschinen automatisch. Thema dieser Diplomarbeit war die Verifikation eines sicherheitskritischen System-on-Chips (SoC) im Herzen dieser Einheiten. Dem System wurde der Safety Integrity Level 2 (SIL 2) des IEC 61508 Standards für funktionale Sicherheit zugewiesen.
Das SoC besitzt einige Eigentschaften, die eine Herausforderung für seine Verifikation darstellen. Es ist hoch konfigurierbar, mit Parametern, die sehr große Wertebereiche haben können. Es wird bis zu 10 Jahre ununterbrochen in Betrieb sein. Gelieferte Messwerte sind abhängig von vorhergehenden Messwerten über die gesamte Betriebsdauer.
Das Ziel dieser Diplomarbeit war die Definition und Demonstration einer SIL 2 konformen Verifikationsmethodik für das genannte SoC. Ein automatisiertes Softwareframework zur Verifikation sollte entwickelt werden. Dieses sollte auf Systemebene und für die Reverifikation von zukünftigen Versionen des Systems wiederverwendbar sein.
Eine Methodik für unabhängige funktionale black-box Verifikation wurde definiert. Ihr Ablauf beginnt mit der Spezifikation von semi-formalen Verifikationsanforderungen. Natürlichsprachliche Eigenschaften (Properties) wurden vorgestellt und ihre Übersetzung in SystemVerilog Assertions wurde definiert. Formal Property Verification wurde mit constrained-random Simulation kombiniert. Für letztere wurde die Universal Verification Methodology (UVM) angewendet. Ein Software Framework wurde entwickelt, das automatisch Verifikationsergebnisse generiert, die zurückführbar auf die zugrundeliegenden Funktions- und Sicherheitsanforderungen sind. Vollständigkeit der Verifikation wurde Anhand von funktionalen, strukturellen und formarmalen Testabdeckungsmetriken (coverage metrics) gemessen. SystemVerilog wurde verwendet um die Testabdeckung von über viele tausende von Clock-Zyklen verstreuten Werten zu messen. Dazu wurden Covergroups mit Assertions kombiniert. Für gefundene Fehler in der Implementierung wurden spezifische Testabdeckungsmetriken für Regressionstests zu den bereits vorhandenen hinzugefügt. Durch die Anwendung der definierten Methodik konnten einige Fehler in der Implementierung gefunden werden, sowie einige Properties formal bewiesen werden.

Acknowledgements (Danksagung)

I want to thank Univ.Prov. Dipl.-Ing. Dr.techn. Axel Jantsch and Univ.Ass. Dipl.-Ing. Dr.techn. Michael Rathmair at TU Wien for supervising my diploma thesis and for their valuable feedback throughout the project.

I want to thank my supervisor Dr. Hamza Boukabache at CERN for giving me all the freedom and support I needed for developing my thesis project, as well for his valuable feedback. I want to thank Daniel Perrin for welcoming me in his section. I further want to thank the whole CROME team for all advice and discussions. Especially, I want to thank Ciaran Toner, the design engineer, for quickly analysing all verification results and providing design updates when needed, which allowed me to keep on progressing with my work. Furthermore, for reviewing my thesis. I also want to thank Markus Widorski, the requirements engineer, for providing all needed information and reviewing my natural language property specification. I want to thank Sarath Kundumattathil Mohanan for his advice for writing my thesis.

Ich möchte mich bei meinen Eltern für ihre finanzielle und persönliche Unterstützung während meines gesamten Studiums bedanken. Ebenso bei meinen Brüdern, die mich immer besuchen, egal in welchem Land ich mich gerade befinde.

I want to thank my husband Yaya for all his patience and advice in stressful times, for cooking delicious food for me throughout my whole Master's studies and especially for encouraging me to fulfil my dream of working at CERN.

Furthermore I want to thank all other family members and friends who supported me and joined me throughout the course of my studies.

Table of Contents

Figures and Listings

Tables

Abbreviations

ARCON	ARea CONtroller
BRAM	Block Random Access Memory
CAU	CROME Alarm Unit
CERN	European Organization for Nuclear Research
CCC	CERN Control Centre
CJB	CROME Junction Box
CMPU	CROME Measuring and Processing Unit
COI	Cone Of Influence
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CROME	CERN Radiation MOnitoring Electronics
CUPS	CROME Uninterruptible Power Supply
DPI	Direct Programming Interface
DUV	Design Under Verification
FEC	Formal Equivalence Checking
FEC	Focused Expression Coverage
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPV	Formal Property Verification
GB	Gigabyte
GHz	Gigahertz
GUI	Graphical User Interface
HSE	Occupational Health & Safety and Environmental Protection Unit
ID	Identification
IEC	International Electrotechnical Commission
IL	Instrumentation & Logistics
LHC	Large Hadron Collider
LS2	Long Shutdown 2
MC/DC	Modified Condition/Decision Coverage
Nr.	Number
PLC	Programmable Logic Controller
RAMSES	RAdiation Monitoring System for the Environment and Safety
ROMULUS	Read-Out Modular electronic for Ultra Low cUrrent measurementS
RP	Radiation Protection
SV	SystemVerilog
SVA	SystemVerilog Assertions
SW	Software
TeV	Teraelectronvolt
TLM	Transaction Level Modelling
TLM1	Transaction Level Modelling 1
UCIS	Unified Coverage Interoperability Standard
UVM	Universal Verification Methodology
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Chapter 1

Introduction and Motivation

At CERN, the European Organization for Nuclear Research, the fundamental laws of the universe are studied. For this purpose the worlds largest and most powerful particle accelerator, the Large Hadron Collider (LHC), has been built. It has a circumference of 27km and is installed underneath the surface of France and Switzerland. The LHC accelerates particles up to 99.9999% of the speed of light where they reach energies of 7 TeV (teraelectronvolt). It collides the two particle beams sent in opposite directions in order to recreate the conditions at the big bang. Under these conditions the interaction of particles and the formation of new particles are studied. These experiments produce ionizing radiation. Being exposed to ionizing radiation can be life threatening if the dose received by a human body is above certain limits. The LHC has been placed 100m underground. The soil above it shields the public from ionizing radiation that is generated during operation of the accelerator [CEG17]. Nevertheless, CERN continuously monitors and controls the amount of ionizing radiation present on the surface above the experiments, on the CERN site where the equipment for the experiments is built as well as directly at the experiments underground. Nevertheless CERN continuously monitors and controls the amount of ionizing radiation present directly at the experiments underground, as well as on the surface above the experiments. It also controls the radiation present on the CERN site, where the equipment for the experiments is built and further experiments of lower radiation are conducted.

Radiation protection is the responsibility of the Radiation Protection (RP) Group within the Occupational Health & Safety and Environmental Protection (HSE) unit of CERN. The Instrumentation & Logistics (IL) section within the RP group is responsible for the design, development, installation and maintenance of specialised radiation monitoring systems. Currently there are three monitoring systems in operation. Figure 1.1[1] provides an overview of their locations [Per17]. The Area Controller (ARCON) monitoring system is already over 30 years old. Because it is difficult to maintain a system of such old technology and also because technology and the state-of-the-art in radiation monitoring have evolved, the ARCON system has reached the end of its lifetime. Starting with the next long shutdown of the LHC in 2019 it will be replaced by the new CERN Radiation MOonitoring Electronics (CROME). Further in the future the currently installed RAdiation Monitoring System for the Environment and Safety (RAMSES) devices will also be replaced by CROME [BPD+16].

[1] With kind permission of the author.

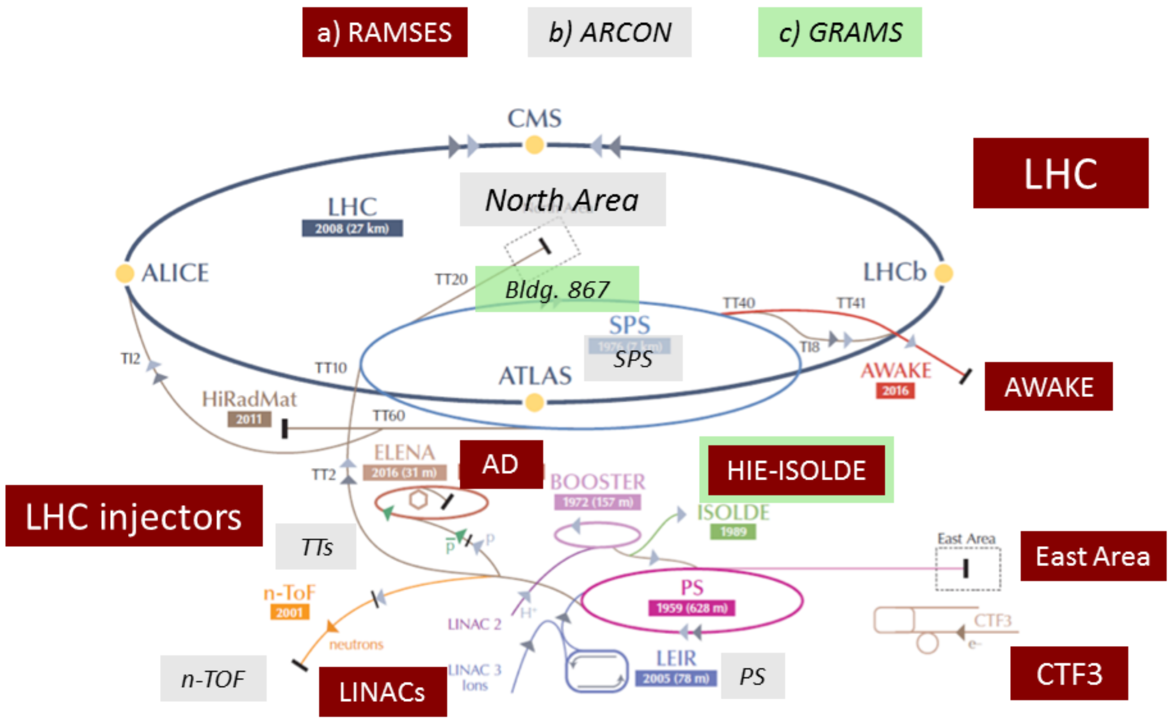

Figure 1.1: CERN Radiation and Environmental Monitoring System [Per17].

The CROME system is currently being developed by the Instrumentation & Logistics (IL) section within the Radiation Protection (RP) group. It consists of autonomous monitoring stations, alarm units, a junction box and an uninterruptible power supply. The autonomous monitoring units, called CROME Measuring and Processing Units (CMPUs), consist of an ionization chamber and an electronic readout system. The readout system can be either directly attached to the chamber or connected with a specialized cable. The cabled version is used for monitoring areas of high radiation. The ionization chamber is placed directly into that area of high radiation, whereas the readout electronics are placed in a area of lower radiation for their protection. The ionization chamber detects ionizing radiation and converts it to current. A read-out chip measures this current, which can be within 2fA to 250nA. Because these currents can be so low, a specialized cable had to be developed for the cabled version. The read-out chip passes the current to a System-on-Chip (SoC) that uses it to calculate the total radiation dose received by an area, as well as the radiation dose rate at the time of measurement. The SoC also monitors the state of the whole CMPU device. At the detection of dangerous conditions, eg. if the radiation dose or dose rate exceeds a defined limit, the CMPU autonomously generates alarms and machine interlocks that stop the LHC particle beams. Parameters like dose and dose rate limits, conversion factors from current to radiation, and many more can be configured remotely by operators in the CERN Control Center (CCC) [BPD$^+$16].

According to the standard IEC 60532 for radiation protection instrumentation, the radiation monitoring instruments at CERN need to fulfil at least Safety Integrity Level (SIL) 2. This is because they are responsible for triggering interlocks of machines and access doors [Hur16]. Therefore CROME is being developed to fulfil the Safety Integrity Level 2 (SIL 2), which is defined in the IEC 61508 standard for functional safety [ESC10]. This means its components have to implement measures to reduce the risk of dangerous failures during operation and its development process has to follow certain rules for avoiding the introduction of systematic faults. Systematic faults are those introduced during the design and development of an electronic system, when an implementation does not comply with its specification. The goal of functional verification is to detect such faults before the system is put into operation. Functional verification of an HDL or software implementation for a safety-critical electronic device shows whether the HDL or software implementation satisfies its functional and safety requirements specification and whether the implementation contains any unspecified functionalities. It therefore minimises the risk of undetected systematic faults [ESC10]. The goal of this thesis is to define an applicable functional verification methodology that fulfils the requirements for SIL 2 for verifying the SoC inside the CMPU. The applicability and effectiveness of the methodology is demonstrated on a subset of the CMPU's design.

1.1 Problem Description and Goals

Problem Description

All safety-critical decisions and actions performed by the CROME Measuring and Processing Unit (CMPU) have been implemented in an Hardware Description Language (HDL) code for a Field-Programmable Gate-Array (FPGA). In order to verify that the CMPU is able to execute its safety-critical functions as specified and does not perform any unspecified functions, it has to be demonstrated that its implementation satisfies its functional and safety requirements specification. To conform to the IEC 61508 standard for functional safety of electrical/electronic/programmable electronic safety-related systems the verification process itself needs to fulfil certain requirements. The verification activities need to be conducted in a systematic and reproducible way. Verification documentation in the form of a verification plan and corresponding results have to be created [ESC10].

The safety-critical decisions taken by the CMPU depend on mathematical functions of sensor measurements and input parameters. The input parameters are sent via the network from a supervision software in the CERN Control Center (CCC) to an embedded software on the CMPU. Based on these input parameters the embedded software calculates the parameters that it sends to the FPGA. Because these parameters influence the safety-critical decisions, the calculations performed in the embedded software have to be considered as safety-critical as well. Therefore the safety-related functional verification of the CMPU has to consist of the verification of the complete HDL code as well as a part of the embedded software.

Different verification methodologies for HDL designs exist, that have different applicability, depending on the characteristics and complexity of a design. The CMPU receives more than 100 input parameters and sensor measurement values. Most of them are 64 bit wide. This creates an input space of around $2^{64^{100}}$ possible combinations of input values. Optimistically assuming it would be possible to apply 1 combination of input values per clock cycle of a 2.5GHz desktop computer which runs the simulation, exhaustive simulation-based verification would take $4.96*10^{1909}$ years. Clearly this is too long to do it. There are little possibilities for parallelising the simulation of the CMPU because the design contains a chain of calculations where the outputs of one block are the inputs to the next one and the design is not fully pipelined. Furthermore the calculations performed by the FPGA depend on the previous internal states over a long period. CMPU devices that monitor environmental background radiation will be operating for over 10 years without ever being reset. Neither in simulation nor in emulation it is realistic that a whole period of 10 years of operation could be ever tested within reasonable time. For this reasons it will only be possible to simulate a subset of the input space and use cases of the device. It has to be possible to conclude from the results of verifying this subset that the device will function correctly for all possible input stimuli over all realistic device operating times. For this thesis only the functional verification of the designs behaviour will be considered. Verification of physical components is out of scope.

It would be desirable to formally proof the correspondence of the implementation with its specification, because this would guarantee that the correspondence has been shown for all possible scenarios. Because the safety-critical decisions each depend on multiple parameters and multiple mathematical calculations that depend on the previous internal states, the state-space for formal proofs becomes very large. It could easily exceed the capabilities of existing formal verification tools. Tools that are available today are usually better at handling control logic and simple arithmetical operations like additions of variables of short bit widths. Proofs that span several clock cycles or include 64 bit wide variables are difficult if not impossible to achieve [SSK15]. The possibilities for the given design have to be studied.

Automated verification of FPGAs often takes as long as the HDL design itself [Fos17]. Clearly manual verification would take even longer, especially when it comes to re-verification. When a design had been manually verified, the whole effort has to be repeated in case the design was modified. When a software framework for automated verification is available, the effort of re-verification can be reduced to starting some scripts. When the software produces meaningful verification results also the effort needed for analysing negative verification results can be reduced. Modifications of the Design Under Verification (DUV) might be necessary in case mismatches between the implementation and its requirements specifications have been found. Modifications might also be necessary due to an update of other components of the system that interact with the FPGA and embedded software. Furthermore additions might be made in future versions of the CROME system that allow to perform new measurements and calculations. In order to keep the benefits of automation, a verification software framework needs to be flexible enough to account for such modifications.

Goals

The first goal of this thesis is to define a verification methodology that is applicable to the CROME Measuring and Processing Unit (CMPU) and similar safety-critical designs. Verification techniques have to be selected that are able to demonstrate the correspondence of the implementation with its specification despite the problems described above. Different techniques may be defined for the different parts and abstraction levels of the design. The verification methodology needs to conform to IEC 61508 for achieving the Safety Integrity Level 2 (SIL 2) or higher.

Typically coverage metrics are used to express the progress and completeness of verification [KPSS14]. Based on the functional and safety requirements specifications, and on the design specifications of the FPGA and embedded software code, a meaningful coverage model has to be designed. The coverage model shall be used to measure the contribution of each applied verification technique to the complete coverage of the verification requirements. By demonstrating the verification methodology on a subset of the CMPU's HDL code it shall be evaluated whether

100% functional coverage can be achieved by the chosen techniques.

The second goal of this thesis is to develop a software framework that automates the verification activities that have been defined. The framework shall allow block-level and system-level verification of the FPGA as well as software system-level verification of the FPGA together with the safety-critical functions of the embedded software. The framework shall be developed in a flexible and reusable way. This will ease modifications of the framework, in case the design is modified or new functionalities are added to the design. It also helps to speed up the development time of the framework itself when a verification environment for one block can be partly reused for another block or on system-level.

To conform to IEC 61508, all verification activities and results need to be documented and they have to be reproducible [ESC10]. This means it has to be possible to recreate verification result documentation by using the same version of the design and verification framework as in a previous run. The developed framework shall be able to automatically create this documentation.

1.2 Structure of the Thesis

This thesis is structured as follows. Chapter 2 describes verification and validation within the safety-life cycle of digital electronic systems. It then provides background information on functional verification, mainly focusing on the verification of HDL designs. It is structured into simulation-based verification and formal verification. For both types it provides theoretical background and details about available methods and tools. It also discusses the state-of-the-art.
Chapter 3 details the verification methodology that has been defined in this thesis for the CROME Measuring and Processing Unit (CMPU) and similar safety-critical electronic devices. It starts with the description of verification planning, suggests a workflow and verification technologies and argues why these have been chosen over others. It ends with the description of necessary verification documentation for complying to IEC 61508.
Chapter 4 provides details about the CROME project and the Design Under Verification (DUV), the CMPU.
In Chapter 5 the defined methodology is demonstrated on a subset of the CMPU, the Integration block. This block is responsible for radiation dose calculation and alarm and machine interlock generation. The chapter explains the steps that have been taken and difficulties that have been encountered during the verification of the CMPU, mainly focusing on the Integration block. It describes alternative paths that have been considered and the solutions that have been implemented. The verification planning is described in great detail, as these activities already lead to updates of the DUV. A translation of natural language to SystemVerilog assertions is suggested, which facilitates the communication between different stakeholders. Then the coverage model for simulation-based verification is presented. The developed simulation-based software framework is described in detail. Then the results obtained through this framework are discussed. Then the formal property verification modules that have been implemented, are presented and the results that could be obtained with them are discussed. At the end a summary of the verification results and a discussion of the applied techniques is provided.
Chapter 6 concludes the thesis and provides an outlook for the verification of the whole CMPU.

Chapter 2

Background and State-of-the-Art

In software engineering a commonly used definition for verification is that it answers the questions "Are we building the system right?". This means verification shows whether a system was developed according to its design specification. The design specification captures the designers implementation decisions. Validation is then defined as "Are we building the right system?". This means, validation shows whether a system was developed according to its requirements specification [Boe84]. The requirements capture the functionalities that a user is expecting, including safety requirements [ESC10]. This chapter first describes verification and validation in the safety life cycle of electronic systems. It then focuses on different methodologies and techniques that could potentially be used to verify the CROME Measuring and Processing Unit (CMPU).

2.1 Verification in the life cycle of Safety-Critical Electronic Systems

This subsection will provide an overview of the safety life cycle of safety-critical electronic devices and what role verification and validation take in it.

The generic standard for functional safety of electrical/electronic/programmable electronic safety-related systems, IEC 61508, specifies target failure measures for four Safety Integrity Levels (SILs). For systems operating continuously or on high demand the target failure measures are expressed as the average frequency of dangerous failures per hour. These are systems that perform their safety functions either more than once per year or continuously. For systems operating on low demand the target failure measures are expressed as the average probability of dangerous failures per demand. These are systems that contain safety functions that are requested once per year or less frequent. For higher SILs the frequency/probability of failures has to be lower [ESC10]. The CMPU is a continuously operating device with some functions that are provided on demand. Even though these functions might not be used more often than once per year, they have to be classified as "high demand" functions, because it is possible to use them more frequent. The CMPU needs to comply with SIL 2 [Per17]. Therefore the following paragraphs focus on the requirements for verification when fulfilling SIL 2.

The functional safety standard specifies requirements per SIL for each phase of a safety-related system's life cycle, from the concept, its realisation, its operation up to its disposal. A system that fulfils these requirements conforms to the standard and the selected SIL. A system that shall conform to a higher SIL also has to fulfil the requirements for the lower SILs. Legal obligations could require that a system is developed according to the standard. It might as well be done as a method of quality assurance.

Verification has to be done for each life cycle phase of the overall safety-related system. The standard defines verification as the demonstration that the output products of each phase fulfil the objectives and requirements specified for that phase. The outputs could be documents, in which case a verification activity might be a review. It could be a software, in which case verification might consist of a walk-trough through safety-critical functions, together with a person different from the developer. Several verification techniques might be combined in each phase. A software walk-through might be complemented by static code analysis techniques and automated testing. The combination of used techniques has to be chosen according to the requirements for the SIL assigned to a system.

The main focus of the standard lies on the system's realisation phase. The base for the whole realisation phase is the system safety requirements specification. The realisation phase is split into several subphases, including the development phase. The development phase of Application-Specific Integrated Circuit (ASICs) and other integrated circuits like FPGAs and of software can be further divided into subphases following the V-model approach [ESC10]. Depending on the complexity of the system the V-model can be tailored, phases could be combined or additional iterations could be necessary. The V-model phases are traversed for each component of the system. A component could be an individual device of the overall system or an FPGA or the software of one device. Figure 2.1 shows the V-model for the CMPU, combined for the FPGA and embedded software. The scope of this thesis is depicted in green. Module/Unit testing is drawn half in blue, half in green to indicate that this activity is shared by the design and verification engineers. Usually a design engineer writes some testbenches to make sure the design works as intended. For complex FPGA blocks it can be beneficial to add independent verification on block-level. For other blocks it might be sufficient to verify them together with higher level blocks that instantiate them.

Before entering the development phase of a component, its safety requirements specification has to be derived from the system safety requirements specification. This specification will then be a guideline for the following subphases. A separate functional requirements specification that contains non-safety-related functions of the system might also exist. In this case it has to be shown that the non-safety-related functions have no influence at the safety-related functions. A clear boundary has to be established. If this is not possible, all functions have to be treated as safety-related.
Based on these requirements specifications, the architecture is designed and documented in the architecture design specification. The architecture design specification could be combined for FPGAs and software. During this phase also the integration test specification is written. The design decisions taken before implementing a hardware description language (HDL) code or software is then written down in a software system design specification. Concurrently the software system integration test specification is written. Large designs are usually divided into modules and sub-modules. For each of them a module design specification and a module test specification is written. Once these documents are ready, the implementation can start. At the same time verification activities can be prepared, for example by developing a verification software framework.

The implementation of each sub-module should be verified on its own against the module's design

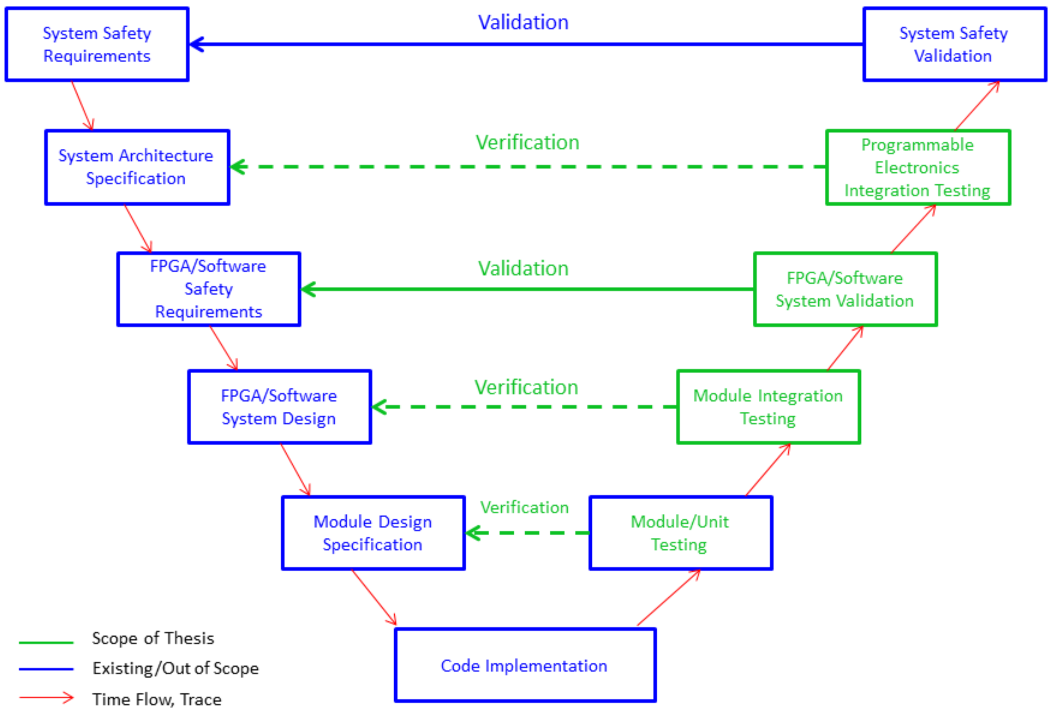
Validation
System Safety Requirements
System Safety Validation
Verification
System Architecture Specification
Programmable Electronics Integration Testing
Validation
FPGA/Software Safety Requirements
FPGA/Software System Validation
Verification
FPGA/Software System Design
Module Integration Testing
Verification
Module Design Specification
Module/Unit Testing
Code Implementation
Scope of Thesis
Existing/Out of Scope
Time Flow, Trace

Figure 2.1: V-model for the CROME Measuring and Processing Unit, based on [ESC10].

specification. Step by step the sub-modules should be plugged together to form the larger modules, which should be verified as well. This is called module testing. The verification of the whole design where all modules are connected together is called software system integration testing. In this activity the implementation is verified against the software design specification. For electronic systems that contain FPGAs the standard adds an additional life cycle phase that contains the integration of the FPGAs and software, called programmable electronic integration [ESC10]. When the full HDL and software codes have been verified on their own, their interaction can be verified by integration testing based on the architecture design. At first the sensors and actuators connected to the electronic device might still be simulated or modelled by verification software. At the final stage the HDL and software code should be integrated into the final hardware system and verified together with the final components like sensors, actuators and communication connections first on device level and later on complete system level. This is called system integration testing. At the final stage of integration testing a device or system of devices might be tested with all its software and hardware components in the real physical environment in which it is going to operate [ESC10, Grü17].

The IEC 61508 standard defines validation separately from verification as an activity that demonstrates that a completely developed component fulfils the system safety requirements specification for this component. The requirements for system safety validation of FPGAs contain several hardware measures like environmental tests or surge immunity tests. Such measures are out of scope of this thesis. This thesis only covers functional and black-box verification, which targets the behavioural aspects of programmable electronic designs. This means it covers the verification of HDL code and embedded software integration.

To conform to IEC 61508 the validation and verification activities have to be planned with a validation and verification plan. It describes the validation and verification strategy, including techniques and tools to be used. It also lists steps that have to be performed for obtaining validation and verification results for each subcomponent. For validation, the plan shows how each statement, both in the system safety requirements specifications and the system design specification, is validated. For verification it shows how it will be ensured that the verification activities for subcomponents have actually been performed. It also describes how to ensure that all created plans and specifications are not contradicting each other. Typically reviews and checklists are used for that.

The results of the validation and verification activities have to be documented together with a procedure to reproduce them. In case the verification and validation activities discovered mismatches between the developed system and the safety requirements specification, the corrective actions performed have to be documented as well. Whenever an existing system needs to be modified, verification and validation activities for affected components need to be redone. Therefore it is highly beneficial if verification and validation activities are automated. In FPGA and software verification the repetition of automated verification activities is called regression testing. Regression testing is facilitated if the verification software framework is easily extendible and modifiable [ESC10].

The IEC 61508 functional safety standard does not explicitly require the verification engineer to be different from the designer. It leaves this decision of independence to application specific standards [ESC10]. Other standards for safety-critical electronic designs, like e.g. the DO-254 Design Assurance Guidance for Airborne Electronic Hardware do require independence [FV14]. It is commonly believed that independent verification engineers find more systematic faults than design engineers who verify their own implementation [MBS12]. In 1999 a study confirmed this believe [AGHH99]. The authors formed two groups of developers with similar skills. One group worked together with an independent verification and validation team that performed verification alongside the development. The second group had to develop and verify its software on its own. The study found that the independent verification and validation team detected more faults during development than the other group. Many of these faults were introduced due to ambiguous requirement statements. The independent team found them, because it had a different view on the requirements. It could focus on verification and validation aspects without being influenced by thinking of how to implement the requirements. The resulting software was tested with a test suite that was developed independently from both teams. This test suite found 8 times more faults in the software that was developed and verified by a single team. Another benefit of independent verification that could be observed was that the independent verification and validation team found the faults during earlier design phases [AGHH99]. This has the advantage that at an earlier phase of the development it is easier to remove a fault, because there are less parts of the system developed that could be affected by the necessary correction [BB01]. Therefore for the verification methodology defined in this thesis it was chosen to establish independence between the people developing the Design Under Verification (DUV) and the person who verifies it, which also implied that black-box testing had to be chosen over white-box testing [MBS12].

2.2 Functional Verification of System-on-Chip Designs

This subsection introduces the state-of-the-art in functional verification of Hardware Description Language (HDL) designs on Register Transfer Level (RTL). To ease reading only the term verification will be used in this chapter for describing functional verification and validation. The techniques mentioned can be applied for verification and validation. The difference between these two activities depends on the specification against which the verification or validation is done. Section 2.1 provided more details on this differentiation. This chapter focuses on functional verification of HDL models, because the main focus of this thesis lies on the definition of a verification methodology for the verification of a safety-critical System-on-Chip (SoC). The SoC that should be verified consists of a Field Programmable Gate Array (FPGA) and a processor core that executes an embedded operating system and software. Verification of embedded software is mentioned to some extent, because the verification of a whole SoC on system level includes the embedded software it executes. This will have to be considered when verifying the Design Under Verification (DUV) of this thesis on system-level, because it implements safety-critical functions that are partially implemented by the FPGA and partially by the embedded software. More details on embedded software verification can be found in [MBS12] and [Grü17]. Other verification techniques like layout or electrical verification [Lam05] are out of the scope of this thesis.

Verification is the process of determining whether a system conforms to its specification. The process should provide evidence that a system behaves like intended by its functional requirements specification and like described by its design specification. When verifying safety-critical systems, it is crucial to also verify that the system does not behave in any unintended way [ESC10]. For example a requirement might be that an FPGA code receives some read-only parameter values provided by the software at specific time intervals. Verification activities have to show that the parameters are indeed read at theses intervals, so verifying the requirement in its positive version. And they should show that the value is not modified and not read at any other time than specified, thus testing also the negated version of the requirement. It should also include robustness-testing, which verifies the reaction of the DUV to unspecified input scenarios [But12].

Ideally the result of RTL and software verification is the assurance that the developed system conforms to its specification under all circumstances. For complex designs it is often not possible to demonstrate this conformance for all possible input conditions within reasonable time. Functional verification of RTL design can be divided into simulation-based verification and formal verification, which both have their limitations. Simulation based verification can only provide evidence that the design behaves as intended for the specific simulation scenarios that have been verified, that is for the specific input stimuli applied at specific design conditions. Simulating all possible input scenarios with currently available computers and tools would take many years. In the contrast formal verification can mathematically prove that a design always behaves as specified for any kind of inputs. Unfortunately formal verification on full system level would often require more human and computing resources than available. It took 11 person years to proof that the implementation of the operating system seL4 fully satisfies its abstract specification [KEH$^+$09]. Formal verification is widely used on smaller design blocks that generate their outputs within a few clock cycles or for verifying protocol satisfaction of communication interfaces. For more complex blocks it can easily reach the limits of available tools or require more resources than available [Cor18b]. In many industry design projects both techniques are combined to use both their benefits [GLH18].

2.2.1 Measuring Verification Progress

A 2016 industry study [Fos17] found that for most FPGA design projects the time spent in verification is around 50% of the whole project effort, with an increasing trend. For application specific integrated-circuits (ASICs) this number reaches even 70%. A way to reduce the verification efforts, is the prioritization of verification requirements in order to identify critical functions that need to be fully verified and less critical ones that might not be verified for all cases [BCHN05]. In safety-critical electronic designs clearly the safety-critical functions need to be fully verified. Only for functions for which it can be shown that they are completely independent from safety-critical functions the verification goals could be relaxed [ESC10].

A common metric to measure verification progress is coverage. Verification goals can be stated in percentage of coverage. It can be distinguished between structural and functional coverage. Both types of coverage can be measured for simulation-based verification and formal verification. These measurements help to quantify the progress of the verification activities by comparing the defined coverage goal with the currently achieved coverage. Coverage measures can only asses verification progress when measured for positive verification results [BCHN05]. During the development of verification suites, when the DUV might not yet be fully implemented or fault free, they can be used for measuring the effectiveness of the verification software. In the following, functional and structural coverage will be explained.

Functional coverage measures how many of a design's functions have been verified. The goal for coverage of safety-critical functions must be 100% [ESC10]. A functional coverage model describes the design's functions that need to be verified. The functions can be identified based on the following points [BCHN05]:

- Statements in the requirements specifications and implicit assumptions:
  These functions can be identified by the verification engineer. They can be described in terms of values of input or output signals and their relationships. Assumptions that were made by the requirements engineers will have to be reflected as well. To ensure that the verification engineers interpret them as intended by the requirements engineers, the coverage model should be reviewed by the requirements engineers or further stakeholders.
- Statements in the design specifications and implementation details:
  An independent verification engineer can identify these functions based on a high-level design specification. For black-box verification they may only be described in terms of input or output ports. The functions implied by low-level design specifications or implementation decision are only known to the design engineers. Their description in the coverage model can also contain internal variables. One approach is to cover them through white-box testing by the design engineer. If the verification should stay independent from implementation details, another approach can be chosen. The design engineer can write the coverage model and keep the actual coverage collecting code within the design. This code can be measured during independent verification, however analysis will have to be performed by the design engineers.

During simulation-based verification functional coverage can be measured in terms of observed input stiumli, internal state variables or output values at certain time steps or under certain conditions [BCHN05]. It is important to only cover an input when it its effect has been observed at an output. Outputs should only be covered when their values can be related to corresponding input values, because different internal states could lead to the same outputs [TK01]. In formal verification, functional coverage could be measured by relating the proven theorems or propositions to the requirements specification. As for simulation-based verification, the theorems or propositions are statements about input or output ports, or internal states. A functional coverage model could be grouped into Mutually Exclusive and Collectively Exhaustive (MECE) groups. This grouping can help to identify and distinguish the functions, thus ensuring that no functionality is described more than once. This is important for simulation-based verification, because coverage collection usually slows down the simulation. A grouping based on the one suggested in [SMG15] could be:

1. Use cases, modes of operations, states in state machines:
   This group reflects the specified requirements. An implementation requirement may be a positive statement whilst a verification requirement is also necessary for its negation [BCHN05]. Considering the requirement statement "An alarm has to be generated when the measured value exceeds the threshold.", an implicit assumption is that there is no alarm generated when the integration value is below the threshold. The verification plan has to contain at least one requirement for each case.

2. Illegal conditions and unexpected use cases:
   This group is especially interesting for robustness-testing of safety-critical designs. It could contain unspecified or clearly illegal inputs in order to verify the correct detection and reaction of the system to such situations.

3. Interesting scenarios:
   These could be atypcial use cases that are not directly described by the requirements, or combinations of use cases that are not very likely to occur. These scenarios might not happen often, but are definitely possible and allowed. Staying with the alarm example, this could be the lowering of the threshold value by the user at exactly the same time that the measurement value reaches the threshold. Should the alarm go on or off?

4. Register, protocol and input port value range coverage:
   This group describes the interfaces to the design. It can describe access modes of registers, signal sequences that describe a protocol or value ranges of ports on block-level. Allowed access to the interfaces can be described, as well as scenarios that are expected to produce an error, like e.g. writing to a read-only register. Boundary values of ports and their combination to corner case values can be specified here.

5. Temporal proximity of operations:
   This group describes temporal relationships between signal changes or their values. Temporal relationships can be well expressed by temporal logic properties. They might be well suitable for formal property verification.

6. Parallel events that may happen at random times:
   This group includes events that are not controlled during real operation of the electronic systems. It could describe interrupts due to dangerous conditions, like overheating of the electronics. Or it could verify that an automatic change of power mode during a critical operation does not affect this operation in a dangerous way.

Functional coverage collection is defined by the verification engineer. When the requirements specifications are written in natural language, the coverage model for coverage collection has to be created manually, which is an error prone process [SMG15]. A review of the verification requirements could contain a review of the coverage model to increase confidence in its accuracy. Functional coverage can also be supported by structural coverage.

Structural coverage in simulation-based verification is a metric for measuring the amount of the HDL code that has been verified. It is also referred to as code coverage. This metric is commonly used in software verification as well. Typical structural coverage metrics for software and HDL code are statement, branch, condition [MBS12] and Modified Condition/Decision Coverage (MC/DC) [HV01]. The Questa Simulator implements MC/DC coverage under the name "Focused Expression Coverage (FEC)" [Cor18c]. Further structural coverage metrics specific to HDL designs exist as well. These include state coverage of finite state machines, toggle coverage of bits, expression coverage of concurrent assignments and event trigger coverage [WCC09]. Electronic Design Automation (EDA) tools for simulation often allow the automatic collection of structural coverage during simulation runs [SSR+09]. Structural coverage in formal verification measures the amount of the design structure that has been in the cone-of-influence (COI) of the proven properties. The COI contains all signals that influence the result of a property. Some tools refine this metric to only consider signals of the COI that actually formed part of a proof when calculating the coverage [Cor18a]. When analysing the results of structural coverage measurements, the verification engineer needs to look at the HDL code. If the design engineers and verification engineers should be independent persons, this analysis has to be done by the design engineers themselves or further verification engineers.

Neither functional nor structural coverage can guarantee that all systematic faults are found [KK10], [TK01]. Structural coverage cannot show that a function is missing in the implementation [WCC09]. It might also not be able to tell whether a function is covered, because different functionalities can be constructed out of different combinations of code parts [TK01]. It can only tell whether all implemented code has been tested at some point. It is can be a useful supplement to functional coverage when verifying safety-related systems. Structural coverage can reveal holes in the verification plan, coverage model or coverage collection [DL00]. It can also be useful to detect unintended or unspecified functionalities in the design [TK01]. Assuming the coverage model is complete and 100% functional coverage is achieved, but less than 100% structural coverage has been measured, it might be a hint for unspecified functionality, which is not allowed to exist when a design should conform to IEC 61508[ESC10]. For finding unreachable code, structural coverage collection has to be performed on unoptimzed code. RTL simulators are able to exclude unreachable default statements in case statements from the analysis [Cor18c].

2.2.2 Simulation-Based Verification

Simulation-based verification is done by simulating the HDL code in an event or cycle based simulation tool. They simulate the behaviour of the Design Under Verification (DUV). Based on the user specified simulated input signals the simulated DUV generates the outputs that are expected to be generated by the FPGA once the HDL code is programmed into the FPGA. The input stimuli are generated by a testbench that can be written in an HDL or verification language, like SystemVerilog. Testbench code is only used during simulation, so it does not need to be synthesizable into hardware structures. This allows to use high-level language concepts like object oriented programming, the usage of operating system libraries like file access and many more.

SystemVerilog combines such features of high-level languages with those of HDLs. Additionally to the high-level features, it supports parallelized blocks that are automatically executed as separate simulation threads, blocking and non-blocking assignments, 4-state logic and many more [ICS18]. Modern tools are event based, which means they simulate the reaction of a component to input stimuli only when an input to this component changed. This approach speeds up simulation of asynchronous designs and such with multiple clocks [Lam05]. Since the clock signal also has to be simulated in a testbench an event is created at each clock tick. Even if no components are evaluated, the toggling of the clock takes time. A way to overcome this is to simulate the application of input stimuli with shorter delays than in real operation. The delay only has to be as many simulated clock cycles as the DUV needs to calculate the outputs [BCHN05].

In early years of the existence of HDLs, verification was often done by applying some stimulus to the inputs of the simulated HDL code and manually inspecting the output values by looking at the wave-form produced by the simulator. The larger designs get, the more time consuming this process gets. When the design is modified and it needs to be reverified, the whole effort has to be repeated. Self-checking testbenches can speed up reverification significantly. Initially it might take more effort to write the self-checking components, but at each process of reverification this step can be either reduced to some modifications or completely omitted. Simple self-checking testbenches consist of the application of input stimuli at certain time steps. This is called directed testing. The output values are checked at matching time steps either against manually calculated values or values computed by the self-checking component. The latter option already allows more flexibility. When additional input values should be verified, the HDL testbench code can calculate the additional results, unburdening the verification engineers from this task [Mey04].

Reference Models

Another way that eases the addition or adaption of input stimuli even more is the usage of a reference model. A reference model is an additional implementation of the DUV functionality based on the requirements for the DUV. A reference model could be called a "golden reference model", if it is guaranteed that it reflects the behaviour intended by the requirements specification. This would be the case if the requirements engineers implement it as prototype that demonstrates the desired functionalities. Another option is to write the specification in a formal language from which the reference model can be automatically generated. In both cases the reference model would be an executable specification [GAGS09].

A typical testbench with a reference model is shown in figure 2.2. The same input stimuli are applied to the DUV and the reference model. If the reference model is written in an other language as the testbench, some interface functions may have to be used for communication between the two [ICS18]. The outputs of both models are compared in a self-checking component, often called scoreboard [Ini14]. Any number of input stimuli can be applied without the need for modifying the self-checking component. If the outputs do not match, one of the two models contains a fault. In case the mismatch resulted from unspecified behaviour for which a design decision was taken by the designer, the reference model might have to be updated, but the scoreboard could remain unmodified. In contrast a scoreboard that contains manually calculated expected output values or functionality to compute expected output values would have to be updated in this case. In the case of manually calculated expected values this would mean that many manual calculations would have to be redone [WCC09].

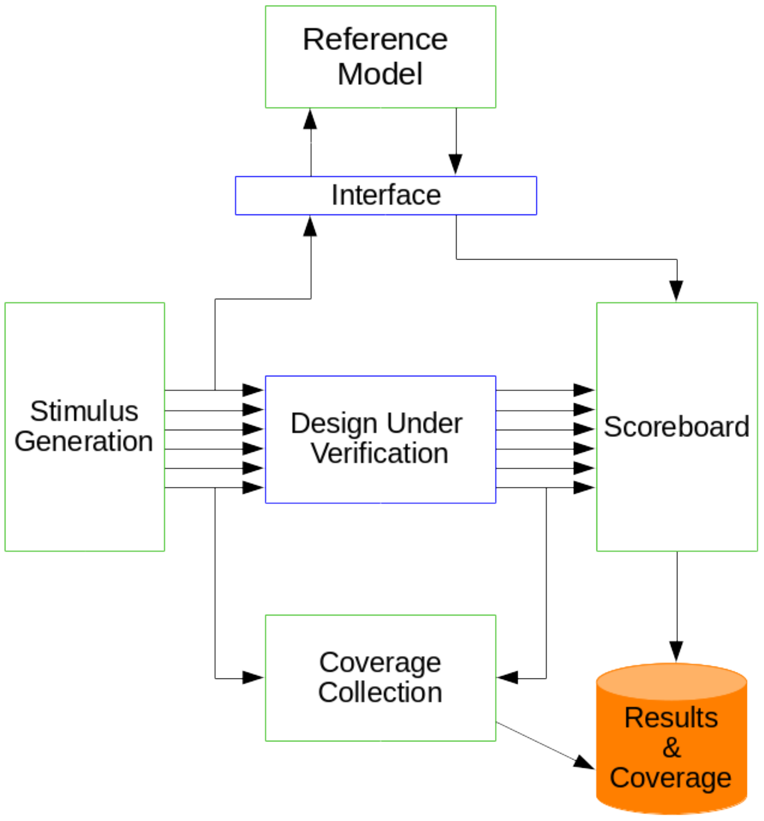

Figure 2.2: Simplified view of a typical testbench with a reference model.

The reference model can be written with a higher level of abstraction than the DUV. A different language can be used. It may not need to model each and every signal and it may not need to be cycle accurate. Such a model is usually easier and faster to implement. Therefore it is also faster to update it, compared to the time it takes to update a scoreboard written in an HDL [Ber03].

Often Transaction Level Modelling (TLM) is used within a reference model or testbench [GAGS09]. If the reference model is written with a higher level of abstraction, comparisons of output values are done on transaction rather than on signal level [WCC09]. Several signal values possibly collected over several clock cycles are grouped together to form a transaction. The reference model can model some of the signals within the transaction, but it does not need to model each and every signal value change. It receives input transactions and produces output transactions. The HDL model outputs are grouped to form an output transactions as well. These transactions coming from the two models are then be compared. If they match, the test passed [Ini14].

Reference models could also be written in the same language as the HDL code. This would avoid mismatches due to different data types and number representations in the languages. On the other hand exactly these similarities between the two models could hide errors that might only be visible when using different languages with a different internal number representations [MK07]. Another advantage of using a cycle-accurate HDL reference model is that the comparisons of output values can be done easily on signal level. If the reference model is written on a higher level of abstraction and is not cycle-accurate, some effort might have to be made to synchronize it with the DUV during simulation. However, once this synchronization code is implemented in the test-bench, there are less comparisons necessary than on system level, which can speed up simulation significantly [WCC09].

A model with a higher level of abstraction could be written in SystemC, which is a C++ library for system-level modelling [Soc12]. A SystemC model has a notion of simulated time. It can model concurrent processes and their communications. It can model a hardware architecture more accurately than an untimed language and is therefore very useful to model processors or System-on-Chips. SystemC uses TLM within the model. It distinguishes between a loosely-timed and an approximately-timed coding style. In the loosely-timed coding style temporal decoupling is possible, which means that a part of the model can run ahead in simulation time. Synchronization can happen when components of the model communicate through blocking transport functions. A component has to wait either when it places a transaction into a blocking send function, that waits until the receiving component accepted it or when it waits for a transaction to be received. In the approximately-timed coding style even phases within a transaction are synchronized to the SystemC simulation time [Soc12].

If timing delays within the model are not of verification interest it can be even more efficient in terms of implementation and execution speed to write an untimed behavioural model at an even higher level of abstraction. This can be done when the outputs are a function on internal state and inputs. Such a model could be a collection of functions that are called from within the testbench. Each function could for example model the function of one hardware block. A system-level model that does not need to model internal timing could model the passing of real-time by a counter and use a rendezvous-based concept to synchronize with the testbench [JS05]. These kind of models can be written in a high-level language like C or C++. Often such models execute faster, because they do not need to model time, synchronization between processes or TLM communication [Ber03]. Data exchange inside the model and with the testbench can be done through function calls and variables. The testbench could group it into transactions and only and extract signal-level data when it communicates with the DUV.

A C or a C++ model that is linked as C code can be accessed from within a testbench written in SystemVerilog through the Direct Programming Interface (DPI) of SystemVerilog. The DPI is usually integrated into a simulator, because it needs some simulator specific code. It allows the SystemVerilog code to call C functions and it also allows the C code to call SystemVerilog functions and even tasks, which can be simulation time consuming. This can be used to bring the notion of time into the untimed C language. It could be used to synchronize the C model with the DUV. The C model needs to be compiled into a shared library that allows late symbol binding at runtime. If any system libraries are used, they have to be linked to it. The resulting C library is then passed to the simulation by a simulator command line argument. At runtime the function calls to the other language are resolved. Apart from the fact that a lot of repetitive code is required, the DPI interface is relatively simple to use. More complex but also less restrictive access is possible through the Programming Language Interface (PLI). VHDL provides a similar interface, called the Foreign Language Interface (FLI) [Cor18c].

Constrained Random Verification

Simulation-based verification verifies only a subset of the whole design input space. Directed testing reduces this subset even more. It only verifies the scenarios that are described by means of input signal sequences in the testbench. With automatically generated input values more scenarios can be verified with less testbench development time than when specifying all these value sequences manually. When a self-checking testbench with a reference model is implemented, there is no need for manually calculating the expected values. Therefore the number of applied input values is only limited by simulation time, but not by development effort [BCHN05].

That it is not feasible to simulate all possible input values within reasonable time, can be easily understood when looking at the time that it takes to simulate all possible input values of a 64 bit wide input port. Assuming a simulation computer with a clock of 2.5GHz and optimistically assuming it would be possible to apply one input value to the simulated DUV at each clock cycle of the simulation computer, it would take 233 years to simulate all possibilities for this one input port. Now assuming that the output depends on the past 10 values of this port, the input space for this port is increased to $2^{64^{10}}$. Under the same assumptions as above it would take $5.8*10^{176}$ years to simulate all possible 10-value long input sequences. Realistically there would even be multiple clock cycles of the simulating computer happening before the simulator can apply a new stimulus because of other operating system processes running. The period that has to pass before new stimulus can be applied to the simulated design also depends on the nature of the design. Clearly, with simulation-based verification exhaustive simulation of highly parametrized designs with large parameter value ranges is not feasible within reasonable time. Therefore a subset of input values has to be chosen. To increase the number of inputs that can be applied within reasonable time, the delay between the application of two consecutive inputs can be shortened. It does not need to be the as long as the delay that would occur in real operation. It is only limited by the processing time of the simulated DUV. In a pipelined design an input value could be applied each simulated clock cycle [BCHN05].

A way to choose a meaningful subset without actually specifying this subset in terms of input value sequences is coverage driven constrained random testing. The input values are generated automatically to be random. Constraints on the random value generation make sure that the generated input values are possible in real operation. Coverage collection ensures that the applied input values are relevant for the verification. The advantage of using random values is that the automatically generated values are spread over the whole range that is allowed by the constraints. This increases the probability of finding errors because more different design states are simulated [MR10]. The value distribution can also be modified to generate more values in the region of extreme values or boundary values, which helps robustness testing. Another advantage of random input values is the automatic generation of corner case input values. Corner cases could be a combination of boundary values of inputs or internal signals. They could also be a combination of extreme values of mathematical functions in a design that could lead to mathematical errors like division through 0. It could be that they result from design decisions that are not visible to a verification engineer [BCHN05]. Corner cases can also be simulation-scenarios that are hard to generate intentionally [VPH09] or are not thought of by the verification engineer. In some cases it can be very complex to manually write or generate a test case that exactly creates a simulation scenario that verifies these corner cases [TK01]. It is often much easier to identify the signals for which boundary or extreme values should be verified. These signals or combinations of multiple signals can be specified in a coverage collecting component by using dedicated verification language constructs. Corner cases of internal signals might only be known to the design engineers, so they should specify how these cases should be covered [BCHN05]. Simulation with random input values can be run until these complex scenarios have been seen by the coverage collecting testbench components.

Applying totally random input values will generate scenarios that are not possible in real operation. Therefore constraints can be applied to the random value generation that limit the generated values to those possible in real operation. Especially when dealing with safety-critical designs the constraints should not restrict the input values to legal or expected values only. Safety-critical designs should also be demonstrated to work under unexpected conditions [But12, But15, ESC10]. Constraints can be used to restrict the input values to possible values, if it can be guaranteed by some other means that another HDL block or device that generates the input values in real operation always generates values e.g. within a certain bound or of a certain pattern. Only then such a constraint can be safely applied [SSK15].

SystemVerilog constraints are fed into a simulator-internal constraint solver. The simulators often combine several different solvers, like Binary Decision Diagram (BDD) or Boolean Satisfiability (SAT) solvers [Yeh14]. These are similar techniques like those mentioned in section 2.2.3 Formal Verification. The simulators find random numbers that fall into the value range described by the combination of all constraints. In SystemVerilog there are several ways how to write constraints. They could be boolean expressions that or logic implications that describe relationships between signals. Value ranges with distributions can be specified as well. Ordering of constraint solving can be expressed, as well as conditions under which a constraint shall be enabled. Listing 2.3 shows an example of a complex constraint that models allowed time values, which are stored in a random variable called "intTime". This variable holds a value that represents a time period in 100ms steps. The constraint in line 5 restricts the value to seconds. The constraint in line 7 assigns weights to some values. These weights are used to modify the distribution of the randomly generated values for "intTime". Line 8 assigns a high weight, stored in a (possibly also random) variable, called "highWeight". Line 9 assigns a lower weight to a range. It is uniformly distributed over all values in the specified range. Values of "intTime" that represent a full minute are of special interest or represent a corner case in this example. There could be a test case that sets the "mode" variable to "minutesOnly", in which case the constraint in line 13 is applied. In SystemVerilog such constraints can be placed in any class that contains random variables. Inline constraints can also be applied to a call to the randomization method of the class. In that case both kind of constraints are combined. This is very useful feature to describe different verification scenarios.

```
rand intTime;

constraint integrationTimes {

  intTime % 10 == 0; //full seconds only

  intTime dist {
    0            := highWeight,
    [1: oneHour] :/ lowWeight,
  };

  if (mode == minutesOnly)
    intTime % 600 == 0;
};
```

Listing 2.3: Example of a complex constraint in SystemVerilog.

Verification Libraries

The **Universal Verification Methodology (UVM)** [Ini14] is a verification methodology that builds on best practices in simulation-based electronic design verification. It combines the benefits of all its predecessors. It defines concepts for implementing testbenches in a reusable way. Reusability is facilitated through a simulator- and therefore vendor-independent open-source SystemVerilog library.

The first verification library developed for facilitating reusability of verification software was the e Reuse Methodology. This methodology already introduced the architectural components and concepts that were later adopted by UVM [MR10]. The first verification library for the SystemVerilog language was the SystemVerilog Verification Methodology Manual (VMM) library. It was a proprietary library developed by Synopsis for use with their simulator. It was desribed in a book that contains many concepts and practical advices that are still applicable to verification with UVM [BCHN05]. UVM is heavily based on the Open Verification Methodology (OVM), which was originally created by Cadence and Mentor as the first verification library that was simulator independent. UVM has been defined by the Accellera Systems Initiative, which is a not-for profit organization who's purpose is to create standards for system-level design and verification. Many major vendors of Electronic Design Automation (EDA) tools which are used for design and verification of electronic systems are members of the organization [7].

UVM facilitates the creation of a typical testbench architecture that uses the architectural components typically also used in the previous mentioned methodologies, however it does not impose any specific architecture. UVM uses many advantages of the object oriented capabilities and preprocessor macros of SystemVerilog. The core concept of UVM is to divide a test run into several phases. Testbench components that are registered with the UVM factory can implement code for these phases. At runtime the phases are traversed in a defined order. Each phase has its specific task like configuring the verification environment, applying stimuli and generating reports. Test scenarios can be written or randomly generated in sequences that can be scheduled in fixed or random orders. When the DUV is accessed through multiple interfaces, sequences can be applied concurrently, using SystemVerilog threads. A SystemVerilog class becomes a UVM component by being derived from a UVM base class. The factory pattern is used to increase flexibility and reusability by allowing object type overriding after compile time. Type overriding could happen at runtime, depending on a testbench internal configuration or condition or it could be enforced from the command line. Several options can be configured through command line arguments that can be easily accessed through UVM functions. This allows e.g. to reuse the same test case and testbench components for sending several different sequences to the DUV, which are specified on the command line and override the sequence originally used in this test case [MR10].

There are several constructs implemented for using TLM within the testbench. DUV signals are abstracted into transactions that are passed around the testbench through blocking and non-blocking ports or sockets. This is a practical concept to synchronize testbench components. Another way of synchronisation are UVM events, which are extensions of SystemVerilog events. UVM adds the possibility to manage the waiting processes, stop waiting and to transfer data with the triggering of an event [Ini14]. The UVM library implements many functionalities that are typically needed in every testbench. It provides highly customizable reporting functions and configurable reporting macros for easing the logging of results. It is possible to limit the generation of log files to a specific simulation time that is configured via the command line. This can be useful when an unexpected result has been observed at some point in a simulation run that was configured to produce only little log outputs. Simulation can then be repeated and logging can be increased only around the time of the test fail [Cor18c].

UVM relies on SystemVerilog for coverage collection. A coverage collecting UVM component can receive the abstracted DUV values as transaction through a TLM port. Events can be used for synchronisation and First-In First-Out (FIFO) ports can be used to store transactions in case the produces transactions faster than the receiver can process them. In SystemVerilog "covergroups" can be specified, that can be parameterized and instantiated like classes. They contain "coverpoints" that describe which variable or condition should be observed. A coverpoint can observe input stiumli, internal state variables in case white-box testing is done, transaction or testbench internal values or output port values. A coverpoint can also observe the outcome of an expression or condition. A coverpoint contains bins that specify values, value ranges or value transitions or repetitions that should be observed. When a value that falls into a bin's range is observed, the coverage of this bin is increased. When the goal that specifies the number of values that should be observed for each bin is reached, the bin is said to be covered. For each bin a condition can be specified, under which an observed value that falls into that bin should add to the coverage goal. Coverage can be sampled concurrently by the simulator or explicitly by calling a covergroups sample method. It can be specified in the coverpoint declaration under which conditions the variable or expression observed by a coverpoint is valid for sampling [ICS18]. Boundary case values and extreme values of mathematical functions could be stated as explicit bin values, because those kind of values are usually more likely to produce errors [MBS12]. When a combination of values should be observed, individual coverage criteria can be combined, which is called cross coverage. Combining multiple boundary values leads to the coverage of corner cases. Functional coverage can also be sampled through temporal logic statements by using SystemVerilog sequences or properties directly in the interface to the DUV. The same set of concurrent cover statements can be used during simulation-based and during formal verification [ICS18].

In terms of cost it is a disadvantage that UVM builds upon SystemVerilog. When verifying VHDL designs with a SystemVerilog testbench, mixed-language simulation is used. Commercial simulators often force a user to buy an additional tool license to enable this feature. Using constraints to guide the random number generation requires a constraint solver inside the simulator.

Because these consist of very complex algorithms, vendors often ask a user to buy an additional license for these features as well [Cor18c]. EDA tool licenses are usually very expensive and only pay off if used for several projects within the organization [PAWJ]. Apart from tool cost, getting started with UVM can also be expensive in terms of personnel cost. It is very time consuming to set up an initial testbench and learn how to use all available features.

Recently the **Open Source VHDL Verification Methodology (OSVVM)** [9] has been introduced. It is a VHDL library that uses similar concepts as UVM. It is written in VHDL and provided as VHDL packages, therefore there are no specific language constructs available that would allow complex modelling of constraints and coverage collection. The advantage of this approach is that there is no constraint solver needed. OSVVM provides VHDL functions for generating random values within specified sets. These sets can be specified as ranges and single values can be excluded. The distribution of generated random values can also be modified. A limitation is that weights for the distribution can only be specified for single values and not for ranges. Compared to the possibilities described above and the constraint example shown in listing 2.3, the use cases for which OSVVM is advantageous over SytsemVerilog are quite limited. Also object-oriented features are not directly available, they could be modelled through protected types though. OSVVM cover point variables are unsigned integers, which leads to further limitations imposed by VHDL. Integers in VHDL are maximum 32 bit wide, which means for a DUV like the one in this thesis, some wrappers would have to be written to cover all 64 bit wide signals. More problematic though is the fact that the binary value where the most significant bit is 1, followed by all zeros ("100....000"), is no valid VHDL integer value. For a signed 32 bit variable that is internally stored as bit vector in two's complement, this value is the most negative value and therefore constitutes an important boundary value. It is not directly possible to state this value as cover point bin.

Nevertheless there are use cases where OSVVM is advantageous over UVM, because of its lower complexity and feature overhead. OSVVM has been successfully used for the verification of a SIL 2 compliant floating point core. This core has has been developed and verified within the CROME project. It is integrated in the DUV of this thesis. This core won't be reverified with the verification suite developed in this thesis, but it will be implicitly used within the blocks that are verified. Because of the mentioned limitations of VHDL integers a bidirectional lookup table was created that maps bit vectors to OSVVM cover points. The verification was partly done by directed tests and partly by randomized tests. Some directed tests were automatically generated by an OSVVM feature called "Intelligent Coverage". It can find uncovered bins at runtime and use this information for generating coverage directed stimuli. While this feature is useful to get single uncovered values or one uncovered range, it is difficult to use when cross coverage of single values and ranges is needed. OSVVM internally flattens the combined values into one set per coverage bin. For example value "5" crossed with range "20-30" would become "(5,20,30)". If there is another cross of value range "5-20" with value "30", the flattened bin would become "(5,20,30)" as well. The "Intelligent Coverage" function does not provide the information which of the two original cross cover bins are uncovered. A lookup table approach was used to overcome this limitation [Ger17].

UVM relies on SystemVerilog for coverage collection. In SystemVerilog only the coverage status of whole cover groups or cover points can be accessed, so if an uncovered bin should be found one would have to create cover points with single bins. A feature like "Intelligent Coverage" would certainly be a useful addition. SystemVerilog though allows much more complex specifications of cover points and bins. Generating input stimuli based on the coverage of such kinds of cover points would require the automatic generation of complex sequences. The generated stimuli would have to have fixed values and orders, so that the cover conditions are fulfilled. This is not trivial [ICS18]. Several researchers are looking for ways to speed up automatic coverage closure [IK15], [YWD14]. Before using such techniques, one needs to carefully define ones goals. If the goal is to find faults that might not even have been thought of when writing the coverage model, coverage driven stimulus generation might be counterproductive. A combination of coverage-driven stimuli and constrained random stimuli could be used.

Several more lightweight open-source verification libraries for different purposes exist. UVVM is another VHDL package library which uses a concept of sending commands to the DUV. It provides only limited functionalities for random value generation and does not contain coverage collection [1]. There are also several python based approaches like for example cocotb [13] and VUnit [2]. Both are open-source libraries that were designed to facilitate the automation of testbenches. VUnit provides python functions that can be used within a VHDL or SystemVerilog testbench. Cocotb tests are written entirely in python. Both work with commercial and open-source simulators. Random test value generation can be used by both approaches. VUnit relies on the randomization functions of VHDL and SystemVerilog, whereas with cocotb python functions can be used. These libraries provide no features for coverage collection, so the measurement of verification progress is left to the verification engineer. Anyway both of them target module/unit testing rather than stress testing on system level. Both of them provide verification libraries for standard bus interfaces. Such verification components also exist for UVM [Cor18c].

Since verification code can often get very repetitive, there have been various products arised that automate the generation of SystemVerilog code. MATLAB contains the HDL Verifier [11], which is able to generate the SystemVerilog DPI interface code for communicating with a C model. Writing this code manually is very cumbersome, especially when fields of C structures need to be accessed, which is not directly possible from within SystemVerilog. Mentor's UVM framework [5] and Doulos' Easier UVM Code Generator [6] generate whole UVM testbench templates. They generate empty testbench components and their Transaction Level Modelling connections. Such tools can be very useful to get started with the methodology. A limitation though is that the component and variable names cannot be completely freely chosen. In [VHR07] and [MW16] frameworks for automatically generating SystemVerilog covergroup code were proposed. The covergroups are generated based on the DUV. This approach can be practical for whitebox testing or testing by the design engineer, but if independent verification is desired, this approach might reveal too much knowledge about the design to the verification engineer. Furthermore extracted coverage reflects the implementation and not the underlying requirements. VerifSudha [10] is a framework for generating SystemVerilog covergroups from specifications written as Python datastructures, as well as whitebox coverage based on the RTL design. In [CCO$^+$18] a requirements management tool was developed that can generate SystemVerilog covergroups from a spreadsheet. The tool also feeds back the result to the verification plan. It utilizes the coverage database of the Mentor Questa Simulator [Cor18c]. This database follows the Accellera Unified Coverage Interoperability Standard (UCIS). This standard was developed to facilitate exchange of coverage information between different tools [Ini12]. Other EDA vendors implement similar databases [7].

Several commercial and open-source verification libraries have been introduced in this subsection. They work with commercial and open-source simulators. Not all of them may be used for the verification of safety-critical systems that should conform to the IEC 61508 standard for functional safety. The standard requires the used tools and libraries to be either certified as conforming to the standard, or be proven in use for many years for the verification of safety-critical designs. This rules out novel and experimental tools or libraries [ESC10].

2.2.3 Formal Verification

Formal verification mathematically proves that a design satisfies its specification for all possible input value sequences. It has been used in hardware verification for many years. Mostly it is employed only on critical subparts of a design and not very often on full system level [**?**], [GLH18]. Many different formal verification techniques for hardware and software exist. The ones most commonly used for hardware verification are detailed in this chapter. These are theorem provers, model checking, formal property verification and equivalence checking.

Automated Theorem Proving (ATP) is the attempt to prove mathematical statements about a design. A mathematical description of the Design Under Verification (DUV) is needed. This description has to be written in a language that can be understood by the proving tool. Usually tool specific languages have to be used, so an HDL design first has to be translated into this language. The mathematical statements, called theorems, are proven on the mathematical design description through logic reasoning, rewriting, mathematical induction and many other mathematical proof techniques. The theorems can be specified in first-order logic, which is very expressive, but computationally complex to solve. Theorem provers are often not fully automated. Proofs are performed in an interactive process guided by the user who analyses intermediate results and selects further proof strategies [**?**]. Provers like ACL2 and HOL have been used for the verification of several hardware designs of Intel, Motorla, AMD and many others. The tools are mostly used for specific subsets of designs that can be well expressed in mathematical terms, like e.g. a floating point unit [8]. An advantage of theorem provers is that they support large designs with any number of states. A disadvantage is that their usage cannot be fully automated. Often it is not easy to trace back a failing proof to the underlying cause of the failure, because the tools do not provide counter examples [GLH18].

Model checking uses the approach of explicit state enumeration. The number of states of a system equals 2 to the power of the number of bits that can change the state of a design, which includes e.g. inputs, internal variables and memories. Symbolic model checking reduces the state space by using representations that group several states into one symbol, e.g. a node in a Binary Decision Diagram (BDD). A Reduced Ordered BDD (ROBDD) is a minimal and unique representation of a boolean function. Several more methods for state representation exist, for example Buechi automata, symmetry reduction and abstraction refinement [**?**]. A symbolic model checker incrementally calculates all possible next states of a design until there are no more new states added. The tool then checks whether a proposition specified in a temporal logic is violated on any path through the function. Model checkers can also use boolean SATisfiability problem (SAT) solvers. SAT solvers iteratively construct an expression that describes all possible execution paths from a design state and concurrently checks that the propositions are satisfied along each path [SSK15].A symbolic model checker can either conclude that a proposition is true for all reached states in a design or it can provide a counterexample. When using commercial tools for FPGA verification the counterexamples are presented as a short waveform, which makes it easy to debug [Cor18b]. Several open-source symbolic model checkers exist. Two examples that are widely used and extended are NuSMV and Uppaal. NuSMV combines BDD-based techniques with SAT-solving. It accepts properties in Computational Tree Logic (CTL) and Linear Temporal Logic (LTL). Numerous customized tools based on NuSMV have been developed. Uppaal is designed for verifying real-time systems by using timed-automata representation and extending temporal logic with timing conditions. Both tools work on models and propositions that are described in tool-specific languages, Uppaal even provides a graphical user interface for designing the automata. Even the reduced number of state symbols that is used by symbolic model checkers might not be small enough to verify an electronic design within reasonable time with currently available tools and computers [GLH18]. Bounded model checking can be used instead. It can prove that a proposition is true up to a certain bound [**?**].

The above mentioned tools for theorem proving and symbolic model checking cannot be used directly for the verification of HDL designs, the model on which the theorems or propositions are proven has to be specified in a tool-specific language. The equivalence of the model with the HDL design would have to be guaranteed in some way. Some proprietary and open-source research tools exist that translate HDL code into these tool-specific languages. The open-source tool Verilog2SMV translates Verilog designs together with SystemVerilog assertions into a word-level representation that can be used by the NuXmv model checker, which is an extended version of NuSMV [ICG+16]. A similar approach has been used in [MHS18]. For the verification of safety-critical designs that should conform to the IEC 61508 standard, the used tool would either have to be qualified by conforming to the standard or being proven in use [ESC10]. Therefore it is difficult to apply theorem proving to safety-critical designs.

Formal Property Verification (FPV) is the application of model checking directly on the HDL code. FPV tools typically implement bounded model checkers to prove properties written hardware verification languages. Properties might be proven for all possible states, it might be shown through a counterexample that they can be violated or they might be inconclusive. Inconclusive properties can be proven from the initial state up to a certain number of clock cycles, which is then the proof bound. If it can be shown that a design is able to calculate all outputs within this number of clock cycles, the design can be considered as verified. Some techniques to determine the needed proof depth are explained in [KPSS14]. Several commercial tools for formal property verification exist that are qualified according to the IEC 61508 standard and that can directly take HDL code as input. The temporal propositions for these tools can be specified in several languages, e.g. in the Property Specification Language (PSL) or SystemVerilog. They extend (LTL) formulas by sequential extended regular expressions that facilitate the writing of repeating statements. Like mentioned formal verification is typically not done on a complete system. Mostly it is accompanied by simulation or emulation. An advantage of using these languages is that they can be directly used in simulation as well, which can serve as a verification of the properties.

Formal Equivalence Checking (FEC) is another formal verification techniques that is widely used for hardware design and verification [SSK15]. It is often used to formally guarantee the equivalence of two versions of the same design. Equivalence means that some defined key points within the models are equivalent. Those could be primary inputs or outputs, state elements or cut points. Cut points describe internal signals where parts of the design are "cut off" and treated as black-box. In that case only the non-black-boxed part of the design is checked for equivalence. The models could be checked for combinational equivalence, where each state of one model is matched to an equivalent state in the other model. They could also be checked for sequential equivalence, which would prove that the outputs produced by the two models are the same in each cycle, no matter what inputs were applied [SSK15].

The two versions of the model can be at different abstraction levels. For example when a netlist is generated from an RTL description, the tool that generates the netlist can formally proof that the two models are equivalent. Formal equivalence checks are also used after optimizing a design. The current trend goes to model-driven design where prototypes or the whole specification are specified in higher level languages like C, C++ or SystemC [PWPD18]. Tools exist that can synthesize these models into RTL models and formally check their equivalence. These tools can also check the equivalence of separately implemented models in a higher-level language and HDL. In [SSM$^+$02] a C model was written and verified with a C testbench. It was then synthesized into a Verilog model by a proprietary tool.

It can be difficult to automatically prove the equivalence of two models that are implemented in very different ways. Often mapping of key points and states has to be provided to the tools. If it is not sufficient to map the primary ports, internal design knowledge is necessary to do so. Some parts of the design might be too different, so that existing tools cannot prove their equivalence. These parts can be treated as black-box and the proof could be done on a subset of the design [SSK15].

Formal verification promises that a fully verified design satisfies its specification under any circumstances, which often makes people believe that formal verification can guarantee that a design behaves as intended. What formal verification cannot protect from though, is an incorrect specification. It also cannot prevent the introduction of faults into the formal models or properties, when they are derived from a specification by a human being [Lam05]. Therefore especially when verifying safety-critical designs all automated verification activities should be accompanied by reviews of all stakeholders [BCHN05], [SSK15]. The Unified Modelling Language (UML) was created to facilitate communication between stakeholders of different backgrounds. A lot of research has been conducted that aims at using UML class diagrams as inputs for model checkers [PWPD18]. In [GJC12] this was applied to safety-critical systems. In [HCL07] safecharts, which are a modified version of UML statecharts, are adapted so that they can be used as input to a model checker. Safecharts model hardware component failures and assign risk levels to each state. A formal specification written in UML or any other (semi-)formal language can be easier understood by people unfamiliar with the used verification and implementation languages, so it could be reviewed by a client, user or high-level system engineer [PWPD18]. In the CROME project this would be physicists who are the domain-experts for radiation protection and main authors of the functional requirements specification, but might be unfamiliar with FPGA verification and design. It is very important to involve them into the design process, because the FPGA development and verification engineer might on the other hand not be familiar with radiation protection. The IEC 61508 standard recommends to use semi-formal methods for specifications [ESC10].

There have been several attempts made to automatically generate properties in a property specification language from requirements written in natural language. Such tools could increase verification productivity, because they avoid interpreting the requirements in different ways. Often they also enforce to use a subset of the English language to ensure the writing of more unambiguous requirements. In [HH16] a proprietary tool was developed that is able to translate requirements that describe signal behaviour very precisely. For requirements that are written on a more abstract level it is more difficult to automatically generate properties on signal level. In [YCC15] a research-prototype has been developed to automatically generate LTL properties out of high level requirements. They allow only a restricted English grammar to be used for the specification of the requirements. Their tool then heuristically partitions variables into input and outputs and extracts timing information. They also apply semantic reasoning to find inconsistencies in the requirements.

In [SSK15] a number of examples of false positives in formal verification of RTL designs that happened at Intel are described in great detail. These examples show how easy false positives can be generated. The book chapter gives hints on how to reduce this risks. It especially emphasizes the verification of the formal assumptions through manual code review, by placing any formal assumptions and assertions into a simulation or by adding cover properties that ensure that the assertions are not passing vacuously, which happens when implications pass because their antecedent is never true.

Structural coverage can also be measured during formal property verification. Formal coverage metrics have been defined in [CKV03]. A tool that calculates the formal proof can count e.g. the state bits and input and output port bits that were part of a proof [Cor18b].

Chapter 3

Methodology for Functional Verification of Safety-Critical Electronic Designs

A methodology has been defined for the verification of the CERN radiation monitoring electronics (CROME) Measuring and Processing Unit (CMPU). It consists of a workflow and a selection of techniques and technologies for each step. The methodology conforms to the requirements for verification of electronic designs which are classified as SIL 2. They are specified in IEC 61508 [ESC10]. The methodology combines automated verification with manual reviews for validation. Simulation-based verification with the Universal Verification Methodology (UVM) [Ini14] is combined with Formal Property Verification (FPV). This chapter will detail the methodology. The steps defined for fulfilling the Safety Integrity Level 2 (SIL 2) are generally applicable to the life cycle of a safety-critical electronic system. The verification methodologies defined are applicable for System-on-Chips (SoCs) with similar design characteristics. These include high configurability, large input spaces and continuous calculations over long periods that are influenced by previous device states.

In [But12, But15] a verification process has been defined that extends the verification process described in the DO-254 standard for safety-critical airborne systems [FV14]. The author suggests to add constrained random testing to directed testing, with an emphasize on robustness testing with input values outside the specified ranges. He combines simulation-based verification with assertions that are added to the Design Under Verification (DUV) code by the design engineers. It differs from the methodology defined in this thesis in the way that it uses assertions during simulation and during the design phase, whereas the presented methodology uses assertions for formal verification.

The IEC 61508 standard recommends fault-injection as one verification technique [ESC10]. It is often done in model-driven design approaches, where faults can be injected on several abstraction layers and the fault-tolerance of the system can be assessed. [WSP$^+$17] and [CA15] both developed frameworks and verification process based on these techniques. In [TCE$^+$15] the Transaction Level Modelling (TLM) interface of SystemC has been extended by fault-injection functions. Fault-injection was out of scope of this thesis. The complete DUV of this thesis is triplicated inside an FPGA and a triple modular redundancy voter monitors the outputs. A separate watchdog circuit also monitors the state of the DUV. A soft-error mitigation core prevents from single event upsets through radiation [Ger17]. Verification on electronic system-level will have to include fault-injection. It can be done as addition to the verification techniques suggested in this thesis.

The safety life cycle phases for Field Programmable Gate Arrays (FPGAs) are covered in part 2 of IEC 61508. The standard mostly includes FPGAs when it specifies requirements for ASICs. Where it differentiates between ASICs and programmable Integrated Circuits (ICs), which include FPGAs, is the specification of techniques for avoiding the introduction of faults during the design. Furthermore it adds an additional life cycle phase for systems that contain FPGAs and software, the programmable electronic integration phase, where FPGA and software are combined and verified together.

For some phases in the safety life cycle, IEC 61508 specifies techniques for avoiding the introduction of faults. The SIL aimed for determines whether specific techniques are mandatory or at least one technique of a list of recommended ones has to be chosen. Table 3.1 lists the requirements for system verification together with the techniques that were chosen to be used in the methodology defined in this thesis. It also provides a link to the respective section in this thesis. Techniques that are highly recommended (HR) for SIL 2 have to be used if no reasonable justification for not using them can be provided. If IEC 61508 recommends (R) techniques, then at least one of the recommended ones has to be chosen. For some requirements, additional (A) techniques that are not specified by IEC 61508 are added by the verification methodology defined in this thesis. The following paragraphs provide an explanation of the table.

Verification Planning:
The IEC 61508 standard lists some points that need to be contained in a verification plan [ESC10], as will be shown in section section 3.2. An additional step has been added in this methodology: Verification requirements are reformulated into temporal properties written in natural language. This step eases communication between the involved engineers. The technique will be described in section 3.2.4.

Requirements conformance in each design and development phase:
This clause refers to the phases of the V-model, as was shown in figure 2.1. Each design and development phase (shown on the left side of the V-model) is accompanied by a verification of that phase (shown on the right side of the V-model). Part 3 of IEC 61508 regards safety-related software. It requires forward traceability from the safety requirements through all phases up to the verification results, as well as backward traceability the other way round [ESC10]. Traceability is also common in other safety standards, eg. the DO-254 [FV14] for airborne systems. Despite not being required for FPGAs, traceability has been added to the presented methodology for FPGA verification as well. It is a practical way to measure progress and completeness of the validation and verification activities.

Verification results documentation:
These two points regard the documentation of the positive and negative results, including corrective actions taken when the results deviated from the expected ones. Requirements traceability is added here as well.

Verification of the system design requirements:
Verification of requirements is the only verification activity for which the IEC 61508 standards requires a second person to be involved. It is done through reviews and inspections. The usage

Table 3.1: E/E/EP system verification requirements and techniques

Sections in IEC 61508:2010 - Part 2	**Verification Requirement by IEC 61508**	**Selected SIL 2 techniques for avoiding faults**	**Section in Verification Methodology of this thesis**
7.9.2.1-4	Verification Planning	**A: semi-formal methods**	Natural Language Properties
7.9.2.5	Requirements conformance in each design and development phase	**A: requirements traceability**	Verification Documentation
7.9.2.6, 7.9.2.10	Verification results documentation	**A: requirements traceability**	Verification Documentation
7.9.2.7, Table B.1	Verification of the system design requirements	Table B.1: HR: inspection of the specification R: semi-formal methods	Review of Requirements and Design Specifications
7.9.2.8, Table B.2, Table B.5, Table F.2	Verification of the system design and development	Table B.2: R: computer-aided design tools R: simulation -: formal methods Table B.5: HR: functional testing M: documentation HR: expanded functional testing R: black-box testing **A: constrained-random input** Table F.2: HR: application of proven in use VHDL simulators HR: functional test on module level HR: documentation of simulation results R: coverage of the verification scenarios **A: coverage for regression testing**	Verification Software Framework
7.9.2.9, Table B.3, Table B.5	Verification of the system integration	**A: formal methods** Table B.3: R: black-box testing Table B.5: HR: functional testing HR: expanded functional testing R: black-box testing	Verification Software Framework

of semi-formal methods is recommended for facilitating communication. The standard mainly refers to diagrams like flow-charts that can be used by the design engineers, eg. to express timing relations or to describe state machines.

Verification of the system design and development:
This phase includes all subphases of the V-model. Computer-aided design tools are used for the FPGA design as well as for simulating the design inside a testbench. Formal methods are neither recommended nor forbidden. The standard classifies the effectiveness of formal verification as low even if done by experienced persons. It only classifies them as high when the verification engineers

have already formally verified similar designs before [ESC10]. One has to add though, that the latest version of the IEC 61508 standard was written in 2010. Since then tools for using formal methods have advanced. Often they can be used without detailed knowledge of the underlying mathematics [SSK15, Cor18b].
Functional testing is extended by "expanded functional testing", which covers stress-testing. Constrained-random simulation is added to increase the probability of faults even more [But15]. "Sufficiently high" functional coverage is required by the standard. There is no mentioning of how "sufficiently high" is defined. The standard leaves the definition of coverage metrics up to the verification engineer. It only requires the documentation of the established goals and whether they are achieved. Additionally to the coverage goals based on the requirements, coverpoints for regression testing are added in the defined methodology. The standard explicitly mentions that code coverage alone is not sufficient and that it usually only serves to show which functionality has not been covered. During functional verification automatic verification of output sequences should be chosen over manual inspection of output signals [ESC10]. This clause is met by the presented methodology with the usage of a reference model.

Verification of the system integration:
System integration verification includes the verification of the interaction between several FPGA blocks or the FPGA and software code. Only (expanded) functional and black-box verification are mentioned by the standard [ESC10]. In the presented methodology formal verification is applied additionally as far as possible with existing qualified tools.

3.1 Verification Methodology

The SIL 2 compliant verification methodology for the CMPU is presented in this and the following sections. A workflow will be presented that combines reviews, simulation-based verification with UVM and formal property verification.

It was chosen to do independent black-box verification. Black-box verification means that no internal design knowledge is used for verification. No internal signals are accessed. The verification code is written as if the Design Under Verification (DUV) would be a black-box from which no internal details are known [WCC09]. Independence means that the verification engineers indeed should not know any implementation details. They are not allowed to read DUV source code or detailed specifications. Independence between the design and verification engineers is not required by the IEC 61508 standard for functional safety [ESC10], but it is a commonly used best-practice. Other safety standards even require it [FV14]. Different people think differently, so they understand the requirements differently. When one person develops the design code and another one develops verification code and both of them interpreted a requirement in a different way, the verification result will deviate from the expected one. This way misinterpretations by the design engineer can be discovered [AGHH99].

Formal verification exhaustively verifies a design [SSK15]. Ideally it could be used without the need for simulation. For several reasons though it was not used as the only technique. Today's formal verification tools are often not yet powerful enough to be used for complete design verification on system level. Usually there are some limitations in terms of design complexity or needed computing power. Another point is that verification code is software and software can be faulty. Formal properties are declarative code, which is different from typical imperative code for simulation testbenches. Writing them requires a different way of thinking. Even for verification engineers who are experienced in formal verification, there is a high risk of false positives. A wrong written property might be proven, so the verification engineers would believe that a certain verification requirement has been formally proven. There could have been a mistake in the property though, that might not even be identified during code review. It could be a simple thing like omitted parenthesis, which changes the meaning of the logical expression. The wrong property might be proven and the intended one might never be checked, so a fault could escape unnoticed [SSK15]. In simulation-based verification it is easier to produce failing test results than to produce false positives, especially when a reference model is used that was written by an independent verification engineer. It is unlikely that both engineers introduce the same systematic faults into their models [AGHH99]. As mentioned, the IEC 61508 standard for functional safety classifies formal verification as a measure of low effectiveness even if done by verification engineers who are experienced in it [ESC10]. For these reasons formal verification was not chosen as the only verification technique.

Figure 3.1 provides an overview of the methodology. The green boxes depict verification activities. A more detailed description of the techniques and methodologies chosen for each of them can be found in the following sections of this chapter. A demonstration of the verification activities on a subset of the CMPU can be found in chapter 5.

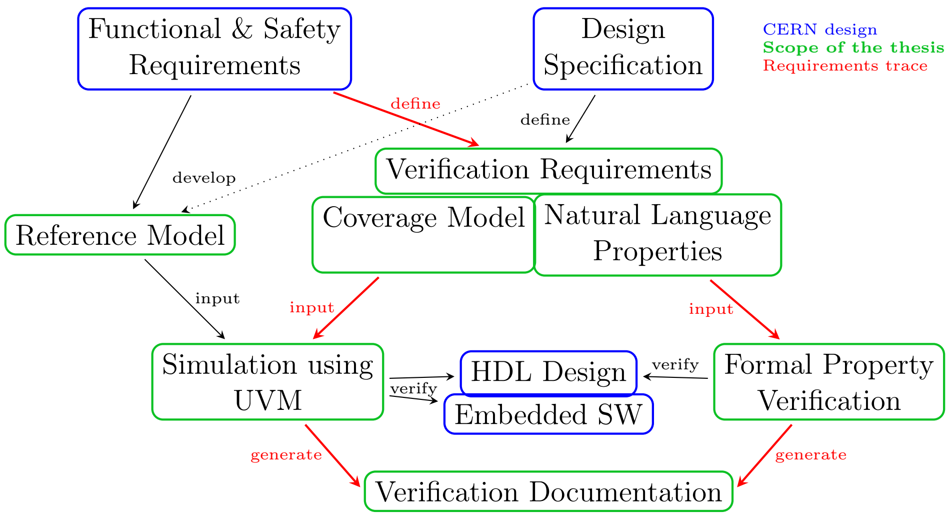

Figure 3.1: Verification Workflow

The workflow consists of these chronological steps:

1. **Review of the functional and safety requirements and design specification:** The functional and safety requirements specification builds the base for all further activities. First they are reviewed by the verification engineers and any ambiguities can be discussed with the stakeholders if they are available. Ideally the verification engineers would base their work solely on the requirements specification in order to have highest independence between the thoughts process of the design engineers and the verification engineers. Often

the requirements leave space for interpretation and design decisions. These decisions are documented in the design specification. In these cases a high-level design specification or design architecture specification can also be taken as verification input, though when independence between the verification and implementation is desired, the verification engineers should not look at detailed specifications or documentation of the implementation. In step 3 the verification requirements will be written. They will be based on the reviewed requirement and design specifications. This step is detailed in section 3.2.1 and demonstrated on the CMPU in section 5.1.1.

2. **Reference model:**
   For automated simulation-based verification (step 6) a reference model will be needed. It will be used for automated calculation of expected results when applying input stimuli to the simulated design. It can be helpful to start with the implementation of the reference model even before writing verification requirements. The development activities will deepen the verification engineer's understanding of the functional and safety requirements. Further ambiguities in the specifications might be noticed during its implementation. This step is detailed in section 3.3.1.1 and demonstrated on the CMPU in section 5.2.1.1.

3. **Definition of verification requirements:**
   The next step is to define one or more verification requirements for each functional and safety requirement. Verification requirements can also be defined for special cases that are described in a design specification. Further design usage scenarios that typically discover faults can be stated. These can be based on best practices [BB01, MBS12] and previous experience of the verification engineers. The verification requirements build the base for simulation-based verification and formal property verification. They are collected in a verification plan. This step is detailed in section 3.2.2 and demonstrated on the CMPU in section 5.1.2.

4. **Natural language properties:**
   Formal Property Verification (FPV) proves that a design satisfies specified properties. These properties will have to be written in a formal language. Formal languages do not allow space for implicit assumptions or ambiguities [CDHK15]. Taking advantage of this attribute, the verification requirements can be transformed into semi-formal statements in natural language. A table is provided in section 3.2.4 that defines a translation from English statements to SystemVerilog statements. Once they are written in SystemVerilog, they can be used as inputs to a FPV tool. The intermediate step of writing them in natural language eases communication between the involved engineers. They can be reviewed by the requirements engineers. This step is detailed in section 3.2.4 and demonstrated on the CMPU in section 5.1.4.

5. **Coverage model:**
   It can be beneficial to write the coverage model for simulation-based verification after the natural language properties have been written. If they had been reviewed by the requirements and design engineers, further ambiguities in the specifications will have been clarified before writing the coverage model. In SystemVerilog, functional coverage can be collected by "covergroups" or "cover properties". These cover properties are written in the same syntax as the properties for formal verification. When the natural language properties are already available, it might be easier in some cases to reuse parts of them for cover property statements than to specify the coverage as "covergroup". This applies particularly to temporal

relationships of signals. The coverage model can be written in an easy readable form first, eg. in a spreadsheet or it can be directly written in SystemVerilog. This step is detailed in section 3.2.3 and demonstrated on the CMPU in section 5.1.3.

6. **Formal property verification:**
   Ideally a design could be fully formally proven, so the whole simulation-based verification could be left out. Therefore it can be beneficial to start with formal verification. Whenever a requirement can be fully formally proven, it can be excluded from the coverage collection goal in simulation-based verification. Formal Property Verification (FPV) was selected as technique in this methodology. It has some advantages over other formal verification techniques. Results can be obtained comparativley fast, so the efficiency of the technique on the DUV can be evaluated quickly. Another benefit is that the formal properties can be placed into the simulation environment for verifying the formal verification code itself. This step is detailed in section 3.3.2 and demonstrated on the CMPU in section 5.2.2.

7. **Simulation-based verification using the Universal Verification Methodology (UVM):**
   The simulation-based verification environment could be developed in parallel, especially when multiple verification engineers are available. When formal proofs are inconclusive, simulation can help to understand the reason behind it. Wherever the limits of formal verification are reached, simulation-based verification can be used as alternative. Embedded software code might be verified as well together with the Hardware Description Language (HDL) design for integration testing. With SystemVerilog, embedded software functions written in C or C++ can be easily integrated into the simulation testbench. This step is detailed in section 3.3.1 and demonstrated on the CMPU in section 5.2.1.

8. **Verification Documentation:**
   All types of verification activities (reviews, FPV and simulation) will generate results in different kinds of representations. For quality and safety reasons, they have to be documented. To conform to the IEC 61508 standard for functional safety, they also have to be reproducible. That means any step to obtain them has to be documented as well. To ease tracking of verification progress and to increase reproducibility, each result should be backward traceable to a verification requirement, which should be backward traceable to a functional or safety requirement. That way it can be clearly concluded which requirements have been verified. The red arrows in figure 3.1 depict this traceability. Forward traceability is given by following the arrows in the direction as they are shown. Backward traceability is given by following them in their opposite direction. This step is detailed in section 3.4. Its demonstration on the CMPU is integrated in section 5.2.1.

3.2 Verification Planning

Verification needs planning similar to the development of an electronic system. An organisation might have a goal that it cannot achieve with available products on the market and therefore decides to develop its own system. Or it would like to develop a new product, in which case the goal could be to sell this product. Whatever the goal is, it will determine the requirements of the new system. The requirements engineers will formulate them and give them to the system architects and developers, which first design an architecture and then writes the design specification. Finally they will implement the design according to the specification. They will prioritize the functionalities to be implemented and define a schedule and responsibilities. Then they might track their progress by the number of requirements implemented.

Similarly verification needs to be planned. The goal of verification is to demonstrate that an implementation corresponds to its requirements specification and its design specification [ESC10]. Similar as for the development, this goal determines the requirements for the verification. These requirements are different from the development requirements, which logically comes from the fact that verification has a different goal than the development. Verification can even be seen as a separate development project. The verification engineers will define the verification techniques to be used. They will design the architecture for the verification software and for physical test setups if needed. Then they might write a design specification for the verification software and test setups. Finally they will implement the verification environment according to this specification. Similar like when developing the design, they will prioritize the requirements, define a schedule and responsibilities and track their progress based on the verification requirements fulfilled [BCHN05]. When verifying safety-critical devices, the verification of the most critical functions has to be given highest priority [ESC10]. In the case of verification, a requirement is fulfilled when it is implemented in the verification environment and when the expected results for that requirement are obtained.

Verification starts with defining a verification strategy. Verification methodologies, tools, resources in terms of people, devices and times have to be chosen and goals have to be specified. These steps can be written into a verification plan. The main part of a verification plan are verification requirements [BCHN05]. They are mainly based on the functional and safety requirements, so it is beneficial to review these requirements first and clarify misunderstandings with the involved engineers. In this methodology, simulation-based verification will be done with constrained random input stimuli. The verification plan will not contain a description of inputs to be applied, but of the inputs and outputs that shall be observed. They are specified in the coverage model. The properties for FPV will first be stated in natural language first and later translated into a formal language.

Figure 3.2 shows how forward traceability from requirements to test results, as well as backward traceability in the opposite direction, is established. For the CMPU, the verification documentation has been created in spreadsheets. The mapping for traceability was added to each entry, as depicted by the red arrows. Verification requirements are based on functional or safety requirements, the design specification, or specified additionally, coming from unspecified design properties. Each of them will be either verified in simulation-based verification or formal verification. For the former an entry in the coverage model is created, for the latter a natural language property is written. The coverage model entry can be directly related to a test result, stating the test case that covered it. The natural language property is translated into a formal property, which will generate a proof result. The mapping can also be tracked in a traceability matrix or in a requirement management tool. Traceability is useful for quality assurance and process tracking. It can also be beneficial for safety certification to demonstrate the completeness of the verification [Pit18].

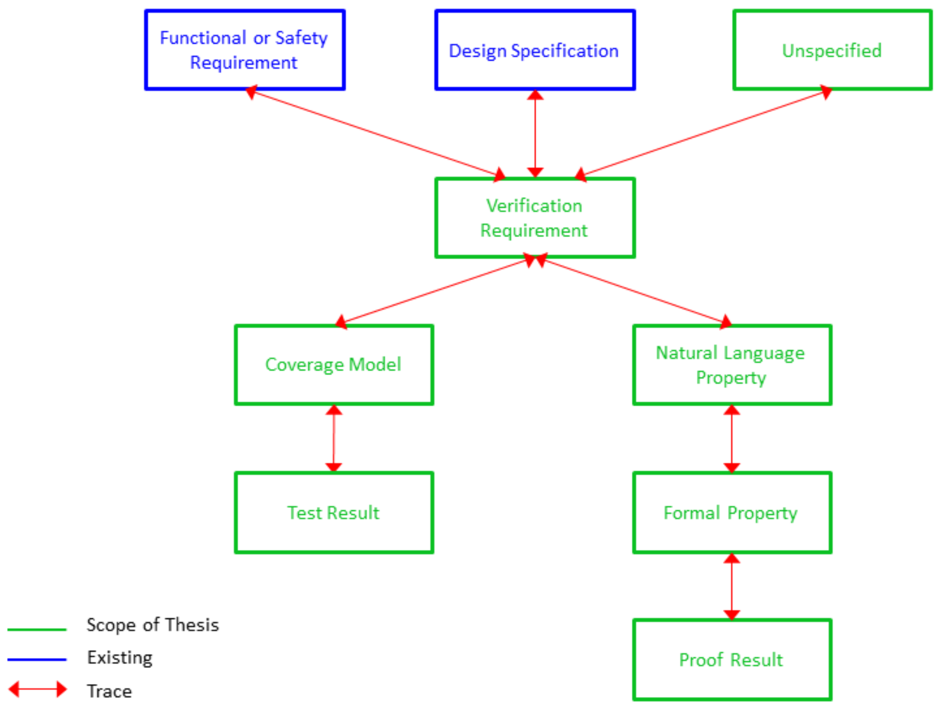

Figure 3.2: Requirements Traceability

3.2.1 Review of Requirements and Design Specifications

Verification engineers need to know the expected behaviour of the DUV. When doing independent black-box verification this knowledge ideally comes from the requirements specification only [FV14]. The designers can document their implementation decisions in the design specification. Unless the requirements specification is extremely detailed, there will be some design decisions that need to be known by the verification engineers in order to perform their tasks [FV14]. These decisions could be written in a separate document to avoid that the verification engineers get more knowledge about the design internals than necessary, thus increasing independence. These specifications build the base for the verification activities. A systematic fault in a specification can have fatal implications. It can turn into a failure once the safety-critical device is in operation. Such a fault could be inside an algorithm or a logical or mathematical formula. These kind of faults cannot be found by automatic verification software that was written to find deviations of the implementation from the specification. Therefore it is essential that the specifications are critically reviewed before deriving the verification requirements.

If it is in the scope of expertise of the verification engineers, they can review the requirements and design specifications [FV14]. Otherwise a separate review has to be performed by a domain expert before starting verification activities. For safety-critical designs, the review should include the consideration of typical design measures that can reduce risks. The person who conducts the review should have knowledge in these. In the IEC 61508 standard this activity is mentioned as safety validation of the requirements. Another purpose of reviews is to detect ambiguous requirements or implicit assumptions that were taken by the requirements engineer. Furthermore they should discover contradictions between the requirements and design specifications [ESC10]. To keep independence high, detailed design specifications should be reviewed by persons who do not perform automated verification of this design. In the process of developing safety-critical devices these reviews might be directly used as verification of a lifecycle subphase, for example for the "system design requirements verification" [ESC10].

The review findings will have to be discussed with the responsible persons and the requirements specifications might have to be updated, which could make changes in the implementation necessary. The benefit of performing these reviews before writing any verification software code is that the effort of modifying verification code is avoided. The verification software can be implemented based on the updated specifications.

3.2.2 Verification Plan

A verification plan that conforms to the IEC 61508 standard for functional safety contains a description of the verification strategy, the tools to be used and the steps to be performed to obtain the verification results [ESC10]. Apart from this activity description, a verification plan contains verification requirements [Ber03]. For each functional requirement and each statement of the implementation specification one or more verification requirements need to be defined. Additional verification requirements might be defined that do not directly result from the design requirements. They might regard typically error prone design specifics, like eg. unusual combinations of use cases [TK01] or boundary value tests of input parameters [MBS12]. A design corresponds to its specification when it completely implements all functionalities described in its specifications and when it does not contain any other unspecified functionalities. That means the absence of unwanted design behaviour needs to be verified as well [BCHN05]. The latter is especially important for safety-critical designs where unspecified behaviour could be life-threatening [ESC10]. In section 2.1 safety validation was described as well. Validation according to IEC 61508 means to verify that the design corresponds to the safety requirements. It is covered by the verification plan if verification requirements for the safety requirements are included.

The verification requirements could specify what should be tested, for example an interesting value combination. The verification requirement does not need to state how this shall be tested. It should not depend on the technique that will be used to fulfil the verification requirement because it gives more flexibility if the technique is interchangeable. This way one can purely focus on the function that should be verified when writing the verification requirement. The writing is not influenced by any restrictions that might exist in a used technique or programming language.

It can be helpful to specify an objective or intent together with each verification requirement. For complex requirements it will help to remember later why a scenario was chosen to be interesting. And even more important, it can support the reviewing activities. When reviewing the verification requirements, the review engineers will hopefully often agree with the verification engineers on the importance of a requirement and approve it. The review engineers might though have an other reason in mind for considering the scenario as relevant. They could think that it covers a certain case, but the design engineers actually thought of some other case. So it could happen that the verification requirement is wrong when considering the case that the verification engineers thought of, but because it covers another interesting case, this mistake could pass the review unnoticed. The other interesting case thought of by the reviewers might be implicitly covered. If the verification requirement gets removed with a later change of requirements or specification, the implicitly covered case would not be covered anymore. A new review might reveal this situation, or it might not. When an intent or objective is stated together with the requirement, it is clear to the reviewers why the verification engineers wrote it. The reviewers can therefore more easily see a mismatch between the requirement and its objective. Furthermore it can inspire them to think of further scenarios and they can clearly see if these are missing in the verification requirements.

The verification plan for the CMPU was written in a spread sheet. It contains one line per verification requirements. The fields per verification requirement are explained in table 3.2. Examples are provided in section 5.1.2.

Table 3.2: Explanation of the fields in the verification plan per verification requirement.

Field in Verification Plan	**Explanation**
Requirement Number	One of: • functional or safety requirement number • input parameter ID • additional requirement statement by requirements engineer, added after requirements review • design specification statement number
Requirement statement	A copy of the requirement or specification statement or the name of the parameter that is verified by this verification requirement.
Verification requirement number	A combination of the requirement number and a consecutive numbering of verification requirements per requirement statement.
Verification requirement	A description of a input condition or scenario that should be verified.
Objective	The intent of the verification requirement.
Kind	One of the coverage types listed in table 5.2.
Safety ranking	Number 1 means highest priority regarding safety.
Coverage model	Either a textual description of the functional coverage model for simulation-based verification or the name of the SystemVerilog covergroup or cover property that covers this verification requirement.
Property in natural language	A temporal logic property that has to hold in order to verify the corresponding requirement. The property is expressed in natural language as described in section 3.2.4.
Property	Either the property in SystemVerilog or the name of the corresponding property in the formal verification suite.
Verification method	States whether a requirement is expected to be verified with formal or simulation-based verification.
Test name	UVM test case name in which the requirement is expected to be verified, in case "Verification method" was simulation.
Criteria to pass	Criteria that needs to be satisfied in order to verify this verification requirement. For simulation-based verification the criteria is that the verification requirement is covered and there were no mismatches between the reference model and the implementation. For formal verification the criteria is that the property was proven. The criteria depends on the "Verification method".
Comment	Comments, if necessary.

3.2.3 Coverage Model

When using random input stimuli for simulation-based verification, there is no need for specifying input stimuli sequences in the verification plan. What has to be stated is a measure to determine when verification activities can be stopped. The verification goal is described as a coverage model. Random simulation is performed until this goal is reached. For safety-critical systems 100% functional coverage has to be reached. The coverage model needs to describe all safety-critical design configurations and usage scenarios. The DUV of this thesis, the CMPU, implements all safety critical functions inside the FPGA. Apart from a few exceptions like the provision of the compile date of the VHDL code, each and every calculation performed inside the FPGA generates results which can be used for safety-critical decisions. Therefore there was no need to distinguish between safety-critical and non-critical functionality.

The CMPU is a configurable measurement device. It contains signal processing algorithms that calculate safety-critical output values from real time measurements as well as configuration values that are modifiable at runtime. All possible configurations and input values that can influence safety-critical functions need to be considered. This creates a huge input space that influences safety-critical functions. As mentioned in section 1.1 it is not possible to verify all input value combinations if there are several parameters with bit widths of 64 bits. The coverage model has to state a selection of a representative subset including some values or value combinations of special criticality.

SystemVerilog was chosen for writing the simulation-based verification software. One reason for this choice were its inbuilt functionalities for coverage collection. There are two ways to measure coverage. It can be done concurrently by cover properties, which are stated in temporal logic. These properties can also be used as assertions. It can also be done by covergroups, which sample specified signals whenever its sample function is called. They contain coverpoints, which contain so called "bins", which specify values or value ranges to be observed. Coverage statements can observe input or output signals, internal DUV signals and testbench internal variables. Independent black-box verification may not state coverage in terms of internal DUV signals [WCC09]. The conditions to be observed can be expressed in many ways, eg. as values, value ranges, value transitions or expression outcomes. Conditions under which the values are valid for coverage collection can be specified as well. It is important that measured coverage is only considered when tests pass. It also has to be ensured that a input value is only covered if it had an effect [BCHN05].

The coverage model for the CMPU has been grouped into the types listed below. Suitable SystemVerilog language constructs for coverage are mentioned. For similar designs the same types might be reused partly or completely. The goal was to keep the grouping Mutually Exclusive and Collectively Exhaustive (MECE). Mutually exclusiveness avoids unnecessary redundancies, which increases simulation performance. Collectively exhaustiveness means that within a group no value or value combination of interest is omitted. The grouping helps the verification engineers to identify incomplete collections [SMG15].

Cover types determined for the verification of the CMPU:

1. Use case:
   This group collects all positive stated verification requirements. It describes the expected design behaviour for all expected usage scenarios and configurations. This group ensures

that all functionality is implemented as specified. They can be stated as cover properties or covergroups. Input and output values can be specified, as well as conditions under which the values need to be observed.

2. Negated requirement:
   This group collects the negated versions of the use cases, where relevant and interesting. It describes scenarios that must not happen. It is not enough to observe the absence of such a scenario at one time only. Rather it has to be shown that it never occurs during the whole simulation. Formal proofs are a better way to verify these verification requirements. The same assertion statements can be used in simulation. Additional cover statement might be necessary to make sure the assertions do not hold vacuously only, which would be the case if the antecedent of an implication never occurs. The additional cover statement would need to ensure that relevant stimuli are generated and the assertion statement makes sure that the negated requirement is never violated.

3. Value range:
   This group contains at least one SystemVerilog coverpoint for each input parameter. The values can be boundary values of the possible value ranges, extreme values of mathematical functions, algorithm-dependent values that might lead to failures in mathematical functions, like eg. a division by zero or any other values that are interesting for verification. At least one bin that covers the value ranges between each individually stated value should be added. To increase chances of finding corner case faults, additional value ranges close to the individually stated ones could be added.

4. Interesting scenario:
   This group contains unusual but perfectly legal device configurations. It is closely related to the last two groups. It describes relationships between input signals. It covers special cases that are not specified in the requirements, but that need to be considered. Often they result from design decisions or implicit assumptions made by the requirements engineer. They can be specified in the same way as use cases.

5. Temporal relation:
   This group is similar to interesting scenarios, but with a focus on temporal relation of value changes. Since the SystemVerilog property syntax contains temporal logic operators, they are the preferred choice for covering this group. These properties can be stated as concurrent coverage properties. They need to be observed one time only, in contrast to the assertions of point 2 that need to hold always.

6. Stress test:
   This group covers the combination of boundary values of input signals. They can be stated as cross coverage of SystemVerilog coverpoints that were created for value range coverage. Conditional sampling of covergroups can be used as well.

3.2.4 Natural Language Properties

Each functional and safety requirement was described by one or more semi-formal temporal logic properties, written in natural English language. The intention of writing properties in natural language was to reduce the risk of introducing faults into the verification code that arise from different interpretations of the requirements by the different involved engineers. Natural English language can be easier understood by a requirements engineer not familiar with SystemVerilog.

The Personnel Safety and Machine Protection group at CERN also recognized the need for formalized specifications to avoid ambiguities and misunderstandings. They used the approach of stating the safety critical functions as boolean logic expressions of function inputs together with a natural language description of the used fields and the purpose of the function. They used these expressions to automatically generate a minimum set of test vectors that verifies all possible input combinations that trigger a safety function. These vectors were generated with MATLAB. Their DUV was a Programmable Logic Controller (PLC) system [VHH+14]. This method is not applicable for the verification of the CMPU, because the inputs to the safety-critical functions are the parameters with its bit-widths of up to 64 bits. As mentioned earlier, the input space is far too large to be simulated exhaustively.

Nevertheless, a semi-formal description could be used as input to formal property verification. Table 3.3 lists a mapping of natural language expressions to SystemVerilog Assertions (SVA) expressions. Some property statements are mapped to multiple natural language statements, because the requirement would otherwise not be easily understandable or sound awkward in English language. Expressions (Expr), Properties (Prop), and Sequences (Seq) are the SystemVerilog meaning of expression, property and sequence as described in [ICS18]. A SystemVerilog sequence can be a sequence spanning multiple clock cycles or a property. Properties can be eg. implications or a single boolean expression. When a requirement is expressed as a natural language property, the Expr, Prop and Seq constructs might still need some translation into SVA language. The wording should be used consistently, then it can be mapped in a design specific table like in the examples in table 5.3. When the mapping is strictly obeyed, automated scripts could be used for the translation.

Table 3.3: Mapping of natural language constructs to SVA constructs.

Natural Language	**SystemVerilog**	**Explanation**
Expr1 equals Expr2	Expr == Expr	An Expression (Expr1) equals another Expression (Expr2).
Expr1 does not equal Expr2	Expr != Expr	An Expression (Expr1) does not equal another Expression (Expr2).
Expr1 equals or is greater than Expr2	Expr >= Expr	An Expression (Expr1) equals or is greater than another Expression (Expr2).
Expr1 equals or is less than Expr2	Expr <= Expr	An Expression (Expr1) equals or is less than another Expression (Expr2).
Expr1 and Expr2	Expr && Expr	Logical and of two expressions (Expr1, Expr2).
Expr1 or Expr2	Expr || Expr	Logical or of two expressions (Expr1, Expr2).
(Expr)	(Expr)	The parenthesis emphasize that an expression (Expr) consisting of several parts belongs together.
Prop implies that: Seq	Prop |-> Seq	Property (Prop) implies that in the same clock cycle the sequence (Seq) happens. "implies:" is placed in a new line to indicate that the whole statement above implies the one that follows.
Prop implies that: Seq in TimeSeq	Prop |-> ##TimeSeq Seq	Property (Prop) implies that the sequence (Seq) happens after some cycles or condition indicated by TimeSeq[1].
Every time when Expr: Seq	Expr |-> Seq	Another way of stating that when an Expression (Expr) is true, a sequence (Seq) should happen.
At Expr: Seq	Expr |-> Seq	Another way of stating that when an Expression (Expr) is true, a sequence (Seq) should happen.
Sig does not change	$stable(Sig)	Signal (Sig) is stable, so it does not change its value.
TimeSeq after Expr: Seq	Expr ##TimeSeq Seq	Expression (Expr) is followed by sequence (Seq) after some cycles or condition indicated by TimeSeq[1].
in TimeSeq Seq	##TimeSeq Seq	After some cycles or condition indicated by TimeSeq[1] the sequence (Seq) happens.
Prop1 until Prop2	Prop1 s_until Prop2	Property (Prop1) holds until another Property (Prop2) holds. They don't hold at the same time.
Prop1 as long as not Prop2	Prop1 until Prop2	Property (Prop1) holds until another Property (Prop2) holds. The whole statement is also true if Prop2 never holds.

[1]TimeSeq could be a number or range alone or it could also include a sequence or condition that should happen before sequence (Seq) happens.
E.g. TimeSeq = ##[0:2] stating that Seq should happen 0 to 2 clock cycles after Expr
E.g. TimeSeq = ##1 input1xDI == 0 ##[2:$] $rose(input1xDI) ##1 stating that after Expr one clock cycle later input1xDI should be 0, then 2 to infinite clock cycles later input1xDI should have risen from 0 to 1 and 1 clock cycle later Seq should happen.

3.3 Verification Software Framework

Automating verification activities increases productivity and facilitates reverification [BCHN05]. When developing electronics conforming to IEC 61508, it has to be possible to regenerate any verification evidence. It is not enough to keep log reports, actually it should not even be necessary to keep them. Instead it should be possible to (re-)create verification results for any desired configuration of design and verification environment [ESC10]. With increasing design complexity manual recreation can soon become very time consuming. Even the manual writing of automated test cases with self-checking capabilities can be a lot of effort. Constrained random simulation reduces this effort and at the same time automatically adds much more input stimuli than one would ever specify in manual test cases [BCHN05]. Formal verification removes the need for input stimuli generation at all [SSK15].

Many verification activities can be automated to save verification software development time and verification execution time [BCHN05]. The developed verification software consists of several scripts and programs written in multiple programming and verification languages. The language choice for simulation-based and formal verification code for the CMPU was SystemVerilog. The reference model was chosen to be implemented in C. Makefiles, Tcl/Tk scripts and shell scripts enabled further automation.

3.3.1 Simulation-based Verification Using the Universal Verification Methodology

Constrained random testing together with functional coverage collection was chosen for simulation-based verification. Applying random input stimuli facilitates the simulation of more verification scenarios with less test development effort [BCHN05]. Furthermore it generates input stimuli that are randomly selected from the whole available input space. This technique increases chances that hidden faults are discovered, because DUV usage scenarios are verified that might not seem relevant to a verification engineer, but which can actually trigger a failure [BCHN05]. Constraints can be used to modify the distribution of input stimuli. They can be used to test the DUV for robustness by guiding the random number generator to generate more values around boundary values or other critical values. They can also be used to narrow the range of generated values to faster obtain DUV usage scenarios described by the coverage model, without actually directly specifying the necessary input values. This way more scenarios are verified than specified in the coverage model, which increases chances of finding systematic faults.

The Universal Verification Methodology (UVM) builds on these techniques. It provides an open-source software framework for SystemVerilog testbenches [Ini14]. SystemVerilog itself has vast support for constrained random stimuli generation and coverage collection [ICS18]. UVM enhances it by adding practical functions that are typically used in testbenches, for example for result reporting and graceful test termination. It extensively uses the object oriented capabilities of SystemVerilog and adds a layer of test phases and transaction level modelling on top of it. UVM was created to increase reusability within the testbench and between testbenches [MR10]. The capabilities for reuse provided by UVM and SystemVerilog will be very beneficial for verifying the CROME Measuring and Processing Unit (CMPU). The CMPU contains several algorithms and design blocks with resembling properties and interfaces.

At CERN, the Universal Verification Methodology (UVM) is used by a few other groups beside the Radiation Protection group [Fie17, MCP^{+}15]. They use it for verifying pixel detection chips. Those are very different from the CMPU. These chips are part of the particle detectors at the collision points of the Large Hadron Collider's (LHC's) particle beams. They detect the location and time of particle hits in order to gain knowledge about the kind of particle that got produced. These chips are not safety-relevant, they are part of the physical experiments. The correct functioning of these chips has higher priority than verifying unusual configurations or other corner case scenarios. They use UVM to simulate different kind of expected particle flights with some random and some fixed properties. They simulate for example a consecutive number of particle hits in a straight line, randomizing the position of that line. These groups do white-box verification. Often they use internal signals to force the DUV into a specific state. These detector chips do not depend on values calculated in previous internal states. For a certain input, a defined output is expected [MCP^{+}15]. The CMPU on the other hand produces a certain output depending on its current and previous inputs as well as its previous outputs (or internal variables). The implication for simulation is that the detector chips can be simulated with many short tests, whereas for the CMPU it would be desirable to run one long test simulating continuous operation. An other difference is that these verification engineers are measuring their verification progress in terms of executed test cases. For the CMPU it was decided that the input values should be as random as possible. That means constraints should only be used where necessary to make the DUV work at all. Therefore verification progress is measured with functional coverage collection.

UVM was also chosen because of the advantages of SystemVerilog. The coverage collection options described in section 3.2.3 are language features of SystemVerilog. They can be smoothly integrated into testbenches and the testbenches are portable between different simulators. SystemVerilog interfaces are not only connections to the DUV, they can also contain internal variables, processes, and concurrent assertions and cover properties. The properties used for formal verification can be placed into the interface to the DUV for verifying the properties themselves. UVM provides base classes for many testbench components. Customized components can be created by deriving from these base classes. Reuseability can be increased by creating customized base classes that derive from the UVM classes. These base classes can contain common functionality that is not interface specific, if possible not even DUV specific. That could be the managing of log files in a test case or a reset sequence that applies to the whole DUV.

Overview of a UVM Testbench

The architecture of the testbench for verifying one block of the CMPU is shown in figure 3.3. Modules that are based on UVM components are described with the prefix "UVM". For better clarity the language of a module or class is sometimes written in parenthesis. SV stands for SystemVerilog. This architecture is representative for verifying other blocks of the CMPU or similar designs. Several blocks can be verified by instantiating multiple UVM environments inside a UVM test. At least one UVM environment has to be created for each interface of a DUV. On block level, one UVM environment is necessary for each block interface. The testbench architecture will be explained in detail in section 5.2.1.

A UVM testbench contains a stimulus generation side that applies input signals to the DUV. It consists of a UVM test case, UVM sequences, a UVM seqeuncer and a UVM driver. A UVM test

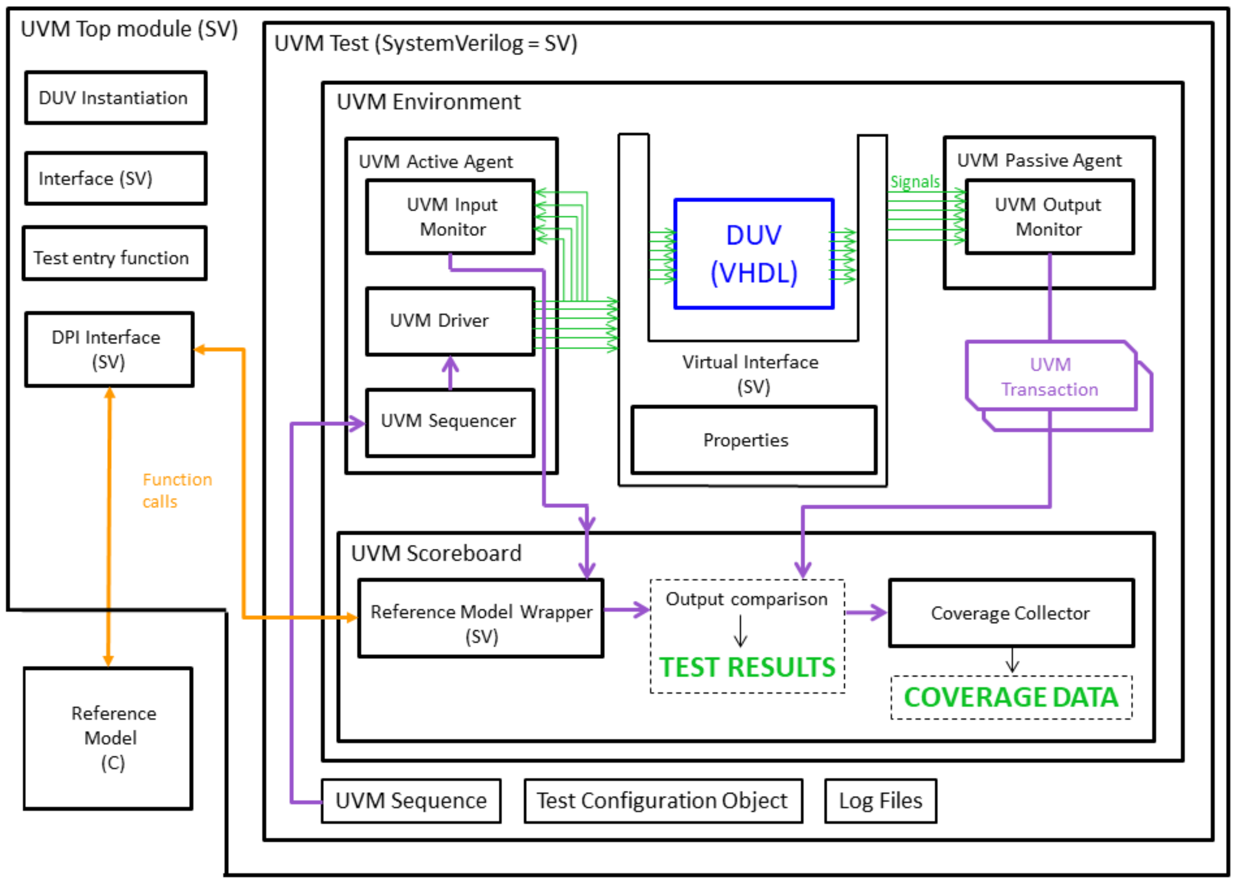

Figure 3.3: UVM Testbench Architecture for the CMPU

case can create one or more UVM sequences, which it executes on a UVM sequencer. A UVM sequence can create one or more transactions (called UVM sequence items). It modifies and sends these transactions to the UVM sequencer, which passes them on to the UVM driver. The UVM sequencer can use different arbitration algorithms for sending these transactions. A UVM monitor is used to monitor the signals that are received by the DUV [Ini14]. It sends this information to the reference model. It could also write it into a log file. On the output side of the DUV, a UVM monitor reads the output signals and passes them to the UVM scoreboard. The UVM scoreboard also receives the output values calculated by the reference model and compares the them to the DUV outputs. If all of them match, the test passes. A coverage collector can receive data from several places, for example from an input or output UVM monitor, from the reference model or the UVM scoreboard.

Transaction Level Modelling

Transaction Level Modelling 1 (TLM1) was used inside the testbench for the CMPU. A UVM transaction groups one or more DUV signals together. It can also contain additional information. Inside a UVM testbench, transactions can be transferred trough ports. Multiple objects can be connected through TLM1 ports. For example a UVM monitor can contain a port object through which it sends a transaction to several other testbench components, for example the scoreboard and the coverage collector. A UVM testbench traverses different phases. A build phase, where objects are constructed. A connect phase, where port connections are made. That means a reference to the receiver objects is passed to the sender object. Later, in the run phase, a testbench component can use a port's "write" function in order to send a transaction to the receiver. Internally this calls a "write" function implemented by the receiving object. This is where data can be received for further processing [Ini14].

When communicating with the reference model through the DPI, the communication happens through function calls. The reference model was implemented on a higher level of abstraction than the DUV. Most transaction variables have a corresponding field in the block-level C structures. Communication happens via the above mentioned getter and setter functions.

When communicating with the DUV, the signals are extracted from the transaction (in a driver) or a transaction is created from the signals (in a monitor). The testbench communicates with the DUV through a SystemVerilog interface. A practical advantage of these interfaces is that they can store signals that are present in transactions, but that are no port signals of the DUV. That way additional testbench internal information can be passed from the input side of the testbench to the output side without further need for synchronization. That information could be for example the length of a delay that the driver has to wait between applying two input signals of the same transaction. It can be useful to store this information in the interface while the DUV is performing its operation. A monitor should monitor the interface signals and not the transactions, because it should monitor the data that is actually sent to the DUV by the driver. The driver could be faulty, so a transaction might not reach the DUV as intended. By storing transaction related information inside the interface, the input monitor component can reconstruct the original transaction that should have been sent to the DUV. It could compare the two transaction at run time, if that level of assurance is needed. Otherwise the additional information is mainly important for debugging test results, which usually forms a great part of the verification activities [Fos17].

3.3.1.1 Reference Model

The test results are obtained by regularly comparing the output values of the DUV with the output of a reference model. For a DUV like the CMPU, an untimed model is sufficient, even though the CMPU is synchronized to the real time. It regularly performs computations in a defined order. While the DUV generates this regular cycle based on its clock, in a more abstract C model it can simply be modelled by a loop and a loop counter. More notion of timing in the model would only add unnecessary detail that would slow down its execution. On block-level even this notion of time is not used. The C model only knows its internal state represented as variables in a structure. The untimed reference model could have been written in any high level language. The decision for C was based on several points. One reason was that C functions can be easily called from SystemVerilog by using the SystemVerilog Direct Programming Interface (DPI). This is very useful for performing block level tests and directly comparing results with the DUT output from within the UVM testbench. It is possible to interface SystemVerilog with C++, but there was no real benefit in doing so. The C++ code would have to be compiled with the 'extern "C"' linker directive. This directive can only be added to functions that are not class members. For class members a wrapper would have to be written. This would have been an unnecessary extra step for interfacing with SystemVerilog. The reference model could be written in plain C without many drawbacks of not having C++'s object oriented functionalities. One advantage of C++ would have been the possibility of operator and function overloading. Since the comparison operator for alarm thresholds can be configured, operator overloading would have been beneficial. Further it would be useful in future when doing fixed and floating point calculation comparisons. An advantage of C is the possibility to call any functions in any order from SystemVerilog. This could be seen as a disadvantage, because the software engineering principle of separation of concern can be easily violated [Vli18]. This possibility is very useful for block level verification. It could also be used to test the testbench by injecting faults into the reference model by calling some subfunction at a "wrong" point in simulation time. Then one can verify whether the testbench reports mismatches as expected.

The reference model for the CMPU was designed in a modular way. For each DUV block that needed to be modelled, one C module and one C structure declaration was created. Parts of, or the complete FPGA design can be modelled by combining several C modules. DUV operation is modelled through function calls. Input and output signals are encapsulated in the block structure. Block internal variables are stored in the structure as well, which means that the module has no state. All state information is stored in the block structure. The state is updated by calling a module function and passing a pointer to the structure as parameter. This architecture allows to model multiple instantiations of the same block. The modular design also increases maintainability and extendibility of the C model.

The SystemVerilog testbench communicates with the C model through the SystemVerilog DPI. C functions that shall be called from within SystemVerilog code, need to be imported by a DPI import statement. Similarly a SystemVerilog function that needs to be called from within C code, needs to be exported. The import and export statements can be placed anywhere outside a class. A SystemVerilog interface can be used to encapsulate all of these statements. C function calls from SystemVerilog are then of the form "interfaceName.functionName();". This way it can be clearly seen when a function call accesses a DPI imported function, which increases maintainability of the code. SystemVerilog datatypes have to be used in the function import statements. One needs to carefully consider the datatype mapping imposed by the SystemVerilog Language Reference Manual (LRM) [ICS18]. Simple data types like integers and even strings can be passed easily. Arrays and 4-state logic signals require the use of specialised data types on the C side.

It is possible to use pointer parameters in the C functions. C pointers can be stored in the SystemVerilog testbench with a special datatype, called "chandle". Unfortunately the pointers cannot be dereferenced from within SystemVerilog. The approach chosen was to store one chandle variable for each modelled block inside the testbench. At the beginning of a test run, the testbench calls a C function that dynamically allocates the memory for the block-level structure and returns a pointer to it. A second C function that frees the memory is called at the end of a test run. The pointer to the structure is passed to C module functions that operate on the field values and calculate expected results. The SystemVerilog testbench cannot directly modify field values, which it has to do in order to pass input values to the C functions. It also cannot read the results from the structure. The approach chosen was to create getter and setter functions that retrieve or modify the values of the fields of a structure. Each time a DUV operation is modelled, first the setter functions are used to pass input values to the C module. Then the operation functions are called. Afterwards the results are requested by the testbench through the getter functions.

Using the DPI interface requires a large amount of repetitive code, especially with the approach of getter and setter functions. There is great potential for automation. Python scripts can serve that purpose. The Matlab HDL Verifier [11] could also be used for automatic generation of the DPI interface. It flattens structures into single function parameters. The usage of getter and setter functions can be omitted.

It was chosen to compare the output values of the DUV and the reference model at runtime, inside the simulation testbench. Another option would have been to write the output values into a log file and use post processing scripts to compare them. When the reference model can be accessed through DPI function calls, the first option is advantageous. Output values of the reference model can be easily obtained through some function calls. The test result can then be used at runtime to control coverage. Coverage data is only collected if the output values of both models match. This restriction mainly helps during the debug and testbench development phase. It should not be necessary, because verification activities cannot stop anyway as long as there are failing test cases.

3.3.1.2 Constrained Random Inputs

When using completely random input values, verification efforts might not be efficient. Input values that are not possible or allowed during real operation might get generated and cause test failures. It might also be more difficult to achieve full functional coverage within reasonable simulation time. The probability of the random generation of certain points of interest that are stated in the coverage model might be too low. Random input value generation can be constrained in several ways in order to avoid these problems [BCHN05].

A transaction class (UVM sequence item) abstracts the DUV interface into a transaction. It contains variables that either directly store the data sent to or received from the DUV or it groups several signals together into one or more variables. Variables that represent DUV input values can be declared with the SystemVerilog keyword "rand", so that their values can be selected randomly. Constraints that shall be applied to every transaction of a kind can be placed into the transaction class. Inside a transaction SystemVerilog data types can be used, which eases signed operations. If the DUV uses bit widths that do not correspond to the available SystemVerilog data types, constraints can be used to restrict the input values to the possible ranges. Like mentioned in section 2, for stress testing of safety-critical designs constraints should not restrict the value ranges to the allowed ones, but to the possible ones. Further constraints might be used to modify the distribution of the randomly generated values.

In [SM14] is explained how SystemVerilog constraints can be used to create a bathtub like input value distribution. A modified version has been defined for verifying the CMPU. It is demonstrated in listing 3.4. For stress testing a higher weight can be given to certain values or value ranges. In the example the higher weight is given to the most negative value, 0 and the most positive value. The weight values are randomized as well. Lines 2-4 make sure that the value selected for highWeight is higher than the one for lowWeight. Lines 6 - 10 show how the value distribution is applied to a random variable called input1xDI. Three individual values are assigned a high weight. The value ranges between these values are assigned a low weight.

Another concept that has been used, is shown in listing 3.4 as well. The value range of input1xDI is configurable through the variable globalUsedMaxInput1. This is a global variable declared in a global package. It can be modified anywhere in the testbench. It is used to configure

```
constraint weightDistribution {
  highWeight > 0;
  lowWeight  > 0;
  highWeight > lowWeight;

  input1xDI dist {(0 - globalUsedMaxInput1)       :/ highWeight,
                  [(0 - globalUsedMaxInput1): -1] :/ lowWeight,
                  0                               :/ highWeight,
                  [1: globalUsedMaxInput1 - 1]    :/ lowWeight,
                  (globalUsedMaxInput1 - 1)       :/ highWeight};
}
```

Listing 3.4: Constraints for weight distribution.

- input constraints in transactions
- coverpoint bins
- default constraints in sequences

Value ranges used for constraints and coverpoint bins have to correspond to the DUV datatype, when it shall be possible to reach 100% of coverage. If constraints are used to restrict some input values to a certain range, eg. -10000 to +10000, but the coverpoint bin keeps the original range of -2^{63} to (+2^{63}-1) of a signed 64-bit field, 100% coverage can never be reached. Therefore a global variable, globalUsedMaxInput1 in the example, is used to configure all value ranges that regard the same DUV field. This technique might be applied if the main focus of a test case are not the value ranges, but some other DUV properties. It might still be of interest to test that other properties together with a range of values of another signal instead of some fixed ones. Furthermore it can help debugging the testbench. The first goal could be to reach 100% coverage with positive test results for reduced value ranges, before scheduling a long test run for the full ranges that might need several days to complete.

Constraints cannot be parametrized, but they can use variables that are modifiable at runtime. Global variables can be modified anywhere in the testbench without control. A cleaner way for configuring constraints would be to use class member variables and modify them through function calls. This approach was not chosen for these value ranges, because they are not intended to be configured multiple times at runtime. It is not a constant value, because it shall be configurable once at runtime by the UVM test case. This allows the creation of different UVM test cases that verify different aspects of the DUV. The reason why it shall be modified only once are some limitations of SystemVerilog covergroups. Once created, they cannot be modified anymore. This is explained in more detail in section 3.3.1.3.

Input constraints that are test case or UVM sequence specific can be added in two ways. A derived transaction type could be created, that adds additional constraints to the base transaction. The UVM factory pattern is useful to override class types at runtime. This allows to exchange a base transaction with a more specialized version without the need for modifying the UVM sequence code. Similarly a whole UVM sequence can be replaced in a UVM test case.

Another option is to use the randomize function within a UVM sequence. Each SystemVerilog class has a randomize function that assigns new random values to all variables declared as "rand" or "randc" [ICS18]. It can be used together with the "with" keyword, as seen in listing 3.5. Constraints can be specified in the parenthesis that follow it. These constraints are combined with the ones specified in the transaction class. The simulator's constraint solver determines the values from which the random number generator can choose [Cor18c]. If constraints are contradicting each other, the randomize function call will fail. The listing shows how a UVM report macro (‘uvm_fatal) is used in that case to terminate the simulation. An example for a invalid constraint is: input2xDI $>$ 0; input2xDI $<$ 0;. It is invalid, because the value of input2xDI cannot be greater than and less than 0 at the same time.

In listing 3.5 only one field of the transaction inTransaction is randomized: field input2xDI. It is specified in the parameter list of the randomize function call (line 1). It is possible to specify several transaction fields. If the list is empty, new values are generated for all random variables of inTransaction. The constraint statements in the "with" section (line 2) can also include other fields of the transaction, that are not randomized in this call. They can also use any other variable that is within the scope of the function call. For example sInputMax could be a class variable of the UVM sequence.

```
1  if (!inTransaction.randomize(input2xDI)
2    with {`constrainTransaction input2xDI < sInputMax;}) begin
3      `uvm_fatal(get_type_name(), {"Failed to randomize input2xDI!"})
4  end
```

Listing 3.5: Constrained call to randomize function.

In line 2 another concept is shown that has been used to facilitate constraints modification and reusability of the sequence classes. When the whole randomize call and its surroundings is repeated several times in a sequence, it would be desirable to place it into a function and use parameters to specify the transaction fields. Unfortunately this is not possible. The transaction field names in the randomize function parameters and constraints are not variables, they are variable identifiers. They need to be statically stated before compile time. SystemVerilog preprocessor macros can be used to increase flexibility. In listing 3.5 the macro‘constrainTransaction was placed into the constraint section. This macro might be empty or contain additional constraints. That way it is possible to quickly modify the constraints of a whole sequence, which can be very useful for debugging. The above mentioned default constraints for sequences can be specified in this macro in a base sequence class. They determine the value ranges that shall be tested with a sequence. A specialised version of the base class might redefine this macro. Additional constraints, like eg. field relationships, can be added. This technique has several benefits:

- Constraints that shall be used within the whole sequence are specified in one place only. They don't have to be repeated in each call to randomize.
- It allows the modification of the constraints only, while the rest of the randomize call (parameters, error handling) needs to be written only once.
- Any constraint statements using any allowed operators, keywords and transaction fields can be added through the macro.

Of course macros have their draw backs as well. They cannot be simply overwritten by derived classes, like functions can be. They need to be undefined first, and redefined again. If a derived class does neither undefine, nor define the macro, the content will not be the one stated by the parent class, but by the class which was compiled before the mentioned derived class. This has to be kept in mind when using macros to modify constraints. Another drawback is not relevant in this example, but applies to macros containing procedural code. They can be difficult to debug with simulator GUIs. The UVM library contains many useful macros. Some of them, eg. the ones that automate the communication between a UVM sequencer and a UVM driver, should be avoided if it is desired to debug that section of the code. Stepping through the macro code with a debugger GUI does not always work.

Regression Testing

SystemVerilog provides random stability. Each class and thread has its own random number generator. It generates a sequence of random values which can be requested by calling one of SystemVerilogs randomize functions. The sequence generated in one class or thread is not influenced by other classes or threads. For a selected initial random seed and unmodified testbench code it is guaranteed that the same random values will be generated if the simulation is repeated. This makes it easy to run regression tests with the same input values. Problems arise when the testbench code is updated. A single call to a randomize function influences all subsequently generated random values within one thread and all threads spawned by this thread. The reason behind this is that whenever a new SystemVerilog thread is spawned, its thread-internal random number generator is seeded by the next available random number of the parent thread. So if one call to a randomize function is inserted before spawning a thread, it will get a different seed and therefore the random sequence produced inside this thread will be different [ICS18]. That means the modified version of the testbench will not be able to generate exactly the same input value sequences for regression runs.

Verification results need to be reproducible if the design shall conform to the IEC 61508 standard [ESC10]. Version control can be used to store each version of testbench, reference model and DUV code. If needed for evidence, any test result can be reproduced this way. For regression testing though it is not practical to use different versions of the testbench. Furthermore this might not even be possible. Regression tests are executed after modifications of the DUV [ESC10], so an old testbench version might not be usable together with the new DUV.

Functional coverage collection ensures that a regression test run still tests the same functionality. It also needs to be ensured that a regression test run would find the same faults that a previous test run found. Each time a test fails, an entry could be added to the coverage model, which models the input and output conditions at the time of the test failure. A verification run on the faulty DUV has to cover this entry with a failing result. (If coverage collection is only enabled if test results pass, this would have to be bypassed.) It might not be trivial to create a coverage model entry that exactly describes the test failure scenario. It should not be coverable by input scenarios that cannot produce the failure of the faulty DUV. It might be that it is more efficient in terms of engineering time to write a directed test case for the failure scenario. It can be added to the regression test suite. This test should fail with the old DUV and pass with the new one. Since the test case might have to be adapted to a modified DUV, it is left to the engineers to ensure that the test case still produces the same scenario. If the DUV was changed significantly, even this might not be possible. Another option could be to write a formal property that ensures that the fault does not exist anymore.

3.3.1.3 Coverage Collection

Functional Coverage Collection

Elements of the functional coverage model are descriptions of DUV usage or operation scenarios. When doing black-box verification, only input and output signals of the DUV and testbench internal signals can be used for these descriptions. The use of internal DUV signals is not allowed [WCC09]. Within the SystemVerilog testbench, the coverage model can be implemented as SystemVerilog covergroups or SystemVerilog cover properties [ICS18].

It is important to cover an input signal only when its effect could have been observed [TK01]. For black-box verification this means that the effect needs to be visible at the outputs. Therefore the approach chosen was to create one coverage collector for inputs and outputs. This class instantiates all covergroups. UVM does not provide a coverage collector component. However, another UVM component can be used, a UVM subscriber. This is a component that subscribes to another component by implementing the receiver side of a TLM1 port. The CMPU does not receive new input values until its calculations are complete. This is due to the system architecture. For a DUV with this property, all input information can be passed from the input transaction through the DUV interface and read by the output monitor. The coverage collector can receive this transaction through its TLM1 port. For a DUV without this property, the correct input and output transaction would have to be matched. UVM provides for example First In First Out (FIFO) TLM1 ports that could facilitate the matching. A UVM subscriber, like any UVM component, can implement an arbitrary number of TLM1 ports.

In some cases it is desirable to combine all coverage information into a single log file. In the chosen testbench architecture, all log files are created and configured inside a test case. UVM report IDs are used to determine the file into which a UVM report message is written to. For example each report message with the ID string "covCollector" is directed to a specific file. The coverage collector first samples the covergroups and afterwards retrieves the percentage value of coverage. SystemVerilog covergroups provide functions for both actions. Whenever the percentage increased, a log message is written.

Cover properties are concurrent statements that need to be contained in a package, module, interface or similar SystemVerilog component that allows concurrent statements. The choice for the CMPU was the interface to the DUV. There the properties can access all DUV ports. The coverage of cover properties can be automatically displayed in simulator GUIs or reported by simulator functions [Cor18c]. Another option is to write coverage information to a log file, which can be the same as used by the coverage collector class. Concurrent cover properties can be followed by an action block. It is executed each time the property is covered. A UVM report macro can be used there directly, with ID parameter "covCollector". Another option is to trigger a UVM event inside the action block. This UVM event needs to be declared inside the interface. The UVM event, in contrast to a SystemVerilog event, can send data to any component waiting for its occurrence. Instead of writing a message to a log file each time a property is covered, which can be very often for some properties, it is probably more interesting to increase a counter. It can be implemented as a static variable in the action block and its value can be sent with the UVM event. A UVM event can only send objects, so a simple data type like an integer or string needs to be encapsulated in a custom object. A "message data" object can be created that can store any relevant information. The coverage collector class could spawn a thread that waits for the

UVM event to be triggered. This can be easily done through the SystemVerilog fork statement. Whenever of interest, a UVM report message can be written to the coverage log file by using the report ID "covCollector".

Similarly UVM events can be used to provide coverage feedback to the stimulus generation. When a certain signal or signal combination is covered, the constraints that led to the coverage could be relaxed. The UVM sequence could be reconfigured or even replaced through type overriding. The only architectural limitation is that the component that waits for the triggering of a UVM event, needs to have access to it. A UVM event inside the interface can be waited for by any component through a virtual interface. A virtual interface is like a pointer to the interface. It can be stored into the UVM configuration database by the UVM top module. Any class can retrieve it from there. A UVM sequence could get the virtual interface, but it cannot access any other testbench components. It is actually a UVM object that communicates with other testbench components trough UVM functions. A UVM test case can wait for the triggering of a UVM event. Since it instantiates the UVM sequence object, it can pass information to it by calling its methods.

Like mentioned above, SystemVerilog covergroups have some limitations regarding dynamism. Covergroups are instantiated like classes. Their constructor and the covergroup itself can be parametrized. The coverpoint bins can use variables that are accessible within the scope of the covergroup. Nevertheless, once a covergroup is instantiated, it is created with the values of all parameters and variables at that time. All values are passed in per value, even if the variables are no parameters. Coverpoint bins cannot be updated after the covergroup instantiation. The only place where SystemVerilog covergroups can be instantiated is in the constructor of a class [ICS18]. UVM phases come in handy to control the order of object creation. In the testbench architecture chosen (see figure 3.3), the coverage collector class is created in the build phase of the scoreboard, which is created in the build phase of the UVM environment. The UVM environment is created in a test case. The build phase is traversed top down, which means that the UVM test case's build phase is followed by the UVM environment's build phase, followed by the UVM scoreboard's build phase. This is where the coverage collector is instantiated, which in its constructor instantiates the covergroups. That means that the values that should be used to create the covergroup and its coverpoint bins must be assigned to the relevant variables latest in the build phase of the scoreboard. The scoreboard is a component that is not modified for different test cases, so the best place to modify variables used by covergroups, is the build phase of a UVM test case.

Structural Coverage Collection

Structural coverage collection can be switched on additionally to functional coverage. It is not recommended to use structural coverage collection alone [Meh14], but it is a very useful tool for finding so-far untested functionality. Often it can be collected automatically by a simulator tool. The DUV only needs to be compiled and optimized with some tool specific command line arguments that turn on annotations for structural coverage collection. Then the simulation run has to be executed with structural coverage collection enabled.

Optimization is useful to speed up simulation. It modifies logic to be more efficient and removes unreachable code. For structural coverage collections it can have some unwanted side effects though. An optimization that removes unreachable default cases of case statements can be useful, because when they are included it would never be possible to achieve 100% code coverage. Some simulators are able to exclude such parts from coverage with specific tool settings [Cor18c].

In black-box verification of saftey-critical designs, structural coverage can be used to find unused or unreachable functionality. The goal is to determine whether the design contains unspecified functionality [ESC10]. Optimization that removes unreachable code is counter productive in this case. It needs to be carefully decided which level of optimization is used.

When it is desired that the verification engineers gain no knowledge about the code, then structural coverage analysis has to be done by somebody else, since it means analysing the source code.

Many simulators [Cor18c] implement the Unified Coverage Interoperability Standard (UCIS) [Ini12] in the form of coverage databases. They can store both functional and structural coverage, collected during simulation and formal verification.

3.3.2 Formal Property Verification

Several formal verification techniques were mentioned in section 2.2.3. For the functional verification of the CROME Measuring and Processing Unit (CMPU) a verification technique was needed that allows black-box verification. A requirement by the IEC 61508 functional safety standard is, that only qualified tools are used [ESC10]. Therefore any research tools in experimental stage were ruled out. Proven in use open-source tools would have been allowed. The ACL2 theorem prover [8] was considered. One reason for not choosing ACL2 or a similar tool was that these tools perform formal property checking on a model that is written in a specific input language which is not a Hardware Description Language (HDL). The model would have to be somehow related with the original VHDL code of Device Under Verification (DUV). It would have to be ensured that the model exactly represents the HDL code. Another reason for not using such kind of tool was that there would have been a higher risk of introducing faults into the verification code. The input to ACL2 is a specific functional language. Not only would it have been a new programming language for the verification engineer and any engineer who will review the verification code, but also the concept of functional programming languages is different from common imperative programming languages or HDLs. Since the CMPU is a safety-critical electronic device, risk reduction applies also to its development and verification process. The IEC 61508 standard goes even further and specifies any kind of formal verification executed by an engineer experienced in formal methods as a measure of "low effectiveness" for avoiding systematic faults. Only if the engineer performing formal verification has had experience with formal verification of similar designs it is considered as to be a measure of "high effectiveness" [ESC10]. This was also a reason why formal verification could not have been the only verification methodology chosen. It cannot be guaranteed that CERN will always find verification engineers experienced in formal verification of similar designs. Furthermore the design has very unique characteristics like its long lifespan and large range of input values, so it might not be possible at all to find someone who worked on a similar design.

Formal Equivalence Checking (FEC) was considered as well. It might have been possible for some blocks of the DUV. Qualified tools would exist that synthesize C code into RTL code [12] and that perform sequential equivalence checking between two RTL models [12, 3]. They are intended to prove the equivalence of the same model on different levels of abstraction. In that case the tools know very well the mapping of ports and keypoints between the two models, because they have generated one or both models. The DUV of this thesis and the reference model have been written independently and additionally on a different level of abstraction. It would have not been straight forward to match the two models. The tools would have to be guided with additional logic and constraints. A typical case is the reset signal. The C model does not have a reset signal, but a function can be called to reset it. A simple one-to-one mapping of input and output ports is not possible. Therefore the same is true as mentioned for the ACL2 and similar provers. It would have been an error prone activity. Furthermore the tools might even need some internal mappings of keypoints. Design knowledge is needed to identify them, so the independence of the verification would be lost.

Of course formal property verification is error prone as well [SSK15], but it has an advantage over the other techniques in that regard. Formal property verification can be well combined with simulation-based verification. One language can be used for both kinds. The language of choice was SystemVerilog. The same is possible when the simulation-based testbench would be written in VHDL. Properties could be written in the Property Specification Language (PSL) in VHDL flavour. PSL can also be written in a SystemVerilog flavour. In that case though, the DUV would have to be written in Verilog [CS10]. The advantage of combining both techniques is that any assertion written for formal verification can be added to the simulation-based environment for verifying the property itself.

This is a measure to find false positives. A formally proven property has to hold during simulation as well, as long as the same input assumptions or constraints are used. When the same signal names are used in the simulation interface and the formal verification module, the same properties and assertions can be used in both environments. They can be placed into a separate SystemVerilog file and included into both verification environments. There could be compilation problems when the code in the shared file is not allowed outside a interface or class environment or when it references variables that are declared in another file. This problem is solved by including the shared file with the SystemVerilog include statement and not compiling the file on its own. The compiler copies the content into the SystemVerilog file which includes it and compiles the resulting file.

In comparison to developing a UVM testbench, formal property verification can be started very quickly. Figure 3.6 provides an overview of the tool and test module architecture. First one SystemVerilog (SV) module is needed that contains the input assumptions and assertions. A second module is needed that binds this verification module into the DUV by using the SystemVerilog "bind" keyword. That way the SV module can directly access any DUV ports. For the SV module these ports are all input ports. The formal tool compiles the SV modules together with the DUV into a formal module on which it performs the proofs. A minimal setup contains not more than that. Further modules, constraint files and automation scripts, eg. for report and coverage generation can be added in later steps.

The SystemVerilog Assertion (SVA) language combines linear temporal logic inspired operators [CDHK15] with sequence matching over multiple clock cycles. The latter concept is called Sequential Extended Regular Expressions (SEREs) in the PSL standard [CS10]. A property written in SVA describes an expected state of the DUV signals, followed by its expected future states. It can refer to the state of the previous clock cycle through SystemVerilog sampled value functions. With the $past function it can refer to a value at a constant number of clock cycles in the past. It can refer to future states in any number of clock cycles [ICS18].

The state of the CMPU always depends on its states in the past, often many clock cycles be-

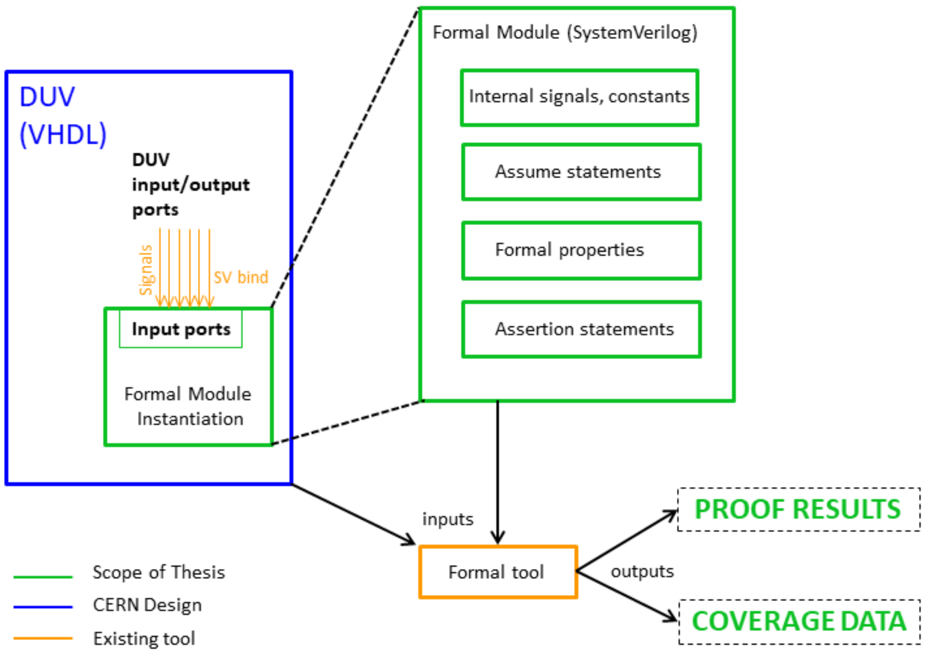

Figure 3.6: Formal property verification architecture for the CMPU

fore a property should hold. This number varies, depending on the input signals. To verify such kind of designs, some additional SystemVerilog code is needed. Concurrent "always procedures" can be used inside the verification module to capture previous signal values in module variables. They can also perform calculations based on the values in several states. The module variables can be accessed by the concurrent assertions. That way it is possible to refer to past states.

Coverage collection

The verification result of each property can be traced back to its corresponding natural language property, which belongs to exactly one functional or safety requirement. Functional coverage through formal properties can be calculated with the following formula:

$$Functional\ coverage[\%] = \frac{Number\ of\ proven\ properties}{Number\ of\ properties} * 100.00$$

Formal verification tools also allow to collect structural coverage during the model checking process [Cor18a]. It provides information about the percentage of DUV logic that was used by all proofs. As in simulation it can be used to discover so far unverified logic.

3.3.2.1 Workflow

In the following paragraphs a workflow is defined. It can be summarized into these steps:

- Formulate each requirement as SVA properties.
- Start without input assumptions.
- If assumptions are used, use them as assertions in previous blocks and add cover properties.
- Prove assertion or add it to simulation-based verification.
- Check assertions and assumptions in simulation.

Formulate each requirement as SVA properties:
For each functional and safety requirement one or more natural language properties is defined. The natural language properties should be written in English language phrases that combine the language constructs suggested in table 3.3. The mapping in the table can be used to translate each of these natural language properties into a SVA property.

Start without input assumptions:
Ideally one would have to place no assumptions on the input values, because each SystemVerilog assumption removes a set of input values from the proof. Input values that are assumed to not occur will lead to the formal tool not considering these values in its proof, so the proof won't be exhaustive. Design blocks that receive some of their inputs from other blocks might require certain input value ranges or input sequences to function as specified. In that case input assumptions have to be placed into the formal verification environment. For robustness testing of safety-critical designs the input assumptions should not rule out illegal, but possible input sequences. The properties would have to consider these cases. Another way is to write separate verification modules for legal and illegal inputs and write according properties for both cases.

If assumptions are used, use them as assertions in previous blocks and add cover properties:
A proof that contains input assumptions is only valid if it can be guaranteed by other means that these input values will never happen [SSK15]. This can be done for example by placing the input assumptions as output assertions into the proof of the previous block when doing block-level verification. Furthermore the assumptions can be placed into the simulation verification suite, where they are treated as checkers by the simulator. A failing assumption would show that either a legal value was ruled out by assumptions in the formal verification suite or that an illegal value was simulated. Another way for generating confidence that the assumptions do not rule out possible input sequences is to add cover sequences or properties for important legal input sequences, eg. ones affecting safety-critical functionality.

Prove assertion or add it to simulation-based verification:
Once the necessary assumptions are written, the model checking tool can be run on the design and the assertion statements. When a property is formally proven, it is proven for all input value combinations. The corresponding functional or safety requirement can be excluded from simulation-based verification or the coverage goal could be set to 0. If it is inconclusive, it can be placed into the simulation-based verification software as a checker. If the property was not faulty,

it should never fail in simulation.

Check assertions and assumptions in simulation:
The proven properties should also be placed into the simulation testbench for checking the properties themselves. This helps to discover false positives. The input assumptions can be placed into the simulation testbench as well. If the simulation applies stimuli that are not within the constraints imposed by the assumptions, it will have to be further investigated. It might be that the simulation uses input stimuli that are not expected to happen in real system operation. It would have to be validated whether the verification results are not influenced by this. Another reason could be that the input assumptions for formal verification were too tight and the formal proofs are not valid for the all possible usage options of the design.

3.4 Verification Documentation

There are two main documents that need to be created for conforming to the IEC 61508 standard for functional safety [ESC10]:

- The documentation of the final verification results, showing that all safety-critical functionalities have been verified. See table 3.5.
- A tracking of encountered faults and the corrective actions applied. See table 3.4.

The verification activities need to generate reproducible results. The way to obtain them was stated in the verification plan. Even though reproducible, they are usually stored as evidence that the verification activities have been actually performed for a specific version of the DUV [FV14].

UVM provides many reporting features that can be used to generate log files. The amount of log messages generated as well as where they should be printed to, can be controlled at runtime from within the testbench. Together with simulators that support UVM, the logging can be controlled after compilation through UVM command line arguments. They can be used for example to change the severity of log messages or to redirect them to a specific file. A dedicated reporting phase is traversed at runtime after the test run phase has finished. It can be used for writing summary information of test runs [Ini14].

The Accellera Systems Initiative created a standard called "Unified Coverage Interoperability Standard (UCIS)" [Ini12]. The goal of the standard is that coverage information can be shared between different Electronic Design Automation (EDA) tools of different vendors. It is implemented by several EDA tools in the form of a coverage database. It can combine the following information, collected during simulation and formal verification:

- Functional coverage information collected with SystemVerilog.
- Structural coverage collected during simulation.
- Assertion results during simulation.
- Formally proven assertions.

The result documentation for the verification of the CMPU has been stored as spread sheet tables. Tracing information from results to requirements is available, but it could be enhanced by using requirements tracking tools [Pit18]. Table 3.4 and table 3.5 list the mismatch tracking and verification result tables that have been created. The left column states the table name, the right column lists the fields of the table. For fields that are not self explanatory, a description is provided in parenthesis. The columns for simulation input and output values have several subfields. A simple and effective way of tracking testbench, reference model and DUV versions is the usage of a version control system like git [4]. The git commit hashes can be used as version number. This avoids the danger of forgetting to update a version number manually and allows to easily recreate the exact code state that produced any verification result.

Backward traceability of negative verification results to functional and safety requirements or the design specification is given through the field "Verification requirement". The verification requirement can be looked up in the verification plan, where it is linked to a requirement or specification statement. The final positive simulation results table only contains entries of fully verified modules or subsystems. Backward traceability to the verification requirements is given through the covergroups and cover properties of the testbench. Coverage report files can be listed in field "Log files". Several UVM test cases might be needed to achieve full coverage. EDA tools are able to print the UVM test name that lead to the coverage of a covergroup into the coverage report [Cor18c]. The test results of each test case should be reported into a log file which can be listed in field "Log files" as well. Backward traceability of the formal proofs is given through the property name, which is associated with a requirement in the verification plan. The verification requirement can be stated redundantly in the formal proof result tables.

Table 3.4: Documentation of negative verification results plus the correction of their cause.

Table name	**Fields in table**
Mismatches Simulation	Number Date Verification requirement DUV Module(s) Input Values: Signal name, Second last input value, last input value DUT Output: Signal name, Second last output value, last output value Reference Model Output: Signal name, Second last output value, last output value Mismatch Description Cause Solution DUV version - mismatch DUV version - solved Reference model version - mismatch Reference model version - solved Results log file - mismatch Results log file - solved UVM testbench - mismatch UVM testbench - solved
Mismatches Formal	Number Date Verification requirement DUV Module(s) Failing property Expected behaviour Actual behaviour Mismatch Description Cause Solution DUV version - fail DUV version - proven Log files - fail Log files - proven Formal testbench version - fail Formal testbench version - proven
Inconclusive Proofs	Number Date Verification Requirement DUT Module Property DUT version Log files Formal testbench version Status (inconclusive, or proven if proven later on) Cause Solution (If proven later on, proof information can be found in "Proven Properties" table.)

Table 3.5: Documentation of positive verification results.

Table name	**Fields in table**
Simulation Results	Number Date DUV Module(s) DUV version Log files Reference model version UVM testbench version
Proven Properties	Number Date Verification requirement DUV Module DUV configuration (full or reduced bit widths) Property DUV version Formal testbench version Log files

Chapter 4

Radiation Protection at CERN

The European Organization for Nuclear Research (CERN) conducts particle-physics experiments. For that purpose particle accelerators have been built that accelerate particles to very high speeds, close to the speed of light. The particles gain very high energies of up to 7 TeV. When these particles interact with matter, eg. the accelerator itself, and with other particles during the intended collisions, ionizing stray radiation is emitted. It can be life-threatening for human beings to be exposed to radiation. Therefore it is strictly forbidden for people to be close to the experimental machines when they are operating.

The Radiation Protection Group at CERN is responsible for protecting the people and environment around CERN from ionizing radiation. It has several sections where one of them is the Instrumentation and Logistics section. This section is responsible for developing, installing and maintaining radiation monitoring systems. These systems monitor the ambient radiation dose equivalent rate at a specific point in time, as well as the ambient radiation dose equivalent received by an area over a certain period. These numbers can be related to the harmful effects that a human body would experience if it were in this area. Limits are defined by the responsible physicists, according to French and Swiss laws. The monitoring system autonomously shuts down the experiment machines and prevents access for personnel in case the defined limits are exceeded.

Currently there are three monitoring systems in operation, the ARea CONtroller (ARCON) system, the RAMSES system and the GRAMS system, as was shown in figure 1.1. The ARCON system is already over 30 years old and reaches the end of its lifetime. It will be replaced step by step with a newly developed system called the CERN Radiation MOnitoring Electronics (CROME). During the next shutdown, Long Shutdown 2 (LS2), of the LHC which takes place from 2019-2020, the replacement will start. Later on the RAMSES system will be replaced by the CROME system as well [BPD^{+}16].

4.1 The New CERN Radiation Monitoring Electronics

The New CERN Radiation MOnitoring Electronics (CROME) system is a complete in-house development of CERN. Its modularized and reconfigurable design increases maintainability in comparison to the old systems. It is highly configurable at runtime, which allows it to be used for various monitoring purposes with different requirements. It consists of several autonomous units, the CROME Measuring and Processing Units (CMPUs). Around 190 units will be installed during the LS2 of the LHC. Figure 4.1 provides an overview of the CROME system architecture.

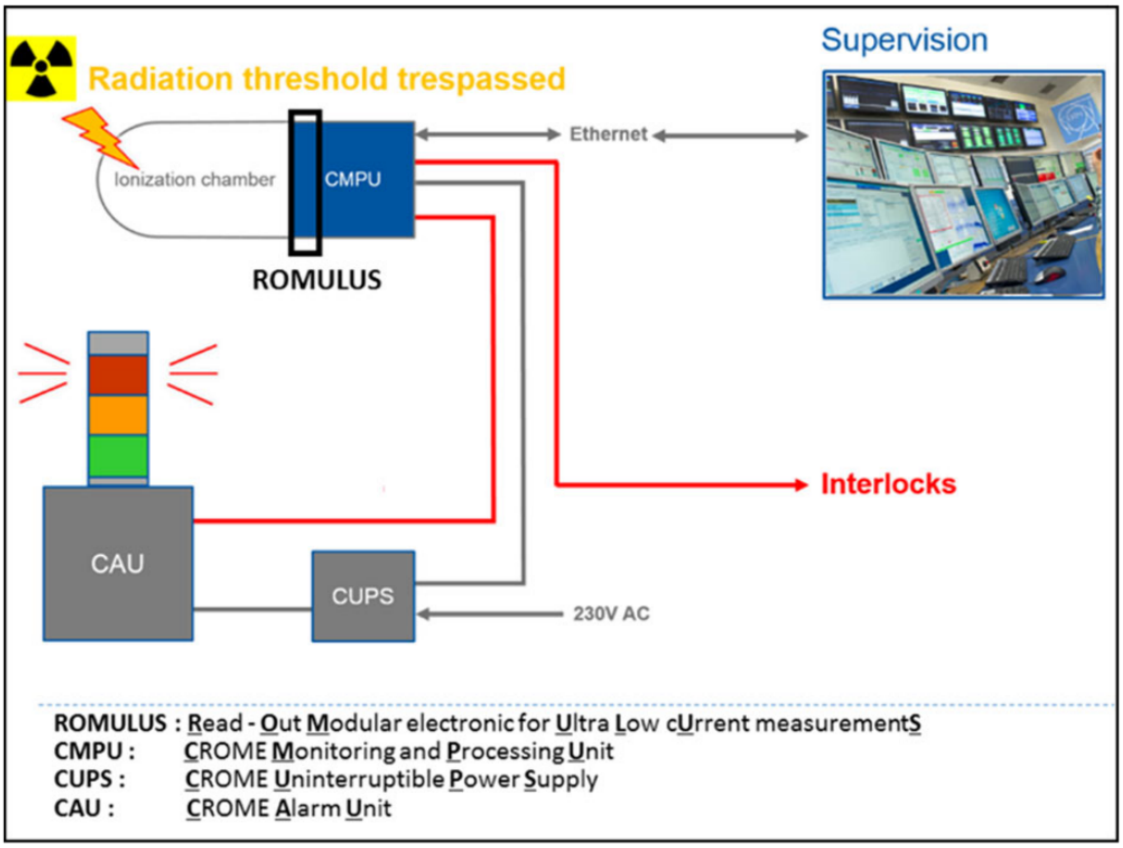
Radiation threshold trespassed
Supervision
Ionization chamber
CMPU
Ethernet
ROMULUS
Interlocks
CAU
CUPS
230V AC
ROMULUS : Read - Out Modular electronic for Ultra Low cUrrent measurementS
CMPU : CROME Monitoring and Processing Unit
CUPS : CROME Uninterruptible Power Supply
CAU : CROME Alarm Unit

Figure 4.1: CROME's System Architecture [BPD+16][1].

The CROME system consists of the following main components:

- CROME Measuring and Processing Unit(CMPU)
- CROME Alarm Unit (CAU)
- CROME Uninterruptible Power Supply (CUPS)
- CROME Junction Box (CJB)

A CROME Measuring and Processing Unit (CMPU) consists of an ionization chamber and an electronic readout system. When placed into areas of low radiation the electronic system is directly attached to the chamber. In areas of very high radiation, that is for example directly beside the accelerators, only the chamber is placed into this area. The electronic system is placed in an area with less radiation and connected to the chamber by a special cable that can transport currents in the magnitude of femtoampere with very negligible losses. It has been developed at

[1]With kind permission of the author.

CERN for this purpose.

When ionized particles interact with the gas inside an ionization chamber, an electrical current gets produced. It is measured by the Read-Out Modular electronic for Ultra Low cUrrent measurementS (ROMULUS), inside the CMPU. The CROME system is able to measure a large range of current, from 2 fA to 250 nA. Depending on the type of ionization chamber attached, the CMPU has to be configured with a different conversion factor that leads to radiological units. The CROME system can measure an ambient dose equivalent rate from 50 nSv/h - 0.1 Sv/h [BPD$^+$16]. The CMPU is connected to the supervision software that runs on a Supervisory Control And Data Acquisition (SCADA) server in the CERN Control Center (CCC) [Per17]. Through the supervision software, the operators of the physics experiments can configure around 150 parameters of the CMPU, eg. the conversion factor from current to Sievert (Sv). Many of those parameters enable different functionalities of the CMPU. The two major safety-critical functions of the CMPU are the alarm and interlock generation. If interlocks are generated by the CMPU, machines are stopped and access to irradiated areas is blocked. This signal can be routed to several receivers via the CROME Junction Box (CJB). The alarm signal is sent to a CROME Alarm Unit (CAU), which produces a visible and audible alarm. It can also transfer the alarm state to another CAU [Hur16]. The CROME Uninterruptible Power Supply (CUPS) contains a battery to ensure that the CROME system stays operative even if the main power supply would fail [BPD$^+$16].

4.2 The CROME Measuring and Processing Unit

The CROME Measuring and Processing Unit (CMPU) is the Device Under Verification (DUV) of this thesis. It contains several monitoring functions that can be configured to directly cause a machine interlock, to generate a warning signal with visual and audible alarm only or to not influence the outputs at all. This allows the monitors to be used for many different use cases. They can be placed directly at physics experiments, where they shut down the experiments as soon as radiation limits are exceeded or a door is opened. They can as well be placed somewhere on the CERN sites for informational purposes only. In that case alarm and interlock functions may be switched off completely.

The assignment of SILs per function of the CMPU can be found in its functional requirements specification [Wid18]. Some functions of the DUV are required to fulfil SIL 1 and some have to fulfil SIL 2. This assignment is regulated by the IEC 60532 standard for radiation protection instrumentation [Hur16]. Because these functions are not independent, the higher safety integrity level has to be applied for all of them. Therefore the methodology developed and used in this thesis had to comply to SIL 2 [ESC10]. The CMPU is a continuously operating device with some "on demand" functions that have influence on the continuous safety functions. Therefore the requirements for the average frequency of a dangerous failure per hour for continuous mode of operation had to be applied. The average frequency has to be less than 10^{-6} per hour [Hur16].

The CMPU contains several printed circuit boards that house e.g. temperature and humidity sensors, supply voltage monitors, a high voltage source needed for the ionization chamber and the read-out electronics (ROMULUS). A System-on-Chip (SoC) monitors and processes the measurement values and controls other components of the systems. This SoC is responsible for the safety-critical functions of the CMPU, the alarm and interlock generation. It is configurable through a supervision software via Ethernet. The SoC consists of a processor hard core and a

Field Programmable Gate Array (FPGA). The processor executes an embedded Linux operating system, which executes embedded software applications. This software builds the interface between the FPGA and the supervision software. It receives the configuration, performs some calculations on it and safely passes the result to the FPGA. All safety-critical functions have been implemented in the FPGA. The complete logic is triplicated and a triple modular redundancy voter algorithm monitors the outputs. A watchdog implemented in a Complex Programmable Logic Device (CPLD) monitors the state of the FPGA. Therefore the FPGA is the main target for verification. The communication functions between the FPGA and the software need to be verified as well. The communication with the supervision needs to be secured in the direction of supervision to software. The reverse way is less critical, because the processed data is returned merely for informational purposes. No safety critical decisions have to be taken by the human operators. The CMPU is directly connected to a CAU and the interlocking system. All safety critical decisions are taken autonomously by the FPGA in the CMPU.

The CMPU continuously reads the output of the read-out electronics (ROMULUS) and monitors the components of the system. It is synchronised to the real time and performs its operations at configurable time steps. It calculates several values based on the raw read-out signal, which it passes to the two main signal processing algorithms. These are the "averaging" and the "integration" functions. The "averaging" function calculates the ambient radiation dose equivalent rate per hour. Three different algorithms are available. The "integration" function calculates the ambient radiation dose equivalent measured over time. Both generate their dedicated alarm signal. The activation of these alarms depends on several configuration parameters and internal conditions. Based on that it can be sent to an interlock output, to an alarm unit (CAU) or ignored [Wid18].

Because the CROME system has to be able to measure very low background radiation as well as very high and highly varying radiation during the execution of the particle physics experiments, the CMPU needs to be able to handle large dynamic ranges of measured current. To be able to process these numbers, the CMPU uses large bit widths for its input and output ports. In total around 150 input parameters are received by the CMPU. It outputs around 20 measurement results and 40 processing results. This large number of input parameters makes exhaustive verification through simulation infeasible. Simulation is further complicated by the long operating times of the CMPU. A monitor may operate as long as 10 years without ever being reset. The large bit widths make formal verification challenging.

4.2.1 The Integration block

The methodology defined in section 3 will be demonstrated on the Integration block of the CMPU. The demonstration can be found in section 5. Therefore this block will be explained here in the necessary detail. Figure 4.2 shows where it is located in the system.

The Integration block accumulates the ambient radiation dose equivalent that the area where the monitor is located, receives during an integration period. It contains one of the two main signal processing algorithms of the CMPU. When the integrated value, ie. the ambient radiation dose equivalent value, exceeds a configurable threshold, the integration alarm is triggered. Based on the configuration it can for example trigger a visual alarm or immediately shut down the radiation producing machine, for example the Large Hadron Collider (LHC).

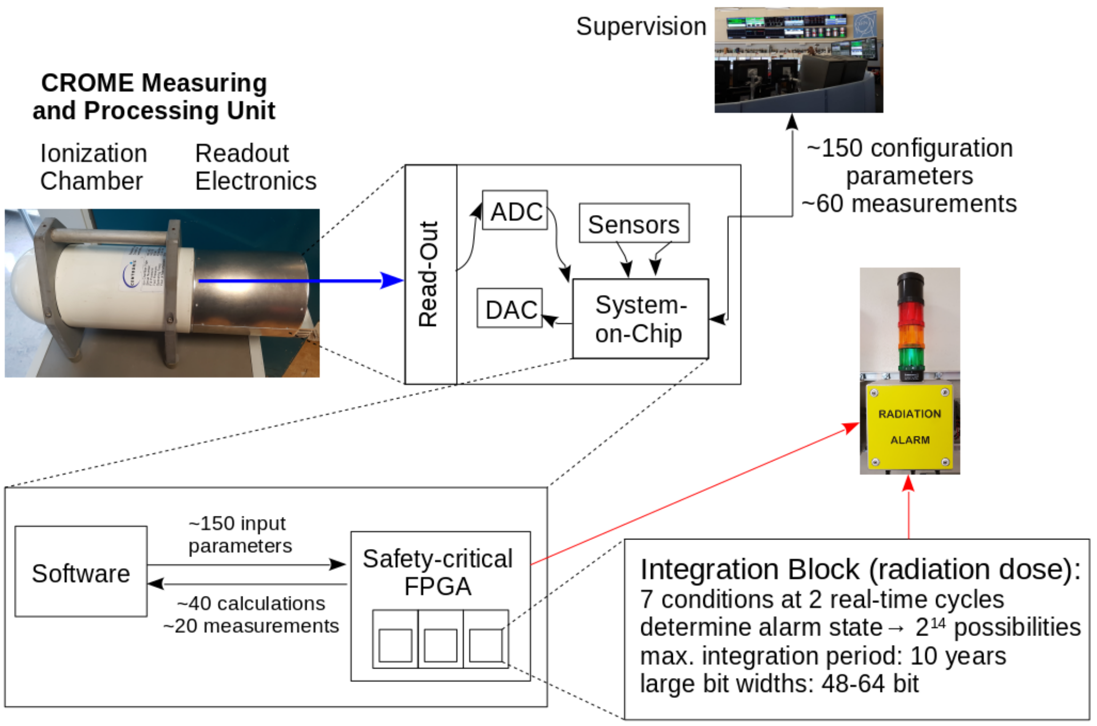
Supervision
CROME Measuring and Processing Unit
Ionization Chamber
Readout Electronics
~150 configuration parameters
~60 measurements
Read-Out
ADC
Sensors
DAC
System-on-Chip
RADIATION ALARM
Software
~150 input parameters
~40 calculations
~20 measurements
Safety-critical FPGA
Integration Block (radiation dose):
7 conditions at 2 real-time cycles
determine alarm state → 2^{14} possibilities
max. integration period: 10 years
large bit widths: 48-64 bit

Figure 4.2: The Integration block inside the CMPU.

The Integration block has the following input signals:

- clock, reset
- a signal to indicate a measurement cycle
- a signal to indicate a device cycle and when the integral calculation can be started
- integral reset signal
- integral alarm reset signal
- a signal to configure the alarm as latched or free running
- a signal to (de)activate the alarm
- integration period length (32 bit, unsigned)
- offset value (48 bit, signed)
- integral threshold (64 bit, signed)
- measurement value, an internal representation of the radiation dose rate (52 bit, signed)

The clock, reset and two cycle indication signals control the general operation of the Integration block. The other input signals are used for determining the output values. Apart from the measurement value, they are all derived from the input parameters which are sent by the supervision software. According to the requirements, new parameter values shall only be taken over at measurement cylces. The only signal change that has to be taken into account between measurement cycles is the alarm reset signal. The embedded software in the CMPU performs several calculations before passing the final parameters to the CMPU. Internally, the Integration block uses a different value representation than visible to the human operators. The embedded software calculates the human-readable result values from the Integration block's outputs. This information is not used for any safety-critical decisions. These are performed autonomously by the FPGA, using its internal value representation.

The block generates 3 output values:

- integration value (64 bit, signed)
- elapsed integration time (32 bit, unsigned)
- integral alarm signal

The **integral value** is calculated from:

- previous integration value (64 bit, signed)
- offset value (48 bit, signed)
- measurement value (52 bit, signed)

It can be seen that the inputs for the integral calculation are very large values. Formal tools might not be able to handle such large counter values [SSK15].

The **elapsed time value** is a continuous count of device cycles (100 ms) since integration start. An integration period can be as long as 10 years, which means 3 153 600 000 cycles. The supervision Graphical User Interface (GUI) allows a minimum measurement time of 1 second. That is the time at which a new measurement value is passed to the integration. That means that it has to be possible to calculate 315 360 000 output values with all input values at their maximum or minimum without any overflows. This number of cycles is too large to be simulated while respecting block internal delays. Even if it would be possible to limit the maximum allowed time of continuous device operation to 1 year, 31 536 000 input stimuli would have to be simulated.

In a Measurement Cycle (MC) the **value of the alarm signal** depends on the following conditions. The alarm reset signal is the only signal that can be received outside a MC. In that case the alarm depends on a slightly different conditions, indicated in parenthesis.

- integral reset signal at the previous MC
- alarm latched configuration at the previous MC
- alarm reset signal in the current device cycle
- alarm value of the previous device cycle
- alarm function activation signal at the current MC
  (at the previous MC if alarm reset happened outside a MC)
- whether the integral value equals or exceeds the threshold at the current MC
  (at the previous MC if alarm reset happened outside a MC)
- whether the elapsed time passed the integration period in the previous MC

For verifying the correct function, 2^7 (128) input stimuli are needed. This is absolutely feasible in simulation and formal verification. But also the absence of wrong functionality needs to be verified. Therefore each condition has to be true and false once at the present MC, and once at the previous MC, leading to 2^{14} (16 384) required input stimuli. It is possible to simulate this number of stimuli, but it can get difficult to randomly generate all these conditions.

Internally the integration block instantiates several smaller blocks that are not subject of independent verification. They were tested by the designers module tests. The verification of the Integration block can be seen as integration test on block level.

Other blocks

There are several similarities between the Integration block and other blocks of the FPGA design. The averaging block has similar configuration options as the Integration block. Instead of calculating the radiation dose, it implements 3 different algorithms for calculating the radiation dose rate. As the Integration block, it generates an alarm depending on a configurable threshold and the alarm can be latched or reset. In some special conditions though the behaviour of the two blocks is different. The FPGA also monitors other components of the CMPU. It generates a fault, when these components do not operate as expected. Faults are configurable in similar ways as alarms, eg. they can be latched or reset. Through the supervision software it can be configured how a fault or an alarm shall be further handled by the CROME system. Similar as the alarm signal of the Integration block, the final output depends on several combinations of input signals. In both cases the output value can be expressed in a logical equation. The verification techniques used to verify the Integration block will be applicable for the other blocks wherever there are similarities.

Chapter 5

Functional Verification of the New CERN Radiation Monitoring Electronics

The initial goal of this thesis was the complete verification of the CERN Radiation Monitoring Electronics (CROME) Measurement and Processing Unit (CMPU). That meant the verification of the Field Programmable Gate Array (FPGA) and software code plus potentially including the communication with the supervision software. There had been only very little systematic verification performed before the start of this thesis project. The FPGA designer had tested his design blocks with directed testbenches, that were already out of date at the start of this project. Only the FPU core had been fully verified with a testbench that was never intended to be extended. On the software side there existed unit tests. Apart from that the system had so far only been tested through the user interface tests of the supervision software, which communicated with the CROME software remotely. These tests can be seen as integration tests on full system-level, but their main target was the test of the supervision software itself. There had been no measurements performed to which extent these tests covered the embedded software and FPGA functionalities. There were no means implemented that could measure such coverage.

All mentioned tests had been performed by the design engineers, no independent verification had been done. On the one hand this provided the opportunity to freely choose the verification methodology for the safety-critical part of the design, since there was nothing that had to be reused. On the other hand exactly this fact made the initial scope of the project too large to be completed within the given time frame of 8 months. It was soon clear that the verification of the complete FPGA and safety-critical parts of the software code without any existing verification software to reuse, was not feasible within this time frame. Therefore the scope of the thesis was reduced to developing a verification methodology for the CMPU and demonstrating its suitability on a representative subset of the design. The verification software framework had to be developed in an extensible way to facilitate the later addition of further parts of the design from block-level up to software system-level.

After an extensive literature study the methodology described in section 3 has been defined. The mentioned verification planning activities have been performed. The functional and safety requirements and design architecture specification were carefully reviewed. Then verification requirements were defined and written into in a verification plan. Safety-qualified tools for applying the chosen verification techniques have been selected. The tool chain was set up and configured from scratch. Several Makefiles and scripts were necessary to ease development activities. Then the verification software framework was developed. First a reference model for simulation-based verification was written. The most time consuming part was to set up the tool chain for simulation-based verification and create the UVM testbench. It had to be developed from scratch before the actual verification of the Design Under Verification (DUV) could be started. This activity took around 1 month. In its minimal version the UVM testbench already consisted of 14 SystemVerilog classes. It contained the necessary mixed-language interfaces and UVM component definitions, together with their Transaction Level Modelling (TLM) port connections. Interfaces between the VHDL design and the SystemVerilog testbench, as well as between the SystemVerilog testbench and the C reference model were necessary. The Questa Simulator [Cor18c] and SystemVerilog provide practical features for implementing these interfaces. Only after these preparations the development of the input stimulus generation and coverage collecting components, which are the most complex components of the simulation-based FPGA verification, could be started. In parallel the formal verification tool chain was set up and the formal verification modules were prepared. This took considerably less time in comparison to the preparation of the UVM testbench. In less than a week the first formal SystemVerilog module was ready and first properties could be fed to the tool.

Short after starting the actual verification activities on the DUV it became clear that there would be multiple changes in the FPGA and embedded software code necessary. The first reviews of the requirements and architecture specification of the FPGA already revealed some ambiguities in the specifications as well as unspecified assumptions. This emphasized the necessity for independent verification and for a modularized and extensible verification software framework. The ambiguities and unspecified cases in the requirements had either not been found or not been documented during the whole past 2 years that had already been spent in the CMPU development project. In some cases the requirements and design engineers agreed on a specification, but never wrote it down. In other cases the designers took design decisions and noted them in the design architecture, but did not discuss them with the requirements engineer. During the course of this thesis project several meetings with all involved engineers were held for establishing a consistent interpretation of the functional requirements. After the meetings one interpretation was agreed on and written down. Specifying formal properties and using their natural language version during meetings helped to identify many special cases that needed to be discussed. The further the verification activities progressed, the more unspecified or ambiguously specified cases were found and often the agreement on requirements for these cases led to updates in the embedded software and FPGA code. The benefits of developing a modularized and flexible verification software framework could be fully exploited even during the time of its development. Most adaptions could be done on very localized portions of the verification code and therefore the modifications did not require much time.

In the following subsections the methodology defined in section 3 is demonstrated on the Integration block of the CMPU. The reference model has been developed for several more blocks. Initial reviews of the requirements and high-level design specifications have been performed for the whole CMPU. Considering the amount of ambiguities found during the course of verification of the Integration block, more findings during the verification of further blocks are to be expected.

5.1 Verification Planning

Verification planning started with an extensive review of the requirements and design architecture specification. Several meetings were held with the requirements and design engineer to clarify unspecified details. Then the verification requirements were determined. In parallel the coverage model for simulation-based verification and properties in natural language were specified. The following subsections provide details of these activities.

5.1.1 Review of Requirements and Design Architecture Specification

Reviews of the CROME functional and safety requirements specification and of the design architecture specification have been performed. The review of the requirements specification targeted mainly unspecified or ambiguously specified requirements. No judgement about system-specific logical formulas and ambient equivalent radiation dose rate calculations could be made, because expert knowledge in radiation protection would have been needed. These kind of reviews will have to be performed by specialised physicists. Review meetings were held with the requirements engineer where the encountered ambiguities were reformulated and unspecified assumptions were specified. An important point to note here was that several review meetings were held with the requirements engineer only and not with the design engineers in order to increase the independence between design and verification engineers. By updating the specification before starting with verification activities some time could be saved during verification software development, because misinterpretations of requirements could be avoided.

5.1.1.1 Review of Requirements Specification

The requirements specification was reviewed in order to derive verification requirements. It was found that many requirements were specified on a very high level, leaving many implementation decisions to the designer. After several short discussions with the requirements engineer, the strategy was changed. One detailed review meeting was held between the requirements engineer, the design engineers and the verification engineer. The purpose of this meeting was to identify the loosely specified requirements and decide together on an adequate implementation. When preparing for this meeting, the idea of stating the formal properties in natural language arose. The meeting was started with a list of open questions and the natural language properties as input. The natural language properties were actually too complex to be discussed during the meeting, so the requirements engineer decided to review them on his own before discussing them. 22 requirements had to be clarified in this meeting. 5 of them regarded the Integration block. They are included in table 5.1. After the meeting both the design and the reference model had to be updated rigorously. The natural language properties were updated as well. The new version was given to the requirements engineer for review and another meeting was held later to discuss his findings. In the following two cases the requirements specification was wrong and had to be updated. These did not regard the integration block.

1. Temperature and humidity values were specified as average values, although they should be momentary measurement values.
2. The comparison operator for thresholds and hysteresis were not consistent.

Luckily the before mentioned meeting was effective, so the second meeting regarding the natural language properties did not cause many code modifications. Some mistakes in the translation from verification requirements to properties could be found, but there were no disagreements about the interpretation of requirements in that cases. Only in one case there was still a difference in the understanding of the requirements between the requirements engineer and the design and verification engineers. The latter two assumed that a non-latched alarm could be reset by the alarm reset signal, whereas the requirements engineer intended to specify that the alarm reset signal only applies to latched alarms. That means that the DUV would have switched off an alarm in a situation where the alarm should have stayed on, and this DUV version would have passed verification. After the review, the DUV, reference model and properties had to be updated. This case clearly demonstrates the effectiveness of reviews. Since the design and verification engineers had the same view on the requirement, all verification activities would have passed successfully. It also demonstrates the limitations of verification. Verification results are positive if the DUV functions as expected by the verification engineer, but there is no guarantee that these expectations are the ones intended by the requirements engineer. Reviews of the verification requirements should be performed by the requirements engineers and other stakeholders.

Table 5.1 lists the results of the review of the requirements specification for the Integration block. It counts the number of unspecified functions that had to be added to the specifications. It further counts the number of updates that were necessary in the DUV implementation and verification code due to the review findings. Finding number 8 was the one discovered when discussing the natural language properties with the requirements engineer.

Table 5.1: Review of requirements and design specifications - Integration block

Nr.	**Ambiguous or not specified requirement**	**Update of specification**	**Update of implementation**	**Update of verification code**
1	New parameter values shall only be taken over at measurement cycles.		1	
2	At the time of an integral reset or the end of the integration period, the integral value shall equal the accumulated dose over the whole period. At the next measurement cycle, the integral value shall equal the accumulated dose over one measurement period. Internally the integral value shall be reset at the time of integration end.		1	
3	At the time of an integral reset or the end of the integration period, the elapsed integral time output shall equal the number of measurement periods that passed since integration start. At the next measurement cycle, the elapsed integration time output shall equal the measurement period.		1	
4	Integration shall be possible for at least 10 years.			1
5	The integration time is a multiple of the measurement time.			1
6	In two specific cases in which the alarm is configured as non-latched, the alarm condition shall be re-evaluated.	1		1
7	The alarm condition shall be re-evaluated at the time of an alarm reset, regardless whether the reset happens at a measurement cycle or between two measurement cycles.	1		1
8	Alarm reset is not valid for non-latched alarms.	1	1	1
9	The latest offset value shall be applied to all values over the whole integration period.	1	1	1
	Total	**4**	**5**	**6**

5.1.1.2 Review of Design Specifications

A part of the design specification and architecture have been reviewed. The main focus laid on the Integration block and on the parameter conversion formulas that are performed in the embedded software (SW) and the FPGA. Fault-detection in the FPGA is implemented through triplication of the FPGA logic and a triple modular redundancy voter. This was not yet fully specified and implemented at the start of this thesis project and was therefore not included into the verification and reviews.

The following list presents the mismatches between requirements and design specification (point 1 and 2) and safety-related weaknesses (point 3-5) that have been found during the review. The first two points concern the Integration block. Suggestions on how to address them are provided as well. The first two examples demonstrate how sever mismatches in implementation and specification can be removed without a single test case. Reviews of specifications with the intent in mind of verifying a design with test cases or formal properties can reveal mismatches before writing a single test case or formal property in software code.

1. **Parameter ranges:**
   All formulas for the calculations of the input parameters and measurement results, which are split between the SW and FPGA, have been reviewed. No mistakes in the formulas could be identified. In some cases though they lead to large truncations of values. This is due to integer divisions performed in the software and the chosen bit widths. Some parameters cannot be configured to their full range as specified. These limitations will have to be identified and validated with the requirements engineers. Either the specifications or the implementation will have to be updated. Even though only trained personnel is allowed to configure the system, internal monitoring would further reduce the risks of failures. Sanity checks on the parameters could be implemented that notify the user if the device is parametrized with a configuration that cannot be handled by the system.

2. **Non-corresponding parameters were used for the integral (ambient radiation dose equivalent) calculation:**
   One systematic fault in the implementation was found in the integral calculation. The measurement value in radiological units is calculated from the measured current and three input parameters, including the offset. To derive the ambient radiation dose equivalent from the raw current measurement value, the offset and the other two parameters need to be in relation. The SW modified the offset so that the FPGA only had to integrate the internal measurement value, together with the offset value in a modified representation. This worked under the assumption that the other two parameters were constant during a measurement period. In the initial version of the CMPU design specification, this assumption was violated. The FPGA performed the integration multiple times within a measurement period, each time using the latest received offset value. At a measurement cycle it sent back the integrated value to the SW, which then calculated the final integrated value together with the other two parameters. The problem arose when the two parameters and the offset were updated between two measurement cycles. The FPGA took any new offset value into account immediately at the time of update, whereas the SW applied the parameter values present at a measurement cycle to the whole previous measurement period. That meant that the three associated parameters of the calculation were not updated at equal times, which lead to wrong derived radiation dose values. This problem was solved anyway by the meeting with the requirements engineer, because he stated that any parameter updates should only be taken into account at measurement cycles. This automatically solved the problem. The FPGA code had to be updated by the design engineer.

3. **Parameter reception:**
   The CMPU is highly configurable at runtime. All safety-critical operations have been implemented in the FPGA, where they are triplicated. Still, the FPGA needs to regularly interface to less safe components of the system. The input parameters coming from the supervision determine the conditions, based on which the FPGA takes safety-critical decisions. It is crucial that the configured parameters reach it uncorrupted. The communication between the supervision software and the embedded SW, as well as between the SW and the FPGA has been protected against data corruption [Ger17]. In the version implemented at the time of the review, it was possible that the operators configure a parameter through the supervision software only a single time during the whole system operation. When the SW received a new parameter, it transformed it into the representation needed by the FPGA. These critical transformations lacked any redundancy. If there were software implementation error that

would cause a wrong calculation, a memory corruption due to other processes or a random hardware fault, it would have never been detected. The risk of systematic errors in the implementation can be reduced through verification of the hardware-software integration, with a special emphasize on these calculations. Nevertheless, to conform to the IEC 61508 standard, each critical function needs to be performed in redundant channels. To reduce risks at runtime, the calculations could be repeated at different times and the results could be compared. This measure has been shown to protect against transient hardware faults [Kop11]. A typical method for run-time error detection is the performance of reversal checks [Tp00]. The parameter calculations could be executed inversely, calculating the originating parameter value from the representation that is sent to the FPGA and the results could be compared.

4. **Secure data transmission:**
The devices are connected to the network, so they could be target of a hacker attack that modifies the configuration. The communication should be encrypted and the supervision should authenticate itself. Only encryption was stated as optional requirement for the CMPU [Wid18]. At the moment the CROME devices are only supposed to be installed in the technical network of CERN that is not connected to the Internet and strongly protected and monitored by the Information Technology (IT) department. As long as this measure is met, the responsibility for protecting against undesired access could be left with the IT department, which performs this on network level.

5. **Monitoring of safety-critical outputs:**
The safety-critical outputs of the CMPU (the alarm and interlock signals) should be monitored at run-time in order to ensure that the sent signal actually reached the target. It should be read back through a redundant channel [Kop11].

5.1.2 Verification Plan

A detailed verification plan for the integration block has been written. For 31 functional and safety requirements, 76 verification requirements were defined. They were grouped into two categories. One category contained verification requirements regarding the parameters. They included the verification of their value ranges, times of updates and interdependencies. The second category contained behavioural descriptions of the DUV. A coverage type like defined in section 3.2.3 has been assigned to each verification requirement. Then a SystemVerilog covergroup, coverpoint or cover property has been specified for each of them. For each functional or safety requirement at least one formal property has been specified, expressed in natural language. It was not necessary to specify a formal property for each verification requirement, because some of them are implicitly covered by other properties. This is due to the fact that for simulation-based verification, a verification requirement might state a description of input stimuli that should be applied whereas formal properties cover all possible input stimuli [SSK15].

Examples

In the following some exemplary entries of the verification plan are presented. The requirements are reformulated and in some cases slightly modified, because for safety and security reasons it is not possible to state the original requirement and to reveal all implementation details. Some requirements are presented together with some of their corresponding verification requirements.

For the latter ones the coverage type is stated before the requirement text.

Example requirement - 1:

Requirement:

"It shall be possible to manually trigger a reset of an integration alarm through the supervision software."

Verification requirements:

1. Temporal: "Verify that an alarm reset happens immediately when the reset parameter was set to 'reset'."
   This is the positively stated verification requirement that describes the functionality that shall be verified. It is categorized as "temporal", because the temporal relationship of the signals is important.
2. Interesting scenario: "Verify that the alarm reset signal has no effect in case the alarm condition is true and the alarm function is not deactivated at the time of receiving the reset."
   This is a tricky verification requirement. First it is not obvious that it is even needed, since the alarm condition is not mentioned in the requirement. The next questions are which configuration values for evaluating the alarm conditions should be used and when shall the condition be evaluated. The way to handle this case was only clear after several discussions with the requirements engineer.
3. Negated requirement: "A latched integration alarm is only reset after the alarm reset signal has been received."
   The CMPU should definitely not implement more alarm reset signals than specified. If it would contain an additional unspecified reset signal or any other condition could cause an alarm reset, this could have life-threatening consequences. An unintended alarm reset when high radiation is present could open the interlocks and allow people to enter the irradiated zones. It has to be verified that there is no other alarm reset signal available than the one specified and that no other condition can cause an alarm reset.

Example requirement - 2:

Requirement:

"An Integral Alarm shall be generated when the integral value exceeds the corresponding threshold."

Verification requirements:

1. Use case: "Verify that an integral alarm is generated when the integral value exceeds the threshold and the alarm function is not deactivated."

2. Temporal: "Verify a case where the alarm condition becomes true one measurement cycle after integration restart."
   If the alarm is not configured as latched, the mentioned measurement cycle would be the point in time at which the alarm is reset. This verification requirement verifies whether the automatic alarm reset is overwritten by the alarm condition being true.

3. Stress test: "Verify that an integral alarm is generated as expected for different threshold values spread over the whole possible range. For each value verify a case where the integral values is larger, smaller and equal to the threshold value. "
   This is a typical stress test requirement that aims at testing boundary values and equivalence classes of input parameters. This is one of the verification requirements for which no formal property needs to be written. If the use case is proven, the stress test is implicitly proven as well.

5.1.3 Coverage Model

The coverage model entries were grouped into several types. They were explained in section 3.2.3. Table 5.2 shows how many verification requirements have been defined for each cover type.

Table 5.2: Count of verification requirements per coverage type.

Coverage Type	**Number of Verification Requirements**
Use case	21
Negated Requirement	4
Interesting Scenario	14
Temporal relation	19
Value range	5
Stress test	7
System Level	6
Total Block Level	**70**
Total Integration Function	76

It can be seen that there were more "interesting scenarios" and "temporal" relationships than "use cases". This is because many special cases were not specified in detail. On the other hand some functions were specified as a positive and a negative statement. This lead to a low number of "negated requirements". The requirements that were stated in the specification in its negated form have been included into the use cases. Some tests cannot be performed on block-level, eg. the instantiation of two integration blocks, therefore they were classified as "system level" tests.

5.1.4 Natural Language Properties

26 natural language properties have been specified for the Integration block. As mentioned above, there were 31 functional and safety requirements. 5 of them regarded the allowed value ranges of parameters (as can be seen also in table 5.2). Since formal properties would be proven for the complete range of their used variables, no specific properties are needed to verify parameter ranges. For each other functional and safety requirement one natural language property has been

defined.

After a review by the requirements engineer these natural language properties were manually translated into SystemVerilog Assertion (SVA) language properties according to the mapping defined in section section 3.2.4. Table 5.3 adds a DUV specific translation of natural language expressions into SystemVerilog.

Table 5.3: Mapping of CROME requirement snippets in natural language to SVA constructs.

Natural Language	**SystemVerilog**	**Explanation**
Cycle is a MC	mcValidxDI == 1	This device cycle is a Measurement Cycle (MC).
Cycle is no MC	mcValidxDI == 0	This device cycle is not a Measurement Cycle (MC).
The time passed since integration start	== etCount	etCount is a signal of the formal verification suite. It is increased in each device cycle.
at the previous MC	signalNameLastMC [1]	It is not possible to access past signal values with a linear temporal property. This functionality was needed because the outputs in one MC depend on the input parameters at the previous MC.
At integration end time	elapsedTimexDO >= integralTimexDI	When integration time has been reached or exceeded.
In the next MC	##[1:$] (mcValidxDI==0) ##[1:$] (mcValidxDI==1)	The MC indication signal is low at some point, then it becomes high again.
Alarm is on	alarmxDO == 1	Alarm is turned on.
Alarm is off	alarmxDO == 0	Alarm is turned off.
Alarm is/was latched	alarmLatchedxDI == 1	The alarm signal is/was configured as latched.[2]
Alarm is/was not latched	alarmLatchedxDI == 0	The Alarm is/was not configured as latched.[2]
Alarm function is/was deactivated	alarmActivexDI == 1	The alarm function is/was deactivated.[2]
Alarm function is/was activated	alarmActivexDI == 0	The alarm function is/was activated.[2]

[1] Store the value of a signal at a MC in a test module internal variable and access it at the next MC.
[2] If the statement refers to the past, then a previously stored value has to be used instead of the actual signal, eg. alarmLatchedLastMC.

Examples

Some natural language properties for the exemplary requirements stated in section 5.1.2 are presented here, together with their translation into SystemVerilog. MC stands for Measurement Cycle.

Example requirement - 1:

Requirement:

"It shall be possible to manually trigger a reset of an integration alarm through the supervision software."

There are several formal properties necessary to fully proof this requirement.

Natural language property - 1:

"(Cycle is no MC and (alarm was latched at the previous MC) and alarm is on and alarm reset equals 1 and (integral value is less than (threshold at previous MC) or alarm function was deactivated at previous MC)) implies that:
alarm is off"

This natural language property is translated into the following SystemVerilog property:

```
property pIntAlarmResetBetweenMT1();
  (mcValidxDI == 0 && latchedLastMc == 1 && ALARMxDO == 1 &&
  integralAlarmResetxDI == 1 &&
    (signed'(integralxDO) < signed'(thresholdLastMc) ||
    alarmActiveLastMc == 0))
  |=>
  (ALARMxDO == 0);
endproperty
```

Natural language property - 2:

"(Cycle is no MC and alarm is on and alarm reset equals 1 and (integral value is greater than or equal to the (threshold at previous MC) and alarm function was activated at previous MC)) implies that:
(in two clock cycles, alarm is on)"

This natural language property is translated into the following SystemVerilog property:

```
property pIntAlarmResetBetweenMT2();
  (mcValidxDI == 0 && ALARMxDO == 1 && integralAlarmResetxDI == 1 &&
    (signed'(integralxDO) >= signed'(thresholdLastMc) &&
    alarmActiveLastMc == 1))
  |=>
  (ALARMxDO == 1);
endproperty
```

Here it is not relevant whether the alarm was configured as latched or not, because in both cases we expect the rest to have no effect.

Example requirement - 2:

Requirement:

"An Integral Alarm shall be generated when the integral value exceeds the corresponding threshold."

Natural language property:

(Cycle is a MC and alarm function is not deactivated and integral value is greater than or equal to threshold)
implies that:
alarm is on.

This natural language property is translated into the following SystemVerilog property:

```
sequence sIntAlarmOn();
  ##‘nrCyclesAlarmRdy (ALARMxDO == 1);
endsequence

property pIntAlarmOn();
  ((mcValidxDI == 1)
   ##‘nrCUntilCalcRdy
   ((alarmActivexDI == ) && (signed’(integralxDO) >= signed’(thresholdxDI))))
    |-> sIntAlarmOn();
endproperty
```

The alarm and integral reset conditions are overwritten by the alarm condition, therefore the corresponding signals do not need to be included into the property. The signal delay that needs to be accounted for is specified in SystemVerilog macros (‘nrCyclesAlarmRdy, ‘nrCUntilCalcRdy). This ensured that the same delays are used in all properties. If the DUV was changed, the delays could be easily adapted in one place by changing the macros’ values. Because the statement in "sIntAlarmOn()" was needed several times, it was placed into a sequence that can be used by other properties as well.

5.2 Verification Software Framework

This section presents the verification software framework that has been implemented for verifying the CMPU. First the simulation-based verification suite is explained, followed by the formal property verification suite. The results achieved with each software are presented at the end of the corresponding subsection. A summary and discussion will be provided at the end of this chapter.

5.2.1 Simulation-based Verification using the Universal Verification Methodology

The Questa Simulator has been used for executing the simulation-based verification software. Any other simulator could have been used as well. Except from a few coverage options, no simulator-specific code has been implemented. These coverage options could be replaced by other coverage means if the code were migrated. All other SystemVerilog and UVM specific code is tool-independent. The simulator's capabilities have been used for coverage report generation and combination in the Questa Simulator's Unified Coverage Database (UCDB) [Cor18c], which is an implementation of the Unified Coverage Interoperability Standard (UCIS) mentioned in previous chapters [Ini12]. To migrate to another simulator, the scripts that use these capabilities would have to be adapted.

Figure 5.1 shows the tool chain used for the mixed-language simulation. A Makefile compiles the C reference model into a shared library. It then compiles the VHDL Design Under Verification (DUV) with Questa's VHDL compiler. Afterwards it compiles the SystemVerilog testbench with Questa's (System)Verilog compiler. The Questa Simulator contains the Universal Verification Methodology (UVM) library precompiled in its installation directories [Cor18c]. (It could also be downloaded from [7] and compiled locally.) The Makefile can also be used to start the simulation with several command line arguments:

- A UVM test case can be selected by its name.
- The reference model library needs to be provided.
- Structural coverage collection can be configured.

Functional coverage collection does not need to be configured. It is activated automatically if the testbench code instantiates covergroups.

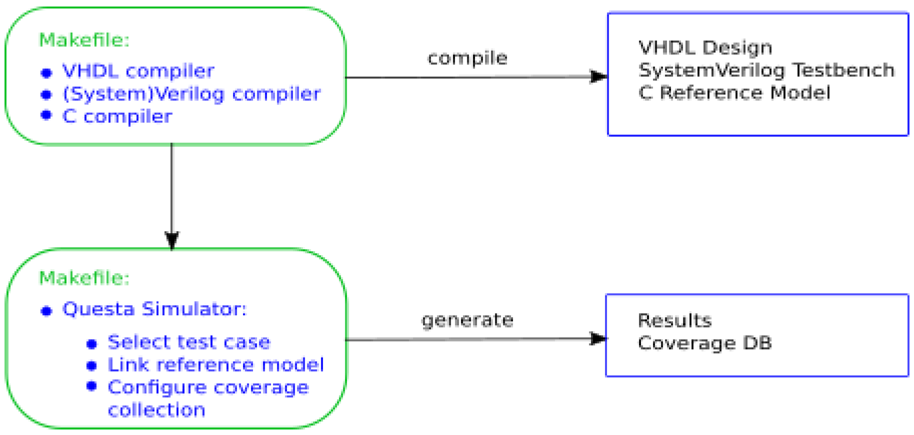

Figure 5.1: Simulation-based Verification Toolchain

Overview of the UVM Testbench

The testbench for simulation-based verification was written in SystemVerilog. Mixed-language simulation was used to simulate the VHDL Device Under Verification (DUV). The concepts and library functions of the Universal Verification Methodology (UVM) were used. Constrained random stimuli were applied to the DUV and the outputs of the DUV were compared to a reference model written in C. To measure the progress of the verification activities, functional coverage collection was implemented, accompanied by structural coverage through the simulator. Transaction Level Modelling 1 (TLM1) was used for communication between testbench components. Figure 3.3 gave an overview of the testbench architecture designed for the CMPU. It is described now in more detail.

The top module of the UVM testbench instantiates the DUV. The DUV is written in VHDL. The VHDL entity is instantiated as if it were a SystemVerilog module. The port type mapping is defined by the used simulator program. The top module also instantiates a SystemVerilog interface that is connected to the DUV entity and one that is connected to the reference model. The latter contains the SystemVerilog Direct Programming Interface (DPI) function import declarations that allow the SystemVerilog testbench to access C functions. UVM components are classes that derive from UVM library classes. SystemVerilog classes cannot directly access module instantiations or interfaces, but they can access virtual interfaces. A virtual interface is basically a pointer to an interface. It enables access to the DUV through the interface ports. Concurrent cover properties/sequences and assertions have been placed into the interface, where they can directly access the DUV signals.

UVM uses the object oriented factory pattern. The top module of the UVM testbench does not create an object of a UVM test class. Instead it starts the factory and tells it which UVM test class object should be created by providing it the test name. This test name can be overwritten on the command line. That way, one UVM testbench can run many different test cases simply by exchanging the test name. UVM has a global resource or configuration database that exits once per simulation. It can be accessed from anywhere within the testbench through static methods of the uvm_config_db class. The top module can put the virtual interfaces to the DUV and the reference module into the UVM configuration database and UVM components can read it from there.

Configurations that are test case specific, have been grouped into a configuration class. Each test case creates an instantiation of this class and passes it to other components of the testbench. The UVM configuration database could be used for that, but it would have a negative impact on simulation runtime, because it needs to look up the objects by name or type each time it is accessed. The test case class is also responsible for handling log files. One base UVM test case class has been created that contains all these general tasks. Specific test cases derive from this class. They execute one or more UVM sequences. Different sequences can be chosen through type overriding at runtime. The base test case instantiates a UVM environment. The environment contains all DUV interface specific testbench components:

- UVM agents: The agents encapsulate drivers, sequencers and monitors.
- UVM driver: The driver is the only component that sends stimuli to the DUV. It has knowledge of timing relations between signals.
- UVM sequencer: This class only derives from the UVM sequencer. It does not implement any specific functionality. All functionality comes from the UVM library.

- UVM input monitor: This class reads the signals that actually arrived at the DUV from the DUV interface. It creates a transaction and sends it to the UVM scoreboard.
- UVM output monitor: This class reads the output signals from the DUV interface, creates a transaction and passes it to the scoreboard.
- UVM Scoreboard: The scoreboard instantiates two further components:
  - Reference model wrapper: This class receives the input transaction from the scoreboard (that was received from the input monitor). It extracts the fields and passes them to the C reference model as function parameters. Similarly it returns the results to the scoreboard in a transaction.
  - Coverage collector: This class contains all covergroups. It receives the output transaction (that was received from the output monitor). This transaction also contains the inputs. The CMPU never receives new inputs before producing outputs, therefore all coverage can be measured based on the output transactions. The testbench can be configured to collect coverage only if tests pass, or any time, which was mainly used for debugging and testbench development.

  The scoreboard compares the output transaction received from the reference model wrapper and the DUV output monitor. If all output signals match, the test passes.

Transaction Level Modelling

The ports of the interface to the DUV are of the SystemVerilog reg datatype and have the the same bit widths as their corresponding DUV ports. There are no signed or unsigned interpretations on this level. The interface to the Integration block also contains some internal variables, most importantly one that stores the time between two measurement cycles. This time is modelled in the testbench, but the DUV has no knowledge of it on block level. Inside the interface, this variable is used for some cover properties and it is provided to the monitor component for reconstructing the output transaction. In the output transaction it is mainly used for recording and interpreting the test result.

Whenever a testbench component accesses the DUV it does not directly access the interface ports, but it accesses them through a clocking block. Clocking blocks more closely resemble real hardware behaviour. An output of the DUV becomes an input to the clocking block and an input of the DUV becomes an output of the clocking block. When the default input read time of the clocking block is set to #1step, it means that at a clock edge at which the clocking block is triggered, it samples all DUV output values that were present exactly 1 simulation step before the clock edge happened. This avoids race conditions that could happen when a signal changes in the same simulation step as the clock edge happened. For the same reason the clocking block delays the driving of DUV inputs to happen a few simulation steps after a clock edge. This matches with the sampling time of properties. Signals in properties are sampled 1 simulation step before the clock event that triggers property evaluation.

Inside the testbench the DUV input and output signals are encapsulated in a transaction. One transaction for the Integration block has been created. The variables in the transactions match the datatypes that are used inside the DUV, they are signed or unsigned accordingly. SystemVerilog integer datatypes can have 8, 16, 32 or 64 bits. For DUV signals that have another bitwidth than these, constraints are used to constrain the allowed values to fit these bitwidths and the next longer datatype that can store this number of bits is used. Most variables in the transaction are declared as rand, which means that random values within the provided constraints and datatype restrictions can be generated for these variables when requested. The clock and reset signal are not randomized and neither are two signals that are generated by a timing block of the DUV. The integration block only works correctly if these two signals have a certain relationship, which has to be modelled in the testbench. The integration transaction derives from the uvm_sequence_item class. It is passed around the testbench components through TLM1 ports.

5.2.1.1 Reference Model

A partial reference model was written based on the CROME functional and safety requirements. It was written in C 2011.

The following blocks have been modelled:

- Measurement value calculation
- Integration
- Averaging using 3 different algorithms

For the mentioned blocks the reference model was written for block-level and FPGA system-level verification. The C model computes the outputs that are computed by the FPGA in the DUV, based on the FPGA inputs. On block-level the model is untimed. Block functions are triggered by certain inputs per block. The C functions need to be called whenever one of these triggering inputs changes. Some triggering inputs, like e.g. the cycle counter of the integration block, could be abstracted to calculations inside the C model. It was not necessary to call the C integration function each time the integration block is triggered. It could be written with a higher level of abstraction. On system-level the real time is modelled as a cycle counter. It is needed to know the order of computations. Some computations are only performed in dedicated time steps or cycle counts that are defined by input parameters. Apart from that no notion of time or modelling of delays was necessary.

The data structure for the FPGA system-level parameters was taken over from the embedded software code of the DUV. In future this will allow to replace the FPGA code with the C model when doing software system-level verification. Pure software verification on software-system level will be much faster than simulating the FPGA in a simulator and using the embedded software code inside the testbench to model the interaction of the two [SSM$^+$02]. The structure uses the C 2011 alignas() specifier, which enforces the tight alignment of the structures values into a consecutive memory space. In the DUV the embedded software interfaces to the FPGA by copying the structure as it is into a Block Random Access Memory (BRAM) of the FPGA whenever new data has been received from the supervision. It uses the same approach to read the outputs of the FPGA. Methods for avoiding faults and for establishing data coherency have been implemented and can be found in [Ger17]. The interfacing between the embedded software and the C model of the FPGA can be done in the same way, by copying the structure. For the block-level verification, a structure for each block has been created which serves as interface between the system-level and block-level within the C code and also as interface between the C code and the SystemVerilog test framework. The advantage of using structures as function interfaces in C is that updates of parameters can be done quickly in one place and there is no need for modifying function signatures.

The C model is connected to the SystemVerilog code through the Direct Programming Interface (DPI) of SystemVerilog. The UVM testbench contains a reference model wrapper class that is responsible for calling the C model functions. It was not necessary to add any specific code to the C model, because it does not call any SystemVerilog tasks or functions and does not use DPI specific data types. The reference model wrapper receives an input transaction from the scoreboard. It passes the transaction field values to the C model through the C model's parameter setter functions. Depending on the inputs that trigger the Integration block of the DUV, it decides when to call the C model functions that calculate the output. The reference model wrapper can either start the C model's computations through one block entry function or it can call subfunctions of this entry function at different simulation time steps. For the verification of the Integration block, the reference model wrapper calls the block entry function regularly and any time an alarm reset was received asynchronously to the integration calculation period, it calls the alarm reset setter function and the alarm reset evaluation function.After the computation it retrieves the outputs through getter functions of the C model and creates an transaction that it sends back to the scoreboard for comparison with the DUV.

5.2.1.2 Constrained Random Inputs

One base test case and one base sequence have been created. They are both derived from the corresponding UVM base class. They contain all functionality that is common to all tests or sequences. That is for example the configuration of log files and other test configurations in the test base class. The base sequence sends a reset signal to the DUV at the beginning of a sequence. Then it starts a loop that regularly generates measurement cycle signals in a transaction. Other parameter values in this transaction can be modified by functions that are overwritten by derived classes.

Apart from the base classes, 19 test classes and 22 sequence classes have been written. Each test case executes one sequence. Test cases can configure eg. globally used maximum values, as was described in section 3.3.1.2. They can also spawn threads that wait for UVM events. This concept has been used to receive feedback from other testbench components and use this knowledge to modify the behaviour of the executed sequence at runtime. A test case can receive a notification from the output monitor when an integration period is finished. This allows to keep certain signals stable during a whole integration period and to prevent integration resets. Another UVM event can be received from the coverage collector, which informs a test case that all coverpoints regarding the integration reset signal are covered and therefore the test can disable the generation of integration reset signals in the running sequence. Sequences cannot receive UVM events, because they are UVM objects. UVM objects cannot access UVM components, they are not part of the component hierarchy. But test cases are, so they can wait for the triggering of an event and pass on the information to a running sequence by accessing its methods or public fields.

Most sequences update parameter values with a certain probability. A probability of 50% means for example that a parameter is updated every second measurement cycle. Except for the alarm reset, parameters are only updated in (or shortly before) measurement cycles. One sequence was created that is configurable by the information received from the UVM events mentioned above. This sequence aims at verifying full integration periods and measurement periods. It limits the probability of integral reset and alarm reset signals. It furthermore keeps the measurement period small in most cases. This sequence was the base class for several more sequences that modified eg. the constraints or disabled the integral reset completely. Constraints of a class can be disabled by a SytemVerilog function. This technique was used to run some sequences with a modified value distribution that increased the probability of generating boundary and extreme values. Other test cases executed the same sequence but with uniform distribution by disabling this constraint.

Sequences that targeted the verification of the alarm conditions used short measurement periods, which increases the number of measurement periods that can be simulated and therefore the number of parameter combinations that can be sent to the DUV. To verify all kinds of input combinations before, at and after the end of a integration period, some dedicated sequences with very short integration periods were necessary. To achieve the final percentage of coverage the constraint had to be very tight. Especially when the end of a integration period should be verified together with the integration reset signal, which does not allow a period to end.

Example - Verifying the effect of the configured threshold on the alarm state

Because the DUV has such large parameter widths, covering the effect of threshold configurations on the alarm state needed some tight constraints as well. The goal was to see the effect of integral value comparisons to threshold values spread over the whole possible range. The measurement and offset values were constrained to be between 80% and 100% of their maximum value in order to increase the integral value in large steps. They were not constrained to be at their maximum in order to increase the total number of different stimuli that have been applied. After all, the goal of a test run is not to reach coverage faster, but to ensure that the DUV passes all kind of error-prone test scenarios. In this sequence the integral reset signal had been constrained to 0 and the integration time was set to its maximum, because a reset of the integration would have interfered with the goal. The threshold value was constrained to be below a certain sequence internal signal which got increased each measurement cycle, therefore allowing higher threshold values each time. Once the threshold value was greater than 0 it was further constrained to stay positive until both the sequence internal variable and threshold would overflow and be negative again. In 50% of the measurement cycles the alarm function had to be switched off in order to allow the alarm to cease and be generated again with the next threshold comparison.

Regression Testing

In section 3.3.1.2 it was mentioned that regression tests need to demonstrate that a fault encountered in a previous version of the DUV has been actually removed in its later version. They also need to show that the fault has not been reintroduced. As mentioned, constrained random inputs are only guaranteed to be the same in each test run if the test case and UVM sequence have not substantially changed. One suggestion for creating a regression test suite was to model input scenarios that led to the discovery of faults in the coverage model. This method could be successfully applied to the Integration block. Each output value depends on the present input values and the internal state of the block. The internal state can be reproduced by applying several consecutive input values.

The following steps need to be taken to create a regression test:

1. Identify the input conditions that lead to the situation in which the fault occurs. Add a new "regression" covergroup, coverpoint or coverpoint bin for this situation.
2. Rerun the failing test and make sure that the new regression coverpoint (bin) is covered in the same simulation time step as the test failed.
3. Rerun the failing test with the updated DUV. Ensure that the coverpoint (bin) is still covered in the same simulation time step. Ensure that there are no more mismatches in the outputs
4. Optionally: Copy the previously failing test and UVM sequence class. Rename them to match the regression coverpoint (bin) and implement a stop condition. The test can stop when it has covered its corresponding coverpoint (bin). Add this test to the regression test suite. Do not update this test.

When doing black-box verification, steps 1-3 cannot guarantee that exactly the same sequence of input stimuli is retested. To do so could be a very time consuming process, especially if the fault was found after a long simulation time. When it can be ensured that the added regression coverpoint (bin) models the input scenario that lead to the fault and cannot be covered by another unrelated stimuli sequence, it is sufficient to perform steps 1-3. If this cannot be ensured by the available knowledge of the DUV (input and output values), step 4 can be added.

The integral value and alarm state of the Integration block are always determined through the input and output values of the previous measurement cycle and the input values of the presently observed device cycle. Therefore steps 1-3 are sufficient to create regression tests. Covering a fault in the elapsed time value might require the addition of step 4. The alarm state depends on the logical OR of several conditions. A regression coverpoint bin can be easily added by using value transition coverage. This is explained in section 5.2.1.3. For the integral value it is possible to add an expression coverpoint. To determine the exact condition that led to the fault (which has to be modelled by the expression) might not be possible without internal knowledge of the block design. For highest assurance, step 4 should be added.

Figure 5.2 shows an example of a failing test case and a regression coverpoint bin that has been created. It can be seen that the coverpoint bin "bIntRegression12" is covered at a simulation time of 27300 ns, which corresponds to the time of the test failure. The failure message was printed at runtime with a UVM report macro. The coverage report was printed after simulation by the use of a simulator command. The number appended to the test name (_200) indicates the random seed that was used as initial seed when running the test. The test successfully passed with an updated DUV. In order not to overload the log files, no message is printed when there was no mismatch. The coverage report of the successful test shows exactly the same output. The condition covered by this coverpoint was a very specific one that was not part of the functional coverage model otherwise. This demonstrates the capabilities of random testing. Test scenarios that were not even planned are verified as well.

```
UVM_INFO src/scoreboard.sv(142) @ 27300: uvm_test_top.ieEnv.mScoreboard
[INTGRATION_TEST_InpCondNoIntReset] Test of alarm failed. DUT: 1, refModel: 0
```

```
COVERGROUP COVERAGE:
---------------------------------------------------
Covergroup             Metric    Goal    Status
---------------------------------------------------
Cross cpIntRegression  100.00%    100    Covered
covered/total bins:          1      1
missing/total bins:          0      1
% Hit:                 100.00%    100
bin bIntRegression12         1      1    Covered

---------------------------------------------------
Cover Testname                         Cover Time
---------------------------------------------------
integrationTestInpCondNoIntReset_200   27300ns
```

Figure 5.2: Regression Coverage

5.2.1.3 Coverage Collection

Functional Coverage

In the testbench architecture defined for the CMPU (see figure 3.3) a coverage collector is subscribed to the UVM scoreboard, which receives the output transaction from the UVM output monitor. It passes it on to the coverage collector if the test passed (this can be disabled). The coverage collector class instantiates the covergroups. It is important that the covergroups can access corresponding input and output values, because covering an input value is only meaningful if its effect can be observed. Since black-box testing was chosen, this means that the effect needs to be visible at the outputs. It is possible to activate sampling of a covergpoint only in case of a certain condition is true, which could represent the observable effect.

4 covergroups and 52 cover properties have been implemented for the Integration block. They are shown in table 5.4. In total 54 coverpoints (including the cross points) have been implemented. For each verification requirement (76) there has been at least one coverpoint or cover property implemented. In some cases more than one were required, eg. to generate cross coverage, which is created from the combination of coverpoints. Each coverpoint or cover property describes some device function or conditions in terms of DUV input and/or output values and their relationships. As mentioned in section 4.2 there are 7 conditions at two consecutive measurement cycles that determine the alarm state. This would lead to roughly 16 384 cover bins. Not all combinations are actually possible if the DUV functions as specified. The most important condition combinations, covering all requirement statements, have been specified as several cross coverpoints and collected in the cover group "cgIntConditions".

If one bin could be covered by applying one input stimulus, only 1515 input stimuli would be required to achieve full coverage. Since the system depends on previous states and inputs, this direct mapping is not possible. Some conditions can be covered by applying one stimulus, some value ranges, like eg. the range of the integration period require as many stimuli as the length of the integration period in 100ms steps.

Table 5.4: Covergroups for the Integration block

Covergroup name	**Number of Coverpoints**	**Number of cross cover-points**	**Number of bins**
cgIntRobustness	6	2	392
cgIntValueRanges	14	3	656
cgIntConditions	15	13	466
cgIntRegression	1	0	1
Total	**36**	**18**	**1515**

The robustness covergroup has been created for the verification of the long operating periods of the system. It also contains cross coverpoints between long measurement periods, long integration periods and the whole ranges of input values of the measurement and offset values. It is unlikely that this covergroup can be fully covered within a reasonable time. The value ranges covergroup contains the values specified by the requirements, as well as the whole bit-widths of parameters for further robustness testing. It mainly represents "value range" and "stress test" cover types. The largest covergroup contains the conditions that can influence the alarm state. It will be explained in more detail in the following paragraphs. The regression covergroup contained one very specific condition in which a implementation fault had been found. Further regression cross coverpoints were included into the other covergroups in order to avoid repetitions.

Example - cgIntConditions

The alarm value depends on input and output values at previous measurement cycles and the presently observed device cycle. They can be modelled as the logical OR of 7 conditions at these two observation times. Measurement cycles can be many thousands of clock cycles apart. SystemVerilog cover properties (or sequences) could be used to cover the conditions that influence the alarm state, but they have some drawbacks in this case. In every cycle in which the start of a sequence, or the antecedent of an implication property matches, a new thread is spawned and kept alive until the property (or sequence) either passes or fails. This can have a great influence on the simulation runtime. What was further needed was the possibility to combine several different conditions to model interesting scenarios. Cross coverage of covergroups can generate these combinations automatically.

The first approach tried was to create an array that contains one field per condition outcome for both observation times, thus having 14 bit values. Each possible value of the array would represent one combination of conditions. Interesting conditions could be filtered with "wildcard bins", which allow to specify a place holder for each bit. Impossible conditions could be added to an "illegal_bin". A coverage of an illegal bin (which is part of the SystemVerilog language) does not contribute to coverage and causes a run-time error. This approach was not very successful. It turned out that many conditions are very unlikely to occur at the same time, and some combinations are even impossible. Even though the meaning of each bit is known, it is extremely time consuming to figure out which condition combinations are uncovered. Bins are displayed in this form: `bin auto[1082]` The binary value of 1082 is 10000111010. Each bit represents one condition, e.g. "integralValue >= threshold". This condition could be for example represented by the left most bit in this value. If all other condition outcomes keep their values as described in the bit string and only the mentioned condition changes its outcome, the following two bins would be covered: `bin auto[58], bin auto[1082]`. It is not very visual what has been covered. It is time consuming and error prone to convert the decimal value into its binary representation, find the bit that toggled and look up its meaning in the documentation. Therefore another approach was needed.

Transition coverage at measurement cycles

A method has been developed that eases the specification of interesting conditions. Condition outcomes at consecutive measurement cycles (but many device cycles, and therefore even more clock cycles apart) are modelled through value transitions. Covergroups developed this way are easily maintainable and extendible, because coverpoints are created for each condition with descriptive names. This eases the analysis of coverage results. It helps to find out which bins were not covered and to modify the input constraints accordingly. New conditions and signals can be added conveniently. Regression coverpoints can be added this way as well.

SystemVerilog provides value transition cover bins. They can state value sequences of the monitored variable or expression, which happen at several sample points. Coverage sampling points for the Integration block need to be the device cycles, because the alarm reset signal can arrive at any of those. For all other conditions though, only value transitions happening at measurement cycles are relevant. Moreover, between measurement cycles the signals involved in these conditions are not expected to change. This DUV characteristic allows to use the value transitions measured at device cycles to obtain the value transitions over measurement cycles. Since values do not change outside measurement cycles, the values read at the last device cycle before a measurement cycle are the same as the ones that were present at the last measurement cycle. Of course this is only true if the DUV functions as specified. Therefore, when using this method, it is essential to add an assertion for each input and output that is not expected to change. These assertions need to ensure that the signals remain stable. It is possible to sample coverpoints only if certain conditions are true (eg. at a measurement cycle). This approach is not sufficient though, because the alarm reset signal needs to be sampled outside measurement cycles and combined with conditions that were present at the previous measurement cycle.

The developed method allows to reuse the same coverpoints for covering

- the combination of condition outcomes at two consecutive measurement cycles.
- the combination of condition outcomes at one measurement cycle with the alarm reset signal at any following device cycle.

Condition outcomes at two consecutive measurement cycles can be covered by stating a transition between two values. Their combinations over both measurement cycles can be automatically generated by cross coverpoints. Figure 5.3 shows an example of this approach. It shows three coverpoints. The first one, declared in line 1, models the condition outcome at the time of sampling. Bin "one" is covered if the condition is true. The second one, declared in line 7, models two value transitions. Bin "goesHigh" is covered if the condition was false at some point of sampling the coverpoint and if it was true at the next time when the coverpoint was sampled. The bin "goesHigh" is covered if the alarm condition was false at one measurement cycle, and true at the next measurement cycle.

The third coverpoint, declared in line 15, is a cross coverage of two bins shown. It is covered if the alarm condition was false at the previous measurement cycle, and is true at the current measurement cycles, and if the period ended at the current measurement cycle. In the coverage report and simulator GUI the cross bin is shown with its name "sameTime". If it is not covered, its meaning can be easily looked up in the coverage model source code. In this case further automatically generated coverage points have been prevented with an option (line 19). (This option is not clearly defined in the SystemVerilog language and therefore leads to a simulator warning. The Questa Simulator implements it in the way it was needed.) If automatic bin generation is desired, they would be shown as

```
bin<staysHigh,one>
bin<staysHigh,zero>
bin<goesHigh,zero>
```

It can be easily seen which condition outcomes are uncovered. This approach requires many more lines of code than the one mentioned before, but it increases maintainability of the testbench a lot, so it is worth the effort. Furthermore there are certain conditions that are impossible if the DUV functions as specified. Because each condition has its own coverpoint representation, it is possible to specify only the combinations needed. Like mentioned above, regression coverage can be easily added by identifying the conditions that lead to a fault and specifying its combination as a cross coverpoint.

```
1       cpIntPeriodEnd: coverpoint
2         (sequence_item.intElapsedTimexDO >= sequence_item.integralTimexDI) {
3         bins zero = {0};
4         bins one  = {1};
5       }
6
7       cpIntAlarmCondChanges: coverpoint
8         (sequence_item.integralxDO >= sequence_item.integralThresholdxDI) {
9         bins goesHigh  = (0 => 1);
10        bins staysHigh = (1 => 1);
11      }
12
13      /* At the same MT the alarm condition becomes true and the
14       * integration period ends */
15      cpIntAlarmCondIntEnd: cross cpIntAlarmCondChanges, cpIntPeriodEndChanges {
16        bins sameTime = binsof(cpIntAlarmCondChanges.goesHigh) &&
17                        binsof(cpIntPeriodEnd.one);
18
19        option.cross_auto_bin_max = 0;
20      }
```

Listing 5.3: Alarm Condition Coverage - Covergroups

Structural Coverage

Structural coverage collection could be simply turned on and off by simulator command line options. The Questa Simulator provides Focused Expression Coverage, which is known as Multiple Condition/Decision Coverage (MC/DC). This is the most effective structural coverage metric it provides. It is typically very difficult to reach 100% of this type of coverage [HV01].

When using the Questa Simulator, the file name containing the top module of the testbench, which instantiates the DUV, must not start with "uvm_". The simulator assumes excludes files starting with this prefix to avoid the collection of coverage data of the testbench. Unfortunately it also excludes any sub modules. The problem can be solved by using a different file name.

5.2.1.4 Simulation-based Verification - Results

Simulation-based verification has been tracked in tables as described in section 3.4. Several mismatches between the DUV and the reference model were found. Many of them happened due to synchronization problems between the two models, because they were written with different levels of abstraction. Furthermore there were several updates done to the DUV, even in its interface, which caused mismatches, but were no design faults.

15 mismatches were found that required further investigation. They are grouped by its cause in Table 5.5. There are still more mismatches that have not yet been analysed at the time of writing. Giving a count would not be meaningful, because faulty states propagate to future states. As mentioned, the design depends on its previous input signals and internal states.

Table 5.5: Simulation-Based Verification - Faults Discovered

Cause	**Count**
Faults in DUV implementation	9
Unspecified or ambiguous requirements	3
Unspecified design decisions	2
Faults in reference model	2
Total	**15**

The following list describes the DUV faults that were discovered. In some cases the visible mismatches on the outputs appeared to have the same cause. Analysis by the designer showed that there were several different faults causing these mismatches. Regression coverage points that described the exact input and output conditions confirmed that the mismatches were actually not the same.

1. DUV alarm output rose only for one clock cycle.
2. Alarm switched on unexpectedly.
3. Alarm was on, even though the alarm function was deactivated.
4. Rounding to was wrong for negative offset values.

5. Alarm reset was not effective, even though alarm function was deactivated (which means an alarm reset should be effective, regardless of the alarm condition).
6. Wrong rounding for offset values less than 2^{20}.
7. Another fault in rounding for offset values less than 2^{20}.
8. A third fault concerning offset rounding of small values.
9. Alarm switched on, because old integral value was compared to a new threshold value.
10. Another case where the alarm switched on unexpectedly.

Functional Coverage

Table 5.6 presents the results of the measured functional coverage. At the time of writing, not yet all mismatches between the DUV and the reference model had been analysed and corrected. In order to demonstrate the applicability of the chosen approaches and the implemented UVM testbench, coverage has been measured once for all tests, regardless of the test results and once for passed tests only. These numbers can be found in the second and third column of the table respectively. This means that the coverage data shown in the second column demonstrates the capabilities of the testbench. The data in the third column provides information on the verification progress.

The table lists the 4 covergroups as well as the cover properties. For each cover type the number of test runs that were needed to reach the shown coverage, are presented. Several UVM test and UVM sequence combinations with were executed with several different random seeds. The last column shows the time it took to reach the shown coverage. That means, after the listed number of test runs or time there could be no further improvements of coverage observed when further test runs were executed. These times and numbers of stimuli are not the minimal necessary. Directed test cases with stimuli directly targeting the coverage holes would require lower numbers. But the goal was not to achieve fastest coverage, but to find implementation faults. Therefore at first sequences were executed which send very random input. If the coverage percentage stalled, these test runs were stopped and more tightly constrained sequences were applied. The Questa SIM coverage database has been used to incrementally collect and combine the coverage information of all test runs.

It can be seen that the coverage of the robustness covergroup is very low. A further test run dedicated to cover the cgIntRobustness covergroup was executed for additional 38.11 hours of simulation, which increased total simulation time to 74.63 hours. It increased the coverage of the robustness covergroup only by 1.09%, to 8.24%. Another interesting observation, which is not visible from the table, was that the regression covergroup was only covered by 4 test cases, which used very similar sequences. It represented a very specific case related to the offset rounding.

The test runs were executed on the following two workstations:

- WORKSTATION1: 3.40GHz, 4 cores, 8GB RAM
- WORKSTATION2: 4.00GHz, 4 cores, 32GB RAM

Table 5.6: Functional Coverage of the Integration Block

Cover type	**Covered - all tests**	**Covered - passed tests**	**Number of test runs needed**[1]	**Number of stimuli applied**[1]	**Time needed (CPU time)**[1]
Cover properties	100.00%	100.00%	6	16355	5.72h
cgIntConditions	100.00%	93.98%	19	324647	39.00h
cgIntRegression	100.00%	100.00%	1	250	13.79min
cgIntValueRanges	91.95%	73.02%	18	249327	37.23h
cgIntRobustness	7.15%	6.02%	6	280977	28.94h
Total	**79.82%**	**74.60%**	**21**	**454200**	**39.89h**

[1] to reach the listed coverage with constrained random input stimuli.

5 of the test runs (16.76 hours) in table 5.6 as well as the additional 38.11 hours run for robustness testing were executed on WORKSTATION1. All other test runs were executed on WORKSTATION2. WORKSTATION2 was roughly two times faster. Therefore the total simulation time stated in the table can be probably reduced to 31.89 hours if all test runs were executed on WORKSTATION2. The additional robustness test run would probably need only around 19 hours. Which is still more than reasonable for achieving only 1.09% of additional coverage. What can greatly speed up simulation time is the disabling of cover properties. Executing the complete test suite on WORKSTATION2 with disabled cover properties and disabled assertion checks took less than 3 hours. Only 6 test runs actually increased the coverage of the cover properties. Once this number is known after a first test run with coverage collection, these tests can be scheduled first in regression test runs and the rest of the test suite can be executed with disabled cover properties. Assertion checks though should not be disabled! Cover properties need to be covered once, but for assertions it has to be shown that they are never violated during the whole run of the test suite. Certain sequences had a large contribution to the total runtime. Simulating the sequence that was detailed in the example in section 5.2.1.2 with disabled cover properties on WORKSTATION2 took still 50.81 minutes (129381 stimuli). A sequence that was dedicated to the integral reset signal combined with integration period endings needed 66.11 minutes (133018 stimuli).

Structural Coverage

Figure 5.4 shows the simulator output for code coverage. 100% statement and branch coverage has been achieved. FEC stands for Focused Expression Coverage. Only 95.34% are reported. There are 2 condition terms that contain a logical AND of two signals. When the Integration block is integrated into the whole system, these two signals are always high at the same time. On block-level this cannot be known. For system-level verification it means that there will have to be an assertion placed on the output signals that drive the Integration block, eg.

```
assert property ((signalA == 1) |-> (signalB == 1));
```

The toggle coverage is also quite low. When looking at the detailed report, it can be seen that only bits of internal signals for the integration period and the elapsed time value were never toggeled. This is because the very large possible integration times cannot be simulated within feasible time.

In table 5.6 the amount of hours that was needed to reach only 7.11% of the robustness tests could be seen. The low coverage in both cases has the same cause.

```
==================================================================================
=== File: integration.vhd
==================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           47        47         0    100.00
    Branches                        37        37         0    100.00
    FEC Condition Terms             43        41         2     95.34
    FEC Expression Terms             2         2         0    100.00
    Toggle Bins                   2518      1960       558     77.83

Total Coverage By File (code coverage only, filtered view): 94.63%
==================================================================================
```

Figure 5.4: Simulation-based Verification - Structural Coverage

The verification of safety-critical systems should reach 100% coverage. Impossible conditions, like the missing FEC condition, could be safely excluded from the coverage collection, because the device configuration (block instantiation and connection) in that case is not configurable. Another configuration would require system-level verification of that other architecture. What is more problematic is the missing functional and structural coverage for the integration and measurement periods. If no measure for verification can be found, the system specification will have to be adapted and the device usage will have to be limited to the verified cases.

5.2.2 Formal Verification

For formal property verification the tool Questa PropCheck has been used. It belongs to Mentor's Questa Formal Verification tool suite [Cor18b]. The choice was made because Mentor's Questa SIM simulator [Cor18c] had been chosen for simulation-based verification, so it provided some advantages to stay with the same product family. One advantage was that both tools can write into a combined coverage database. Another advantage was that the installation, tool setup and tool familiarization took less time than for a completely different tool, because there were some similarities with Questa SIM. The tool choice did not influence the writing of the SystemVerilog Assertions (SVA) or impose the language decision. Property Specification Language (PSL) could be used equally for verification of VHDL code. Some assertions have been written in PSL in VHDL flavour in order to double check some initial SVA assertions. The SVA language was chosen for the reasons mentioned in section 3.3.2. Any other formal property verification tool that supports mixed-language verification could have been used for verifying the VHDL design with the SystemVerilog code that has been written.

Figure 5.5 shows the tool chain used for formal property verification. A shell script has been created that copies the latest DUV version into the verification software directory. The git repository of the DUV has to be checked out and the path has to be provided to the shell script. The script then updates the repository and calls an encryption script created by the designer that instructs the Questa Simulator to generate a ".vhdp" file, encrypted with a Questa internal key.

This file contains all DUV blocks that have been specified in a configuration file. Only the entity declarations stay in plain text. The verification engineer does not need to open any plain VHDL file, all information she needs for black-box verification are the port declarations. This way higher independence between the design and verification engineer could be established. The encryption script also appends the git hash of the last git commit to the end of the file. This was done to enable the reconstruction of verification activities as required by the IEC 61508 standard. When the verification code is updated in its own git repository, the copied protected VHDL file is included and it can be easily related to the corresponding point in the DUV's repository's history.

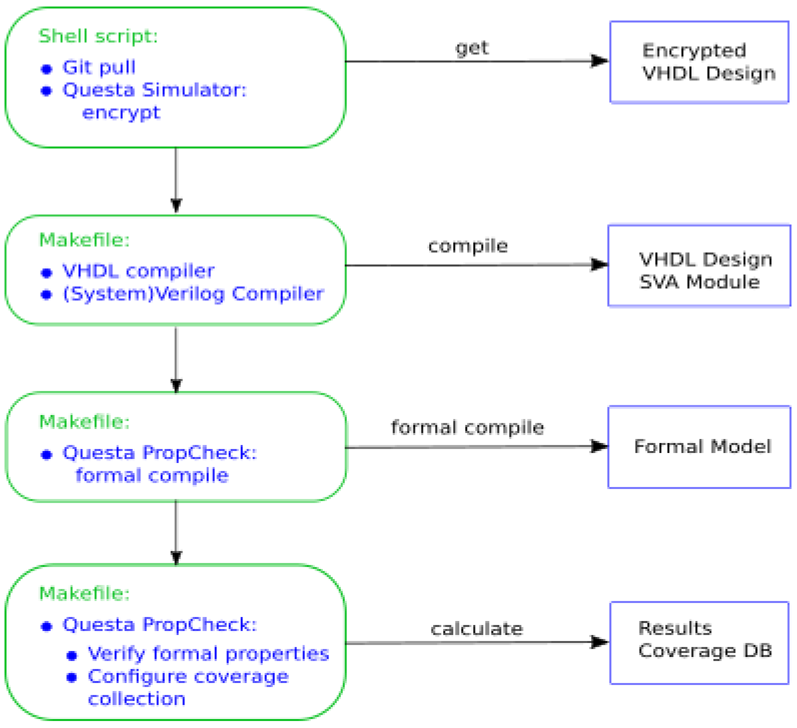

Figure 5.5: Formal Verification Toolchain

A Makefile then compiles the encrypted VHDL Design Under Verification (DUV) with Questa's VHDL compiler. Since the VHDL compiler belongs to the same Questa tool suite, it knows the key for decryption. The Makefile then compiles the SystemVerilog modules that contain the assertions and additional verification code. The SystemVerilog module gets bound into the VHDL design by using the SystemVerilog "bind" statement. This creates an instantiation of the SytemVerilog verification module inside the VHDL design. All ports of the SystemVerilog module are input ports. They are connected to all input and output ports of the VHDL design. If the VHDL code would not be encrypted and if white-box testing would have been the chosen approach, the SystemVerilog module could also access internal signals. The access possibilities of internal signals inside VHDL code through a bound SystemVerilog module is simulator dependent [Cor18c].

Then the Makefile passes a Tcl script to the Questa Formal Verification tool suite. The Tcl script triggers formal compilation of the DUV and the verification code, which generates a combined formal model. Then it invokes PropCheck that performs model checking for proving that the properties hold or showing that they can be violated. A separate Makefile target has been created for performing the model checking together with structural coverage collection and generating a coverage report afterwards. It is more efficient to call coverage collection in a separate run, because it increases the execution time of the model checker. The formal compilation and PropCheck generate several reports and result database files. Results of property proofing and the collected structural coverage can then be stored into the Mentor's Unified Coverage DataBase (UCDB). This database can combine coverage information from formal and simulation-based verification.

For debugging, the formal database can be loaded into the Questa Formal Graphical User Interface (GUI). If a property "fired", which means there exists a legal input sequence that violates the property, a counter example is presented as a waveform. For proven properties no counter example is generated, because a proven property holds for any legal input sequence. In both cases the assumption statements decide which input sequences are legal. In order to verify whether the proven property is not a false positive it can be transformed into a cover property by changing the SystemVerilog keyword "assert" to "cover" and rerunning PropCheck. Usually the goal of using cover properties is to find a legal input sequence that fulfils the cover property. If the property was already proven, one knowns that PropCheck will be able to find such a stimulus and generate a waveform. This can be helpful to debug vacuously proven properties. A property is vacuously proven if it is always true, because the DUV actually did not receive any inputs and remained in a stable state. This can mean for example that the input assumptions were not adequate [Cor18a].

Formal Property Verification of the Integration Block

Formal properties have been written for the Integration block of the CROME Measuring and Processing Unit (CMPU). In figure 5.6 the formal netlist statistics printed by the formal compiler is shown for the Integration block. The design consists of roughly 25 000 logic gates. The Questa Formal Verification tool suite supports designs of up to 250 000 logic gates [Cor18a], so the DUV should be suitable for formal verification on block level. On full system level the gate count could come close to the limit.

A SystemVerilog module that contains the assertions for the Integration block has been written. This module was bound into the VHDL Integration block as was shown in figure 3.6. All ports of the VHDL block were of type std_logic or std_logic_vector. Ports of type std_logic were connected to SystemVerilog ports of type bit. The std_logic_vector ports were mapped to integer ports of different lengths and of signed or unsigned representation according to the requirements and design architecture specification. Ports that have lengths different from the predefined SystemVerilog data types had to be specified as vectors of type "reg". Otherwise the connection to the DUV signals would have failed during compilation. In case these ports were signed, they had to be explicitly cast to signed whenever they were used in any properties or sequences.
The workflow presented in section 3.3.2.1 has been used for writing formal assertions. Step by step natural language properties have been taken from the verification plan and translated into SVA properties by using the mapping defined in table 3.3. Initially there were no input assumptions written, apart from the ones necessary for representing the correct data types. As mentioned in section 3.3.2.1, the idea was to specify as little input assumptions as possible, to verify the design

```
---------------------------------------
Formal Netlist Statistics         Count
---------------------------------------
Control Point Bits                  202
  DUT Input Bits                    201
  Cut Point Bits                      0
  Black Box Output Bits               0
  Undriven Wire Bits                  1
  Modeling Bits                       0
State Bits                         1590
  Counter State Bits                288
  RAM State Bits                    480
  Register State Bits               427
  Property State Bits               395
Logic Gates                       25361
  Design Gates                    23422
  Property Gates                   1939
---------------------------------------
```

Figure 5.6: Formal Netlist Statistics of Integration Block

for as many input scenarios as possible. As expected, the first assertions failed for input scenarios that were not considered to be possible when the Integration block is placed into the whole CMPU design. It turned out though that the design only allows very small deviations from the expected case and 24 input assumptions were necessary. This does not degrade the verification results, as long as the Integration block will be integrated into the whole system as specified and the input assumptions will be used as assertions on the output signals of the blocks which drive the Integration block.

7 of the input assumptions forced the initial state of the input parameters to all zeros, to avoid the propagation of 'X' values into the design. Another 7 assumptions ensured that the input parameters would only change when a certain testbench signal rises. 1 assumption ensured that the parameter value ranges would not be violated. Furthermore, the following relationships between input signals had to be ensured by input assumptions:

1. The testbench internal signal that indicates when parameters might change, stays high for a certain number of clock cycles (determined by an internal design delay).
2. As long as this signal is high, no measurement cycle starts and the integration start signal does not rise.
3. This signal never rises within a measurement cycle.
4. When a measurement cycle starts, this signal ceases.
5. Input parameters do not change immediately after a measurement cycle finished.
6. A measurement cycle is a certain number of clock cycles long (determined by an internal design delay).
7. A start of a measurement cycle is followed by an integration start signal.

8. There is exactly 1 integration start signal inside a measurement cycle.
9. The integration start signal is a pulse.

The first idea was to specify the input assumptions as relationship between signals. This would have lead to proofs that are valid for all possible input conditions. Expressing such relationships as SystemVerilog assumptions increases the computational complexity for the formal tool, when it attempts to prove the specified properties while not violating the assumptions. This approach had to be abandoned, because the formal tool quit with an error message, saying that the assumptions were too complex to be handled. Therefore they had to be specified much more statically with fixed signal delays.

11 cover properties were written as well to validate the assumptions. This should ensure that the assumptions would allow all legally expected stimuli, since any proof is only valid for stimuli that do not violate the assumptions. These properties modelled some aspects of the expected design behaviour. They ensured for example, that parameters could change their value or that it was possible that more than one "integration start" signal arrived between two measurement cycles.

The Integration block contains several calculations and logical value combinations that influence its most critical output, the integration alarm signal. The calculations and logical value combinations depend on many previous internal states, possibly spanning several thousand clock cycles. Questa PropCheck can only handle SystemVerilog sequences that span a few clock cycles, ideally not more than 10 in the antecedent of an implication [Cor18b]. Therefore some procedural SystemVerilog "always" blocks have been added to the formal verification module. This code is compiled into the formal model as if it were part of the DUV. It collects signals of interest, which are then used in properties for comparison with the real DUV's outputs. One could argue that this code could contain faults, but the same is true for the properties themselves. Formal property verification can proof that properties of a design are always fulfilled, but it can't proof that the properties are fault free.

5.2.2.1 Formal Verification - Results

Formal property verification has been tracked in tables as described in section 3.4. All formal proofs were executed on WORKSTATION2 (4.00GHz, 4 cores, 32GB RAM), the same as mentioned in section 5.2.1.4.

8 properties could be fully proven on the design. In the following they will be described:

1. **assert__pIntAlarmOn**
   This assertion proves that whenever the alarm condition is true in a measurement cycle and the alarm function is activated, the alarm switches on. Since formal verification is exhaustive, this means that the alarm is always generated as specified, regardless of the values of other parameters. As mentioned in section 4.2.1, there are 16 384 possible combinations of input signals values combined with previous output signal values that can influence the alarm state. None of them is allowed to influence the alarm signal if the alarm condition is true

within a measurement cycle, and the alarm function is not deactivated. This assertion has been proven for all possible combinations within some minutes. In comparison one of the simulation sequences for covering some of the input conditions regarding integral resets had a runtime of 66 minutes.

2. **assert__pIntAlarmOnIfCondTrue**
This assertion ensures that an alarm, once switched on, stays on at least until the next measurement cycle. It ensures that whenever the alarm condition is fulfilled outside a measurement cycle (and the alarm function is not deactivated), the alarm stays on. This implies automatically that an alarm reset signal cannot reset an alarm in this case. The proven property does not need to use the alarm reset signal. It proves that the alarm reset signal, as well as any other input signal, has no influence on the alarm in this case. This and the previous assertion are both proven for all 2^{128} possible integral value and threshold value combinations. This was proven in less than a minute. The corresponding simulation sequence was described in the example in section 5.2.1.2. It took nearly an hour to simulate it.

3. **assert__pIntEtStableBetweenMT**
This assertion ensures that the elapsed time value does not change between measurement cycles. This is particularly important to support simulation-based verification. Coverage collection relies on the trueness of this assertion. Values are only collected at measurement cycles. If this assertion is proven, the simulation testbench does not need to monitor this output signal between measurement cycles. This significantly reduces the number of needed comparisons between the DUV and the reference model.

4. **assert__pIntIntegralStableBetweenMT**
This assertion is similar to the one above. It proves that the integral output value does not change between measurement cycles.

5. **assert__pIntAlarmResetBetweenMT1**
This assertion ensures that a latched alarm can be manually reset outside a measurement cycle, as long as the alarm condition is false or the alarm function is deactivated.

6. **assert__pIntAlarmResetBetweenMT2**
This assertion ensures that an alarm reset outside a measurement cycle is not effective if the alarm condition is true and the alarm function is activated. This proof is actually not necessary anymore, because this case has already been proven with "assert__pIntAlarmOnIfCondTrue". Initially the proof of the latter property did not conclude. On a more powerful computer it could be proven.

7. **assert__pIntThreshZeroNoAlarmGoesOn3**
This assertion proves that whenever the alarm switches on, the alarm function must be activated. The actual intent of this proof is to show that the alarm state cannot change from off to on if the alarm function is deactivated.

All of these assertions were placed into the simulation-based verification software as well. Non of them was ever violated.

3 properties were inconclusive. In the following they will be described:

1. **pIntManualReset**
   This property intends to prove the correct output value of the integral value at the time of a manual integral reset. The difficulty is that the integral value is calculated over all previous measurement cycles since integration start, which can be 10 years long. Since a property cannot span that many clock cycles, the integral value has been modelled in the formal verification module. The complete integral calculation algorithm has been modelled. A testbench internal integral value is calculated on each measurement cycle. It is used for comparison with the output value in this property.

2. **pIntIntegralValue**
   This property intended to proof the equivalence of the integral value calculated by the DUV and the one calculated in the testbench. After 29 hours of calculation with full computing power the tool had only calculated the correctness of the proof for 44 clock cycles. To give a comparison: The formal tool reported intermediate proof radii of the proven properties of around 6000 clock cycles. Once a property is proven, it is proven for any number of clock cycles.

3. **pIntElapsedTime**
   The elapsed time count can span as many measurement cycles as the integration calculation. Therefore it was also modelled in the testbench. This property tried to prove the equivalence between the DUV and testbench signal for the elapsed time. As the one above, it did not conclude.

Table 5.7 summarises the cases in which properties fired unexpectedly. In 3 cases the cause was a fault in the DUV. They were all related to the alarm signal. In one case it toggled before settling to its final output value. In the other two cases it rose for 1 clock cycle at an unexpected time. In simulation these fault would have never been found, because the output signal comparisons happened only once, several clock cycles after they were expected to be ready. The severity of this fault depends on the connected module or device which receives the alarm signal as input. It depends on when it reads the alarm output. There was 1 case where the designer and requirements engineer had agreed on an addition to the requirements, but had not documented it. In another case the input assumptions allowed signal sequences that will never occur if the Integration block is integrated into the complete design as foreseen. This restriction was also not obvious from the specifications. The adapted input assumptions will have to be placed on the outputs of the block that will drive the Integration block on system-level to ensure that this case will not happen in real system operation. In 2 cases the design specification was not detailed enough. It was usually detailed enough for simulation-based verification, but in formal property verification each clock cycle needs to be accurately described. In 4 cases there were faults in the implementation of the properties and test module. Since there are so many conditions that can influence the alarm state, it is not trivial to specify a property that always holds. Each and every special case and parameter constellation that leads to a different alarm state than expected in the normal case, needs to be specified in the property. That is why it was possible to prove the first two assertions mentioned above that regarded the alarm condition (pIntAlarmOn and pIntAlarmOnIfCondTrue). The alarm condition together with the alarm activation overwrite all other conditions and input parameters, Some more properties have been written. Therefore it was possible to find a general valid statement that could be written as a property.

Table 5.7: Formal Verification - Faults Discovered

Cause	**Count**
Fault in DUV implementation	3
Unspecified or ambiguous requirements	1
Unspecified design decision	2
Input assumptions allowed unexpected DUV usage	1
Fault in property	2
Fault in the calculation of reference signals used by properties	2
Total	**11**

Functional Coverage

Functional coverage can be calculated according to the formula stated in section 3.3.2. 8 out of 26 properties could be proven, therefore functional coverage calculates to:

$$\textit{Functional coverage[\%]} = \frac{8}{26} * 100.00 = 30.77\%$$

Structural Coverage

To enable structural coverage calculation, the unencrypted design had to be used. Figure 5.7 shows the structural coverage report of the formal tool. The cumulative summary includes all sub-blocks that are instantiated by the integration block. These numbers are not expected to reach 100%, because it is not of interest to verify the sub-blocks. They need to be verified on block-level. The instance summary shows the structural formal coverage of the Integration block. These numbers only include elements that have been used for the proof. They represent a subset of the Cone-Of-Influence (COI) of the properties. The COI could as well contain signals that are not relevant for the proof, but that have been used in the property statements. The percentage of verified inputs roughly matches the functional coverage. The coverage of internal design elements is much higher. This shows that structural coverage alone cannot be used to asses the progress of verification.

Figure 5.8 presents the process statistics printed by the formal tool for the proof run with coverage calculation. The tool was run with 8 parallel formal engine processes. The highest peak memory usage per process was 16.7 GB. That means that the computer on which the proofs are calculated needs to have at least 16.7 GB of RAM available for one process. According to the tool support, the tool is not expected to conclude on proofs if there is not sufficient RAM available.

```
----------------------------------------------------------------
Questa Formal Coverage Report
----------------------------------------------------------------
Questa Formal Version 10.7c_3 linux_x86_64 14 Nov 2018
----------------------------------------------------------------
Report Generated : Thu Jan 24 12:24:31 2019
----------------------------------------------------------------
Hierarchy : integration
Number of assertions used to compute coverage: 8
Proven Assertions :
  svaBind.assert__pIntAlarmOn
  svaBind.assert__pIntAlarmOnIfCondTrue
  svaBind.assert__pIntEtStableBetweenMT
  svaBind.assert__pIntIntegralStableBetweenMT
  svaBind.assert__pIntAlarmStableBetweenMT
  svaBind.assert__pIntAlarmResetBetweenMT2
  svaBind.assert__pIntAlarmLatchedOnUntilReset2
  svaBind.assert__pIntThreshZeroNoAlarmGoesOn3
----------------------------------------------------------------
Formal Coverage Cumulative Summary
----------------------------------------------------------------
                           Total          Formal       % Covered
                                         Covered
----------------------------------------------------------------
State Bits                  1281             296           23.1%
Nets                        1086             246           22.7%
Assignments                   54              33           61.1%
----------------------------------------------------------------
Formal Coverage Instance Summary
----------------------------------------------------------------
                           Total          Formal       % Covered
                                         Covered
----------------------------------------------------------------
Inputs                       203              70           34.5%
Outputs                       97              97          100.0%
State Bits                   328             230           70.1%
Nets                         377             246           65.3%
Assignments                   46              32           69.6%
----------------------------------------------------------------
```

Figure 5.7: Formal Verification - Structural Coverage

```
--------- Process Statistics -----------
Elapsed Time                    5250 s
-------- Orchestration Process ---------
---------- WORKSTATION2:19450 ----------
CPU Time                           3 s
Peak Memory                      0.2 GB
---------- Engine Processes ------------
---------- WORKSTATION2:19475 ----------
CPU Time                        5182 s
Peak Memory                     14.0 GB
CPU Utilization                   98 %
---------- WORKSTATION2:19474 ----------
CPU Time                        5203 s
Peak Memory                      0.4 GB
CPU Utilization                   99 %
---------- WORKSTATION2:19476 ----------
CPU Time                        5200 s
Peak Memory                      0.4 GB
CPU Utilization                   99 %
---------- WORKSTATION2:19477 ----------
CPU Time                        5194 s
Peak Memory                      0.8 GB
CPU Utilization                   98 %
---------- WORKSTATION2:19470 ----------
CPU Time                        4085 s
Peak Memory                     16.7 GB
CPU Utilization                   98 %
---------- WORKSTATION2:19479 ----------
CPU Time                        5199 s
Peak Memory                      6.1 GB
CPU Utilization                   99 %
---------- WORKSTATION2:19478 ----------
CPU Time                        5197 s
Peak Memory                      1.1 GB
CPU Utilization                   98 %
---------- WORKSTATION2:19473 ----------
CPU Time                        5210 s
Peak Memory                      0.8 GB
CPU Utilization                   99 %
----------------------------------------
```

Figure 5.8: Formal Verification - Process Statistics

5.3 Summary of Verification Results and Discussion

Table 5.8 summarizes all faults that were discovered when applying the verification methodology defined in chapter 3 to the CROME Measuring and Processing Unit(CMPU). 12 faults were found due to reviews and the specification of semi-formal properties in natural language. 26 faults were found due to automated verification techniques. Even though with automated methods more faults were discovered and even more are expected to be found, the number of faults discovered by reviews can not be neglected. It shows the importance and capabilities of reviews and communication between the involved engineers. Those review findings that lead to updates of the specification and implementation could save some development and all corresponding debugging effort during verification. The ones that lead to an update of the verification code came mostly from ambiguously specified functionality, which emphasized the need for (semi-) formal specifications. The more knowledge was gained about the design, the more not or ambiguously specified details were found. Like mentioned in section 5.1.4, the finding that was made when discussing the natural language properties with the requirements engineer would have never been found by automated verification techniques. The design and verification engineers had both interpreted the requirements in the same way. All tests would have passed. However, the requirements engineer intended to specify something else, which means the DUV would have not conformed to its specification and therefore the test should have failed.

Table 5.8: Verification Findings - Integration Block

Found by	**Update of Specification**	**Update of Implementation**	**Update of Verification Code**	**Total found by method**
Review of Requirements Specification	4	5	6	**9**
Natural Language Properties	1	1	1	**1**
Review of Design Specification	1	2	0	**2**
Simulation using UVM	5	9	2	**15**
Formal Property Verification	4	3	4	**11**
Total	**15**	**20**	**13**	**38**

The results of constrained-random simulation using UVM and Formal Property Verification (FPV) cannot be quantitatively compared for two reasons. First, the amount of time spent in formal verification was only 2/3rds of the time spent in simulation-based verification. Secondly, formal verification was started later. This was mainly due to the fact that the initial development of the UVM testbench and the reference model took quite long. One consideration was to stick with UVM only, because using two different verification techniques means that two software frameworks need to be maintained. This is currently and also in the near future done by a single person. The question was whether the benefit of formal verification would justify the overhead. Finally, the choice of adding formal property verification with SystemVerilog assertions was made because of several reasons. Concurrent cover properties had already been used in simulation, therefore there is no additional overhead added for learning the SVA language. Furthermore, the risk of false positives is reduced in comparison to other formal verification techniques, because the formally proven properties can be verified in simulation. There is hardly any effort required to connect a formal property module to a DUV, even of a different language, since SystemVerilog provides the "bind" statement that does this connection smoothly. The only initial effort required was for writing the input assumptions, though this activity was already part of the verification, since it required a detailed inspection of specifications and it could reveal unspecified implementation details. That means nearly all development effort can be dedicated to actual design verification.

Most faults in the implementation have been found with simulation-based verification. Since formal verification was started later, it cannot be judged whether some DUV faults that had already been found at that point would not have been found equally with formal verification. The initial faulty DUV could have been used to study this, but the goal of this thesis was not the assessment of these two verification techniques, but the definition of an applicable verification methodology. An interesting observation can be made when looking at the cause of the discovered faults. Most failures found with constrained-random simulation using UVM came from actual faults of the DUV. Most failures found with FPV came from faults in the properties or ambiguously specified requirements. Formal properties need to be stated very accurately on clock cycle level and all input conditions that could lead to a different expected outcome need to be considered. If there is a general case which shall be proved and there are several exceptions or special cases, all of the latter need to be considered in the property specification. Considering the large number of input and state combinations that influence the alarm state (2^{14}) of the CMPU, this is a very challenging task. Furthermore, the requirements were specified on a very high level. The low-level design specification might contain more details. However, to keep independence between the design and verification engineers high, it should not be read by the verification engineer if it can be avoided.

A qualitative comparison between constrained-random simulation using UVM and FPV is provided in table 5.9. Constrained-random simulation can find faults that are not even considered in the coverage model. This was shown in section 5.2.1.2. Several regression coverage points had to be added to the coverage model that would ensure that those very specific scenarios would in future be verified additionally to the ones described in coverage points that are based on the specifications. This is generally considered to be a strength of constrained-random simulation [But15].

The CMPU operates on large input value ranges over very long operating periods. Input values that are immediately used as inputs to mathematical operations can be verified over their full ranges with the developed UVM testbench. Verifying the effect of some input parameters required tight constraints and relatively long simulation runtimes, but they were still feasible within reasonable time. FPV did not conclude for any of the mathematical operations. Unfortunately both verification techniques applied could not handle the long operating times of the device, which means that neither 100% functional coverage nor 100% code coverage could be achieved. Concluding from passing test cases for short operating periods that the device would also function as specified for long periods is not possible. It is important to state that this conclusion should not be drawn. It cannot be guaranteed that there would be no internal overflows or any other faults when certain input sequences are applied over a long period. Independent black-box verification was chosen, which means that any conclusions can only be drawn from observable outputs. Even though it can be calculated that no overflows happen when the input and output port widths are used, it cannot be shown eg. that there are no smaller bit widths used for internal signals. What this further means is that independent black-box verification of the CMPU is not sufficient. It needs to be complemented either by other black-box techniques, or by white-box techniques such as code reviews and walkthroughs. The developed FPV module could be extended to use cut points. This is a white-box verification technique that "cuts away" parts of the design and allows to use internal signals as inputs. This could be applied to the internal registers that store the integral and elapsed time value. A proof would then consider all possible values of these registers [Cor18b]. Where FPV outperformed simulations was on logical operations. FPV could prove that alarms would always be generated as specified, for any integral and threshold value, which are both 64 bit wide ports. As the example in section 5.2.1.2 showed, there was specific effort needed to cover this function in simulation-based verification and it was one of the sequences that had to run for the longest time. A similar technique could be developed inside the UVM testbench since SystemVerilog is able to access DUV internal signals [Cor18c]. If white-box verification is conducted by the independent verification engineers, this should be their last activity after all black-box activities, so that they gain no internal design knowledge before finishing these. This increases independence and therefore increases verification quality [AGHH99].

Table 5.9: Formal Verification - Qualitative comparison of constrained-random simulation using UVM to Formal Property Verification

Comparison criteria	**Constrained-random simulation using UVM**	**Formal Property Verification**
Fault discovery	• Finds unanticipated faults (outside coverage specification).	• Finds faults related to the stated properites. • Finds faults on clock cycle-level. Might be more accurate than necessary.
Design challenges: large bit widths and long operating times	• Can handle large bit widths and complex mathematical operations for signals that change frequently. • Cannot handle large operating times.	• Cannot handle large bit widths used in complex mathematical operations. • Cannot handle large operating times if mathematical operations are involved. • Can exhaustively prove logical operations.
Known whether a fault has been removed.	Yes, by repeating exact stimuli.	Hard to say, because a different counter example might be found first on updated DUV.
Regression	Regression coverpoint might be hit by unrelated stimuli if not stated specific enough.	Rerun of proof guarantees that a property is never violated by updated DUV.

Table 5.10 compares constrained-randdom simulation using UVM to FPV in terms of human and computing resources, which in the end for an organization are financial resources. The initial development effort to set up a mixed-language simulation environment, develop a reference model and the base classes of the UVM testbench can take quite long. For this thesis it was around 1.5 months. FPV can become productive within a week. Once verification activities start, constrained-random simulation leads to results in shorter time. There can be simulation results available within seconds. The same is true for counter examples of failing formal properties. More

hidden faults might appear later in simulation, but no applied sequence ran for much more than one hour. That means that within one hour all passing and failing test results are available. The proof of a single formal property can already take one hour or more time. Some properties are proven within minutes, but others can run for several hours without conclusion. For inconclusive properties it is hard to tell whether they would find a proof or a counter example. The proving process needs to be run for some time (often hours) and progress as well as proof depth need to be monitored in order to judge when to stop it manually and add the property to the list of inconclusive ones. It can easily happen that the proof did not conclude because the property statement was wrong. Updating it and rerunning the proof is a lengthy process. Since properties have to be completely accurate they are not easily reusable for non-standard interfaces. Even though the CMPU has several design blocks with similar characteristics, hardly any property will be reusable between blocks because of many different special conditions. Since the interface are similar, the UVM testbench will be easily reusable. Modifications will be mainly necessary in signal connections and constraints. Expensive tool licences are necessary for both methods, but in terms of needed computing power, UVM is by far cheaper than FPV.

Table 5.10: Formal Verification - Comparison of resources needed for constrained-random simulation using UVM to Formal Property Verification

Comparison criteria	**Constrained-random simulation using UVM**	**Formal Property Verification**
Initial setup	Lot's of effort: • Development of reference model • Development of UVM testbench (minimum 14 SystemVerilog classes) • DPI connection between testbench and reference model • Synchronization between DUV and reference model	Minimum version requires • 1 verification module • 1 SystemVerilog bind statement • Makefile
Results availability	Most of the time within seconds or minutes.	1 proof can take several hours. It might fail or never conclude.
Faults in verification code (once the input constraints/assumptions conform to the specification and the DUV and reference model are synchronized.)	Faults only expected in reference model. Because of higher level of abstraction: • Typically fewer faults [Ber03]. • Easier to correct.	Faults in properties happen often: • Complicated logic statements combined with sequential regular expressions necessary to account for every input combination in the COI of the property. • Clock cycle accurate description necessary.
Reusability (for other blocks of the CMPU)	Yes, small modifications necessary.	No, too specific.
Needed computing resources	Sufficient to run simulation and develop code simultaneously • 1 core per test run • 8 GB RAM	Inconclusive proofs with • 4 cores/8 processor threads • 32 GB RAM Typical workstation [BT16] • 12 cores • 100 GB RAM

Chapter 6

Conclusion and Outlook

6.1 Conclusion

The core of this thesis was the definition of a SIL 2 compliant methodology for the verification of safety-critical electronic designs. It has been demonstrated and evaluated on a subset of the CROME Measuring and Processing Unit (CMPU).

The methodology encompasses all steps required by the IEC 61508 standard for functional safety of electrical/electronic/programmable electronic safety-related systems [ESC10] for the behavioural verification of SIL 2 classified programmable electronic devices. It adds 5 additional techniques:

- Semi-formal methods for verification requirements specification (natural language properties).
- Requirements traceability for programmable electronic designs.
- Constrained-random inputs for simulation.
- Regression coverage.
- Formal Property Verification (FPV).

Independent black-box verification has been chosen as overall technique. A workflow has been defined that interlaces review and planning activities with preparatory work for the automated verification activities. It was suggested to start with reviews of the relevant requirements and design specifications, without going into detail of low-level design specifications, to keep independence high. Based on the knowledge gained, a reference model with a higher level of abstraction than the Design Under Verification (DUV) can be written, which further increases knowledge of the design. Then verification requirements can be written that state the functions to be verified and create a link to the functional and safety requirements specification. There is no need to specify test cases or input stimuli for the chosen automated verification techniques. It could be shown that specifying semi-formal properties in natural language as preparation for formal verification also benefits simulation, because formal languages do not leave space for interpretations [SSK15]. A table for translating the natural language properties into SystemVerilog properties has been provided. It could even be shown that these reviewing and planning activities revealed faults that could have never been found by automated verification techniques. There was a case where the design and verification engineers interpreted an ambiguously specified requirement in the same way, while the intent of the requirements engineer was a different one. All test cases or proofs would have completed successfully because the same fault would have been present in both the DUV and the verification code. During FPV, functional coverage was measured in terms of proven properties. For simulation-based verification a coverage model written in SystemVerilog was necessary. A grouping into Mutually Exclusive and Collectively Exhaustive (MECE) cover types has been provided, that helps to identify incomplete models.

After this preparatory work, the verification software can be developed. This has been done for the Integration block of the CMPU and prepared for further block-level and system-level verification. A flexible and extendible simulation-based testbench has been developed in SystemVerilog. It made use of the Universal Verification Methodology (UVM) concepts and library. A reference model has been written in C and seamlessly integrated into the testbench trough the SystemVerilog Direct Programming Interface (DPI). Several development concepts have been presented in this thesis. These include concepts for flexibly modifying constraints, coverage collection over large numbers of clock cycles and coverage for regression testing. Several faults in the DUV could be found with constrained-random simulation. It has been shown that the runtime of simulation with disabled cover properties is more than 10 times lower. For a design like the CMPU, cover properties should be avoided or executed only once at the beginning of a regression test suite run.

A workflow for formal property verification has been defined and demonstrated on the Integration block of the CMPU. 8 properties could be formally proven and some faults in the DUV could be found. Due to the large operating times of the CMPU, 100% coverage could not be achieved with the chosen black-box verification techniques. White-box verification techniques have been suggested as addition. The missing conclusions will most likely have to be drawn from code reviews, because it is unlikely that automated verification techniques can achieve full coverage of such large chains of consecutive device states.

Verification documentation is generated largely automatically during both simulation and formal verification. Several scripts execute the verification suites and collect and combine the various outputs of the simulator and formal tool. This enables the reproduction of verification results and the execution of regression test suites. It also generates backward traceability from verification results over verification requirements to functional and safety requirements. Templates for documentation tables have been provided. All software code and documentation produced during verification activities were additionally stored in a Git repository. This provides evidence for safety-certification as well as an additional guarantee for reproducibility of verification results.

In the last section the results from applying the defined verification methodology on the CMPU have been discussed. It could be seen that verification planning was an important activity that also led to the discovery of design faults. A comparison between constrained-random simulation using UVM and formal property verification has been provided. It can be concluded that the two techniques complemented each other very well. A benefit was that the same development language, SystemVerilog, could be used for both.

6.2 Future Work

Building on the work of this thesis, the next step will be the completion of the independent black-box verification of the Integration block as far as possible. This means that simulation-based verification will be done until no more faults are found with all test cases in the regression test suite that was used to measure the achievable coverage. Further effort will be put into formal property verification as well. Since some logical conditions could be proven, it can be expected that it will be possible to proof at least some of the remaining ones.

Once these techniques lead to no more further results, white-box testing techniques will be applied to close functional coverage completely. One possibility that was already mentioned is the extension of FPV with cut points. It might allow proofs of the value calculations spanning over long operating periods, because the state keeping logic would be removed and replaced by an artificial input port. Therefore all internal states of the cut out register could be verified in one device cycle. The large bit widths would have to remain tough, because verifying shorter parameter widths instead would not be meaningful. Further formal verification techniques can be assessed as well. Combinational equivalence checking might be a good candidate. The developed C model could be reused as reference model. Emulation might be considered instead of simulation, but it might be similarly limited in terms of possible operating lengths due to internal delays inside the DUV that need to be respected. If the mathematical operations spanning over long periods cannot be verified by any technique, a characterisation of the parameter ranges and interactions, together with the possibilities for their verification within reasonable time will have to be made. Input values and operating scenarios that cannot be verified can be stated in a safety manual, which describes the safe usage of a safety-related device [ESC10].

Once the Integration block has been fully verified as far as possible, the defined verification methodology will be applied to further blocks of the CMPU. After that, verification will be done on FPGA system-level. As soon as the triple modular redundancy voting is implemented, fault-injection techniques will have to be applied as well. Furthermore, the integration of the FPGA together with the embedded software will have to be verified. One option could be to include the software functions that perform the parameter transformations into the UVM testbench through the DPI interface. Input stimuli can be applied to the parametrization functions first. Their results can be used as DUV inputs. The DUV outputs can be passed to the software readout functions and the final results could be compared to the expected ones, calculated by a software-system level reference model. The latter can be an extension of the already developed C reference model. Another option for system-level verification might be the new Accellera standard "Portable Stimulus" [7]. Once the software and FPGA integration has been verified, further system-level verification can be done with the C model replacing the FPGA design. This should allow faster runtimes, since no more simulation of HDL designs will be necessary [SSM$^+$02].

Requirement management tools might be used in future instead of spread sheets for automating the connection between the various tables. The DPI connection between the C model and the UVM testbench contains a lot of repetitive code. It can be automated either by a custom script or by using eg. the MATLAB HDL Verifier [11]. The translation from natural language properties into SystemVerilog is a another good candidate for further automation. The safety-related weaknesses mentioned in section 5.1.1.2 will have to be removed, possibly by the mentioned techniques, though this is to be done by the design engineers. Once available, these modifications will have to be verified as well.

Literature

[AGHH99] Arthur, J. D. ; Groner, M. K. ; Hayhurst, K. J. ; Holloway, C. M.: Evaluating the effectiveness of independent verification and validation. In: *Computer* 32 (1999), Oct, Nr. 10, S. 79–83. http://dx.doi.org/10.1109/2.796141. – DOI 10.1109/2.796141. – ISSN 0018–9162

[BB01] Boehm, Barry ; Basili, Victor R.: Software Defect Reduction Top 10 List. In: *Computer* 34 (2001), Januar, Nr. 1, 135–137. http://dx.doi.org/10.1109/2.962984. – DOI 10.1109/2.962984. – ISSN 0018–9162

[BCHN05] Bergeron, Janick ; Cerny, Eduard ; Hunter, Alan ; Nightingale, Andy: *Verification Methodology Manual for SystemVerilog*. Berlin, Heidelberg : Springer-Verlag, 2005. http://dx.doi.org/10.1007/b135575. http://dx.doi.org/10.1007/b135575. – ISBN 0387255389

[Ber03] Bergeron, Janick: *Writing Testbenches: Functional Verification of HDL Models*. 2. Springer US, 2003. http://dx.doi.org/10.1007/978-1-4615-0302-6. http://dx.doi.org/10.1007/978-1-4615-0302-6. – ISBN 978–1–4615–0302–6

[Boe84] Boehm, B. W.: Verifying and Validating Software Requirements and Design Specifications. In: *IEEE Software* 1 (1984), Jan, Nr. 1, S. 75–88. http://dx.doi.org/10.1109/MS.1984.233702. – DOI 10.1109/MS.1984.233702. – ISSN 0740–7459

[BPD+16] Boukabache, Hamza ; Pangallo, Michel ; Ducos, Gael ; Cardines, Nicola ; Bellotta, Antonio ; Toner, Ciarán ; Perrin, Daniel ; Forkel-Wirth, Doris: TOWARDS A NOVEL MODULAR ARCHITECTURE FOR CERN RADIATION MONITORING. In: *Radiation Protection Dosimetry* 173 (2016), Nov, Nr. 1-3, S. 240–244. http://dx.doi.org/doi:10.1093/rpd/ncw308. – DOI doi:10.1093/rpd/ncw308

[BT16] Baranowski, Rafal ; Trunzer, Marco: Complete Formal Verification of a Family of Automotive DSPs. In: *Proceedings of 2016 Design and Verification Conference and Exhibition Europe (DVCon 2016)* Accellera Systems Initiative, 2016

[But12] Butka, B.: Is the current DO-254 verification process adequate for the future? In: *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, 2012. – ISSN 2155–7209, S. 6A6–1–6A6–11

[But15] Butka, Brian: Advanced Verification Methods for Safety-Critical Airborne Electronic Hardware. (2015), 1. https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/media/TC-14-41.pdf

[CA15] Cipitria Ayestaran, Iban: *Simulated fault injection for time-triggered safety-critical embedded systems*, TU Wien, Diss., 2015

[CCO+18] Craw, Brian ; Crutchfield, David ; Oberkoenig, Martin ; Heigl, Markus ; O'Keeffe, Martin: Using Automation to Close the Loop Between Functional Requirements and Their Verification. In: *Proceedings of 2018 Design and Verification Conference and Exhibition United States (DVCon 2018)* Accellera Systems Initiative, 2018

[CDHK15] Cerny, Eduard ; Dudani, Surrendra ; Havlicek, John ; Korchemny, Dmitry: *SVA: The power of assertions in system verilog, second edition.* Springer,Cham, 2015. – 1–590 S. `http://dx.doi.org/10.1007/978-3-319-07139-8`. `http://dx.doi.org/10.1007/978-3-319-07139-8`

[CEG17] CERN Education, Communications ; Group, Outreach: LHC the guide. (2017), Feb, 1-58. `https://home.cern/sites/home.web.cern.ch/files/2018-07/CERN-Brochure-2017-002-Eng.pdf`

[CKV03] Chockler, Hana ; Kupferman, Orna ; Vardi, Moshe Y.: Coverage Metrics for Formal Verification. In: Geist, Daniel (Hrsg.) ; Tronci, Enrico (Hrsg.): *Correct Hardware Design and Verification Methods.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2003. – ISBN 978–3–540–39724–3, S. 111–125

[Cor18a] Corporation, Mentor G.: *Questa Formal Technology Command Reference.* Software Version 10.7a. Wilsonville, Oregon, United States of America, 2018

[Cor18b] Corporation, Mentor G.: *Questa PropCheck User Guide.* Software Version 10.7a. Wilsonville, Oregon, United States of America, 2018

[Cor18c] Corporation, Mentor G.: *Questa SIM User's Manual.* Software Version 10.7b. Wilsonville, Oregon, United States of America, 2018

[CS10] Computer Society, IEEE: IEEE Standard for Property Specification Language (PSL). In: *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)* (2010), April, S. 1–182. `http://dx.doi.org/10.1109/IEEESTD.2010.5446004`. – DOI 10.1109/IEEESTD.2010.5446004

[DL00] Dupuy, A. ; Leveson, N.: An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In: *19th DASC. 19th Digital Avionics Systems Conference. Proceedings (Cat. No.00CH37126)* Bd. 1, 2000, S. 1B6/1–1B6/7 vol.1

[ESC10] Electrotechnical Standardization (CENELEC), European C.: IEC 61508:2010 - Functional safety of electrical/electronic/programmable electronic safety-related systems / European Committee for Electrotechnical Standardization (CENELEC). Brussels, B, Mai 2010. – Standard

[Fie17] Fiergolski, A.: Simulation environment based on the Universal Verification Methodology. In: *Journal of Instrumentation* 12 (2017), Nr. 01, C01001. `http://stacks.iop.org/1748-0221/12/i=01/a=C01001`

[Fos17] Foster, H. D.: Trends in functional verification: A 2016 industry study. In: *Proceedings of 2017 Design and Verification Conference and Exhibition United States (DVCon 2017)* Accellera Systems Initiative, 2017, S. 1–13

[FV14] Fulton, Randall ; Vandermolen, Roy: *Airborne Electronic Hardware Design Assurance.* CRC Press, 2014. – ISBN 978–1–4822–0606–7

[GAGS09] Gajski, Daniel D. ; Abdi, Samar ; Gerstlauer, Andreas ; Schirner, Gunar: *Embedded System Design: Modeling, Synthesis and Verification.* 1st. Springer Publishing Company, Incorporated, 2009. – ISBN 1441905030, 9781441905031

[Ger17] Gerber, Nicola: *Contributions to the SIL 2 Radiation Monitoring System CROME (CERN RadiatiOn Monitoring Electronics).* CERN internal, 3 2017

[GJC12] Grant, Emanuel ; Jackson, Vanessa ; Clachar, Sophine: Towards a Formal Approach to Validating and Verifying Functional Design for Complex Safety Critical Systems. In: *GSTF Journal on Computing (JoC)* 2 (2012), Nr. 1. `http://dl6.globalstf.org/index.php/joc/article/view/1008`. – ISSN 2010–2283

[GLH18] Grimm, Tomás ; Lettnin, Djones ; Hübner, Michael: A Survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip. In: *Electronics* 7 (2018), Nr. 6. `http://dx.doi.org/10.3390/electronics7060081`. – DOI 10.3390/electronics7060081. – ISSN 2079–9292

[Grü17] Grünfelder, Stephan: *Software-Test für Embedded Systems : Ein Praxishandbuch für Entwickler, Tester und technische Projektleiter*. 2. Wieblinger Weg 17, 69123 Heidelberg : dpunkt.verlag, 2017. – ISBN 978–3–96088–148–3

[HCL07] Hsiung, P. ; Chen, Y. ; Lin, Y.: Model Checking Safety-Critical Systems Using Safecharts. In: *IEEE Transactions on Computers* 56 (2007), May, Nr. 5, S. 692–705. `http://dx.doi.org/10.1109/TC.2007.1021`. – DOI 10.1109/TC.2007.1021. – ISSN 0018–9340

[HH16] Harris, C. B. ; Harris, I. G.: GLAsT: Learning formal grammars to translate natural language specifications into hardware assertions. In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016. – ISSN 1558–1101, S. 966–971

[Hur16] Hurst, Saskia: *Reliability Analysis of the CERN Radiation Monitoring Electronic System CROME*. CERN internal, 12 2016

[HV01] Hayhurst, K. J. ; Veerhusen, D. S.: A practical approach to modified condition/decision coverage. In: *20th DASC. 20th Digital Avionics Systems Conference (Cat. No.01CH37219)* Bd. 1, 2001, S. 1B2/1–1B2/10 vol.1

[ICG$^+$16] Irfan, A. ; Cimatti, A. ; Griggio, A. ; Roveri, M. ; Sebastiani, R.: Verilog2SMV: A tool for word-level verification. In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016. – ISSN 1558–1101, S. 1156–1159

[ICS18] IEEE Computer Society, IEEE Standards Association Corporate Advisory G.: IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. In: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), Feb, S. 1–1315. `http://dx.doi.org/10.1109/IEEESTD.2018.8299595`. – DOI 10.1109/IEEESTD.2018.8299595

[IK15] Imková, M. ; Kotásek, Z.: Automation and Optimization of Coverage-driven Verification. In: *2015 Euromicro Conference on Digital System Design*, 2015, S. 87–94

[Ini12] Initiative, Accellera S.: Unified Coverage Interoperability Standard (UCIS). (2012), June

[Ini14] Initiative, Accellera S.: *Universal Verification Methodology (UVM) 1.2 Class Reference*. Napa, CA 94558, United States of America, 2014

[JS05] Jantsch, A. ; Sander, I.: Models of computation and languages for embedded system design. In: *IEE Proceedings - Computers and Digital Techniques* 152 (2005), March, Nr. 2, S. 114–129. `http://dx.doi.org/10.1049/ip-cdt:20045098`. – DOI 10.1049/ip–cdt:20045098. – ISSN 1350–2387

[KEH$^+$09] Klein, Gerwin ; Elphinstone, Kevin ; Heiser, Gernot ; Andronick, June ; Cock, David ; Derrin, Philip ; Elkaduwe, Dhammika ; Engelhardt, Kai ; Kolanski, Rafal ; Norrish, Michael ; Sewell, Thomas ; Tuch, Harvey ; Winwood, Simon: seL4: Formal Verification of an OS Kernel. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. New York, NY, USA : ACM, 2009 (SOSP '09). – ISBN 978–1–60558–752–3, 207–220

[KK10] Kandl, Susanne ; Kirner, Raimund: Error Detection Rate of MC/DC for a Case Study from the Automotive Domain. In: Min, Sang L. (Hrsg.) ; Pettit, Robert (Hrsg.) ; Puschner, Peter (Hrsg.) ; Ungerer, Theo (Hrsg.): *Software Technologies for Embedded and Ubiquitous Systems*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010. – ISBN 978–3–642–16256–5, S. 131–142

[Kop11] Kopetz, Hermann: *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 2nd. Springer Publishing Company, Incorporated, 2011. – ISBN 1441982361, 9781441982360

[KPSS14] KIM, Namdo ; PARK, Junhyuk ; SINGH, H. S. ; SINGHAL, Vigyan: Sign-off with Bounded Formal Verification Proofs. In: *Proceedings of 2014 Design and Verification Conference and Exhibition United States (DVCon 2014)*, 2014

[Lam05] LAM, William K.: *Hardware Design Verification: Simulation and Formal Method-Based Approaches.* Prentice Hall, 2005. – ISBN 978–0–13–143347–2

[MBS12] MYERS, Glenford J. ; BADGETT, Tom ; SANDLER, Corey: *The Art of Software Testing.* 3. Hoboken, New Jersey : John Wiley & Sons, Inc., 2012. – ISBN 9781119202486

[MCP$^+$15] MARCONI, Sara ; CONTI, Elia ; PLACIDI, Pisana ; SCORZONI, Andrea ; CHRISTIANSEN, Jorgen ; HEMPEREK, Tomasz: A SystemVerilog-UVM Methodology for the Design, Simulation and Verification of Complex Readout Chips in High Energy Physics Applications. In: *Applications in Electronics Pervading Industry, Environment and Society - APPLEPIES 2015, Rome, Italy, May 5-6, 2015*, 2015, 35–41

[Meh14] MEHTA, Ashok B.: *SystemVerilog Assertions and Functional Coverage.* 1. Springer-Verlag New York, 2014. `http://dx.doi.org/10.1007/978-1-4614-7324-4`. `http://dx.doi.org/10.1007/978-1-4614-7324-4`. – ISBN 978–1–4614–7324–4

[Mey04] MEYER, Andreas: Chapter 3 - Methods for Determining the Validity of a Model. Version: 2004. `http://dx.doi.org/https://doi.org/10.1016/B978-075067617-5/50003-1`. In: MEYER, Andreas (Hrsg.): *Principles of Functional Verification.* Burlington : Newnes, 2004. – DOI https://doi.org/10.1016/B978–075067617–5/50003–1. – ISBN 978–0–7506–7617–5, 21 - 48

[MHS18] MINHAS, M. ; HASAN, O. ; SAGHAR, K.: Ver2Smv — A tool for automatic verilog to SMV translation for verifying digital circuits. In: *2018 International Conference on Engineering and Emerging Technologies (ICEET)*, 2018, S. 1–5

[MK07] MATHUR, A. ; KRISHNASWAMY, V.: Design for Verification in System-level Models and RTL. In: *2007 44th ACM/IEEE Design Automation Conference*, 2007. – ISSN 0738–100X, S. 193–198

[MR10] MEADE, Kathleen A. ; ROSENBERG, Sharon: *A Practical Guide to Adopting the Universal Verification Methodology (UVM).* 2. Cadence Design Systems, Inc., 2010. – ISBN 978–1–300–53593–5

[MW16] MANDOUH, E. E. ; WASSAL, A. G.: Covgen: A framework for automatic extraction of functional coverage models. In: *2016 17th International Symposium on Quality Electronic Design (ISQED)*, 2016. – ISSN 1948–3295, S. 146–151

[PAWJ] PUNNOOSE, Ratish J. ; ARMSTRONG, Robert C. ; WONG, Matthew H. ; JACKSON, Mayo: Survey of Existing Tools for Formal Verification. `http://dx.doi.org/10.2172/1166644`. – DOI 10.2172/1166644

[Per17] PERRIN, Daniel: *Description of the CERN Radiation & Environmental Monitoring System.* Nov 2017

[Pit18] PITCHFORD, Mark: Der unendliche Entwicklungs-Lebenszyklus vernetzter Systeme. In: *Elektronik Praxis* (2018), November

[PWPD18] PRZIGODA, N. ; WILLE, R. ; PRZIGODA, J. ; DRECHSLER, R.: *Automated Validation & Verification of UML/OCL Models Using Satisfiability Solvers.* Springer International Publishing, 2018. `http://dx.doi.org/10.1007/978-3-319-72814-8`. `http://dx.doi.org/10.1007/978-3-319-72814-8`. – ISBN 978–3–319–72814–8

[SM14] SINGH, Mahak ; MUKHERJEE, Siddhartha: Use of Iterative Weight-Age Constraint to Implement Dynamic Verification Components. Version: 2014. `https://verificationacademy.com/verification-horizons/june-2014-volume-10-issue-2`. 2014, 46-50

[SMG15] SPROTT, Jason ; MARRIOTT, Paul ; GRAHAM, Matt: Navigating The Functional

Coverage Black Hole: Be More Effective At Functional Coverage Modeling. In: *Proceedings of 2015 Design and Verification Conference and Exhibition United States (DVCon 2015)* Accellera Systems Initiative, 2015

[Soc12] Society, IEEE C.: IEEE Standard for Standard SystemC Language Reference Manual. In: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (2012), Jan, S. 1–638. `http://dx.doi.org/10.1109/IEEESTD.2012.6134619`. – DOI 10.1109/IEEESTD.2012.6134619

[SSK15] Seligman, Erik ; Schubert, Tom ; Kumar, M V Achutha K. ; Seligman, Erik (Hrsg.) ; Schubert, Tom (Hrsg.) ; Kumar, M V Achutha K. (Hrsg.): *Formal Verification.* Boston : Morgan Kaufmann, 2015. – 1 – 341 S. `http://dx.doi.org/https://doi.org/10.1016/C2013-0-18672-2`. `http://dx.doi.org/https://doi.org/10.1016/C2013-0-18672-2`. – ISBN 978–0–12–800727–3

[SSM+02] Semeria, L. ; Seawright, A. ; Mehra, R. ; Ng, D. ; Ekanayake, A. ; Pangrle, B.: RTL C-based methodology for designing and verifying a multi-threaded processor. In: *Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324)*, 2002. – ISSN 0738–100X, S. 123–128

[SSR+09] Sánchez, Ernesto ; Squillero, Giovanni ; Reorda, Matteo S. ; Pradhan, Dhiraj K. ; Harris, Ian G.: Test generation and coverage metrics. Version: 2009. `http://dx.doi.org/10.1017/CBO9780511626913.005`. In: *Practical Design Verification.* Cambridge University Press, 2009. – DOI 10.1017/CBO9780511626913.005, S. 122–153

[TCE+15] Tabacaru, Bogdan ; Chaari, Moomen ; Ecker, Wolfgang ; Novello, Cristiano ; Kruse, Thomas ; Liu, Kai ; Post, Hendrik ; Hatami, Nadereh ; Schwerin, Andreas von: Fault-Injection Techniques for TLM-Based Virtual Prototypes. In: *Proceedings of Conference: FDL, At Barcelona, Spain*, 2015

[TK01] Tasiran, S. ; Keutzer, K.: Coverage metrics for functional validation of hardware designs. In: *IEEE Design Test of Computers* 18 (2001), July, Nr. 4, S. 36–45. `http://dx.doi.org/10.1109/54.936247`. – DOI 10.1109/54.936247. – ISSN 0740–7475

[Tp00] Torres-pomales, Wilfredo: Software Fault Tolerance: A Tutorial. (2000), 10, S. 66

[VHH+14] Valentini, F ; Hakulinen, T ; Hammouti, L ; Ladzinski, T ; Ninin, P: Formal Methodology for Safety-Critical Systems Engineering at CERN. In: *Proceedings of ICALEPCS2013.* San Francisco, CA, USA, Mar 2014

[VHR07] Verma, S. ; Harris, I. G. ; Ramineni, K.: Automatic Generation of Functional Coverage Models from Behavioral Verilog Descriptions. In: *2007 Design, Automation Test in Europe Conference Exhibition*, 2007. – ISSN 1530–1591, S. 1–6

[Vli18] Vliet, Hans v.: *Software Engineering: Principles and Practice.* 3. Wiley, 2018. – ISBN 978–0–470–03146–9

[VPH09] Verma, Shireesh ; Pradhan, Dhiraj K. ; Harris, Ian G.: SystemVerilog and Vera in a verification flow. Version: 2009. `http://dx.doi.org/10.1017/CBO9780511626913.006`. In: *Practical Design Verification.* Cambridge University Press, 2009. – DOI 10.1017/CBO9780511626913.006, S. 154–172

[WCC09] Wang, Laung-Terng (Hrsg.) ; Chang, Yao-Wen (Hrsg.) ; Cheng, Kwang-Ting (. (Hrsg.): *Electronic Design Automation: Synthesis, Verification, and Test.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2009. – ISBN 9780080922003

[Wid18] Widorski, Markus: *CROME Project - CMPU Functional Requirements.* CERN internal, 11 2018

[WSP+17]

In: Weissnegger, Ralph P. ; Schachner, Martin ; Pistauer, Markus ; Kreiner, Christian J. ; Roemer, Kay U. ; Steger, Christian: *Generation and Verification of a Safety-Aware Virtual Prototype in the Automotive Domain.* United States : IGI Global Publishing, 2017. – ISBN 9781522528456, S. 195

[YCC15] Yan, R. ; Cheng, C. ; Chai, Y.: Formal consistency checking over specifications in natural languages. In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015. – ISSN 1530–1591, S. 1677–1682

[Yeh14] Yehia, Ahmed: The Top Most Common SystemVerilog Constrained Random Gotchas. In: *Proceedings of 2014 Design and Verification Conference and Exhibition United States (DVCon 2014)*, 2014

[YWD14] Yang, S. ; Wille, R. ; Drechsler, R.: Improving Coverage of Simulation-Based Verification by Dedicated Stimuli Generation. In: *2014 17th Euromicro Conference on Digital System Design*, 2014, S. 599–606

Internet References

[weblink1] B. AS. Universal vhdl verification methodology - getting started. https://github.com/UVVM/UVVM/blob/master/GETTING_STARTED.md. Accessed 2018-12-09.

[weblink2] L. Asplund. Vunit. https://vunit.github.io/. Accessed 2018-12-16.

[weblink3] Cadence. Jaspergold sequential equivalence checking app. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/jaspergold-sequential-equivalence-checking-app.html. Accessed 2019-01-09.

[weblink4] G. Community. git. https://git-scm.com/. Accessed 2019-01-26.

[weblink5] M. G. Corporation. Uvm framework. https://verificationacademy.com/topics/verification-methodology/uvm-framework. Accessed 2018-12-16.

[weblink6] Doulos. Download the easier uvm coding guidelines and code generator. https://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_generator/. Accessed 2018-12-16.

[weblink7] A. S. Initiative. Accellera systems initiative. http://www.accellera.org/. Accessed 2018-12-16.

[weblink8] M. Kaufmann and J. S. Moore. Industrial proofs with acl2. https://www.cs.utexas.edu/users/moore/publications/how-to-prove-thms/intro-to-acl2.pdf. Accessed 2018-12-07.

[weblink9] J. Lewis. Open source vhdl verification methodology user manuals. https://github.com/OSVVM/OSVVM/tree/master/doc. Accessed 2018-12-09.

[weblink10] V. T. P. Ltd. Verifsudha framework. http://www.verifsudha.com/. Accessed 2018-12-16.

[weblink11] Mathworks. Hdl verifier. https://ch.mathworks.com/help/hdlverifier/index.html. Accessed 2019-01-07.

[weblink12] a. S. B. Mentor. Catapult high-level synthesis. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/hls-verification. Accessed 2019-01-09.

[weblink13] PotentialVentures. cocotb. https://cocotb.readthedocs.io/en/latest/introduction.html. Accessed 2018-12-09.

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, am 15.02.2019

[Katharina Ceesay-Seitz]