Decentralized and Pulse-based Clock Synchronization in SpaceWire Networks for Time-triggered Data Transfers

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt in der Fakultät für Mathematik und Informatik
der Julius-Maximilians-Universität Würzburg

Autor:
Kai Borchers

Gutachter:
Prof. Dr.-Ing. Sergio Montenegro
Prof. Dr.-Ing. Görschwin Fey

Februar 10, 2020

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Bremen, Februar 2020

Kai Borchers

Acknowledgements

I would like to thank all the people who supported me throughout the years by either technical discussions or by motivating me to finish this project. Additionally, I am quite thankful for the excellent infrastructure of the DLR Institute of Space Systems, where I have had access to throughout all the years.

A special thanks goes to Sergio Montenegro and Görschwin Fey who agreed to spend a lot of their time in supporting me and reviewing this work.

Finally, I have to say rather sorry than thank you to my friends and especially to my family for whom I should have had more time.

Abstract

Time-triggered communication is widely used throughout several industry domains, primarily for reliable and real-time capable data transfers. However, existing time-triggered technologies are designed for terrestrial usage and not directly applicable to space applications due to the harsh environment. Instead, specific hardware must be developed to deal with thermal, mechanical, and especially radiation effects.

SpaceWire, as an event-triggered communication technology, has been used for years in a large number of space missions. Its moderate complexity, heritage, and transmission rates up to 400 MBits/s are one of the main advantages and often without alternatives for on-board computing systems of spacecraft. At present, real-time data transfers are either achieved by prioritization inside SpaceWire routers or by applying a simplified time-triggered approach. These solutions either imply problems if they are used inside distributed on-board computing systems or in case of networks with more than a single router are required.

This work provides a solution for the real-time problem by developing a novel clock synchronization approach. This approach is focused on being compatible with distributed system structures and allows time-triggered data transfers. A significant difference to existing technologies is the remote clock estimation by the use of pulses. They are transferred over the network and remove the need for latency accumulation, which allows the incorporation of standardized SpaceWire equipment. Additionally, local clocks are controlled decentralized and provide different correction capabilities in order to handle oscillator induced uncertainties. All these functionalities are provided by a

developed Network Controller (NC), able to isolate the attached network and to control accesses.

Contents

List of Figures

List of Tables

Acronyms

ABV	Assertion-Based Verification
AHB	Advanced High-Performance Bus
ASIC	Application-Specific Integrated Circuit
BDD	Binary Decision Diagram
BE	Best-Effort
BFM	Bus Functional Model
CDC	Clk Domain Crossing
CM	Compression Master
COTS	Commercial Off-The-Shelf
CRC	Cyclic Redundancy Check
CRV	Constrained Random Verification
C-state	Controller State
CAS	Collision Avoidance Symbol
CSMA/CD	Carrier Sense Multiple Access/Collision Detection
CTL	Computation Tree Logic
DIRQ	Distributed Interrupt
DLR	German Aerospace Center
DS	Data-Strobe
DSP	Digital Signal Processor
DUT	Device Under Test

EBNF	Extended Backus-Naur Form
EC	Equivalence Checking
EDA	Electronic Design Automation
EDIF	Electronic Design Interchange Format
EEP	Error End of Packet
EMC	Electromagnetic Compatibility
EOP	End Of Packet
ESA	European Space Agency
ESC	Escape
FCT	Flow Control Token
FF	Flip-Flop
FPGA	Field Programmable Gate Array
FPV	Formal Property Verification
FV	Formal Verification
HDL	Hardware Description Language
HLS	High-Level Synthesis
HVL	Hardware Verification Language
IN	Integration Frame
IP	Intellectual Property
L-Char	Link Character
LET	Linear Energy Transfer
LTL	Linear Temporal Logic
LUT	Look-Up Table
LVDS	Low Voltage Differential Signaling
MC	Model Checking
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor

MUL	Multiplier
N-Char	Normal Character
NC	Network Controller
NDCP	Network Discovery and Configuration Protocol
NIT	Network Idle Time
NTP	Network Time Protocol
OBC	On-Board Computer
OVM	Open Verification Methodology
PCB	Printed Circuit Board
PCF	Protocol Control Frame
PPM	Parts Per Million
PSL	Property Specification Language
PTP	Precision Time Protocol
QoS	Quality of Service
RAM	Random-Access Memory
RC	Rate-Constrained
RCR	Remote Clock Reading
RMAP	Remote Memory Access Protocol
RTL	Register-Transfer Level
ScOSA	Scalable On-Board Computing for Space Avionics
SEE	Single Event Effect
SEFI	Single Event Functional Interrupt
SEGR	Single Event Gate Rupture
SEL	Single Event Latch-up

SET	Single Event Transient
SEU	Single Event Upset
SM	Synchronization Master
SoC	System-on-Chip
SUF	Start-Up Frame
SV	SystemVerilog
SVA	SystemVerilog Assertions
SYF	Synchronization Frame
TDMA	Time Division Multiple Access
TDP	Time Distribution Protocol
TID	Total Ionizing Dose
TMR	Triple Modular Redundancy
TT	Time-Triggered
TTCAN	Time-Triggered Controller Area Network
TTEthernet	Time-Triggered Ethernet
TTP/C	Time-Triggered Protocol SAE Class C
UML	Unified Modeling Language
UVM	Universal Verification Methodology
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VMM	Verification Methodology Manual

Part I

Introduction and Motivation

Chapter 1

Introduction

1.1 Motivation and Background

Communication technologies in spacecraft are essential as they connect all subsystems like On-Board Computer (OBC) or payload instruments together. In contrast to terrestrial systems, spacecraft engineers are limited in their selection of communication technologies. This limitation is mainly caused by radiation effects in space environments, which can lead to different undesirable impacts inside the selected hardware [KCR06]. These effects can range from Single Event Upset (SEU) were bit changes occur up to hardware disruptive events like Single Event Gate Rupture (SEGR). A special design and manufacturing process is often necessary to obtain the required resilience. However, the whole process of creating radiation hardened hardware can become very expensive and need to be reasonable from the economic point of view. Additionally, the space industry is quite small and far away from consuming as many electronic parts as commercial industries.

SpaceWire, as a communication technology for serial data transfers, was initially defined by the European Space Agency (ESA) in 2003. Since then, SpaceWire has been used in multiple projects throughout the whole space domain. One of its main advantages is the moderate complexity. This leads to a low utilization if used on a Field Programmable Gate Array (FPGA) or Application-Specific Integrated Circuit (ASIC) and also allows rapid develop-

ment of SpaceWire components like interfaces or routers. The physical layer only requires Low Voltage Differential Signaling (LVDS) driver and receiver, whereas FPGAs and LVDS parts are both available as radiation hardened devices.

Recent projects that utilized SpaceWire are Eu:Cropis [Kot+18] and MASCOT [Hab+13]. Both projects consist of a traditional spacecraft system structure with instruments centered around an OBC without real-time requirements. In contrast to these projects, the decentralized system Scalable On-Board Computing for Space Avionics (ScOSA) is developed inside the German Aerospace Center (DLR) [Tre+18]. It is based on a meshed SpaceWire network with different kinds of computing nodes and targets high performance, reliability, and scalability. These nodes can handle several tasks throughout different mission phases by the use of dynamic system reconfiguration. However, this reconfiguration property has a temporal boundary. As a consequence, real-time data transfers are required, which are not directly supported by SpaceWire.

Besides ScOSA, which is one reason for the research activities reflected in this work, we don't see a full replacement of SpaceWire within the next years.

1.1.1 SpaceWire Issues

The SpaceWire standard does not support true real-time capabilities. One reason for this is the applied wormhole routing scheme [ESA08, p. 97]. This kind of routing allows message forwarding inside routers as soon as logical or path addresses are received. Wormhole routing will drastically reduce buffer sizes inside SpaceWire interfaces or routers but leads to messages that are spread throughout the whole network with the ability to block other data or creating deadlock situations.

Priority based arbitration is proposed to establish at least basic Quality of Service (QoS) [ESA08, p. 99]. For this, logical addresses can be forwarded inside routers prioritized. However, cascading multiple routers can lead to situations where prioritized message forwarding doesn't work any longer [Bor+18]. Additionally, messages are allowed to have arbitrary lengths, which leads to

undefined periods of network resource utilization.

1.1.2 Event and Time-Triggered Networks

In general, communication networks can be classified in *time-triggered* and *event-triggered*. Communication inside event-triggered networks is established as soon as data is available. The inputs of the network can be considered as a trigger event. In contrast to that, time-triggered networks can only be accessed at specific points in time. These accesses are controlled by a static schedule, which is defined before time-triggered networks start their operation. This static schedule definition reduces the complexity of evaluating the network regarding performance and reliability.

Additionally, the schedule has a direct relation to the real-time capabilities of the network because all data transfers are controlled of it. This prevents conflicts of data transmissions between all participating units connected to the network and guarantees data delivery in time. Real-time capabilities can also be established in event-triggered networks by using a *rate constrained* approach. For this, the data bandwidth is limited by applying minimal idle times between consecutive data packets and by defining a maximum packet length [Boy+16].

1.2 Contribution

This work addresses the SpaceWire real-time problem by developing a time-triggered approach for decentralized system structures. For this, a system-wide clock synchronization can be considered as a precondition. Existing clock synchronization approaches for meshed network topologies can't be applied directly without a non-standard modification of existing SpaceWire components. The problem is solved by a transfer of bus-based clock synchronization approaches into the SpaceWire networks as outlined more detailed in the following.

The first research contribution of this work is the development and implementation of new methods to handle start-up phase and clock synchronization

in order to support distributed system structures for SpaceWire networks.

Both methods rely on the *broadcast code* feature of SpaceWire without the need to modify existing routers beyond the currently revised SpaceWire standard [ESA19]. These broadcast codes can be considered as system-wide interrupts, whereas its latency and jitter characteristics are directly correlated to the alignment of all local clocks which need to be synchronized.

A start-up phase is required to establish initially a system-wide or global time, which is a precondition for any kind of time-triggered data transfer. The implementation is based on a majority determination between multiple start-up involved nodes. The process is designed to tolerate n failing nodes, whereas n depends on the number of start-up nodes. The node, which is elected to finish a start-up phase, uses a broadcast code to align the initial set of local clocks. However, the quality of initial alignment requires an application of correction value, which depends on network structure and communication links.

An already established global time needs to be synchronized throughout schedule based operation. Otherwise, schedules would drift apart, which leads to a complete loss of communication. The synchronization process requires knowledge about the values of all other clocks that are part of the clock ensemble. These values are often derived in bus-based communication technologies by comparison between expected and actual reception of messages. However, this is only possible because of constant latencies, which is an implication of the network structure. Latencies can vary drastically in switched networks, which makes the bus-based approach impracticable. Instead, packets contain the accumulated delay throughout the whole path from source to destination node. Based on this accumulated value, the destination node can derive the required information of the source node clock. However, the accumulation requires that each unit, which is traversed by a packet, can determine these delays.

SpaceWire, with its switched network topology, would require a non-standard redesign of its components if latency accumulations are selected to apply existing synchronization approaches. Instead, the bus-based approach is transferred into SpaceWire networks by utilizing broadcast codes

to keep variations of latencies in a tolerable range. This allows the usage of existing SpaceWire components to create systems that can synchronize their clocks. The developed approach provides the ability of decentralized clock synchronization, which is a significant difference to the existing extension SpaceWire-D.

Additionally, a potential non-standard modification was applied to the used SpaceWire interface in order to change its broadcast code handling. The modification leads to a transfer of jitter into a larger but more constant latency with the intent of improving the clock synchronization quality.

The second research contribution covers the broadcast code evaluation concerning its latency range. The latency range defines the jitter, which is vital to get a proper clock synchronization. Hence, a Universal Verification Methodology (UVM) verification environment is created to transfer large numbers of broadcast codes in order to track all encountered latencies. These test runs are applied to standard and modified SpaceWire interfaces. Additionally, formal property verification is applied to get a confirmation of the functional simulation results provided by the UVM environment. However, the application of this formal approach leads to specific issues and limitations that are discussed in the related chapter.

The last research contribution focuses on the evaluation of the developed methods. For this, a complete time-triggered network with different structural configurations and several schedules were created as Very High Speed Integrated Circuit Hardware Description Language (VHDL) design. All relevant statistics were monitored and extracted by a UVM verification environment during multiple test runs. The results show that each applied start-up phase finished successfully with a neglectable number of logical collisions and within an acceptable time. The evaluation of clock synchronization shows that oscillator drifts were compensated, and local clock deviations don't exceed 1.2 μs. However, the maximum clock deviation depends on multiple system parameters. Thus, the monitored maximum value of 1.2 μs can't be treated as an overall upper boundary. An advantage of the modified SpaceWire interface with respect to synchronization quality was observed throughout all executed tests.

1.3 Publications

This work contains parts, extensions, or ideas of previously published material of the author[1]. A complete list of all publications is given in the Appendices IV. However, the following peer-reviewed papers are mainly considered for this thesis:

- The concept of the pulse-based and decentralized clock synchronization approach was introduced in [Bor+18].
- The prototype and first evaluation results were published in [Bor+19]. It contains the investigation of different system characteristics (e.g. start-up behavior or clock synchronization quality) and utilization results.
- The system evaluation was mainly done by functional simulation with the support of SystemVerilog Assertions. An alternative use case of SystemVerilog Assertions in order to handle volatile registers was discussed in [BMD19].

1.4 Thesis Structure

The thesis is separated into four main parts. Part I provides a motivation and background information about the theses subject. Part II gives an overview of existing technologies and approaches which are applied inside this work. Chapter 2 covers the basic concepts and operational modes used in current time-triggered technologies. An overview of FPGAs regarding its structure and design flow is given in Chapter 3. Additionally, several FPGA verification aspects and methodologies are introduced. The foundation part closes with Chapter 4 by providing an overview of the communication technology SpaceWire. Its unique broadcast code capabilities are the backbone for the prototype developed in this work.

[1] The author's birth name is Stohlmann.

The main contribution of this work is given in Part III. An introduction of the general clock synchronization approach is given in Chapter 5. It illustrates how remote clock estimates are gathered based on pulses and shows the architecture of the developed prototype. A modified SpaceWire interface is discussed as well to achieve an improved clock synchronization.

SpaceWire broadcast codes are evaluated separately in Chapter 6. They are the foundation of the introduced clock synchronization approach, whereas they are characterized in two different ways. Functional simulation with a constrained random approach is used on one side. On the other hand, formal property verification is applied in order to confirm or disproof the simulation-based results. Additionally, formal property verification specific problems (e.g. complexity handling) are discussed.

The overall system evaluation of the developed prototype is given in Chapter 7. It starts with an introduction of the UVM verification environment and its main components like metric analyzer and design checking parts. The system evaluation is divided into start-up and distributed clock analysis followed by an overview of synthesis results for different FPGA targets. Possible improvements and further work are discussed in Chapter 8. Finally, all used references and some notations are given in Part IV.

Part II

Foundation

Chapter 2

Time-triggered Communication

2.1 Introduction

Accesses to communication mediums need to be managed in some way. A well-known access method is Carrier Sense Multiple Access/Collision Detection (CSMA/CD), used by the Ethernet protocol. It provides the ability to detect collisions in case two or more parallel messages are transmitted. Additionally, it resolves collisions by applying timeouts before retransmission of collided data. Time-triggered communication applies the Time Division Multiple Access (TDMA) access method to achieve a controlled, decentralized, and collision-free access to the communication medium. This kind of communication relies on the progression of a global time. Each node, required to transmit data, is allowed to access the network at specific points in time. These access points are defined by static schedules, stored locally at every node, and computed before the network starts operation. This kind of network access is considered as decentralized. In contrast to that, a network controlled by a single arbiter or master has a centralized access.

The executed clock synchronization approach is a major characteristic of each time-triggered technology. A distributed approach is present if multiple nodes exchange their clock values in order to derive a converged clock correction value. A centralized clock synchronization is applied if a single source is used to adjust all clocks inside the system. However, this represents a single

point of failure, capable of causing a full loss of communication. This problem is unacceptable for most time-triggered technologies which are designed to provide high reliability or even being used in safety-critical systems. An example of this kind of technology is Time-Triggered Protocol SAE Class C (TTP/C). It was developed to fulfill requirements for distributed safety-critical systems in various domains like automotive or aerospace [TTT02, p. 11].

However, the distributed clock synchronization approach substantially increases system complexity. It requires an initial clock synchronization after system resets or power cycles to switch from an asynchronous to synchronous operation. Throughout synchronous operation, all local clocks need to be synchronized periodically to compensate drifts that occur over time. Finally, late powered or reset nodes require a re-integration into the existing synchronous operation.

The typical structure of a time-triggered communication system is given in Figure 2.1. It consists of multiple nodes connected to a network for exchanging

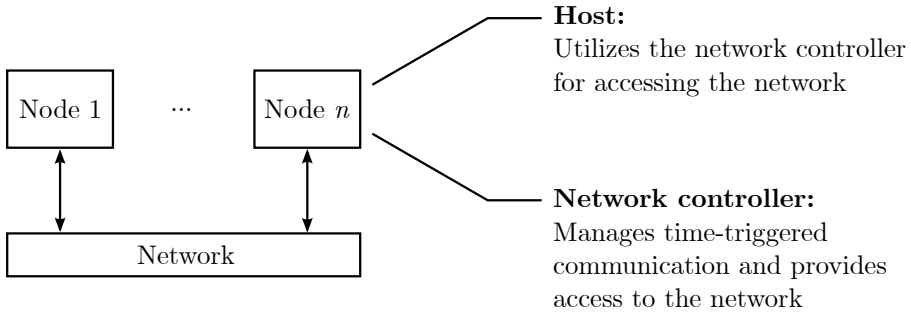

Figure 2.1: Typical structure of time-triggered systems.

information. This network must be considered as a shared resource between all connected nodes. It's in the responsibility of the network controller to ensure all accesses are performed according to the schedule to prevent conflicts. A specific interface, often implemented as a descriptor table, allows the host to provide data for transmission to the network controller. A direct connection between host and network doesn't exist. Additionally, the network

controller handles the start-up, clock synchronization, and integration tasks.

The network itself can consist of arbitrary structures and topologies. The structure has a direct impact on the schedule and how the network is accessed. Figure 2.2 shows two example schedules for bus and switched network topologies. Each schedule consists of four slots utilized by one or more nodes. The bus topology allows only single access at the same time, indicated by an exclusive utilization of each slot. A multicast transmission with one source and n destinations is implicitly given. Several bus-based technologies like TTP/C [TTT02, p. 37], SAFEbus [HD92] or FlexRay [Rau07, p. 127] are available. All of them can be used with dual redundant channels to increase either data throughput or reliability.

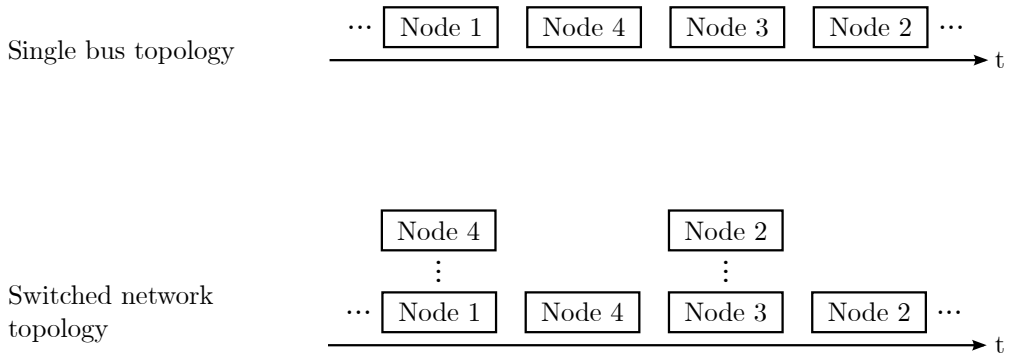

Figure 2.2: Network access depending on its topology.

Switched networks can be set up in a way that multiple accesses at the same time are possible. This multiple write property is given as soon as several data paths throughout the network are available. An example technology, that relies on the IEEE 802.3 Ethernet standard, is Time-Triggered Ethernet (TTEthernet) [TTT11, p. 6]. TTEthernet defines traffic classes to distinguish between different communication modes, whereas Time-Triggered (TT) has the highest priority. Additionally, Best-Effort (BE) and Rate-Constrained (RC) traffic classes for less or even none critical timing requirements are defined [Ste+09]. All these traffic classes share the same network throughout system operation.

Additional time-triggered communication aspects are discussed in the remainder of this chapter, which is structured as follows. Initially, the concept of digital clocks is introduced in Section 2.2 followed by a discussion of clock uncertainties in Section 2.3. A differentiation between clock accuracy and precision is given in Section 2.4 because of its importance to determine the quality of clock synchronization. The notion of global time with respect to distributed systems is given in Section 2.5. Finally, the main operational states are explained that most time-triggered technologies have to execute. An overview of start-up processes and how they are used to establish a global time is introduced in Section 2.6. Possible node integration strategies are shown in Section 2.7 followed by an introduction of different clock synchronization approaches in Section 2.8.

2.2 Digital Clocks

Clocks generally need periodic events and a mechanism to count them. A number of periodic events must be recognized depending on the duration that needs to be measured. Microwave based atomic clocks have been used to define a second since 1976 because of their superior performance. With the occurrence of optical-based atomic clocks, even higher accuracies are achievable [McG+19]. Optical clocks are also subject to research activities regarding synchronization within the femtosecond area [Ber+19]. However, generally electronic oscillators[1] are used for embedded systems in order to create periodic events. They fit in size and reliability with the disadvantage of being less accurate compared to atomic clocks.

For time-triggered systems, these periodic events are also called *microticks*, whereas the distance between two consecutive microticks is called *granule*. The granule can also be considered as the period T of an oscillator, which is defined as the reciprocal of its frequency f. To describe clocks and properties throughout definitions, the following notation $property_i^k$ is applied, whereas k indicates the number of a clock and i represents a specific microtick or

[1] Also indicated by term *clk* for the remainder of this work.

macrotick/tick[2].

The granularity of a clock can only be measured with a clock that provides a finer granularity. However, each measurement by use of an electronic oscillator leads to digitalization errors, as shown in Figure 2.3.

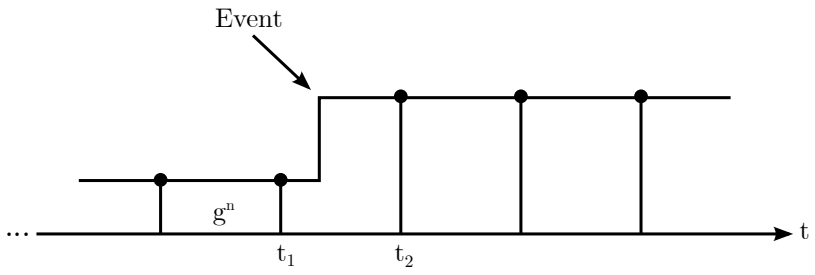

Figure 2.3: Digitalization error as a function of oscillator granularity.

A signal value is sampled continuously by clock n to observe events. The event, indicated by an increase of signal value, takes place between t_1 and t_2. A temporal deviation between real and recognized event occurs caused by the sampling rate with a maximum absolute error value of $\frac{g^n}{2}$. The error value can be decreased by selecting oscillators with higher frequencies, which leads to a shortened granule g^n. Another limitation of digital clocks is related to event ordering. The order of multiple observed events between two consecutive sampling points can't be reconstructed.

2.3 Clock uncertainties

Digital clocks are derived by use of oscillators as introduced in the prior Section 2.2. Hence, uncertainties of used oscillators directly correlate with the quality of digital clocks. These uncertainties are generally defined in Parts Per Million (PPM), which expresses the maximum deviation to the nominal oscillator frequency. Additionally, PPM values depend on different properties like temperature, voltage or aging [Vec17, p. 11].

[2]A detailed definition of macrotick/tick is given in Section 2.5.

The *drift* of a clock can be defined with respect to two consecutive microticks by the following equation [Kop11, p. 54].

$$drift_i^k = \frac{z(microtick_{i+1}^k) - z(microtick_i^k)}{n^k} \tag{2.1}$$

It is assumed that all microticks of clock k are observable by clock z with a negligible digitalization error. This observation provides an actual duration between $microtick_{i+1}^k$ and $microtick_i^k$, which contains a deviation to the nominal duration, expressed by n^k. The ratio between the actual and nominal duration of two consecutive microticks defines the drift for a given clock and a specific microtick.

Inside existing literature, the term *drift rate* is often used to express oscillator uncertainties. It can be defined by the following equation [Kop11, p. 55].

$$\rho_i^k = \left| \frac{z(microtick_{i+1}^k) - z(microtick_i^k)}{n^k} - 1 \right| \tag{2.2}$$

The drift rate describes an unsigned measure for the oscillator frequency deviation compared to its nominal frequency. A perfect oscillator without any deviations has a drift rate of 0.

2.4 Accuracy versus Precision

The *precision* is used to determine how close a clock ensemble is synchronized. The definition of precision requires an introduction of offsets between clocks. An offset is measured between two clocks j and k with the same granularity for a given microtick i by use of a reference clock z [Obe11, p. 15].

$$offset_i^{jk} = \left| z(microtick_i^j) - z(microtick_i^k) \right| \tag{2.3}$$

The digitalization error of reference clock z is considered as negligible. The maximum offset between n clocks for a given microtick i is defined as follows [Obe11, p. 15].

$$\Pi_i = \max_{\forall 1 \leq j,k \leq n,} \{offset\} \tag{2.4}$$

Π_i represents the precision for a clock ensemble at microtick i. The precision for an arbitrary microtick interval is defined as Π. The precision values are expressed by microticks of the reference clock.

Clock deviations, measured to a given reference clock, are called *accuracy*. Similar to precision, the accuracy is also determined for a single microtick i but also for an arbitrary interval. Figure 2.4 shows the differences between precision and accuracy for a specific point in time.

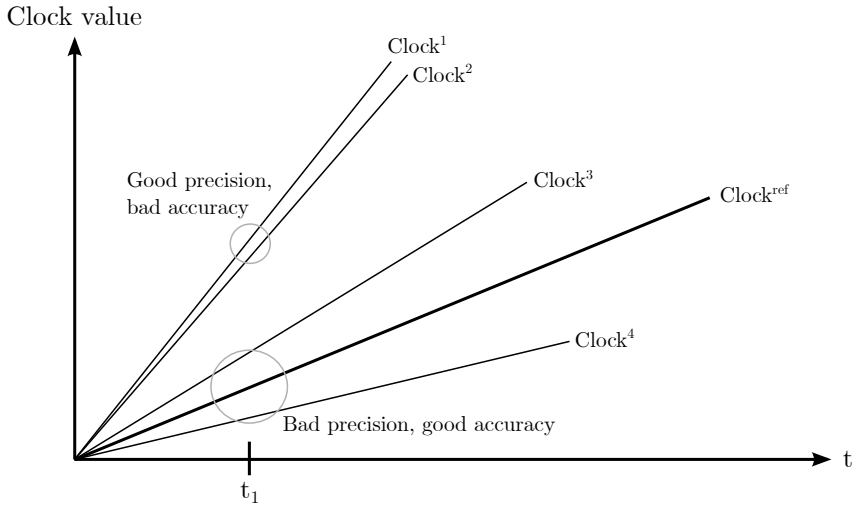

Figure 2.4: Definition of precision and accuracy.

$Clock^1$ and $Clock^2$ have a low offset between each other at t_1, which leads to a good precision between them. However, the offset of each clock to the reference $Clock^{ref}$ is large, which causes a bad accuracy. $Clock^3$ and $Clock^4$ illustrates the opposite by showing a large offset between each other but a small offset to $Clock^{ref}$.

2.5 Global Time

As explained in Section 2.1, each node inside distributed systems generally has to maintain its own clock locally, whereas the value of its clock is derived by an oscillator. These oscillators can vary between nodes in frequency and stability. Thus, microticks of oscillators are insufficient to define a global time throughout all nodes.

Instead, a system-wide temporal duration, often called *macrotick*, is defined [Cen+13, p. 33]. These macroticks are used to define parameters that are shared inside the whole system, e.g. slot lengths or whole schedules/cycles, as shown by Figure 2.5. They are derived by counting microticks until

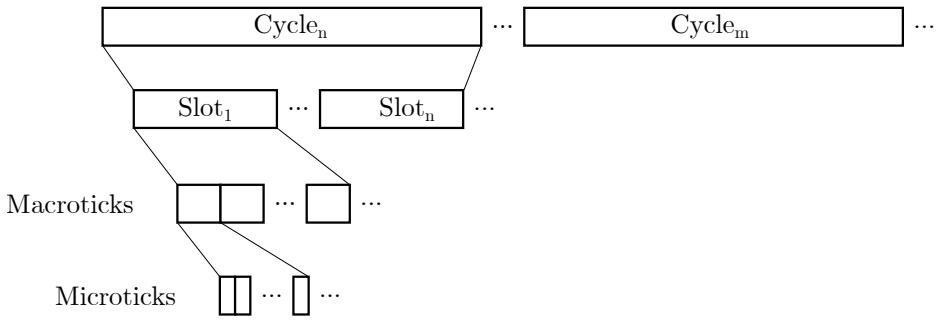

Figure 2.5: Construction of schedule cycles based on microticks, macroticks and slots.

the defined macrotick duration is reached [Han06] and represent the smallest time granularity inside the system. The number of microticks per macrotick can vary between nodes depending on the used oscillators. This number can also vary for a single node as soon as correction methods for clock synchronizations are applied.

2.6 Start-up

A time-triggered system is considered as asynchronous after a global power-on or in case unexpected faults lead to a full loss of synchronization. The system changes to synchronous operation as soon as two or more nodes synchronize

their local clocks in order to execute their schedules. It is the responsibility of start-up algorithms to establish a synchronized time within a maximum temporal duration for a subset of nodes. The start-up phase is generally implemented separately to other processes to encapsulate its high complexity. Additionally, the separation allows a better exchange of the algorithm if required and improves maintenance and testability.

The implementation of a start-up process directly depends on the network topology. Three different start-up algorithms for a bus, with up to nine nodes, were described and evaluated in [Loe99] by use of simulation. Another start-up solution, based on unique message lengths, is provided by [CLS04]. A start-up algorithm and related architectural guidelines are provided in [SP02] with a focus on bounded execution durations and the ability to work under different failure scenarios. An extension in the fault hypothesis, which allows arbitrary failures, is discussed in [SK06]. Additionally, this enhanced algorithm was compared with the FlexRay start-up algorithm. It turned out that FlexRay is vulnerable to simple failure modes.

Bus-based start-up algorithms basically work in a similar way. A set of nodes, which are allowed to perform a start-up, exchange messages to establish the synchronous operation. For this, start-up nodes are initially passive by observing the bus to check for data exchanges that are already in progress. These data exchanges indicate that either synchronous operation is already performed or another start-up node executes a start-up phase to establish a synchronous operation. If no data exchanges are monitored throughout a timeout period, start-up nodes become active by starting their own local clock to provide periodically start-up specific data according to the schedule. These data, in turn, are used by other start-up nodes to determine the currently executed schedule position. This information is finally used by unsynchronized nodes to change into a synchronous operation.

Possible collisions must be handled during the asynchronous operation because no centralized mechanism is used to bring the system into a synchronous operation. Instead, multiple start-up nodes can begin their start-up phases in parallel. However, a bus topology only provides mutual exclusive write access. Thus, collisions need to be detected and resolved in some way.

The technology FlexRay detects collisions by use of the schedule [Fle10, p. 173]. Initially, a Collision Avoidance Symbol (CAS) is transmitted to the bus before start-up nodes start their local clocks to enter a reduced schedule execution for active start-up phase application. Throughout one reduced schedule execution, only a single start-up frame is transmitted per start-up node inside a unique slot. This CAS application can happen in parallel, whereby multiple start-up nodes start the schedule execution. This leads to several transmitted start-up frames throughout a single schedule execution. However, each start-up node is assigned to a unique slot for start-up frame transmission. Thus, all start-up frames are transmitted in a predefined order. The first start-up frame inside this order is recognized by all start-up nodes, which indicates a parallel start-up phase. By this, the collision is detected, and all start-up nodes that don't belong to the first start-up frame stop its active start-up phase. Start-up nodes, who canceled their start-up phase, observe again the bus to detect data transfers. The timeout used for observation is equal for all start-up nodes inside a FlexRay system.

TTP/C uses a different strategy to resolve collisions compared to the equal timeout of FlexRay. In case two or more start-up nodes start transmission within a specific interval, frames collide and are recognized as noise at the receivers [SP02]. The detection of noise leads to a reset of the timeout, which is used for observing the bus. These timeouts are unique for every start-up node. Thus, it is assumed that collisions resolve only by temporal differences introduced by unique timeouts.

Bus-based time-triggered technologies with their broadcast abilities rely on nearly constant message propagation delays for start-up, clock synchronization, and integration. Switched network topologies instead can't use the same concepts. Message delays can vary significantly caused by routing and congestion with a dependability to the network structure. Additionally, message multicasting is not necessarily applied. TTEthernet, as a prominent switched network technology, accumulates propagation delays inside transferred messages to overcome the problem. A request/acknowledge based data exchange between multiple participants is done throughout start-up phases to establish an initial time [TTT11, p. 53]. Thereby, the start-up routine differs for high

reliable configurations.

Algorithm analysis is often done by functional simulation. However, functional simulation is often not capable of checking all system properties concerning system states or state transitions. Formal verification can be applied to provide a mathematical proof that system properties hold under every defined condition. Such a formal verification was done for the DACAPO and TTP start-up algorithm in [LP97]. TTEthernet functionality, with respect to clock synchronization and start-up behavior, was investigated by formal verification in [SD11] and extended in [Dut+12]. A survey of additional formal verifications used for start-up algorithms, like TTCAN, SPIDER, or FlexRay, is given in [SRR16].

2.7 Integration

Nodes need to be integrated into synchronous operation in case they are powered on late, loss of synchronization due to errors or reset throughout operation. For this, an integrating node has to determine the actual system state to bring its own state in synchronicity. This state, or at least the relevant sub-state, is typically provided by data frames of already synchronized nodes periodically throughout synchronous operation. These data frames can be either dedicated or combined ones. Dedicated data frames used for integration reduces the overhead with the disadvantage of utilizing schedule slots. Combined data frames increase the overhead but don't block schedule slots. Typically, most technologies use synchronization frames also for integration.

FlexRay provides a single frame type to cover all functionalities [Fle10, p. 183], including payload transfers. The *startup frame indicator field* inside the header is used for integration, leading, and following start-up as well. This bit is set to one a single time by a subset of nodes throughout each schedule execution.

Different frame types are used by TTP/C. A separated frame for the start-up and so-called normal frames for synchronous operation are used. Normal frames are further distinguished because they contain either implicit or ex-

plicit Controller State (C-state) information [Obe11, p. 98], which represents the system state. Implicit C-state information are only included into the frame Cyclic Redundancy Check (CRC). Already synchronized nodes must use their own C-state to verify the frame by calculating the CRC. However, the C-state can't be extracted from the frame, which makes the integration of unsynchronized nodes impossible. Explicit C-state frames instead, contain all required state information to perform an integration [TTT02, p. 40]. Implicit C-state frames are introduced by TTP/C to reduce the frame overhead.

FlexRay and TTP/C provides the system state periodically by use of multiple nodes. A different approach is used by Time-Triggered Controller Area Network (TTCAN), which incorporates the concept of multiple time masters [Füh+01]. However, during operation, only a single time master is active, responsible for providing its view of the system state by periodic reference message transmissions to all other nodes.

TTEthernet, as a switched network technology, encapsulates and distributes the system state inside IEEE 802.3 compliant Ethernet frames. The system state information is transmitted periodically inside Protocol Control Frames (PCFs). PCFs can be specified further to cover different purposes. The Integration Frame (IN) represents such a PCF specification used for node integration [Ste09]. They are created by Synchronization Masters (SMs) and transmitted to all Compression Masters (CMs) where they are processed. The resulting IN is finally transmitted back to all SMs. Compared to the bus-based technologies, it is not possible to use the received system state information without initial preprocessing by the CMs. The processed IN contains the actual system state and a membership field, which represents the number of already synchronized SMs. The integration can be completed successfully if a sufficient number of SMs are active.

2.8 Clock Synchronization

Uncertainties of physical oscillators lead to local clock drifts over time, which can't be entirely removed. Clock synchronization is used to align all local

clocks of a system within predefined boundaries. This alignment process is applied continuously throughout synchronous operation in order to keep the clock deviations shorter or equal to the allowed precision.

The problem of clock synchronization inside distributed systems was first addressed in [Lam78]. However, the introduced approach relies on a fault-free message exchange between all system nodes. As a consequence, a single faulty node can corrupt the whole system. Further research regarding fault-tolerant clock synchronization was published in [LM85]. Its results are the foundation for many other clock synchronization algorithms, including the fault-tolerant midpoint algorithm [LL88] used for this thesis.

Clock synchronization can be applied in different ways [KAH04]. One solution is *clock state correction* intending to correct clock values immediately to remove the deviation that was accumulated over time. However, this correction only removes the effect of oscillator drifts but doesn't prevent clocks from drifting apart again. The clock drifting itself can be reduced by applying *clock rate correction*. As explained in Section 2.2, clocks derive their values by counting oscillator events. This counting correlates to the rate of a clock. Rate correction adjusts the counting to decrease or increase the rate of a clock to compensate oscillator uncertainties.

It might be sufficient to use only rate correction for specific applications. This can be done if systems work on time differences instead with time values, as explained in [Lis91]. However, widely used time-triggered technologies like FlexRay, TTP/C, or TTEthernet combine both correction methods to achieve better synchronization and to tolerate less precise oscillators.

Clock synchronization can be further distinguished in internal and external synchronization. For internal synchronization, an ensemble of clocks is defined and used to synchronize their clocks. External synchronization is used if an ensemble of clocks is defined but synchronized to a separated single or an ensemble of clocks.

2.8.1 Processing Order

The process of clock synchronization can be defined generally in three steps. Initially, a node inside a system collects clock time values of other predefined nodes. These values are also called *remote clock estimates* and represent a relation between local clocks. The remote clock estimation was distinguished in two major approaches by [AP98, p. 13]. For time transmission techniques, a local clock value is sent by node N_i, based on its local time, to node N_n. The reception point in time at node N_n allows the estimation. However, latencies of the local clock distribution must be considered because it can affect the remote clock estimation quality drastically. The Remote Clock Reading (RCR) technique describes a request based remote clock estimation. Node N_i can trigger a local clock transmission at node N_n if required. The remote clock estimation additionally contains the request transmission latency.

Nodes can gather single or multiple remote clock estimates depending on the system. All these estimates, or a subset of them, are used to calculate a correction value for its local clock. Multiple remote clock estimates allow a fault-tolerant correction value calculation by discarding the extremes. The computation is done by *convergence functions* which are outlined more detailed in Section 2.8.2.

Finally, nodes use the prior calculated correction values to correct its local clock. The point in time and the way of correction value application differs between clock rate and state corrections.

2.8.2 Convergence Functions

Convergence functions take a set of remote clock estimates as inputs and provide a correction value used to correct a local clock. A detailed introduction of convergence functions is provided by [Sch86]. The work also contains evaluations of precision and accuracy boundaries for each investigated convergence function. An extended evaluation and classification of convergence functions is provided in [AP98, p. 19]. It defines *convergence-average* and *nonconvergence-average* techniques. For convergence-average techniques, a concrete clock value inside remote clock estimates is required to define

the correction value. Nonconvergence-average techniques only work with the presence of remote clock estimates. Convergence functions generally differ in computation complexity and their ability to tolerate faulty remote clock estimates.

Two convergence functions which are integrated in widely used communication technologies are given in the following. Notation $f(p_i, x_i, ...x_n)$ identifies a convergence function, where p_i is the processor or node that intends to execute the convergence function and $x_i, ...x_n$ represents the received remote clock estimates of p_i.

Fault-tolerant midpoint function. This function $f_{ftm}(p_i, x_1, ...x_n)$ is used by FlexRay [Fle10, p. 213] and has been introduced initially by [LL88]. The algorithm discards the k highest and lowest remote clock estimates. From the remaining values, x_{high} and x_{low} are used to calculate a midpoint that serves as the correction value. Parameter k depends on n, but the maximum number of discarded values is bounded to four (two lowest, two highest). The algorithm complexity can be considered moderate because it requires only the sorting of values and a division by two.

Fault-tolerant average function. This function $f_{fta}(p_i, x_1, ...x_n)$ is utilized by TTP/C [TTT02, p. 56] and has been published in [Dol+83]. The algorithm has been further analyzed for usage in a loosely coupled distributed real-time system in [KO87]. f_{fta} discards a prior defined number k of n received remote clock estimates. The remaining $n - 2k$ values are all used to calculate its average which is the final correction value. The algorithm complexity is similar to function f_{ftm}.

2.8.3 Technology Specific Application

The synchronization approach depends on multiple factors like network topology or required reliability. Bus-based network technologies like TTP/C and FlexRay often takes advantage of small message latencies and low jitters. Both technologies are designed to provide a fault-tolerant clock synchroniza-

tion. The general concept of collecting remote clock estimates is the same for FlexRay [Fle10, p. 209] [Rau07, p.53] and TTP/C [Kop03; KB03], although these technologies differ regarding its number of different synchronization frames. Several nodes transmit synchronization messages periodically, whereas all receiving nodes know the expected dispatch point in time due to the schedule. The comparison between actual and expected synchronization frame reception provides the remote clock estimates used for error correction. Multiple nodes are selected to provide these synchronization messages to allow fault tolerance depending on the underlying correction algorithm (convergence function).

Another bus topology is TTP/A that relies on a master based clock synchronization [KB03; KHE00]. A master periodically transmits specific *fireworks frames* that contains relevant information to establish a global time inside each reception node. Clock state correction is applied immediately after the reception of the second byte of each fireworks frame. The temporal distance or interval between the first two bytes of each fireworks frame is known and additionally measured by all receiving nodes. The interval value is used to adjust the local clock rate based on the measured deviation to the expected interval.

A different approach for aligning local clocks is applied by SAFEbus. It uses three different kinds of synchronization messages. *Initial Resync* messages for start-ups, *Long Resync* messages for integration and *Short Resync* messages to correct oscillator drifts [HD92]. These messages contain relevant information of the overall system state required to synchronize or integrate. However, all nodes finally synchronize on sync pulses, which is a low value on the communication bus, applied by all nodes at fixed points in time. The low value is recognized by all nodes and causes a freeze of local clocks. All local clocks continue running at the release of the sync pulse, which completes the synchronization process.

Message latencies in meshed networks can vary drastically, which causes significant jitters. However, the precise determination of latency and jitter is the foundation of synchronization for the bus-based technologies introduced in this Section. Hence, nodes connected by a meshed topology are not able

to collect remote clock estimates within an acceptable uncertainty by just detecting the reception point in time of synchronization frames. Instead, other mechanisms need to be established that allow a determination of the synchronization message latency between transmitting and receiving nodes. Network Time Protocol (NTP) [Mil91] was defined to solve the problem for very large networks like the internet. It is suitable for applications with synchronization requirements of a few milliseconds. A better synchronization quality is often required for industrial automation, military systems, and many other domains. These applications can utilize the Precision Time Protocol (PTP), defined in standard IEEE 1588, to achieve a synchronization quality in the area of microseconds or even sub-microseconds [IS08, p. 2]. Although it is possible to realize PTP in software, hardware timestamping is required to get high accuracy clock synchronization [LEG12].

TTEthernet deploys the PTP of standard IEEE 1588 to apply a delay accumulation inside synchronization frames [AK07]. A synchronization frame can pass multiple switches until the destination nodes are reached. Each traversed switch adds its delay, which enables the receiving node to determine the expected dispatch time of the respective synchronization frame [Ste+06]. The synchronization process itself requires two distinct applications of convergence functions [TTT11, p. 22]. Initially, a set of nodes, defined as SM, send a synchronization frame to all connected nodes marked as CM. CMs, which generally provides switching functionality, collect all synchronization frames to apply a first convergence function. The result, a compressed synchronization frame, is sent back to all connected SMs where a second convergence function is executed. Depending on the number of CMs, each SM finally receives n compressed synchronization frames used for the clock correction. The maximum limit of compressed synchronization frames is set to three for each synchronization attempt [Obe11, p. 200].

Chapter 3

Field Programmable Gate Arrays

3.1 Introduction

FPGAs are electronic devices capable of implementing user-defined hardware designs combined with a high degree of flexibility. The resources inside FPGAs are connected by control of configuration memories to provide required user functionalities. These memories are reconfigurable for the majority of available FPGAs, which allows rapid prototyping, inexpensive bug fixing, or functional updates. However, the fixed layout of FPGAs, e.g. hardware primitives, clk trees, and interconnect possibilities, are placed throughout manufacturing, which introduces disadvantages compared to ASICs. These disadvantages comprise an increase in silicon area and power consumption. A decreased performance must also be expected. Additionally, FPGA utilization is expensive if used for high quantity products. A detailed comparison about the area, performance and power gap between FPGAs and ASICs are given in [KR07]. The configuration of currently available FPGA devices is realized by three different memory types and introduced in the following.

Flash technology. These memories are based on the *floating gate* technology and utilized by Actel FPGAs [BSV11, p. 12]. An advantage of this

technology is the non-volatility. Additonally, radiation tests for a particular flash based FPGA have shown, that the configuration memory is not sensitive to Single Event Effects (SEEs) caused by radiation [Urb+13]. A drawback of this technology is the limited number of reconfiguration cycles [Mic15, Tab. 2-3].

SRAM technology. Utilized by most available FPGAs but requires a configuration each time the device is powered on because of its volatility. The configuration time is very short compared to flash-based devices. Additionally, this technology allows an infinite number of reconfiguration cycles [BSV11, p. 14]. A significant disadvantage of this technology is the sensitivity to ionizing radiation [Man+08; Caf+02].

Antifuse technology. The configuration of this memory type can only be applied once without any possibility of reconfiguration. High voltage is used to melt a resistance which permanently stores the required value inside each memory cell [BSV11, p. 13]. This type of configuration memory provides the maximum resistance against radiation effects.

FPGA designs are mainly written by use of Hardware Description Languages (HDLs) like VHDL [Soc08] or Verilog [IEE06] to provide a Register-Transfer Level (RTL) representation of the required functionality. Additionally, major Electronic Design Automation (EDA) vendors have started to support SystemVerilog [SG17] as a design language. Efforts in increasing the abstraction of hardware design development is ongoing by use of High-Level Synthesis (HLS). HLS enables FPGA designers to describe hardware in programming languages like C, C++, or SystemC, depending on the utilized compiler [Nan+16]. Evaluations between HLS and traditional HDL designing have shown that performance gains and reductions of development times are possible [Wan+15; PR09].

The remainder of this chapter is structured as follows. Section 3.2 provides an overview of the general FPGA architecture. The impact of radiation effects on electronic devices and its implications to FPGAs are discussed in

Section 3.3. The overall design flow is introduced in Section 3.4 followed by an overview of formal verification approaches given in Section 3.5. Aspects regarding traditionally applied functional simulation are covered in Section 3.6. Assertion-based verification is introduced in Section 3.7 due to its importance for simulation and formal verification. Finally, an overview of the most adopted verification framework is given in Section 3.8.

3.2 General Architecture

An abstract architecture of modern FPGAs is given in Figure 3.1 [KTR08]. It basically consists of functional blocks with the possibility of being intercon-

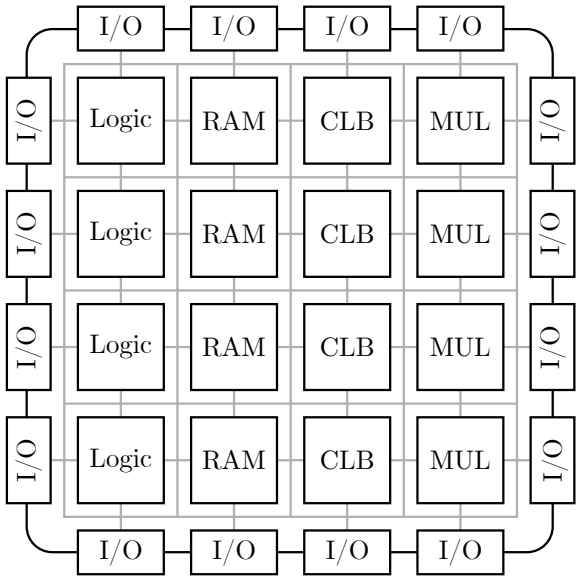

Figure 3.1: Basic FPGA architecture.

nected by routing channels. The configuration memory defines how the routing channels are configured and which blocks are actually connected to each other. A more detailed view on interconnect capabilities of meshed-based FPGA architectures is discussed in [PMM15, p. 48]. The given functional blocks can be classified into three types: I/O ports are used to connect the FPGA logic with external signals. Basic logic primitives provide the core

logic gates that are utilized for designs. Other blocks like Random-Access Memorys (RAMs), Multipliers (MULs), or hardware, used for creating and maintaining clks, can be treated as specialized ones.

A logic primitive can be configured to represent small boolean functions by utilizing Look-Up Tables (LUTs). Due to the interconnection capabilities, multiple logic primitives can be combined to implement arbitrary algorithms. Additionally, each logic primitive generally provides Flip-Flops (FFs), multiplexer, and special arithmetic functionality. Arithmetic functionalities are often provided by Digital Signal Processor (DSP) units as well.

It is recommended to use dedicated RAM units to store large data amounts instead of using FF. Otherwise, it is possible to run out of hardware resources that are required to implement the algorithms.

3.3 Radiation Effects

Electronic devices may be exposed to different types of radiation, which can be classified into two major categories: Charged particles (e.g. electrons) and electromagnetic radiation (e.g. ultraviolet light) [KCR06, p. 9]. The amount of energy which is deposited into a device is called Linear Energy Transfer (LET). It is defined by the deposited energy per unit path length (MeV/cm) divided by the density of the material (mg/cm^3) [BSV11, p. 44]. Electronic devices can be affected by radiation in different ways, ranging from transient to destructive effects. Devices can be shielded to reduce the overall dose level. However, shielding doesn't protect against SEEs caused by high energy particles [Duz05]. An overview of common radiation effects is given inside the following Sections 3.3.1 and 3.3.2.

3.3.1 Single Event Effects

Different types of SEEs exist and generally classified by *soft errors* and *hard errors* [BSV11, p. 45].[1] An overview of common SEEs are introduced in the following with respect to typical circuit elements [KCR06, p. 13] given

[1] SET, SEU and SEFI are soft errors. SEL is a hard error.

in Figure 3.2. These circuits generally consist of FFs and *combinational*

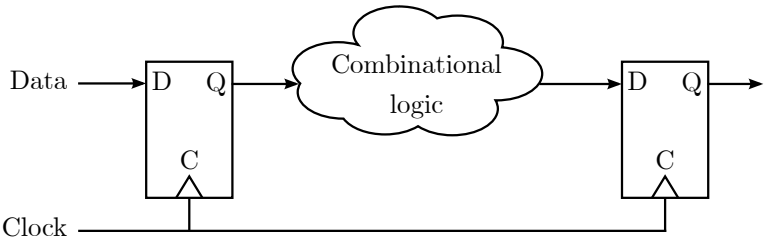

Figure 3.2: Typical circuit elements (combinational logic and flip-flops).

logic. FFs are used to define state machines or to store input values for further processing. Signals that are applied to combinational logic directly propagate through it. Additionally, combinational logic doesn't store any values. A circuit that defines its outputs based on its inputs and the actual memory content (state) is called *sequential logic*.

Single Event Transient (SET). SETs can occur in combinational logic of circuits. Charged particles that hit combinational logic can cause transient current spikes. The current spike can propagate to the output of the combinational logic, depending on its intensity. This, in turn, provides a probability of corrupting the system state, which happens in case current spikes and relevant clk edge events are present at FFs at the same time. The probability of corrupting FFs increases with higher system operating speeds [Dod+10].

Single Event Upset (SEU). SEUs can cause corruptions of memory elements [BSV11, p. 44]. It can happen to FFs, RAM cells used inside the circuit, but also to the configuration memory if SRAM based FPGAs are used. These effects are usually removed if the correct value is rewritten to the affected memory element. SEUs not necessarily implies an incorrect system behavior.

Single Event Functional Interrupt (SEFI). A SEFI is present if the radiation effect leads to a system that is not capable to continue a fault-free

operation [Kog+97]. This can happen if memory elements of state machines are corrupted or a program counter of processors.

Single Event Latch-up (SEL). SELs can cause permanent errors that induce a high current in the affected device [EDN04]. These high currents can be destructive if not recovered early enough by powering-off the device.

3.3.2 Total Ionizing Dose

Total Ionizing Dose (TID) describes a cumulative radiation effect depending on exposure time, particle flux and its LET [BSV11, p. 49]. TID radiation hardness of Commercial Off-The-Shelf (COTS) microelectronic devices has been extended over the past decade [Dod+10]. However, it is still an important aspect to consider because of its degrading effects inside electronic devices. These degradation effects are various, depending on the electronic parts and the accumulated TID. For example, Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) threshold voltages can change which leads to increased currents or even result in complete losses of functionality [Mau+08]. During flash device tests, an increased single bit upset sensitivity was observed for relative low TID levels [Bag+10]. The correlation of TID to SEEs has be investigated in [Sal+16]. However, no correlation was observed for the tested parts.

3.4 Design Flow

There are several approaches with different abstraction levels in use to implement hardware designs. A simplified design flow for recent Xilinx FPGA devices is shown in Figure 3.3. It can be divided into an optional HLS flow on the left side and the traditional RTL synthesis flow on the right side.

HLS is not a new approach. It started in the 1980s but failed to be adopted until the early 2000s. HLS tools generally focus on specific applications to achieve better results. Thus, it should not be utilized by default for all design types [MS09]. Xilinx HLS tool can process abstract hardware descriptions

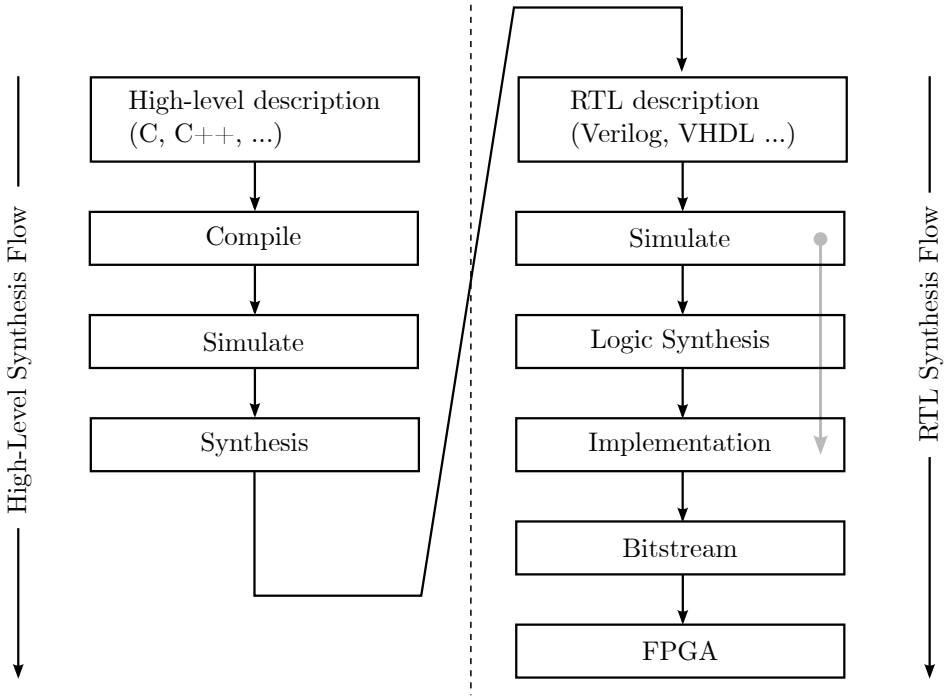

Figure 3.3: High-level and register-transfer level synthesis flow.

written in C, C++ or SystemC [Xil18b, p. 12]. However, multiple academic and commercial tools are available, able to process hardware descriptions based on other inputs (e.g. MATLAB or Handel-C) [Nan+16]. The high-level model is initially compiled, followed by simulation to ensure correct functionality on the abstract level. Finally, high-level descriptions are transferred into RTL descriptions by the use of synthesis. This transfer requires three important operations [Cou+09]. All required FPGA hardware resources need to be allocated. Additionally, the abstract hardware description doesn't contain clk cycle accurate behavior. A specific scheduling process is executed to solve this. Finally, variables and operations are bound to suitable functional units.

The RTL synthesis flow starts with verification/simulation of the RTL design. Different verification approaches exist and further discussed later in this chapter. The verification of hardware designs can be applied pre-syntheses, post-synthesis, and after implementation. The logic synthesis transfers RTL designs into technology independent representations of gates and macros.

These independent representations are finally mapped to specific FPGA resources [KB06, p. 229]. The final result of logic synthesis is a netlist, usually provided as standardized and vendor independent Electronic Design Interchange Format (EDIF). The netlist is further optimized throughout implementation in order to fulfill user constraints. Additionally, the implementation provides all capabilities to place and route the netlist into a concrete FPGA device [Xil18a, p. 7]. Finally, the targeted FPGA can be configured by use of the bitstream file.

3.5 Formal Verification

Formal Verification (FV) and its applied approaches rely on mathematical concepts to determine correct system behaviors. It has been considered as a verification technique applied to special or rare corner cases for a long time. Today, FV is often utilized alongside functional simulation to compensate weaknesses for certain types of designs or specific domains. However, FV has even started to replace traditional verification methods if the design is suitable [Kai+09]. The number of problems which can be addressed by FV are diverse. Three major approaches, commonly used throughout FPGA and ASIC design, are introduced in the following.

3.5.1 Equivalence Checking

Three different types of equivalence checking are commonly used. *Combinational equivalence* is applied to combinational design parts only as explained in [DS07]. *Sequential equivalence* incorporates state elements to proof cycle-accurate equivalence between a given reference model and the alternative implementation [MS05]. It is often used between different representations of the same system [DH02]. Several RTL re-designs are usually applied throughout FPGA developments. These re-designs often take place due to optimization purposes, e.g. new logic is inserted to reduce power consumption or critical paths are eliminated to increase the operating frequency. Instead of applying functional simulation for each re-design, it is sufficient to ensure the system

output behavior is equal for all input combinations compared to the reference RTL. Additional computation prior to Equivalence Checking (EC) is required, if a one-to-one interface correspondence between two system representations doesn't exist [MM04, p. 3].Functional simulation can take weeks or even months before completed. EC instead, may provide the results in hours or even minutes[2]. However, the actual computation time directly depends on the system state-space. Equivalences can also be checked between RTL and corresponding gate-level circuits to ensure synthesis was performed correctly. In contrast to model checking, it is not necessary to describe formal system properties or specifications to apply EC.

Transaction-based equivalence can be considered as the most recent variant. It is used for comparing high-level and RTL descriptions. It provides the freedom of utilizing models without a cycle-accurate specification [SSK15, p. 229].

3.5.2 Clk Domain Crossing

Recent studies have shown that most FPGA and ASIC designs utilize 3 or 4 clk domains. But also designs with 20 or even more clk domains were developed [Fos18a, Fig. 1-2] [Fos18d, Fig. 7-3]. Multiple clk domains require data is passed between them. Several ways are described in [Cum08] to handle Clk Domain Crossing (CDC) properly. CDC techniques shall prevent the propagation of metastability into the system. Metastability occurs if setup and hold times of sampling units (FFs) are violated, which results in unpredictable states on the FF output. These violations can't be removed for asynchronous multi-clk designs, but isolated by CDC techniques to prevent system operation is failing.

These checking tools investigate all the design parts involved in CDC, applicable to RTL and gate level. Typical design flows for RTL and gate level checks are published in [CH17]. Gate level CDC checks are recommended to ensure synthesis has not broken existing CDC paths. Synthesis flows might

[2]Durations are provided by Mentor Graphics for tool *Questa Sequential Logic Equivalence Check (SLEC)* [Gra19].

also introduce inadvertently new CDC paths. However, the setup and debug effort throughout gate level CDC checks is considered as much higher compared to RTL CDC checks.

3.5.3 Model Checking

This kind of verification applies property checks to a formal model of the system intended to be verified. These models are usually extracted from RTL descriptions if used for ASIC or FPGA designs. Additionally, system specifications need to be defined in a way that Model Checking (MC) tools can apply them. These specifications are often expressed by *temporal logic* like Linear Temporal Logic (LTL), first introduced in [Pnu77], or Computation Tree Logic (CTL). Throughout evaluation, the MC tools apply legal input combinations to the model in order to violate system specifications [Coh+16, p. 238]. Each violation is reported as a counterexample, which represents the sequence that leads to the failure. MC explores whole system state-spaces, which is a major difference to functional simulation. If no counterexample is found, a specification holds under every legal input sequence. MC is frequently represented by Formal Property Verification (FPV) in the domain of digital circuit design. A concrete problem is addressed by FPV inside this work and outlined in detail in Section 6.3.

MC tools often using Binary Decision Diagrams (BDDs) [DS01] to express the system states and transitions between them. They apply optimization and reduction to the BDDs in order to keep the overall system state manageable. However, it has been observed that BDD can grow exponentially if the model size increases [FLS15]. As a consequence, larger designs are partitioned into smaller parts to reduce the explorable state size. Each part is then individually checked. Additionally, constraints are applied to MC tools by use of *assume* statements, which can reduce the state-space further.

A major advantage is the early application of MC throughout FPGA development cycles. It can be applied as soon as RTL code is available without the need to have a testbench, as it is required for functional simulation.

MC can be additionally used to explore designs by defining cover state-

ments [SSK15, p. 111]. Thus, MC tools provide input sequences that show the design behavior intended to be investigated, without the need of having a testbench ready.

3.6 Functional Simulation

Functional simulation represents the verification method mostly used for FPGA designs. The Device Under Test (DUT) is placed inside a testbench which is able to control DUT inputs. These inputs lead to a reaction of the DUT, which is compared with its requirements. Simulation can be applied in different stages during FPGA verification flows. RTL simulation is used to verify the correct behavior of RTL code without concerns about timing delays. Gate level simulation incorporates the timing information generated by place and route but requires more time for execution [Sim15, p. 181].

Several approaches exist in the field of simulation. They differ in the way of providing stimulus to the DUT or how they check for correct DUT behavior. An overview of the most relevant concepts and their differences is introduced in the following.

3.6.1 Constrained Random Verification

Stimulus, applied to the DUT, must be defined in some way. This can be done either manually or by utilizing a randomization engine. The manual stimuli definition targets a specific operation of the DUT and is also known as *directed testing*. Additionally, directed testing often comprises manual evaluations of DUT responses to ensure they are compliant with the requirements [ST12, p. 5].

The randomized stimuli definition is part of Constrained Random Verification (CRV). Constraints are provided to a randomization engine in order to generate a specific range of input stimuli. However, this approach implies that different related activities are automated. The actual applied DUT stimulus and responses must be tracked by *functional coverage* to ensure all DUT functionalities were tested according to its requirements. Additionally,

all DUT responses must be checked automatically in order to handle the huge amount of test cases.

In contrast to directed testing, more effort is required to prepare a test environment able to handle all CRV related activities. The usual progress of CRV and directed testing is given in Figure 3.4 [Meh18, p. 66]. CRV per-

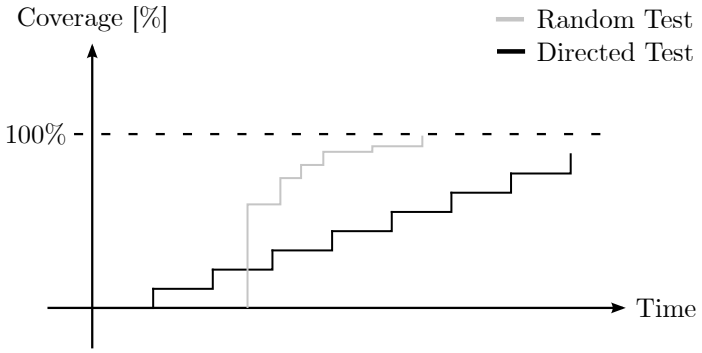

Figure 3.4: Verification progress of random versus directed testing.

forms the verification process faster, although it requires a longer setup time. Once started, CRV rapidly collects a large amount of coverage. Constraints are modified to target the unexplored DUT behavior in order to collect the missing coverage. The randomized stimulus application also reveals bugs in design parts where they are not anticipated [ST12, p. 7]. In contrast to CRV, directed testing provides a constant but flat progress.

3.6.2 Languages

Testbenches are written in different languages. However, recent studies indicate thatVHDL, Verilog and SystemVerilog are used mostly for verifying FPGA designs [Fos18c, Fig. 6-2].

VHDL and Verilog are dominant for system design. They were also selected as primary verification language for a long time. Continuous increases in system complexity and a limited capability of HDLs to cover all necessary verification needs lead to the definition of Hardware Verification Languages (HVLs). OpenVera, e, SystemC, and SystemVerilog are the most

commonly known HVLs. They typically provide stimulus randomization to apply CRV, functional coverage tracking, and high-level programming features, e.g. object-oriented class-based design.

Additionally, some languages are used to establish methodologies or frameworks. These methodologies provide patterns to solve typical verification problems, starting from stimuli application for different access types[3] up to support for register modeling. Current study results that represent the adoption of different methodologies are given in [Fos18c, Fig. 6-3]. It shows that UVM [Soc17], which is based on SystemVerilog, dominates in the application of methodologies.

3.7 Assertion-based verification

Assertions are used to describe system properties in a formal way. They are mostly used to express design behavior over time if used for verifying hardware designs. Assertions mainly defined by SystemVerilog Assertions (SVA) [SG17, p. 364] or Property Specification Language (PSL) [IEE10] whereas both languages are well supported by the main EDA vendor tools. It has been published that Assertion-Based Verification (ABV) adoption can lead to a drastic reduction of verification efforts up to 50% [Y+00]. Several benefits, identified in [Cer+10, p. 7], can improve the overall verification process by utilizing ABV.

Formal design specification. A system property expressed by assertions is distinct and doesn't provide room for interpretation. Instead, system specifications written in natural languages are often ambiguous.

Improved bug detection. Simulation can be considered as *black box* testing in most cases. Input stimuli are applied to the DUT and possible faults need to propagate from internal structures to the interface boundary to be recognized by the test environment. However, it is possible that DUT faults

[3] E.g. to support *Bidirectional Non-Pipelined* or *Out of Order Pipelined* transfers.

are masked or isolated internally [RA16], possibly leading to non-trivial bugs that are found late in the design cycle or completely missed. Assertions can be placed at unit interfaces on arbitrary hierarchy levels or inside design code to address this problem. They can also reference arbitrary signals inside the design if required. Assertions that detect faulty behavior indicate a violation of the concrete covered system property. Problem root causes are localized very fast by the help of assertions, often with additional tool support like *assertion thread viewer*.

Formal and simulation-based application. Successfully applied assertions can be adopted for use by formal and simulation-based methods, which increases efficiency [Y+00].

3.8 Universal Verification Methodology

UVM provides an IEEE standardized verification framework [Soc17] based on the HVL SystemVerilog. It was defined and developed by the main EDA vendors based on the experiences collected during previous methodology standardization processes like Open Verification Methodology (OVM) or Verification Methodology Manual (VMM). By now, UVM is the de-facto standard methodology for functional verification of RTL designs as indicated by the latest survey results in [Fos18c, Figue 6-3] [Fos18b, Figue 10-3]. The study shows an approximate UVM usage of around 50% for FPGA developments and about 70% for ASIC developments in 2018 which represents a large gap to all alternative methodologies. Additionally, an ongoing utilization uptrend is present for the given statistics, ranging from the year 2014 to 2018.

3.8.1 Architecture

A basic UVM testbench architecture is shown in Figure 3.5. It consists of dynamic SystemVerilog (SV) class objects which are connected to static SV

interfaces in order to interact with the DUT. The architecture is divided into

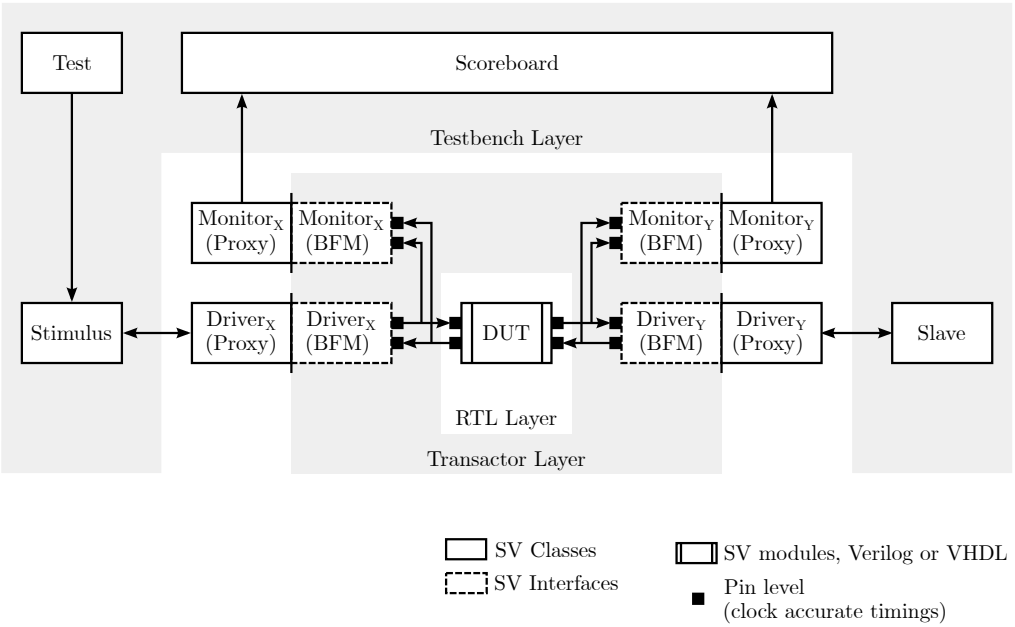

Figure 3.5: Basic UVM architecture.

three layers to increase reusability. Its testbench layer solely contains SV class objects, able to communicate with other objects inside this layer by use of transaction objects. These transaction objects are exchanged by function or task calls without specific DUT timings. Connections to DUT signals can be established for individual cases like reacting on specific hardware events, by referencing SystemVerilog interfaces. However, for most cases, no direct DUT connections exist for the testbench layer. The *Monitor* and *Driver* proxies act as an interface between testbench and transactor layer. Finally, Bus Functional Models (BFMs) operate at the DUT pin level with respect to clk accurate timings, required to drive the DUT input signals and to store its responses.

It is also possible to include the proxy objects into the transaction layer with the disadvantage of limiting portability. However, the integration increases flexibility because BFM tasks would be executed by SV class objects rather than SV interfaces[4].

[4]SV interfaces are limited because they don't provide object-oriented features like in-

3.8.2 SystemVerilog Assertion Integration

SVAs are used to describe and check system properties. These assertion checks can be of type *immediate* and *concurrent*. Assertions that check boolean values without relation to clks are immediate. Concurrent assertions are used whenever system behavior is checked for single or multiple clk cycles. SystemVerilog provides *properties* and textitsequences to express system specifications. Properties become true or false at the end of each evaluation, whereas sequences are used to define more complex properties.

An example sequence, used within a property and checked by an assertion, is shown in Figure 3.6. The property evaluation starts for each rising edge of variable *req* in order to check for a high value of variable *ack* within one to three clk cycles. SVAs are commonly placed inside SV interfaces or modules

```
1  default clocking @(posedge clk); endclocking
2
3  property p_req_fol_ack;
4    $rose(req) |-> s_ack_high;
5  endproperty: p_req_fol_ack
6
7  sequence s_ack_high;
8    ##[1:3] ack;
9  endsequence: s_ack_high
10
11 a_reqack: assert property (p_req_fol_ack)
12 begin
13   // Action block positive evaluation
14 end else begin
15   // Action block negative evaluation
16 end
```

Figure 3.6: Code example of an assertion check based on property and sequence definitions.

to build an encapsulated and reusable verification unit. A dedicated checker statement exists to provide specific capabilities for creating verification units.

heritance and polymorphism.

However, these units also introduce some limitations compared to modules or interfaces [Cer+10, p. 447]. SV interfaces are the only unit that can be referenced by class-based objects. Thus, checkers need to be implemented there in case UVM environments are required to exchange data with SVAs.

These direct interactions between UVM environments and SVAs can improve verification processes is different ways. It might be required to deactivate assertions for specific test runs as introduced in [Lit13]. Additionally, SVA evaluations can be modified over time depending on the DUT system state as published in [Coh13]. UVM environments can provide this information to all relevant SVA in order to update their behavior dynamically throughout verification runs. Another use case of UVM scoreboard support is introduced in [BMD19], which targets the verification of volatile status registers. The author uses SVAs to capture volatile register values at specific points in time. These values are forwarded to the UVM scoreboard, which allows a combined register model comparison for volatile and non-volatile registers. Volatile register verification must be considered as non-trivial because it is not covered by the general UVM verification flow [Lit14].

Chapter 4

SpaceWire

4.1 Introduction

SpaceWire is a communication technology mostly used on-board spacecraft to connect instruments, on-board computers, or other units. It provides full-duplex, bidirectional and serial communication transfers with data rates between 2 - 400 MBits/s [ESA08, p. 13]. Two SpaceWire interfaces are connected point-to-point to establish a *link*.

The remainder of this chapter is structured as follows. Section 4.2 introduces the SpaceWire layers and discusses problems concerning real-time capabilities. An overview of existing SpaceWire modifications and extensions is given in Section 4.3. These extensions either provide concrete solutions, or they contribute concepts that might help to address the real-time issue.

4.2 Layers

SpaceWire is divided into six different layers to define its complete functionality, which is outlined in the following.

4.2.1 Physical Layer

The physical layer defines the properties of cables and connectors. Additionally, Printed Circuit Board (PCB) design recommendations are given to ensure proper track routing. The standard allows cables with lengths up to 10 m and considers certain characteristics (impedance, skew, attenuation, crosstalk) to achieve high data rates up to 400 MBits/s. Extensive Electromagnetic Compatibility (EMC) tests were done in order to meet the requirements for typical spacecraft [ESA08, p. 24].

4.2.2 Signal Layer

LVDS is used as a transmission standard that provides a high immunity to signal noise combined with low power consumption. The Data-Strobe (DS) encoding scheme is used for all data transfers and originally defined in [IEE96]. An example trace for this encoding is given in Figure 4.1. Signal *D* provides

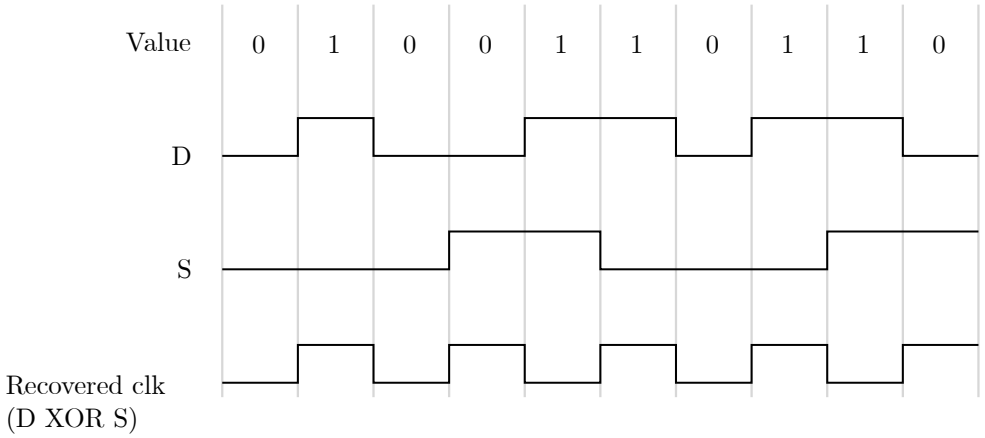

Figure 4.1: Data-strobe encoding with recovered clk.

the actual transmitted/received value. Signal *S* changes its value in case D remains constant. However, it is not allowed that both signals change their values at the same time. Clk recovery can be applied by XORing both signals.

DS encoding tolerates signals skews between D and S to almost 1-bit time. In contrast to that, synchronous data transfers, controlled by a separate clk

signal, only tolerate 0.5-bit times [ESA08, p. 24].

4.2.3 Character Layer

Transferred bits are interpreted in characters. SpaceWire defines *data characters* and *control characters* [ESA08, p. 52] as illustrated by Figure 4.2. Both

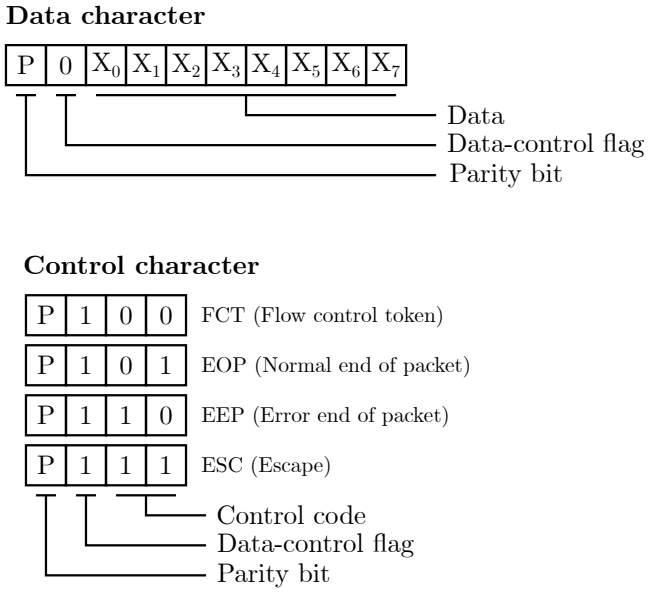

Figure 4.2: SpaceWire data and control characters.

character types contain a parity and data-control flag. The data-control flag is used to distinguish between both character types. The parity bit protects the data-control flag and the parity bit of the actual character. Additionally, it protects user or *control code* values of the previous character. These are either eight bits (X_0 to X_7) for data characters or two control code bits used for control characters.

Four control character types exist. Normal End Of Packet (EOP) and Error End of Packet (EEP) characters represent a packet boundary and mark packets as faulty or fault free. Single Flow Control Tokens (FCTs) indicate a receiver has memory space left for further data character receptions in order to prevent congestion. Escape (ESC) characters only used in combination

with FCTs or data characters to form two possible *control codes* [ESA08, p. 53] as shown in Figure 4.3.

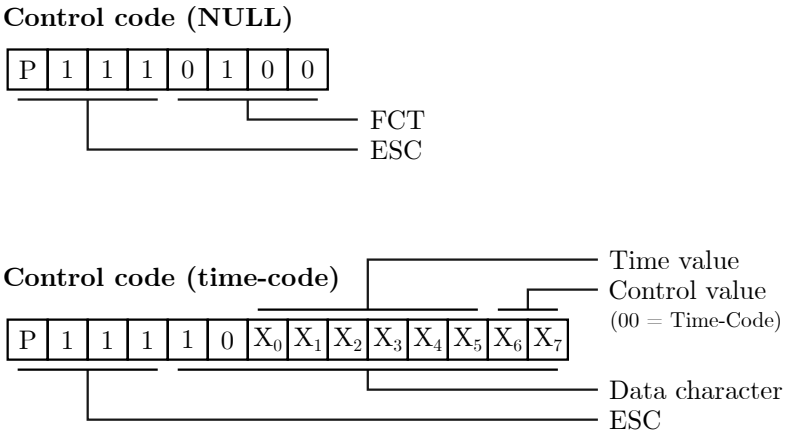

Figure 4.3: Initially defined control codes (NULL and time-code) before actual SpaceWire standard revision.

NULLs are transmitted periodically to keep already established links up. A timeout is recognized in case NULLs are missing, which starts a link recovery procedure. They are also used during link initialization.

Time-codes are used to transmit high priority messages, consisting of six bit time value and two bit control value. SpaceWire data transfers can be considered as a stream of the three introduced elements (data characters, control characters, and control codes). Each element is transferred individually and can't be interleaved during transmission. However, these elements are transferred with different priorities. Time-codes have the highest priority which allows them to interleave large data streams consisting of multiple data characters.

Distributed interrupt codes are introduced by the revised SpaceWire standard [ESA19, p. 88] and shown in Figure 4.4. Time-codes and distributed interrupt codes are now classified as *broadcast-codes*. Distributed interrupt codes have the same structure as time-codes and provide request/acknowledge based interrupt capabilities. They have a lower priority than time-codes but still precedence over data characters. Distributed interrupts allow trans-

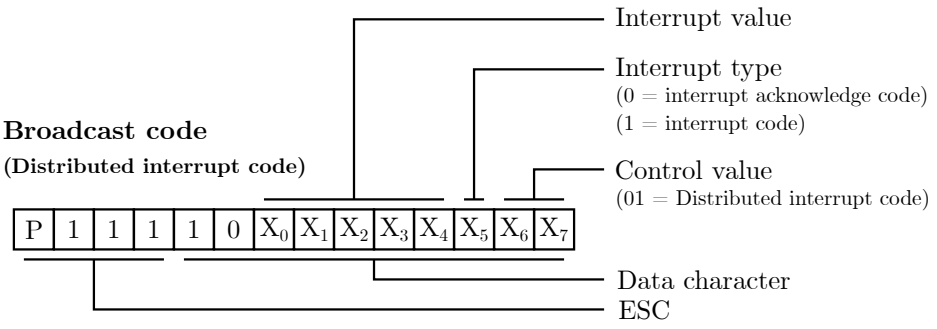

Figure 4.4: Recently added distributed interrupt classified as broadcast code.

mission of arbitrary interrupt values. This is a major difference to time-codes who require time value increments of 1 ($value_n = value_{n-1} + 1$) [ESA08, p. 84]. Additionally, distributed interrupts can be extended by incorporating the interrupt type bit into the interrupt value field. In these cases, the interrupt value consists of six bits by removing all acknowledge capabilities.

4.2.4 Exchange Layer

The exchange layer defines how links between two SpaceWire interfaces are initialized, how the data flow is controlled throughout normal operation, and how errors are detected and recovered. This layer additionally classifies Normal Characters (N-Chars) and Link Characters (L-Chars). Characters and control/broadcast codes, introduced in Section 4.2.3, are classified by their layer accessibility. N-Chars are those that are passed to the packet layer (data characters, EOP and EEP). All others remain inside the exchange layer and part of L-Chars.

Link initialization between two connected SpaceWire interfaces is done by following a handshake procedure. It starts with a transmission of NULLs by $SpaceWireInterface_x$ in order to wait for a NULL response of $SpaceWireInterface_y$. After that, $SpaceWireInterface_x$ transmits FCTs. The procedure is completed once $SpaceWireInterface_x$ receives a FCT of $SpaceWireInterface_y$. After this procedure, both SpaceWire interfaces are

allowed to transmit user data, e.g. data characters and broadcast codes (time-codes, distributed interrupt codes).

Flow control is used after link initialization has finished. It ensures that N-Chars are only transmitted if the receiver has memory space left to store them. For this, $SpaceWireInterface_x$ provides FCTs to $SpaceWireInterface_y$. A single FCT, received by $SpaceWireInterface_y$, indicates that $SpaceWireInterface_x$ has memory space available for at least eight N-Chars.

Several error conditions can be detected. A *link disconnect error* is recognized on receiver side if no data is received for 850 ns. A *parity error* occurs if the parity field inside data or control characters indicates data corruption. Invalid control codes are marked as *ecape errors* whereas an unexpected reception of N-Chars causes *credit errors*. Specific handshake mismatches during initialization are covered by *character sequence errors*. All errors lead to an exchange of silence. This is required to ensure both ends of the link reinitialize the connection.

4.2.5 Packet Layer

The structure of SpaceWire packets is fairly simple, as shown in Figure 4.5. Identifier *spwPacket* defines a complete SpaceWire packet. It allows an arbitrary number of *destinationAddress* values used for routing packets through networks. Value *destinationAddress* may be skipped for point-to-point connections if required. The arbitrary sized *cargo* field[1] contains user data. All packets must be closed by the *endOfPacket* field.

A protocol identifier can be used if multiple different protocols are required to be transmitted [RE10, p. 12]. However, value *destinationAddress* must be present for these cases to provide a logical address in order to determine the correct position of the protocol identifier within received data streams.

[1] Allowed to be skipped from the technical point of view.

```
spwAddress         = data character;
logicalAddress     = data character;
cargo              = data character;
EOP                = control character;
EEP                = control character;
endOfPacket        = EOP | EEP;
destinationAddress = spwAddress | logicalAddress;
spwPacket          = {destinationAddress}, {cargo}, endOfPacket;
```

Figure 4.5: Structure of a basic SpaceWire packet.

4.2.6 Network Layer

Routers are used in case more than two SpaceWire interfaces are required to be connected. These routers can be treated as a collection of SpaceWire interfaces connected by a crossbar. Crossbars can connect each SpaceWire input to an arbitrary SpaceWire output, whereas each output arbitration is done independently and in parallel. Prioritization can be used to control the arbitration. It allows a prioritized forwarding of specific packets to establish ordinary QoS.

Packets that arrive at routers are forwarded to the required router output, depending on their addresses. Addresses are located at the first packet byte and can be of type *logical address* or *path address*. Logical addressed packets are forwarded according to routing tables, which provides the relation between address and required output. Path addressed packets are used to select the required output directly without routing table access.

Path addresses are deleted before packets are forwarded [2]. This allows the traversal of multiple cascaded routers by using path addresses only. Logical addresses are also allowed to be deleted. However, this must be enabled explicitly inside the router configuration for each logical address individually.

Packets inside routers are assigned to the required outputs as soon as addresses are received. This kind of switching technology is known as *wormhole routing*. [ESA08, p. 92]. Wormhole routing is utilized to reduce the required

[2] This address deletion is also known as *header deletion*.

memory size drastically inside routers. A survey of research contributions and commercial ventures in the field of wormhole routing is published in [NM93].

A prioritization scheme is used to allow packets are arbitrated and forwarded prioritized depending on their addresses. However, cascaded routers can cause congestion situations that remove the benefit of prioritized packet transfers, as outlined in Figure 4.6. Prioritized packets are labeled with p,

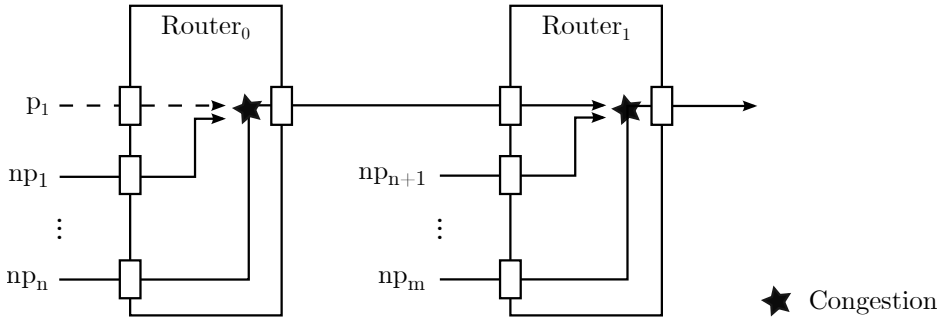

Figure 4.6: Prioritization problem for cascaded routers.

non-prioritized packets are labeled with np. For the given scenario, p_1 competes with $np_1 ... np_{n-1}$ to get access to the output of $Router_0$. However, p_1 will be forwarded next because it is the only prioritized packet at the input of $Router_0$. np_n currently blocks the output of $Router_0$ and is requesting access for the output of $Router_1$. In a worst case scenario, all other packets at the inputs of $Router_1$ are granted to the output before np_n which in turn blocks p_1 as well. As a consequence, the prioritization capabilities of p_1 are restricted to $Router_0$.

4.3 Extensions

An overview of SpaceWire extensions that contribute concepts and ideas, relevant for real-time transfers, are introduced in the following.

SpaceWire Time Distribution Protocol (TDP). The approach relies on a master based time distribution. A single *initiator* transmits periodi-

cally time-codes which are received by an arbitrary number of *targets*. TDP is capable of determining and compensating time-code latencies between the initiator and each target by utilizing distributed interrupts and hardware timestamping. Jitter and drift mitigation is performed inside targets. They determine the local clock differences to the initiator based on time measurements between consecutive time-codes [Sak+14]. A tailored version of TDP is used the first time inside a real flight mission for the *JUpiter ICy moons Explorer* project [TIM16].

SpaceWire Network Discovery and Configuration Protocol (NDCP). This protocol is used to discover and configure SpaceWire networks. It considers every SpaceWire router and SpaceWire end-point as a *Device*. Furthermore, NDCP defines *Control Devices* which are used to manage *Peripheral Devices*. Multiple parameters are available for Peripheral Devices. They are accessed by Control Devices to identify the kind of device (Node or Switch) and its properties. A graph model of the whole system is created if all available information about the network and its nodes are collected [Rom+16]. Establishing an overall view of the system can be important because its structure often influences compensation methods used during clock synchronization.

SpaceWire deterministic (D). This approach defines a time-triggered data transfer over SpaceWire with data exchanges controlled by the Remote Memory Access Protocol (RMAP) [Gib+16]. The global time is distributed centralized by using time-code values. These time-code values directly define the executed schedule slots. Hence, the overall schedule length is limited to 64 slots caused by the time-code structure. The time-code distribution frequency can be parameterized but need to be constant once the system starts time-triggered operation. This leads into equal slot lengths for all 64 available slots.

SpaceWire reliable (R). This protocol provides reliable data transfers by utilizing various functionalities [Mic+16]. It uses automatic packet retrans-

mission based on acknowledged data exchanges. Additionally, large sizes of user data can be segmented over multiple SpaceWire-R packets. It is also possible to establish several parallel data streams between source and destination. Destinations can apply flow control in case they are not able to process the overall incoming data. For this, receivers inform transmitters about the number of packets they can process. The concept is similar to the flow control implemented by SpaceWire on the character layer. Heartbeat packets are transferred in case no user data are available to detect faults like broken connections or non-responding destinations. However, all these properties don't guarantee real-time data transfers.

Part III

Contribution

Chapter 5

Approach and System Design

5.1 Introduction

The concept of clock synchronization, introduced in Chapter 2, can be considered as a precondition for time-triggered data transfers. Furthermore, clock synchronization requires the collection of remote clock estimates. This chapter introduces a novel approach for collecting these remote clock estimates and provides an overview of the complete system architecture, which is used for evaluation purposes.

The introduced approach allows clock synchronization without accumulating packet latencies between source and destination. Instead, remote clock estimates are gathered by recognition of pulses/interrupts which are transferred over the network. The utilization of SpaceWire evaluates this concept because it provides these interrupts built-in. However, the concept could also be transferred to other communication technologies if the character layer of SpaceWire is imitated. The introduced approach additionally provides the ability to handle clock synchronization in a decentralized way. This allows a fault-tolerant time distribution, which is a major difference compared to all existing SpaceWire extensions.

The remainder of this chapter is structured as follows. A detailed introduction of the pulse-based remote clock estimation is given in Section 5.2 followed by an overall system architecture discussed in Section 5.3. The sys-

tem start-up, integration, and clock synchronization process are outlined in Section 5.4. Finally, a modified SpaceWire interface is introduced in Section 5.5, with the intent of reducing broadcast code transmission jitters to achieve better synchronization qualities.

5.2 Pulse-based Remote Clock Estimation

The estimation of remote clocks if fundamental to establish a system-wide clock synchronization. Pulses with predefined characteristics are used for the introduced approach in order to receive clock estimates instead of applying delay accumulation techniques. Its general working principle and usage, in combination with SpaceWire as a concrete communication technology, is given in the following.

5.2.1 General Approach

Clock synchronization is performed while time-triggered systems execute their schedules. During schedule execution, data are exchanged in order to collect remote clock estimates, which are used for the convergence function during clock synchronization. These remote clock estimates represent a relation between local clocks inside the system. For centralized approaches, only a single remote clock estimate is used to synchronize clocks. Decentralized approaches, like the one introduced in this thesis, collect multiple remote clock estimates throughout schedule executions and allow fault-tolerant clock synchronizations.

The pulse-based remote clock estimation uses a schedule shown in Figure 5.1. The schedule is executed periodically and consists basically of three schedule slot types. *Data* slots are assigned to arbitrary nodes. They are used to exchange user data and don't affect the clock synchronization. *Time* and *pulse* slots are executed by a subset or all available nodes ($Node_0$ - $Node_n$). Each selected node transmits this slot pair during schedule execution to all other nodes. Time slots contain information about the executed slot, cycle, and a unique reference number that belongs to the actual transmitting node.

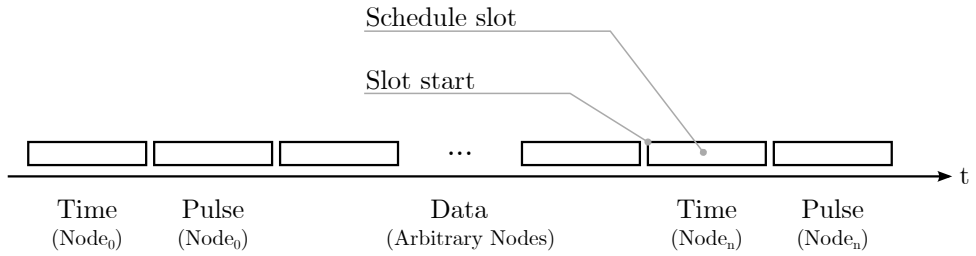

Figure 5.1: Example schedule used to distribute time information.

This reference number is also transmitted inside the pulse slot. Nodes that receive time and pulse values validates the connection between both slot information. Time slot values are transmitted like user data. It must be ensured that transmission is completed inside the slot without any further temporal constraints. In contrast to that, pulse slot values are transmitted with priority in order to provide predictable transmission latencies. It is technically possible to transmit pulse slot values inside Data slots to increase the overall efficiency instead of assigning them to dedicated slots.

The concrete remote clock estimation for a particular pulse slot is explained in the following based on Figure 5.2. It illustrates a pulse slot from the trans-

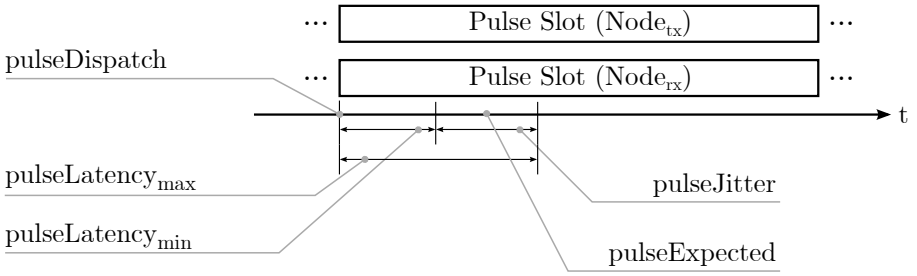

Figure 5.2: Pulse reception for perfectly synchronized local clocks.

mitting and receiving node point of view. This is required because slots are executed based on the local clock of each node. Both local clocks are perfectly synchronized for the given figure. $Node_{tx}$ transmits a pulse value at the slot

beginning according to its own local clock[1]. $Node_{rx}$ (and all other receiving nodes) has information about the pulse dispatch point in time because of the schedule. No local clock drifts are present between $Node_{rx}$ and $Node_{tx}$ for the given example. Thus, the pulse dispatch happens at the same local time for both nodes. Additionally, minimum and maximum pulse latency values are known. These latencies depend on the network path between the source and destination node and determined before system operation. Pulse jitters directly correlate to the quality of the collected remote clock estimates and defined by the temporal variance of pulse receptions (Equation 5.1). The expected pulse reception time (Equation 5.2) must be calculated in order to determine the final remote clock estimate *rce* (Equation 5.3) which expresses the temporal difference between the local clock of the pulse transmitting and receiving node.

$$pulseJitter = pulseLatency_{max} - pulseLatency_{min} \tag{5.1}$$

$$pulseExpected = pulseDispatch + pulseLatency_{min} + \frac{pulseJitter}{2} \tag{5.2}$$

$$rce = pulseExpected - pulseActual \tag{5.3}$$

The closer *rce* values converge to 0, the better local clocks are aligned. The expected pulse reception time *pulseExpected* is defined based on the local clock of each receiving node and set into the *pulseJitter* center. The subtraction between pulseExpected and the actual pulse reception point *pulseActual* defines the local clock difference which can be negative or positive. The following system properties affect the *rce* value and in turn influence the overall clock synchronization quality.

[1]This representation is simplified for illustration purposes. In fact, the pulse value dispatch is delayed by the system precision value and possibly further delayed to allow slot preparation.

Jitter. An increased size of *pulseJitter* directly enlarges its introduced uncertainties. The maximum *rce* value is defined as follows

$$rce_{max} = \pm\frac{pulseJitter}{2} \tag{5.4}$$

for perfectly aligned local clocks. Jitter values represent constant uncertainties that can't be compensated by clock synchronization algorithms. As a consequence, *rce* values are processed only beyond a specific threshold to ensure uncertainties, caused by real clock drifts, are present. However, jitter values may be decreased by modifying the overall system behavior. This work presents such a modification for the used communication technology SpaceWire, introduced in Section 5.5.

A general overview of approximated jitter ranges of synchronization messages is given in [Kop11, p. 70]. It indicates that hardware implementations are capable of maintaining jitter ranges of less than 1 μs. In contrast to that, kernels of operating systems imply jitter ranges between 10 - 100 μs. Even worse jitter ranges of 10 μs - 5 ms are expected if synchronization messages are handled at the application software level.

Local clock drift. This causes clock value differences, which leads to offsets in schedule slot executions, as shown in Figure 5.3. This local clock offset

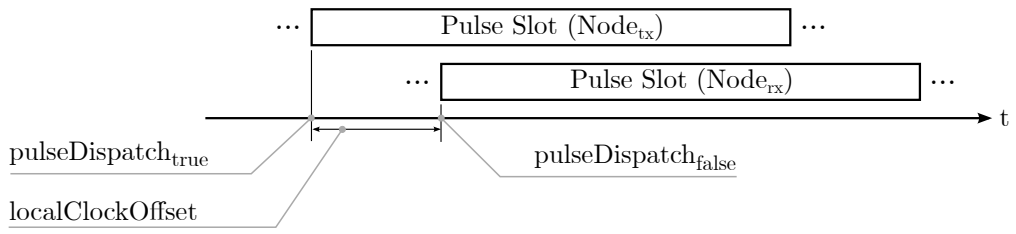

Figure 5.3: Local clock drift effect on pulse determination.

doesn't affect relative values like latencies or jitter. However, as shown in Equation 5.3 and 5.2, the remote clock estimation also includes the pulse dispatch as an absolute value to determine the expected pulse value reception.

Thus, local clock offsets can increase the deviation between expected and actual pulse reception.

A centered position of *pulseExpected* allows a balanced reaction on local clock differences for positive and negative values. Figure 5.4 shows the distance from *pulseExpected* to the jitter boundaries. As already stated, remote

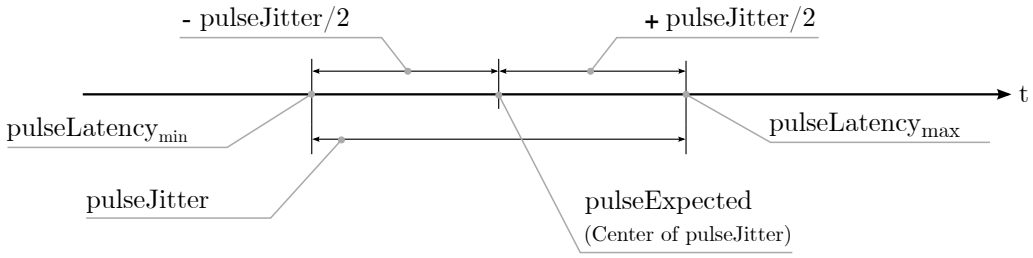

Figure 5.4: Distance from expected pulse location to jitter boundaries.

clock estimates must have a value that clearly contains clock drift uncertainties in order to use them for clock synchronization. Thus, remote clock estimates can be used for synchronization purposes if its values are larger than $+pulseJitter/2$ or smaller than $-pulseJitter/2$. The range of allowed remote clock estimates changes if *pulseExpected* is located at $pulseLatency_{max}$. In this case, remote clock estimates smaller than $-pulseJitter$, and every positive value would be a valid input for the clock synchronization. Clock drifts up to a total duration of *pulseJitter* can be masked if pulses are transmitted with its maximum jitter. This happens, for example, if a pulse is received at $pulseLatency_{max}$ and a clock drift of $-pulseJitter$ is present.

5.2.2 Concrete SpaceWire Utilization

Pulse distribution over serial communication links is provided built-in by SpaceWire broadcast codes [ESA19, p. 84]. This work utilizes Distributed Interrupts (DIRQs) which are a subset of broadcast codes. Figure 5.5 shows the basic connectivity responsible for transmitting and receiving DIRQs. Each SpaceWire interface has a dedicated port for transmitting user data and broadcast codes. User data are stored inside memories and processed in

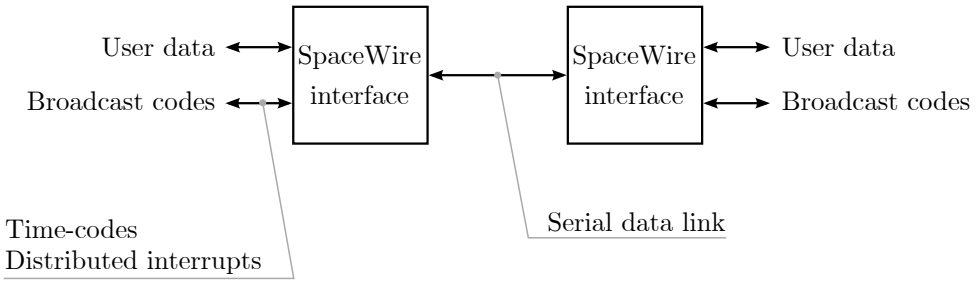

Figure 5.5: SpaceWire interface connection with typical user interface.

the order they are applied to the SpaceWire interface. DIRQs, instead, are processed immediately at the time they are assigned to the broadcast code port. Its transmission over the serial data link is started as soon as currently executed token[2] transfers are finished.

Routers, which are generally part of SpaceWire networks, relay incoming broadcast codes without arbitration to all other outputs. Thus, a constant latency is given for broadcast code forwarding inside routers. Existing uncertainties that affect DIRQ transmissions are outlined in Section 5.5 as well as strategies to reduce them.

5.3 System Architecture

A complete system capable of establishing and maintaining a global time is shown in Figure 5.6. It consists of multiple NCs connected by a SpaceWire network. All NCs are capable of synchronizing their local clocks and of performing a system start-up to establish an initial synchronized global time. The synchronized clocks are used to provide time-triggered data transfers between arbitrary connected hosts.

The implemented clock synchronization incorporates the pulse-based remote clock estimation which is introduced in Section 5.2. The network can be considered as isolated by NCs to prevent hosts from influencing the overall communication by illegal accesses. A basic overview of NCs is given in Fig-

[2]Data characters, FCTs, NULLs, Time-codes, Distributed interrupts.

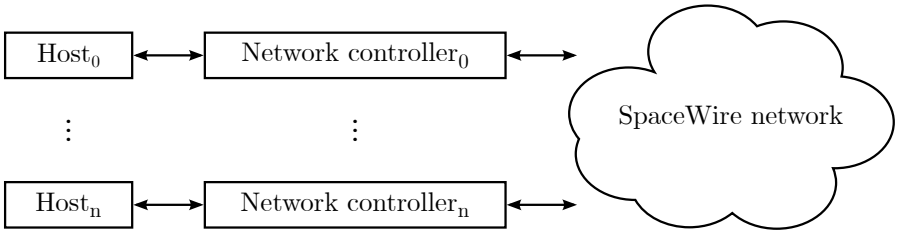

Figure 5.6: Network controller based data exchange by utilizing SpaceWire networks.

ure 5.7. They are connected to the network by the use of standard SpaceWire interfaces. Hosts can interact with NCs over the internal Advanced High-Performance Bus (AHB), which is implemented as an interconnect to improve performance. The start-up unit is active after each power-on, reset, or loss of clock synchronization. It performs the start-up routine to establish an initial global time in case no communication is recognized.

The integration unit collects time and pulse information[3] in case already synchronized NCs exchange data. The integration completes if a sufficient number of time and pulse information is received. The synchronized state is kept as long as the observer unit recognizes a minimum number of time and pulse information. Otherwise, another start-up attempt is initiated.

The schedule execution unit provides information to all other units about actual executed slots and cycles. Its progress is controlled by the macrotick generator unit, which provides ticks with a system-wide defined frequency. The macrotick length can be modified by the state and rate correction units, which are part of the clock synchronization.

Interactions between NCs and hosts are based on instructions stored inside host interface RAMs. The executed schedule is located inside network controller RAMs. The packet exchange RAM either contains host data required for transmission or provides memory space for received data. A detailed introduction of each RAM purpose and the way they are accessed is given in Section 5.3.1. The NC prototype is described in VHDL and implemented on a

[3]Received in time and pulse slots related to Section 5.2.1.

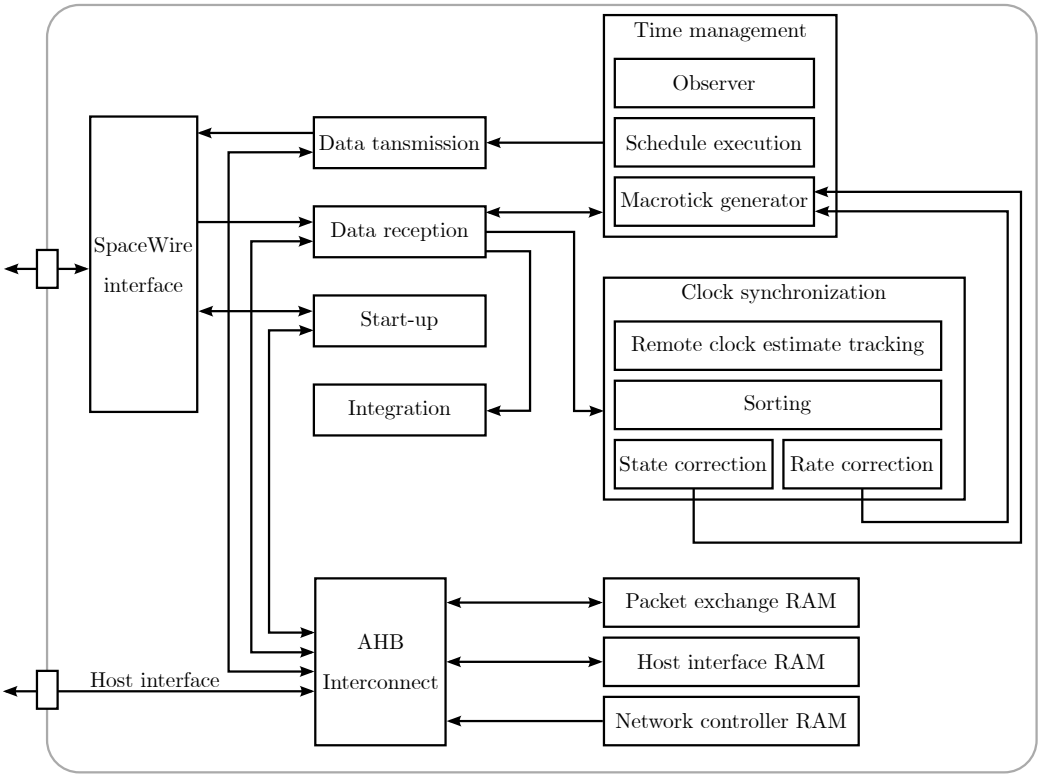

Figure 5.7: Network controller structure.

FPGA with each RAM realized in a separated block RAM. It is also possible to use a single external RAM accessed by a suitable memory controller.

5.3.1 Host to NC Interaction

Hosts need to communicate with NCs to access the network. However, host network access capabilities are restricted to ensure all data transfers are initiated according to the given schedule. Schedules generally define the point in time and the way data is routed through the network. They are fixed before system operation and located inside the network controller RAM. Hosts are allowed to define the payload data, which is transferred throughout slot executions. Additionally, memory space must be defined by hosts for received data. Payload data transmission and reception handling are controlled by the host interface RAM without possibilities to affect the schedule.

Figure 5.8 shows the detailed relation between the schedule, located inside network controller RAMs, and host interface RAMs. Each schedule slot relates to an entry inside both RAMs for transmissions and receptions separately. Each entry consists of three 32 bit words and can be read by hosts and NCs. Only the host interface RAM entries are allowed to be modified to enable handshaking and information exchanges between NCs and hosts. Slot sizes are configurable and defined by each network controller RAM entry (slet). A virtual link identifier (vlid) can be used to validate packet occurrences inside the network. However, vlid values are ignored in the actual prototype implementation because schedule policy checks are not performed inside routers.

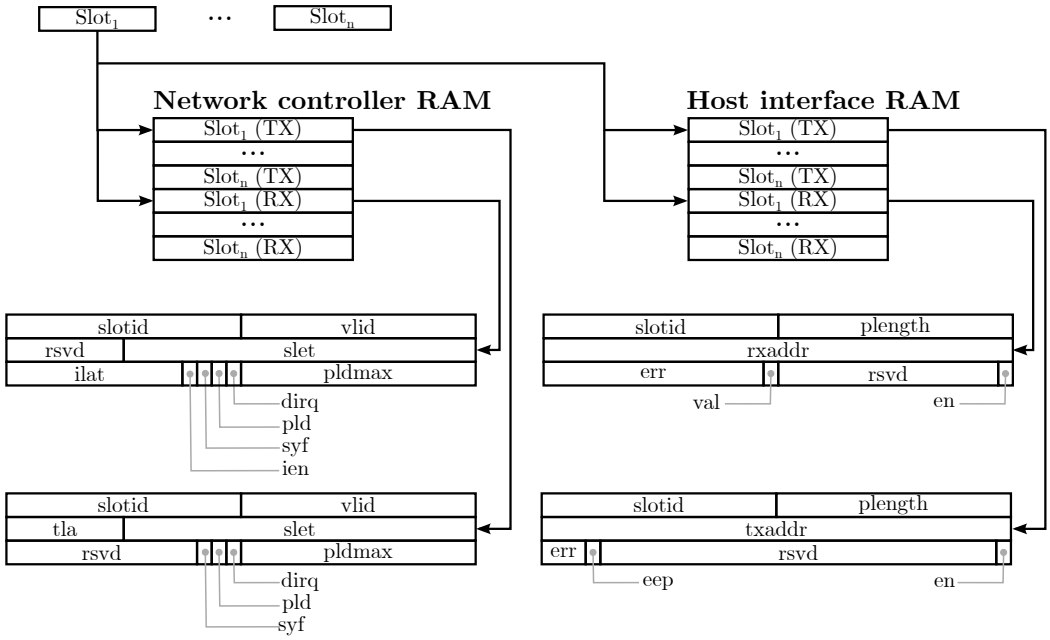

Figure 5.8: Relation between schedule, network controller RAM, and host interface RAM.

Transmission slots. These slots can lead to three different transmission types depending on the control flag values (dirq, pld, syf). Synchronization frames (syf) and DIRQs (dirq) are used to apply the clock synchronization

based on the pulse based-remote clock estimation which is introduced in Section 5.2[4]. Synchronization frames and DIRQs are created and transmitted by NCs without interaction with hosts and their related host interface RAM. Hosts provide information to NCs by modifying host interface RAM entries in case payload data (pld) can be transmitted for a given slot. The modified entry provides information about the payload length (plength), the location of payload data (txaddr), which is stored inside the packet exchange RAM, and how the packet shall be closed (eep). Additionally, entries are enabled (en) by hosts and disabled by NCs after they are processed. Packets that contain synchronization frames or payload data are routed through the network by the use of logical addresses (tla). The maximum size of payload lengths is defined by the schedule to ensure packets transfers are completed inside the related slot. Violations or problems that are observed throughout slot execution by NCs are reported to the host by modifying the error field (err) of related host interface RAM entries.

Reception slots. These slots provide information about the kind of data that is expected to be received, indicated by the control flags (dirq, pld, syf). The combination of received synchronization frames and related DIRQs is primarily used for synchronization purposes inside every NC. However, it is also used to mark each synchronized NC that provides synchronization frames and DIRQs. The interrupt enable flag (ien) is used to select specific or all NCs and its DIRQs as input for the remote clock estimates. Thus, it is possible to track every synchronized NC with the ability to utilize only a subset of synchronized NCs for clock synchronization purposes. Additionally, DIRQ latencies (ilat) are provided and used for uncertainty compensation during clock synchronization. Hosts provide memory addresses to NCs by modifying the related host interface RAM entry (rxaddr) in case payload data (pld) is expected. Each unexpected behavior during reception is stated by NCs inside the error field (err). The packet is marked as valid (val) in case receptions were successful. The actual received number of payload bytes is stored by the

[4] *time slots* transfer synchronization frames, *pulse slots* transfer DIRQs.

NC inside the related length field (plength). Each host interface RAM entry must be enabled and disabled, as already explained for transmission slots.

5.4 Core Functionalities

NCs provide three core functionalities to cover all relevant activities required to establish and maintain a global time. The core functionalities are represented as states in Figure 5.9. NCs either initiate a start-up or they try to

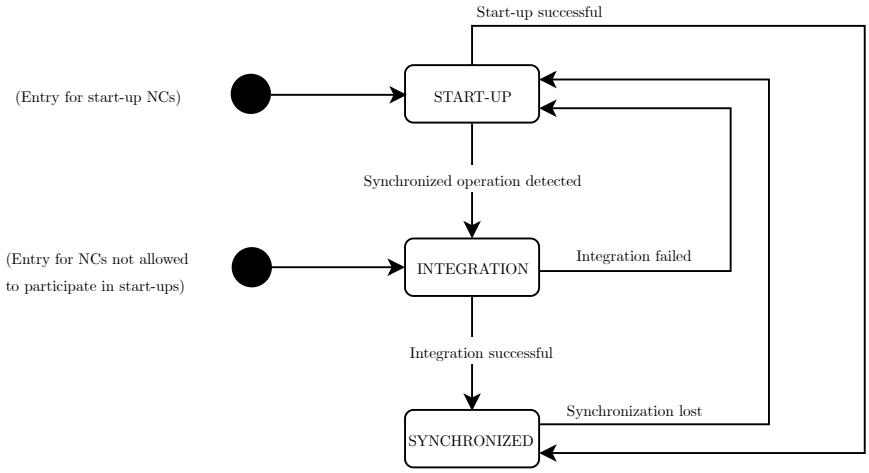

Figure 5.9: Network controller core functionalities.

integrate at the time they are powered on. However, it is the objective to enter the synchronized operation independent on the entry state in order to exchange data through the network. The following sections provide detailed information about each functionality.

5.4.1 Start-up

Start-ups are required to synchronize local clocks of NCs initially after the system encounters a power-on, reset, or loss of synchronization. The introduced system relies on a decentralized start-up process. Thus, a subset of start-up participating NCs is determined before system operation. This subset is allowed to communicate asynchronously to select a single NC. The

selection process is done by exchanging Start-Up Frame (SUF) commands that are confirmed by SUF acknowledges. The NC that collects the majority of acknowledges finalizes the start-up process by transmitting a DIRQ. The DIRQ reception starts the schedule execution in all NCs that provided acknowledges. All unsynchronized NCs need to integrate into a synchronized system from this point.

Figure 5.10 shows the detailed start-up process from the perspective of a single NC. Each start-up participating NC initially listen to the network

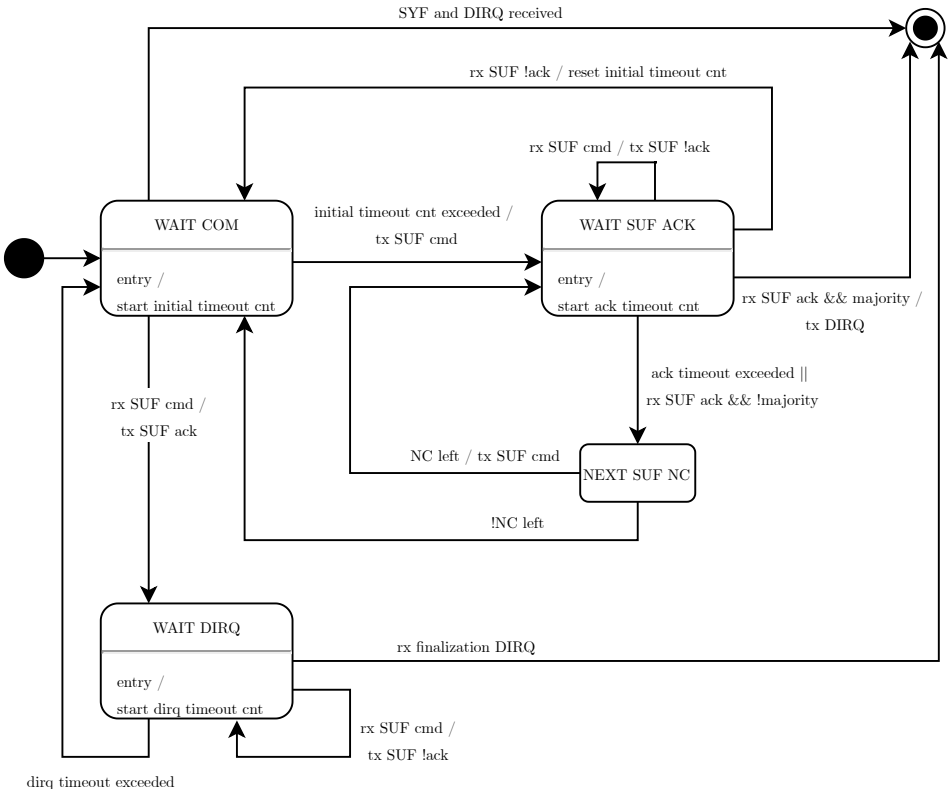

Figure 5.10: Network controller start-up sequence.

in order to detect existing communication. The following situations can be classified.

System already synchronized. A reception of Synchronization Frames (SYFs) and DIRQs indicates a synchronized operation based on an already

established global time. Thus, NCs need to integrate as explained in Section 5.4.2.

System start-up in progress. Receptions of SUF commands are acknowledged as long as the initial timeout is not exceeded. These commands indicate that other start-up participating NCs have already started to establish a global time. The start-up process must be finalized by a related DIRQ reception within a predefined interval. Otherwise, the start-up process restarts. NCs provide only a single acknowledge for the first SUF command they receive. All other received SUF commands are responded with not acknowledges while waiting for the finalization DIRQ.

System waits for start-up. Start-up participating NCs start transmitting SUF commands as soon as the initial timeout has exceeded and no communication was recognized. NCs always wait for a related acknowledge or timeout before starting the next SUF transmission. This procedure terminates successfully if the majority of acknowledges has been collected. However, the sequence fails if SUFs were transmitted to all allowed NCs without having the majority of acknowledges collected. The sequence can also fail if not acknowledges are received as a response on SUF commands, which is also called *logical collision*. The whole start-up sequence restarts in both fail cases. The start-up sequence is finished by transmitting a DIRQ as soon as the majority of acknowledges are collected. However, the start of schedule execution is delayed by the DIRQ latency in order to enhance the synchronicity with all involved NCs.

5.4.2 Integration

NCs are integrated into synchronous system operation if a global time is already established. The whole process is given by Figure 5.11. Already synchronized NCs transmit time (SYF) and pulse (DIRQ) information throughout schedule execution periodically. Integrating NCs, which operate asynchronous related to the schedule, take the first received SYF/DIRQ pair as the

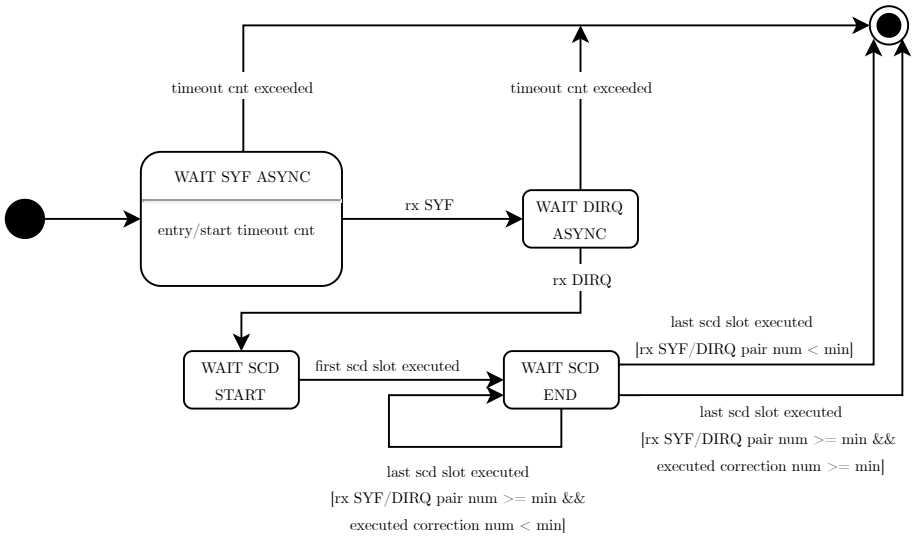

Figure 5.11: Network controller integration sequence.

entry point into the schedule execution. The integration process is stopped if the initial SYF/DIRQ pair is not received within a predefined time.

NCs collect SYF/DIRQ pairs based on schedule entries once the schedule entry point is defined. It is required to collect at least a minimum number of SYF/DIRQ pairs per schedule cycle to ensure enough NCs are synchronized. Otherwise, the integration process is stopped and restarted later on. The integration only succeeds if the minimum number of SYF/DIRQ pairs per schedule cycle are present and a minimum number of clock corrections are applied during the clock synchronization process.

5.4.3 Clock Synchronization

The applied clock synchronization combines state and rate corrections to achieve a better alignment of all local clocks. The effect of each correction type and its combination is shown in Figure 5.12. Each graph provides clock deviations that accumulate over time. Rate corrections lower the gradient of clock deviations by reducing or increasing the progression of clocks. This results in clocks that finally operate at the same rate but with different values. State corrections remove the accumulated deviation without affecting

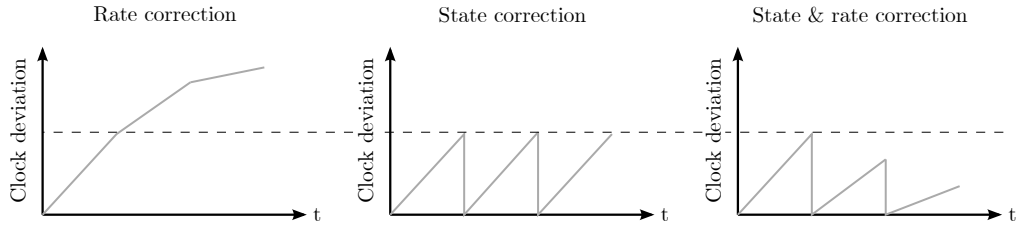

Figure 5.12: Effect of different applied correction methods.

the clock rate. Thus, clocks deviate again after state corrections. The combination of state and rate correction provides lower clock deviations once the rate has been adjusted.

The introduced system and its NCs use a similar cycle structure as FlexRay [Fle10, p. 198]. Two consecutive schedule cycles are organized as double cycles as shown in Figure 5.13. A single cycle consists of slots that are used to

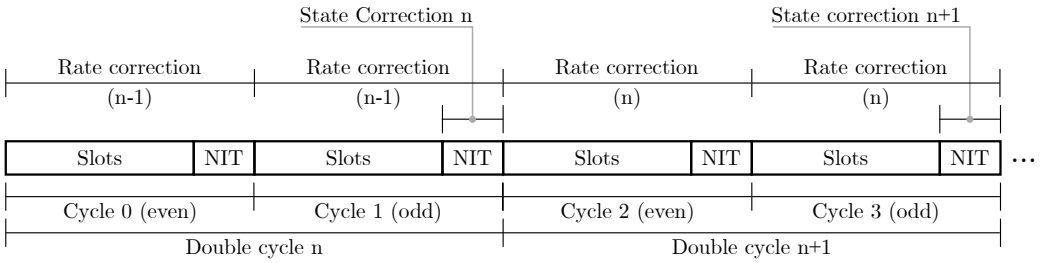

Figure 5.13: Network controller schedule cycle structure.

transfer payload data and SYF/DIRQ pairs in order to allow remote clock estimation. Each cycle ends with a Network Idle Time (NIT), which can be used to apply state corrections. State corrections are applied within the double cycle they are calculated. However, state correction application is only applied at the end of each odd cycle to prevent corrupting rate correction calculation, which is outlined in more detail inside this section. Rate correction values, calculated in double cycle n, are applied throughout double cycle $n+1$. Additionally, rate correction values are distributed equally throughout cycles instead of being applied instantly at specific points in time.

Remote clock estimates that are collected throughout cycle execution are processed by the convergence function introduced in [LL88]. This function is used to calculate state and rate correction values for each double cycle as long as a sufficient number of remote clock estimates is collected. The algorithm was selected because of its ability to tolerate up to two faulty remote clock estimates. However, the number of discarded values depends on the number of collected remote clock estimates, as shown in Table 5.1.

Table 5.1: Number of discarded values depending on valid remote clock estimates.

Remote clock estimates	Discarded number (k)
1 - 2	0
3 - 7	1
> 7	2

Correction value calculations initially start with an ordering of available remote clock estimates. The k highest and lowest ordered values are discarded according to the rules introduced by Table 5.1. Finally, the new high and low values are added and divided by two to receive the correction value used to synchronize the local clock. The simple arithmetic required for correction value calculation is another selection argument for the algorithm. Addition is a basic functionality supported by all relevant FPGAs. Division generally utilizes dedicated hardware blocks. However, dividing values by two can be implemented by simple shift operations, which removes the need for dedicated hardware.

State and rate correction values are used to increase or decrease macrotick lengths in order to influence the clock progression. However, the way remote clock estimates are handled, before they are applied to the convergence function, is different for both correction types and explained in the following.

State correction flow. Remote clock estimates for state correction value calculations are only used within odd cycles as shown in Figure 5.14. It is sufficient to collect a single remote clock estimate within odd cycles to

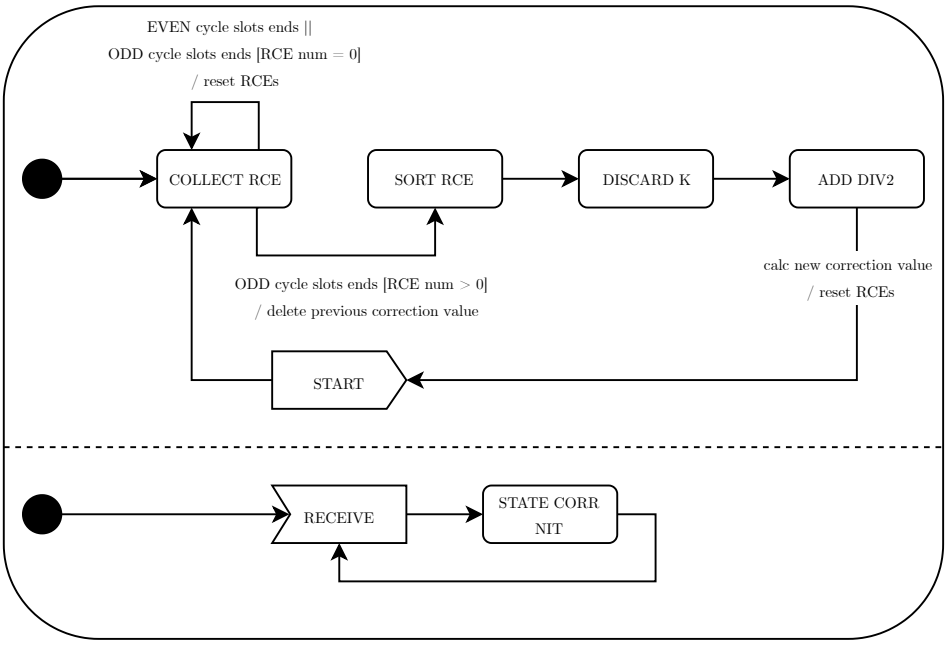

Figure 5.14: Network controller state correction sequence.

start the convergence function. Clock deviations determined inside odd cycles represent the accumulated deviation over the whole double cycle. Thus, it is not required to handle clock deviations inside even cycles separately. The calculated state correction value is deleted once it has modified the clock. As a consequence, actual correction values don't affect future correction values.

Rate correction flow. Remote clock estimates are used indirectly for rate correction value calculations as shown in Figure 5.14. Remote clock estimates are collected for even and odd cycles separately. Differences of related remote clock estimates between two consecutive cycles are calculated after each double cycle execution (Equation 5.5)

$$rce_{diff}(index, cycle) = rce(index)_{cycle+1} - rce(index)_{cycle} \tag{5.5}$$

The rce_{diff} values describe drift rates between a local clock that receives remote clock estimates and the source of each remote clock estimate. This double cycle relation is the reason why state corrections are not allowed to

be applied in even cycles. Otherwise, it would corrupt the determination of drift rates that rely on calculating differences. The convergence function

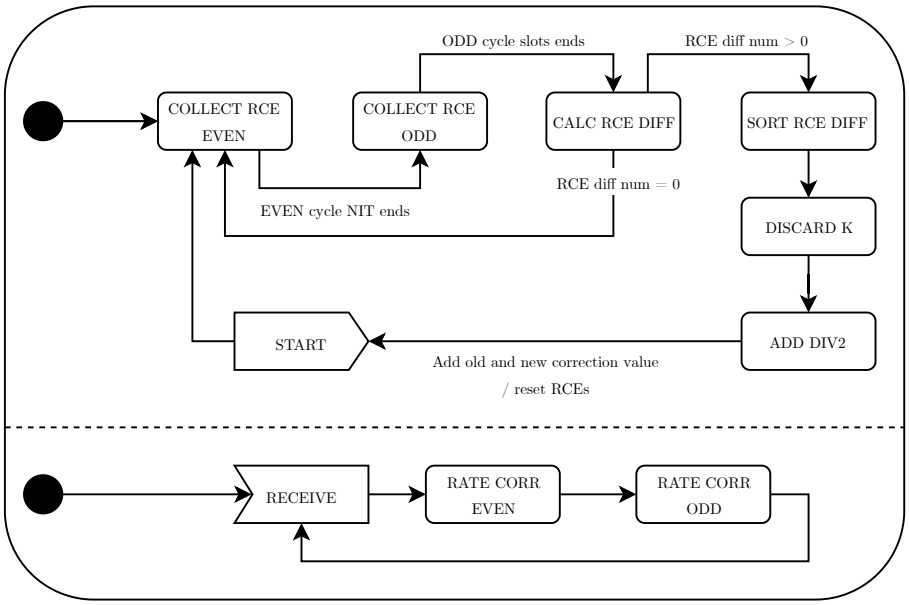

Figure 5.15: Network controller rate correction sequence.

takes all available remote clock estimate differences as inputs in order to calculate the new rate correction value. New rate correction values incorporate the old rate correction value to keep compensated drift rates stable. Without incorporating past values, the following scenario would happen. Clock rate differences are determined by a rate correction value unequal to zero for double cycle n. Double cycle $n + 1$ applies the rate correction value of double cycle n, which leads to a new rate correction value of zero. Double cycle $n+2$ applies rate correction value zero, determined in double cycle $n + 1$. As a consequence, double cycle $n + 2$ reintegrates the initially compensated drift rate of double cycle n in case past rate correction values are not considered. Each updated rate correction value is distributed over the following even and odd cycle individually, which results in a stretched or compressed schedule execution time.

5.5 Jitter Reduction

The clock synchronization quality directly depends on the characteristics of DIRQs. Nodes that receive pulses (DIRQs) have to determine the latency as precise as possible in order to calculate accurate remote clock estimates, which are the inputs for the clock synchronization algorithm. SpaceWire interface implementations are subject to different uncertainties that influence the predictability of broadcast code (DIRQs and time-codes) transfer latencies.

These uncertainties and strategies to remove them partially are explained in the following. The used SpaceWire interface has a structure, as shown in Figure 5.16. It consists of core logic, a receiver frontend, and transmitter

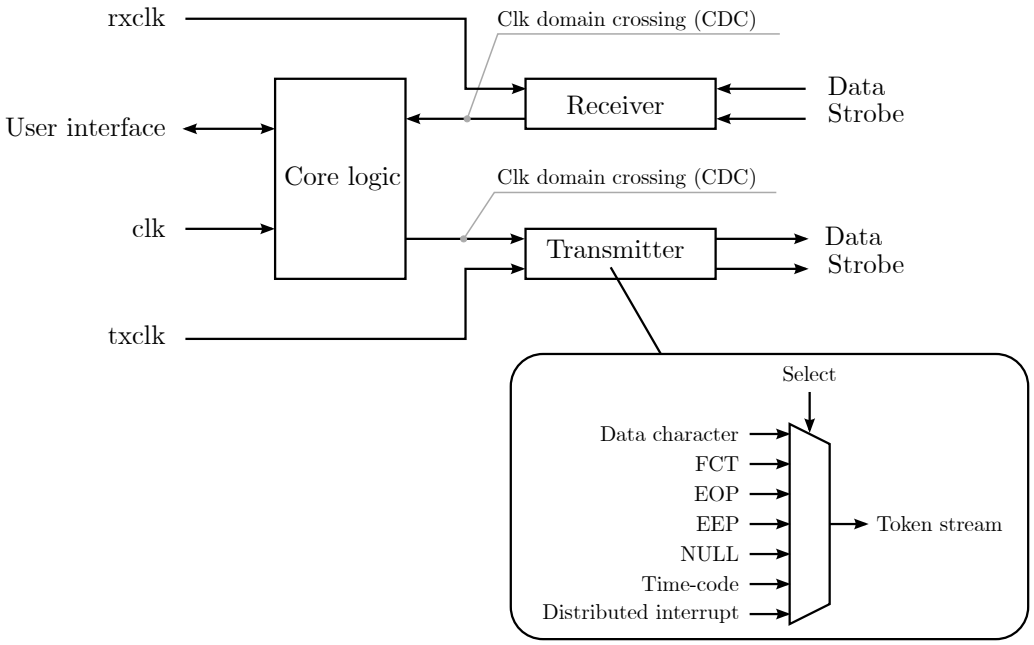

Figure 5.16: Character and code selection for the used SpaceWire interface.

backend. The core logic provides the user interface, which is used to transmit and receive payload data or broadcast codes. Data reception is performed by detecting signal transitions of data and strobe by use of oversampling. The transmission unit generates a token stream consisting of data characters,

control characters, or control codes. Dedicated clk domains are used for receptions (rxclk) and transmissions (txclk) in order to allow much higher data rates. However, the core logic interacts with receiver and transmitter units, which implies CDC issues. Although CDC is covered by proper utilization of synchronization stages, it still introduces uncertainties depending on the phase relation between the relevant clks and its frequency differences.

The problem is illustrated by Figure 5.17. $BcEn_{clk}$ is part of the user interface inside the core logic unit used to enable a broadcast code transfer. The signal is passed into the transmitter unit clk domain indicated by $BcEn_{txclk}$. Value $Uncertainty_{cdc}$ depends on the clk difference between *clk* and *txclk* and its phase relation. Generally, $Uncertainty_{cdc}$ decreases in case frequency of *txclk* increases. A constant preparation time $Latency_{prep}$ is needed to process

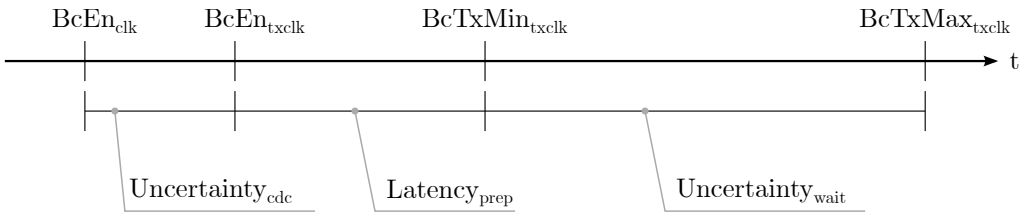

Figure 5.17: Existing uncertainties and latencies for DIRQ transmissions.

the broadcast code transmission request once $BcEn_{txclk}$ is valid. Finally, the required broadcast code token (either DIRQ or time-code) is transmitted as soon as possible.

However, the broadcast code transmission is stalled at minimum until the actual token transfer has finished. In best cases, no token is transferred, which allows an immediate transmission of the broadcast code at $BcTxMin_{txclk}$. In worst cases, another broadcast code transmission of the same type has started, which causes a delay of 14 bits[5] indicated by $BcTxMax_{txclk}$. A corner case is a consecutive time code transmission which has precedence over DIRQ transfers [ESA19, p. 85]. This scenario would prevent DIRQ transfers completely from a technical point of view. However, time-codes are transferred

[5]Broadcast code = ESC (4 Bit) + Data character (10 Bit).

generally with sufficient distances between each other, which eliminates the problem for real applications. Additionally, every broadcast code delay between 0 - 14 Bits is possible depending on the occurrence of $BcEn_{txclk}$ and the actual transmission state. Delay $Uncertainty_{wait}$ influences the latency determination used for remote clock estimate calculations and directly affects the synchronization quality. Time consumed by $Uncertainty_{wait}$ is also depending on the link transmission rate.

In total, two uncertainties are present during broadcast code transmission activities. $Uncertainty_{cdc}$ can be influenced by selecting txclk as fast as possible in order to keep the CDC of BcEn short. Phase relations between txclk and clk could also be controlled to enhance the CDC process. However, $Uncertainty_{cdc}$ represents the much smaller value compared to $Uncertainty_{wait}$. Its full removal is illustrated in the following, according to Figure 5.18. The removal is based on a transfer of uncertainty into latency.

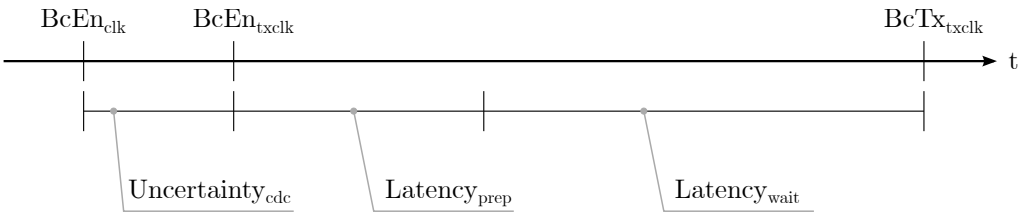

Figure 5.18: Transfer of uncertainty into latency for low jitter DIRQ transmissions.

This is achieved by shifting $BcTxMin_{txclk}$ to $BcTxMax_{txclk}$ for all transfer conditions. An artificial delay is applied to each broadcast code transmission in order to determine the fixed starting point $BcTx_{txclk}$. This introduces idle times and reduces effectively the overall throughput, which results additionally in higher broadcast code latencies. However, the broadcast code latency range and its jitter are significantly reduced, which in turn provides more precise remote clock estimations.

Chapter 6

Broadcast Code Evaluation

6.1 Introduction

In this chapter, the end-to-end transmission characteristics of DIRQs for the used SpaceWire interfaces are analyzed in different ways. This is done initially by the use of HVLs to take advantage of capabilities like object-oriented programming. This kind of functional simulation is a suitable option because of the small design size under evaluation. The correct jitter determination for DIRQs is fundamental in order to achieve proper clock synchronization results, as introduced in Section 5.5. Thus, FPV is applied as an additional verification method to confirm the results provided by functional simulation. The remainder of this chapter is organized as follows. Section 6.2 introduces structure and main components of the verification environment used for functional simulation. The applied FPV and its relevant aspects are explained in Section 6.3 followed by the analysis results given by Section 6.4. Finally, Section 6.5 provides some conclusions.

6.2 Simulation Environment

A separated UVM environment is used to characterize DIRQ end-to-end transmissions for the given SpaceWire Interfaces with an architecture shown in Figure 6.1. The DUT contains two SpaceWire interfaces connected to each

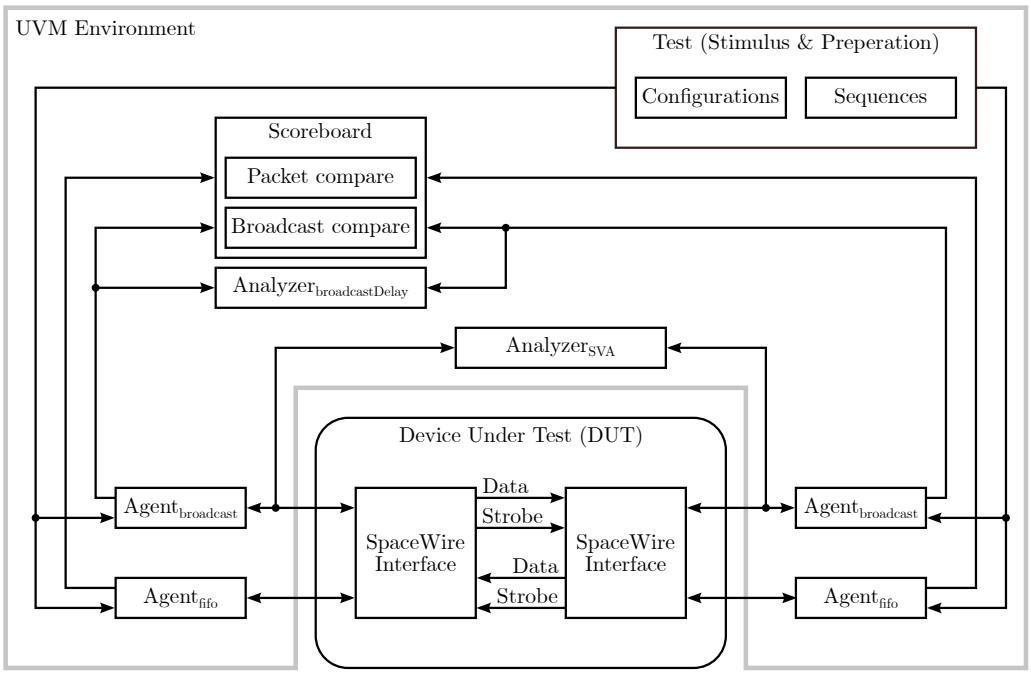

Figure 6.1: UVM simulation environment used for DIRQ evaluation.

other by data and strobe connections. DIRQ forwarding inside routers is constant. Hence, it is sufficient to investigate DIRQ characteristics for a single SpaceWire link. Two UVM agents are connected to the user interface of each SpaceWire unit. $Agent_{fifo}$ transmits and receives user payload data as well as EEP and EOP in order to create SpaceWire packets. Transmitted packets are passed to the *Scoreboard*, where they are used as expected values. Received packets are provided to the scoreboard upon completion, which allows an automated comparison between actual and expected packets. The same actual to expected verification is used for DIRQs. Its latencies are tracked by timestamping each transmission and reception. Thus, a set of latencies is collected, which allows a determination of jitter and the distribution of each latency value. Additionally, latencies are tracked by the use of assertions based on passed clk cycles. This is used to make analysis comparable to FPV because these techniques are generally cycle-based. Thus, a notion of time doesn't exist as it is the case for event-driven technologies like RTL simulators.

The ability to exchange DIRQs, time-codes, and data characters was already verified for the standard SpaceWire interface. However, because of the implemented jitter reduction, it is necessary to verify the modified implementation again.

Simulation runs are handled by a regression tool. A randomized oscillator phase relation is applied for each run before both SpaceWire interfaces exchange data. These phase variations are required because they can affect the latencies of broadcast code transmissions. A different way of oscillator control is applied for system tests introduced in Section 7.2.

6.3 Formal Property Verification

FPV provides a way to prove that system properties for RTL designs hold under every possible input stimulus. For the given case, FPV was selected to confirm simulation-based results, which targets the exact DIRQ latency determination.

Figure 6.2 illustrates the differences between FPV and functional simulation regarding state-space evaluation. FPV checks complete state-space areas in parallel instead of evaluating single traces. Multiple simulation traces are created in order to cover the most relevant properties throughout several simulation runs. However, all traces generated during simulation don't cover the whole state-space, which leaves room for potential bugs in the uncovered areas. This problem of incompleteness is only caused by the huge state-spaces and should not be treated as a fault of simulation techniques in general.

The entire state-space of RTL designs consist of all state elements (like FFs and RAMs) and its inputs with the overall number of states defined by $states = 2^{inputs+stateElements}$. Thus, even minimal designs, like a comparator with two 32 bit input vectors, lead to an overall state-space of 2^{64} elements and a duration of more than 1000 years[1] before exhaustive simulation-based verification finishes.

RTL designs are typically brought into a reset state before property checks

[1] With a simulator capable of checking a vector every 2 ns.

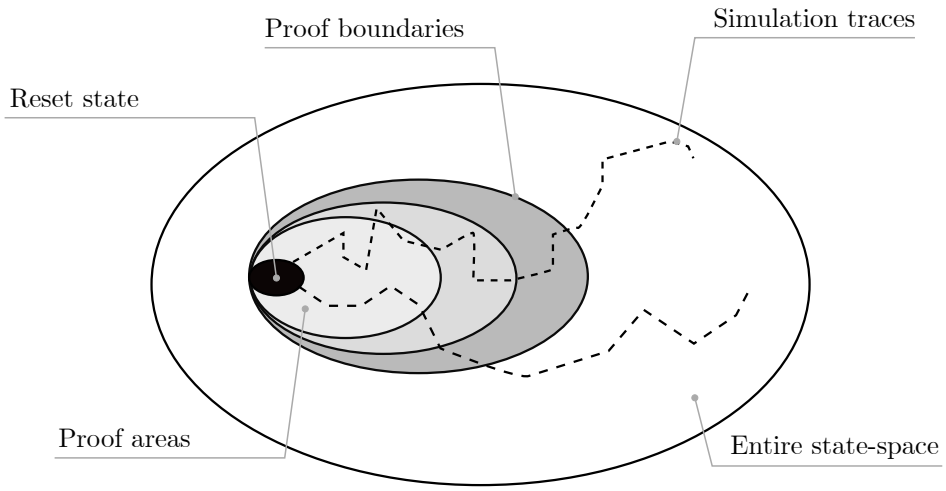

Figure 6.2: Specific simulation traces versus area investigation by formal property verification.

are applied. These reset states are often created throughout dedicated initialization phases. However, they may consist of multiple states in case particular state elements don't receive its default value by reset signals[2]. Alternatively, a predefined reset state can be loaded, which is useful to handle complexity issues.

6.3.1 Property Checking

System properties for this work are written in SVA and stored inside SystemVerilog modules. However, the usage of modules for VHDL designs is driven by a constraint of the selected FPV tool.

As already introduced, the FPV activities within this work focus on determining the minimum and maximum DIRQ latencies between two SpaceWire interfaces. This information can be used to derive the jitter for the entire system safely. A DIRQ transmissions can be initiated by use of *tick_in_spw0* as shown in Figure 6.3. The reception of DIRQs is indicated by *tick_out_spw1*. The main clk domains of both SpaceWire interfaces (*spw0*, *spw1*) are driven by its own asynchronous clks (*clk_spw0*, *clk_spw1*).

[2]E.g. RAMs often don't have a reset signal.

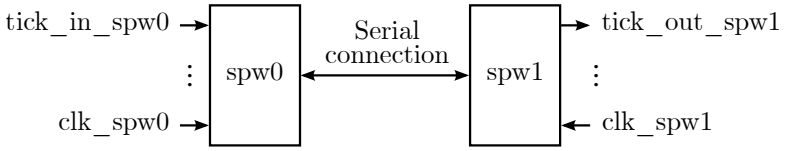

Figure 6.3: DIRQ transmission with respect to different clk domains.

The property that describes the tick_in to tick_out relation for the standard jitter implementation is given in Figure 6.4 and directly applied to an *assert* statement. A valid time window from *tick_in_spw0* high until *tick_out_spw1* high is initially defined by simulation which ranges from 16 - 29 clk cycles. Invalid ranges are defined for areas below and above the valid range, represented by the generated assertions *a_illegalLowArea* and *a_illegalHighArea*. Each property basically checks that *tick_out_spw1* doesn't occur i clk cycles after a rising edge of *tick_in_spw0*.

FPV tools try to violate properties that are placed inside *assert* statements as soon as the formal initialization phase is finished. Evaluated properties can be considered as a *full proof* if no counterexample is found. However, this requires no counterexample is found within the entire state-space, whereas these complete evaluations are not always possible. DUT complexity or insufficient memory resources may prevent full proofs. A complexity issue is present for the asserted properties, which prevents the FPV tool to provide a full proof within 60 hours of execution[3].

This problem is solved by *bounded proofs*. They are used to decrease the search depth in order to limit the explorable state-space. FPV tools automatically increase the search depth incrementally throughout evaluation until a counterexample for a given property is found, or a user-defined proof boundary is reached. An example of these proof boundaries and its related areas is shown in Figure 6.2. However, bugs may not be revealed if proof boundaries are selected to small. A proper selection of these boundaries generally requires good insight knowledge of the DUT.

[3]It might be possible to find a full proof for longer execution times or selecting more solving engines which require more host memory.

For the given work, a maximum search depth of 60 is defined, which allows bounded proofs for the asserted properties given in Figure 6.4. The depth[4] of

```
1  // Illegal low range of broadcast code receptions
2  generate
3    for (genvar i = 5; i <= 15; i++) begin
4      a_illegalLowArea: assert property
5      (@(posedge clk_spw0) $rose(tick_in_spw0) |=>
6         @(posedge clk_spw1) ##i !$rose(tick_out_spw1));
7    end
8  endgenerate
9
10 // Illegal high range of broadcast code receptions
11 generate
12   for (genvar i = 30; i <= 40; i++) begin
13     a_illegalHighArea: assert property
14     (@(posedge clk_spw0) $rose(tick_in_spw0) |=>
15        @(posedge clk_spw1) ##i !$rose(tick_out_spw1));
16   end
17 endgenerate
```

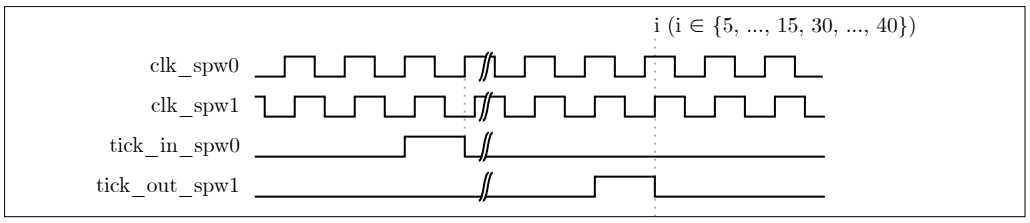

Figure 6.4: Concrete properties used to check distributed interrupt transmission durations for standard SpaceWire interface implementations.

60 allows the tool to find counterexamples within 60 clk cycles with a starting point directly behind the initialization phase. The depth is considered to be reasonable because the underlying DIRQ transmission itself is treated as verified. Additionally, a concrete expectation about DIRQ delays exists because of the prior executed simulation. Thus, the FPV objective is to find small variances around the simulation results rather than finding deep bugs.

[4]The depth is also known as *proof radius*.

However, the selected depth is only suitable because an artificial DUT reset state was applied, which already provides an initialized SpaceWire link as described in Section 6.3.5.

6.3.2 Constraints

Every input of the DUT is treated as a *formal control point* which is driven typically by the FPV tool. Clks are the only exception because they are generally defined statically with its required characteristics. However, clks can also be influenced or even modeled for special cases, as explained in Section 6.3.4.

Constraints to formal control points are required to create valid input stimulus [Fos+07]. These constraints are defined by formal assumptions with the ability to apply properties. DUTs that violate specified behavior on their inputs are called *under constrained*. For the given work, a delay between two consecutive DIRQ transmissions must be defined as shown by Figure 6.5. It causes signal *tick_in_spw0* to be high for a single clk cycle followed by a fixed low period of 40 clk cycles.

```
1 m_tickInDelays: assume property
2 (@(posedge clk_spw0) $rose(tick_in_spw0) |->
3  ##1 !tick_in_spw0 [*40]);
```

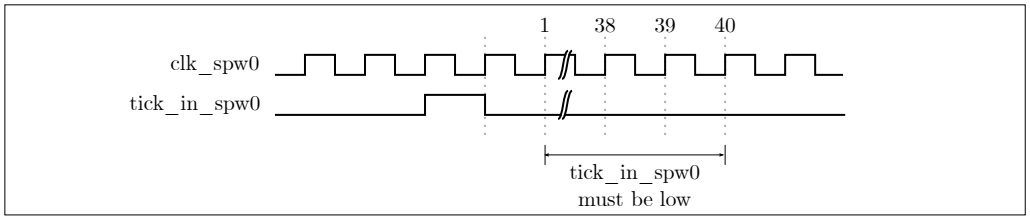

Figure 6.5: Assumption used to ensure a minimum temporal distance between two consecutive DIRQs.

A risk of *over constraining* exists if to many assumptions are applied which can lead to *false positive* results. Required DUT behavior is never triggered by its inputs in these cases. As a consequence, property checks may not be able

to provide any counterexample, which leads to the incorrect conclusion that DUTs fulfill their required behavior. Hence, all the required behavior should be tracked by cover statements, as explained in the following Section 6.3.4.

6.3.3 Coverage

Coverage tracking should be done in order to check that all required functionality of the DUT was tested [SSK15, p. 52]. DIRQ transmission and reception is the key functionality required to be observed as shown in Figure 6.6.

```
1 c_tickInToTickOut: cover property
2 (@(posedge clk_spw0) $rose(tick_in_spw0) |=>
3 s @(posedge clk_spw1) $rose(tick_out_spw1) [->1]);
```

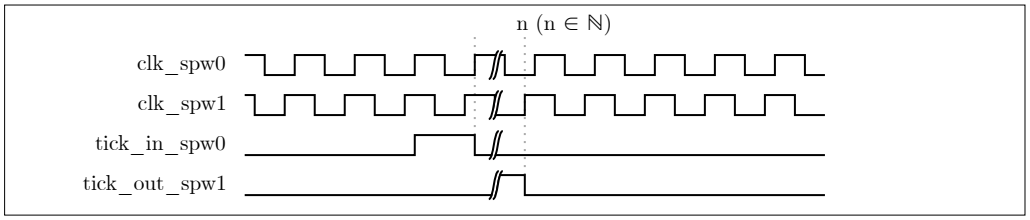

Figure 6.6: Cover statement to track arbitrary DIRQ latencies.

The property to be covered is written in the same way as it is done for property checks and assumptions but passed to a *cover* statement instead. The given cover statement is fulfilled as soon as a rising edge of *tick_in_spw0* is followed by a rising edge of *tick_out_spw1* arbitrary *clk_spw1* cycles later. Possible over constraints that lead to permanent high or low values of signal *tick_in_spw0* would be revealed in this way.

6.3.4 Clk Modeling

Clks used for DUTs are typically defined and provided to FPV tools as part of the overall configuration before verification runs. These clks are created throughout verification runs by FPV tools with user-defined options like phase shifts or unusual duty cycles. Clks that are defined in that way are not

controlled by assumptions or modified during verification. However, there are situations where clks need to be modeled. This may happen for multi-clk designs where phase relations between specific clks are of interest.

A more detailed view of the used SpaceWire interface structure is given in Figure 6.7. The design consists of three clk domains that require CDC

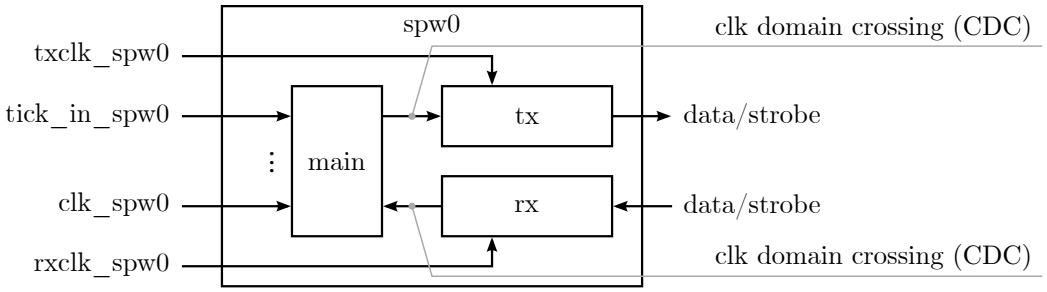

Figure 6.7: Clk domains that are involved for all types of transmissions and receptions.

between different sub-units. In particular from *main* to *tx* and back from *rx* to *main*. In total, six asynchronous clk domains can be used for two connected SpaceWire interfaces. The impact of phase relations regarding transmission timings must be considered as reasonable because data and DIRQs need to pass all clk domains. Thus, a static clk definition is unsuitable for the required property checks. Instead, a set of temporary clks with different phase relations is defined statically but assigned to the real DUT clk throughout verification.

Figure 6.8 shows the relevant part of the code used to model the main clk of SpaceWire interface instance *spw0*. All other clks can be modeled accordingly. Wires *clk0_50mhz* and *clk1_50mhz* are statically defined 50 MHz clks with different phase relations to each other and applied to *clk_spw0* depending on variable *sel_clk_spw0*. These clks can be used to model an arbitrary number of 50 MHz DUT clks by merely adding additional case statements. However, this clk granularity only provides four different clk phases to keep complexity manageable for the FPV tool. It is important to keep *sel_clk_spw0* stable by an assumption because phase relations are considered to be constant once the

```
1  // Static defined clks
2  wire        clk0_50mhz;
3  wire        clk1_50mhz;
4
5  // Clk selection signal controlled by formal tool
6  wire [1:0] sel_clk_spw0;
7
8  // Static clks assigned to DUT clk
9  always @* begin
10   case (sel_clk_spw0)
11     0:    clk_spw0 <= clk0_50mhz;
12     1:    clk_spw0 <= clk1_50mhz;
13     2:    clk_spw0 <= ~clk0_50mhz;
14     3:    clk_spw0 <= ~clk1_50mhz;
15     default: clk_spw0 <= ~clk0_50mhz;
16   endcase
17 end
```

Figure 6.8: Applied clk modeling for formal property verification.

system operates[5]. This allows the FPV tool to select one value (0 to 3) for the beginning of each verification run but prevents a change in between.

It is also possible to model more complex clk scenarios without the need for defining multiple static clks. Instead, a single static clks is used to derive all required DUT clks, as shown in Figure 6.9. The example provides different frequencies to DUT *clk_spw0* depending on *sel_clk_spw0*. Frequencies are allowed to change in an arbitrary way throughout verification runs if *sel_clk_spw0* is not further constrained.

All these clk modeling approaches provide limitations regarding verification run time and accuracy. The more clks are involved, the more time it takes until properties are checked by full or bounded proofs. It may also happen that proofs are not possible any longer due to the increased state-space. On the other hand, multiple clks are required to increase the granularity of possible phase relations applied to the DUT.

[5]Except phase relation changes caused by oscillator uncertainties.

```
1  // Static defined clk
2  wire clk;
3
4  // Clk selection signal controlled by formal tool
5  wire sel_clk_spw0;
6
7  // Clk modeling
8  reg [3:0] cnt;
9
10 always @(posedge clk or posedge rst) begin
11   if (rst)
12     cnt <= 4'h0;
13   else
14     cnt <= cnt + 1;
15 end
16
17 // Assignment to DUT clk
18 assign clk_spw0 = sel_clk_spw0 ? cnt[1] : cnt[3];
```

Figure 6.9: Alternative clk modeling in order to adjust frequencies rather than phase relations.

6.3.5 Complexity Handling

Property checks, which incorporate large state-spaces, often have problems to find full or even bounded proofs. Additionally, FPV tools can run out of memory for complex evaluations. These complexity issues were encountered for the given SpaceWire DUT and addressed in the following ways.

Constraining. Assumptions are used to create valid stimulus on DUT inputs. However, they can also reduce the evaluation space in case they prevent specific DUT behavior to occur. One example is the dynamic transmission rate adjustment, as shown in Figure 6.10.

Each SpaceWire interface can change its transmission rate depending on its related *txdivcnt* signal. However, the use case of the SpaceWire links don't allow these rate changes. Thus, *txdivcnt_spw0* should be kept constant to reduce the evaluation space for the given properties.

```
1 m_spw0TxdivcntZero: assume property
2 (@(posedge clk_spw0) txdivcnt_spw0 == 8'h00);
```

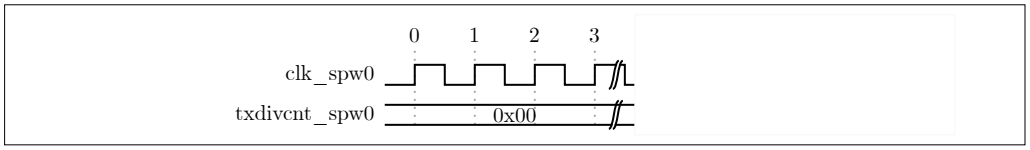

Figure 6.10: Assumption used to reduce state-space.

Blackboxing and Cut points. DUT internal logic can be removed if it represents a complexity problem. This is often done for logic that is not appropriate for formal evaluation in general. BRAMs are an example of this kind of logic and also used inside the SpaceWire interfaces. The majority of them are blackboxed with the effect of transforming BRAM outputs into formal control points, as shown by Table 6.1.

Additional 18 formal control point bits are available after BRAM removal. The significant impact of BRAM resources is shown by the overall design gate number, which is reduced from 10126 down to 6849.

Cut-points represent a more precise way to exclude logic. They define specific signals to be cut away from the logic of interest instead of removing whole units [Agg+11]. However, the signal which is cut away is treated as a formal control point similar to blackbox outputs.

Table 6.1: Blackbox effect on design size.

Element	Unmodified	Blackboxed
Control Point Bits	68	77
DUT Input Bits	62	53
Cut Point Bits	0	0
Black Box Output Bits	0	18
Undriven Wire Bits	6	6
Modeling Bits	0	0
State Bits	2878	1704
Counter State Bits	614	614
RAM State Bits	1216	64
Register State Bits	102	80
Property State Bits	946	946
Logic Gates	11380	8103
Design Gates	10126	6849
Property Gates	1254	1254

User-defined reset state. The applied property checks that ensure DIRQ transmissions within specified boundaries require two SpaceWire interfaces with an established link. However, links are established by executing an initialization sequence that takes at least 19.2 μs. From the FPV point of view, all states that are involved in initializing a SpaceWire link are included throughout property checks. This leads to a large proof area, as illustrated by the dashed oval of the left-hand side in Figure 6.11. In fact, the present proof area is even too large to fulfill cover statements that are used to check for end-to-end DIRQ transmissions, as introduced in Figure 6.6.

This problem is solved by defining a new reset point where the FPV tool starts its evaluation, as shown on the right-hand side of Figure 6.11. The approach is also known as *semi-formal* [MBS18; EY19] or *hybrid verification* [Cer+10, p. 240]. SpaceWire initialization phases can be considered as a pre-condition to any kind of transfers and don't interact with DIRQ transmissions during fault-free operation. Hence, the starting state is shifted to the point

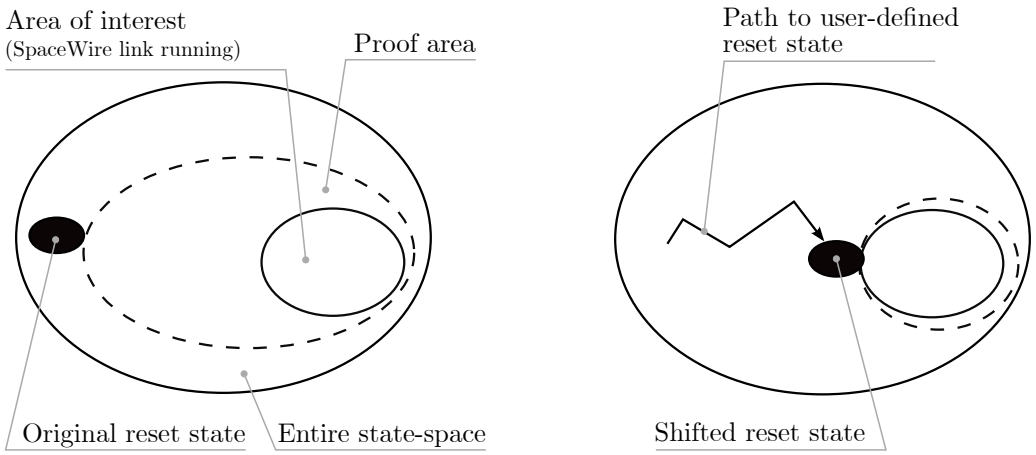

Figure 6.11: Drastically reduced proof area due to user-defined reset state.

where both SpaceWire interfaces have established a link, which drastically reduces the state-space to be evaluated.

These user-defined reset states can be reached by writing cover statements. Fulfilled cover statements typically provide an option to extract the related system state, which is finally applied throughout formal initialization phases. However, it might be impossible to reach the required starting state directly because of complexity issues. In these cases, additional cover points are required to define intermediate states used to create a path to the final state. It is not possible to reach the *running* state, which indicates an established SpaceWire link, for the given design directly. Hence, intermediate states for *starting* and *connecting* are selected to finally reach the *running* state. Functional simulation can also be used to extract a required starting state. This is helpful for complex designs since functional simulation can be guided deeply into large state-spaces.

6.4 Jitter Analysis

Clk frequencies are static for the given tests according to the nominal frequencies used throughout system evaluation in Chapter 7. In particular, 50 MHz for the main clk domains (clk_spw) and 120 MHz for the remaining clk domains (rxclk_spw, rxclk_spw) are used.

The expected latency range and its resulting jitter were initially determined by a set of cover statements during functional simulation, as shown by Figure 6.12. The given code results in 50 individual cover statements to track the occurred latencies of DIRQ transmissions between two SpaceWire interfaces, started by *tick_in_spw0* and completed by reception of *tick_out_spw1*. All covered latencies represent an expected area which is used to define

```
1 generate
2   for (genvar i = 1; i < 50; i++) begin
3     c_determineLatency: cover property
4     (@(posedge clk_spw0) $rose(tick_in_spw0) |=>
5      @(posedge clk_spw1) ##i $rose(tick_out_spw1));
6   end
7 endgenerate
```

Figure 6.12: Cover statement used to track initially all observed latencies.

the final jitter for a given SpaceWire interface implementation. The overall system accuracy depends on the accurate determination of this area. Thus, two illegal areas are centered around the expected area. These illegal areas represent safety properties that disallow the occurrence of latencies within these areas[6]. These properties are defined as shown by Figure 6.13. The given example provides the generation of property checks for one area. Another generate statement is required to cover the second one. The properties target each unhallowed latency separately, whereas the range of each area is defined by the loop variables *low* and *high*. A finite range of each illegal area is considered as reasonable because coverage results already determine a con-

[6]Latency occurrences are illegal in the system context, not by definition of the SpaceWire standard.

```
1  generate
2  for (genvar i = low; i < high; i++) begin
3  a_illegal: assert property
4  (@(posedge clk_spw0) $rose(tick_in_spw0) |=>
5  @(posedge clk_spw1) ##i !$rose(tick_out_spw1));
6  end
7  endgenerate
```

Figure 6.13: Properties used to specify invalid latency areas.

crete area. Additionally, the applied checks target for performance evaluation rather than bug hunting.

The same property checks are applied for functional simulation and FPV. Alternatively, functional simulation could be used to determine absolute latency values measured in ns. However, this absolute measurement is not possible for FPV because of its cycle-based functioning without a notion of time between two consecutive clk cycle events.

6.4.1 Simulation-based

Regression runs are executed for both SpaceWire interface implementations separately. Overall, five regression runs were executed for each implementation. A regression run consists of 800 tests that apply 160000 DIRQ transmissions. Thus, a total of 800000 DIRQs were transmitted for each implementation.

Two connected SpaceWire interfaces are driven by six individual clks. Clk offsets were applied randomly for each test to create arbitrary phase relations between all clks. The offsets granularity was set to 1 ns in order to provide a wide range of different phase relations. The ability to handle a granularity of 1 ns is a major advantage of functional simulation. In contrast to that, FPV ran into serious complexity issues for a lower clk granularity as explained in Section 6.4.2. Tests results for the standard SpaceWire interface implementation are given in Table 6.2. A low number of failed tests were encountered throughout all regression runs for the initially defined latency range. All failed

Table 6.2: Simulation results for standard SpaceWire interface implementations.

Regression ID	Latency range [clk_spw1 cycles]	Passed	Failed
1[a]	16 - 29	784 (98.0%)	16 (2.0%)
2[a]	16 - 29	781 (97.6%)	19 (2.4%)
3[a]	16 - 29	783 (97.9%)	17 (2.1%)
4[a]	16 - 29	784 (98.0%)	16 (2.0%)
5[a]	16 - 29	789 (98.6%)	11 (1.4%)
1[b]	15 - 29	800 (100%)	0 (0%)
2[b]	15 - 29	800 (100%)	0 (0%)
3[b]	15 - 29	800 (100%)	0 (0%)
4[b]	15 - 29	800 (100%)	0 (0%)
5[b]	15 - 29	800 (100%)	0 (0%)

[a] Initial latency range defined by cover statements
[b] Corrected latency range

tests were caused by a DIRQ latency of 15 clk_spw1 cycles, which extends the lower boundary of the initial valid area. Another execution of all regression runs with an extended latency range leads to a fault-free result.

Table 6.3: Simulation results for low jitter SpaceWire interface implementations.

Regression ID	Latency range [clk_spw1 cycles]	Passed	Failed
1	32 - 35	800 (100%)	0 (0%)
2	32 - 35	800 (100%)	0 (0%)
3	32 - 35	800 (100%)	0 (0%)
4	32 - 35	800 (100%)	0 (0%)
5	32 - 35	800 (100%)	0 (0%)

A different situation is present for the results of the low jitter SpaceWire

interface implementation, as shown in Table 6.3. All regression runs were executed with the same constraints but finished without a single failed test for the initially defined latency range.

6.4.2 Formal-based

The valid DIRQ latency range is defined as a cover statement for FPV in the same way as it is done for functional simulation. This is required to ensure that no over constraining is present, which could lead to false-positive results. The FPV tool can select different clk offsets throughout evaluation runs. This allows an investigation of multiple phase relations between all clks. However, an increased number of available offsets slows down the verification runs substantially. Hence, three separated verification runs are applied with different clk offset modeling capabilities, as classified in the following.

- OFST4 - clks are modeled with four possible offsets
- OFST2 - clks are modeled with two possible offsets
- OFST0 - clks are static

The reduction of possible offsets allows a deeper evaluation but reduces the number of possible phase relations between all clks. OFST4 provides the maximum acceptable number of four clk offsets. In contrast to that, functional simulation provides phase offsets with a granularity of 1 ns, which allows more phase relations compared to OFST4. Unfortunately, a dependency between all relevant clks must be supposed concerning the transmission latency of DIRQs. Hence, it is not acceptable to apply multiple tests with a subset of modeled clks to reduce complexity. All verification runs were executed with a limit of 60 hours, with its results are discussed in the following.

The effect of different clk modeling approaches is given in Figure 6.14. It shows the evaluation progress of each verification run by average proof radii over time. The lowest complexity is present for OFST0, which provides the highest average proof radius. OFST0 even shows a clear difference between low and standard jitter implementations, which is not the case for OFST2

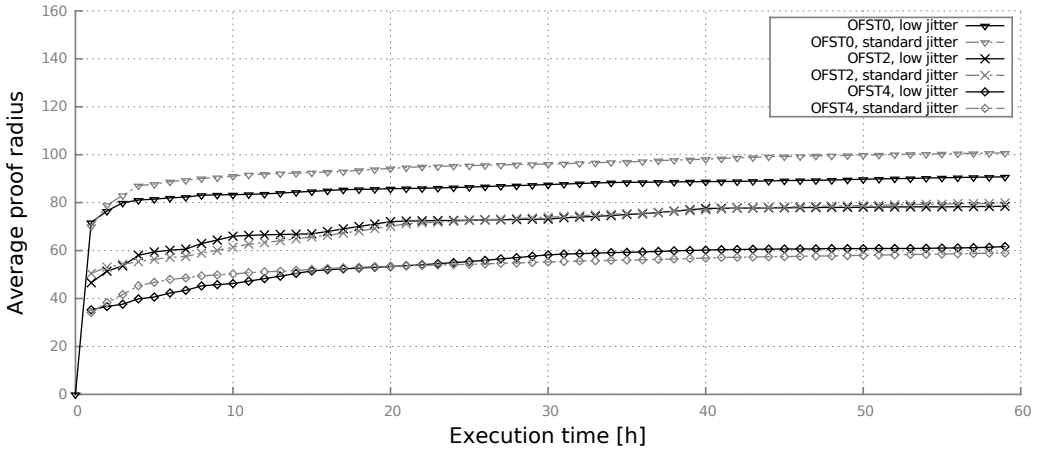

Figure 6.14: Formal property verification progress over time for different clk modeling approaches.

and OFST4. The figure also shows that most progress is made within the first hour for all traces. The collected coverage results for the standard SpaceWire interface implementation are given in Table 6.4. The standard SpaceWire in-

Table 6.4: FPV coverage results for standard jitter implementations.

Offset modeling	Latency range [clk_spw1 cycles]	Covered	Radius [clk_spw1 cycles]	Execution time
OFST4	15 - 29	full	18 - 41	1m 55s - 3h 18m
OFST2[a]	15 - 29	partly[1]	18 - 44	1m 2s - 1d 19h 44m
OFST0[b]	15 - 29	partly[2]	22 - 39	19s - 1d 23h 12m

[1] Latency value 15 is missed
[2] Latency value 15 and 29 are missed
[a] 1 engine with slowed progress
[b] 1 engine with consistent progress, 1 engine with slowed progress

terface implementation provides a wide valid latency range of 15 to 29, with all values being covered for OFST4. OFST2 and OFST0 were not able to cover all latencies, which shows the importance of considering phase relations. Latency value 15 provides a corner case scenario because it was uncoverable for two FPV runs and during functional simulation as discussed in Section 6.4.1. An additional uncoverable latency value is present for OFST0.

Engines are applied to each cover or property check statement until they

are solved. The health of each solving process indicates the progress, which is useful to decide how to proceed with a given verification run. One solving process was left for OFST2 after 60 hours with a single engine that indicated a slowed progress. In contrast to that, two solving processes were active until the end of the OFST0 verification run with one engine that provided consistent progress.

The radius represents the number of clk_spw1 cycles[7] required to fulfill a given cover statement. All covered statements are at 44 clk_spw1 cycles or below, which provides additional information about radii for bounded proofs applied during property checks. The required duration for solved statements is provided by the execution time. It took 3 hours 18 minutes to complete the coverage statements for OFST4. However, the meaning of execution time is different for uncompleted solving attempts (either coverage or property checks). In these cases, each statement is in the progress of being solved with a maximum investigated radius for each point in time. The required time to reach the actual maximum radius is given in the execution time. The execution time is updated as soon as any solving attempt extends the actual maximum radius for a given cover or property check statement.

The low jitter SpaceWire interface implementation provides a shorter range of valid latency values and in turn, a reduced jitter, as shown in Table 6.5. Full coverage is possible for OFST4 and OFST2, which is a difference to the

Table 6.5: FPV coverage results for low jitter implementations.

Offset modeling	Latency range [clk_spw1 cycles]	Covered	Radius [clk_spw1 cycles]	Execution time
OFST4	32 - 35	full	35 - 37	12m 14s - 1h 30m
OFST2	32 - 35	full	35 - 37	4m 11s - 9m 52s
OFST0[a]	32 - 35	partly[1]	36 - 37	1m 19s - 1d 23h 15m

[1] Latency value 32 and 35 are missed
[a] 2 engines with slowed progress

standard SpaceWire interface implementation. However, all active engines were in slowed progress after 60 hours for OFST0.

[7]Counting starts after the formal tool initialization phase.

Property checking results are given in Table 6.6 for the standard SpaceWire interface implementation. The areas to check for are defined by latency ranges 4 - 14 and 30 - 40. The valid latency range of 15 - 29, which was completely covered for OFST4, is located in between. The results show that full proofs were not found at all. This can be sufficient if proof boundaries are defined.

Table 6.6: FPV property check results for standard jitter implementations.

Offset modeling	Latency range [clk_spw1 cycles]	Full proof	Radius [clk_spw1 cycles]	Execution time
OFST4[a]	4 - 14	no	59 - 71	1d 18h 8m - 2d 11h 47m
	30 - 40	no	49 - 56	1d 7h 36m - 1d 16h 45m
OFST2[a]	4 - 14	no	76 - 84	1d 6h 39m - 2d 11h 55m
	30 - 40	no	73 - 82	1d 8h 55m - 1d 18h 0m
OFST0[b]	4 - 14	no	99 - 109	1d 6h 39m - 2d 11h 21m
	30 - 40	no	88 - 103	1d 10h 23m - 1d 19h 25m

[a] 4 engines with consistent progress, 18 engines with slowed progress
[b] 0 engines with consistent progress, 22 engines with slowed progress

A proof boundary of around 60 may be considered as reasonable for the given evaluation. It is because DIRQ transmissions can be considered to be finished in the area of 29 clk cycles for the standard implementation. Additionally, all valid latencies were covered within a radius of 44. Latency violations beyond a radius of 60 are probably caused by faulty DUT behavior and should be addressed by FPV bug hunting approaches. However, results for OFST4 show a minimum reached radius of 49, which represents a relatively short distance to 44. Hence, OFST4 results must be treated carefully.

At least 18 solving engines indicated a slowed progress after 60 hours. Thus, increasing the proof radii slightly might be possible by providing more time for the FPV tool, but it is very unlikely to get full proofs in that way.

The property check results for the low jitter SpaceWire interface implementation are shown in Table 6.7 without significant differences. Full proofs were not expected for the given case as well because the complexity of both implementations can be considered as similar. Almost all solving engines in-

Table 6.7: FPV property check results for low jitter implementations.

Offset modeling	Latency range [clk_spw1 cycles]	Full proof	Radius [clk_spw1 cycles]	Execution time
OFST4[a]	21 - 31[a]	no	64 - 73	1d 17h 27m - 2d 11h 7m
	36 - 46[a]	no	51 - 56	1d 6h 57m - 1d 16h 12m
OFST2[b]	21 - 31[b]	no	69 - 79	1d 17h 55m - 2d 11h 53m
	36 - 46[b]	no	78 - 87	1d 6h 51m - 1d 15h 9m
OFST0[c]	21 - 31[c]	no	91 - 96	1d 7h 3m - 2d 11h 59m
	36 - 46[c]	no	84 - 94	1d 10h 30m - 1d 19h 43m

[a] 3 engines with consistent progress, 19 engines with slowed progress
[b] 2 engines with consistent progress, 20 engines with slowed progress
[c] 1 engines with consistent progress, 21 engines with slowed progress

dicated slowed progress and even execution times are very similar. Finally, no counterexample was found for any SpaceWire interface implementation. This confirms the simulation-based results under the introduced assumptions (e.g. proof boundary) and concerns regarding OFST4 radii.

6.5 Conclusion

The evaluation results provide a precise end-to-end DIRQs latency range for both SpaceWire interface implementations. These ranges are used to define the jitter inside the complete system, which is vital to apply proper clk synchronization throughout all NCs. A latency range of 15 - 29 receiver clk cycles was determined for the standard SpaceWire interface implementation, which represents a latency of 280ns. A huge improvement was achieved for the modified implementation with its latency range of 32 - 35 receiver clk cycles. This is equal to a latency of 60ns and provides a reduction of around 78.6% compared to the standard SpaceWire interface implementation.

The DIRQ latency range determination was initially performed by a set of cover statements, embedded into a UVM test environment used to apply functional simulation. Larger regression runs indicated an insufficient range determination by missing a single latency at the lower boundary. However, a latency value was only missed for the standard SpaceWire interface implementation. Additional regression runs did not show any violation of the corrected latency range.

Formal property checking was applied to confirm simulation-based results. The evaluated design size of two SpaceWire interfaces only consists of 6849 design logic gates, which can be considered as quite small. However, different complexity reduction approaches (e.g. blackboxing and user-defined reset states) were still required to get the final results. A major complexity issue is the clk modeling used to apply phase shifts between all relevant clks. As a result, the maximum proof radius doesn't exceed 73 cycles for the most complex applied offset modeling. In contrast to that, verification runs with static clks provide a proof radius of more than 100 cycles. Additionally, the number of possible clk phase shifts is lower compared to functional simulation, which represents a deviation between both test methodologies. Another important aspect of the given formal results is the inability to find full proofs at all. However, the defined minimal proof radius of 60 cycles is considered as suitable, although not reached by every clk modeling approach.

Chapter 7

System Evaluation

7.1 Introduction

The system evaluation covers the main operational modes of the developed prototype and its utilization results. The prototype size was expected to be manageable by functional simulation. Thus, a UVM test environment was created to check and observe its behavior as introduced in Section 7.2. The application of formal property verification was additionally considered but discarded due to complexity concerns. The start-up behavior is analyzed with respect to successful termination and execution time in Section 7.3. Additionally, logical collisions are investigated to provide estimations on successful start-up sequences. The clock synchronization quality is discussed for different system configurations in Section 7.4 followed by an overview of utilization results (area, frequencies) for different FPGA targets in Section 7.5. Finally, some conclusions are given in Section 7.6.

7.2 Simulation Environment

Verification and performance tracking of the complete system is done by a UVM environment as shown in Figure 7.1. The DUT consists of eight NCs connected to either one or four routers in order to establish and maintain a global time. The network topology is selected automatically throughout the

execution of multiple tests. Three different agent types are used to control

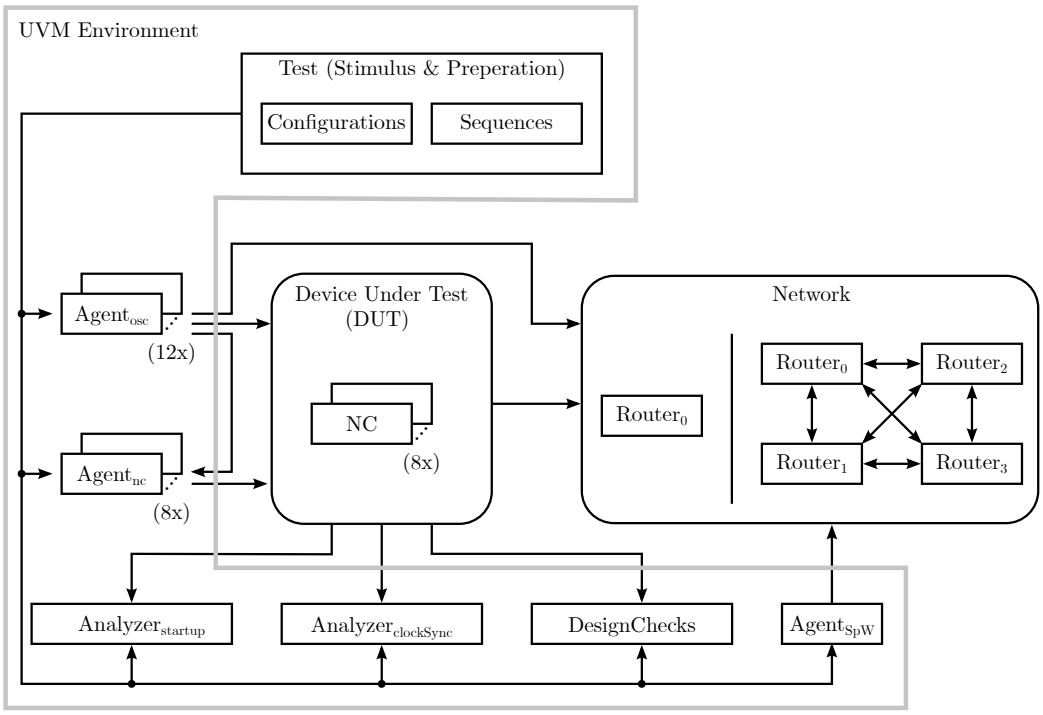

Figure 7.1: UVM simulation environment used for system characterization.

all DUT and network relevant interface signals. Agents generally consist of a driver capable of transferring transaction-level stimuli to the DUT directly or by use of BFMs. Additionally, a monitor observes the DUT to reconstruct transaction objects used for automated comparisons. Sequencers are used inside agents to arbitrate concurrently received transaction stimuli.

$Agent_{ocs}$ is used to mimic the behavior of physical oscillators. It is capable of serving three independent oscillator signals configurable before and during simulation runs. Routers and NCs have three oscillator inputs to operate its internal SpaceWire interfaces and its core logic. Thus, a maximum of 12 $Agent_{osc}$ units are required to provide arbitrary oscillator signals and drifts to each NC and router separately.

Hosts, represented by $Agent_{nc}$, are attached to NCs in order to communicate over the network. They are additionally used to store the schedule

and configuration of each NC. However, the schedule storage or modification would not be allowed to hosts for real applications. Thus, access to schedule related memory locations is limited to prototyping only.

Routers need to be configured before system operation. Routing tables must be set and global parameters like packet multicast or timeouts need to be defined. All these configurations are performed by $Agent_{SpW}$.

7.2.1 Metric Analyzer

Two analyzers are implemented to track and investigate the system properties of interest. The $Analyzer_{startup}$ is used to determine the duration of each start-up performed during related tests. These tests are performed with three or five NCs, which should provide a sufficient fault tolerance without expecting an excessive occurrence of logical collisions. Additionally, both network topologies are used for all executed tests. The analyzer starts a measurement as soon as the majority of all start-up NCs is activated. All participating NCs are activated within a maximum time window to ensure a start-up process can technically be finished within a predefined boundary. Otherwise, the majority of NCs could be held deactivated for years before its activation, which prevents a successful start-up procedure. However, the NC activation order and its constrained point in time are randomized to allow arbitrary start-up sequences.

The system enters a synchronous operation immediately after start-ups are successfully finished. Its overall clock synchronization is investigated by the use of the $Analyzer_{clockSync}$ unit. It observes all slot changes for already synchronized NCs during schedule execution. The analyzer is triggered by the first NC that changes its actual slot to the next slot. From this point, the analyzer creates a timestamp t_{first} and waits until all other synchronized NCs have performed a slot change to create a second timestamp t_{last}. The deviation between all clocks is defined by

$$clockDeviation = t_{last} - t_{first} \tag{7.1}$$

for all slot changes separately. The actual clock precision between all

synchronized NCs is represented by the maximum clock deviation observed throughout all tests.

7.2.2 Design Checks

The prototype has not been completed concerning exhaustive verification based on test requirements and automated scoreboard comparisons. Instead, multiple checker interfaces that contain assertions are described to track and verify the most relevant system behavior.

The schedule execution can be considered as the major functionality provided by each NC to execute a synchronous operation. Each NC is configured before system operation to establish an expected schedule execution. These expectations are described by assertions and checked automated by comparison to real system behavior during operation. It comprises checks for correct slot lengths as described inside the schedule definition and checks for ongoing schedule executions by all NCs that are synchronized. Additionally, schedule executions are influenced by state and rate correction values that are calculated dynamically. However, the expected effects of correction values and their update location inside the executed schedule can still be checked by assertions. The already introduced system architecture, given in Figure 5.7, shows that state and rate correction values influence macroticks. A fixed number of macroticks is executed for each cycle. However, the number of microticks per macrotick can vary to apply correction values. Thus, the number of expected microticks must be determined for each schedule cycle to check that correction values are applied correctly.

The implementation for this particular check is given in Figure 7.2. Property *p_cnt_ut_in_odd_cycle* starts its evaluation whenever an odd schedule cycle begins with slot number 1 and NCs are integrating or fully synchronized. The microtick determination is applied in sequence *s_cnt_ut_in_odd_cycle*. The sequence counts all microticks by use of variable *ut_cnt* once its evaluation starts until the cycle ends. Finally, variable *expected* is calculated by addition of the default microtick number (*reg_utpc_i*) and both correction values in order to compare it with the actual microtick number stored in variable *ut_cnt*. The

```
1  sequence s_cnt_ut_in_odd_cycle();
2    int ut_cnt = 0;
3    int ut_rate = 0;
4    int expected = 0;
5    (1'b1, ut_rate = rate_corr_i,
6    ut_cnt = 0, utcnt_odd_local_rst()) ##0
7    first_match
8    (
9    (oddc_i, ut_cnt++, ut_cnt_ref_odd_set(ut_cnt))
10   [*0:$] ##1 !oddc_i
11   ) ##0
12   (1'b1, expected = reg_utpc_i + ut_rate + state_corr_i,
13   expected_odd_local_set(expected)) ##0
14   ut_cnt == expected + 1 ||
15   ut_cnt == expected - 1 ||
16   ut_cnt == expected;
17 endsequence: s_cnt_ut_in_odd_cycle
18
19 property p_cnt_ut_in_odd_cycle();
20   $rose(oddc_i) && syncpre_i && scd_slotval_o == 1 |->
21   s_cnt_ut_in_odd_cycle();
22 endproperty: p_cnt_ut_in_odd_cycle
```

Figure 7.2: Microtick number check for odd schedule cycle executions.

evaluation for correctness contains three comparisons (lines 14 to 16). It is because the rate correction algorithm is not able to distribute a single positive or negative microtick for odd cycles due to the interaction with the state correction. This specific scenario could be handled by different properties or the expected value calculation need to incorporate these conditions. However, for checks within even schedule cycles, a single comparison is performed that verifies the rate correction value distribution more precise. Thus, for the given prototype, the uncertainty of a single microtick throughout odd checks was considered as acceptable.

7.3 Start-up Analysis

Start-ups of the introduced system are critical because they are a precondition of synchronous operation. The start-up duration depends on several system parameters like timeouts applied initially or caused by logical collisions, the number of network controllers, and the structure of the network itself.

The start-up behavior evaluation is organized in various tests. Each test was executed multiple times to track the behavior under different system configurations. The results of start-up executions and observed logical collisions are given in the following sections.

7.3.1 Execution Times

Overall, 32800 start-ups were executed. Each start-up begins with all NCs are inactive. All start-up involved NCs, either three or five depending on the actually executed configuration, are activated in an arbitrary order with delays in between. These activation delays are randomized by the UVM environment with a maximum delay of one schedule cycle (0.5 ms, 1 ms or 2 ms). These delays result in the activation of all NCs within a single schedule cycle. Each activated NC performs a unique timeout with a minimum duration of at least one schedule cycle. The initial timeout is executed to integrate into synchronous operation or to participate in start-up sequences already initiated by other NCs as introduced in Section 5.4. NCs initiate a start-up sequence in case no communication at all is recognized within the initial timeout.

The measurement of a start-up execution time begins if sufficient NCs are activated to perform a start-up sequence successfully. The measurement completes if a single NC provides the synchronizing DIRQ that finishes the start-up sequence. All NCs are deactivated to prepare the subsequent start-up sequence once the transmitted DIRQ leads to a synchronous operation. Begin and end of each start-up is tracked by the UVM environment automatically.

Figure 7.3 shows the results for the network that consists of a single router. Each cross inside the diagram relates to a single executed start-up. All tests

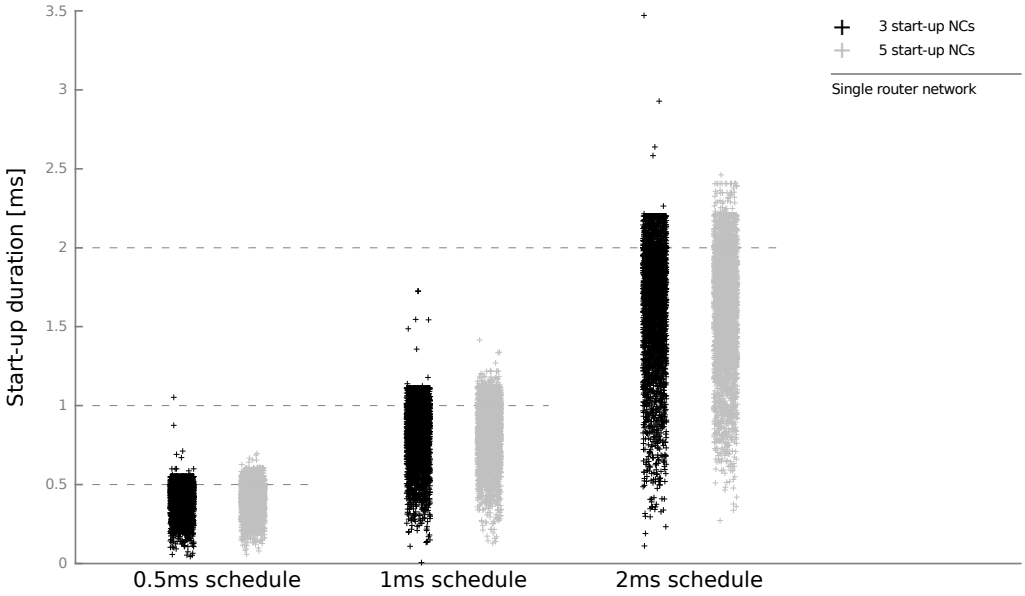

Figure 7.3: Tracked start-up durations for a network consisting of one router.

were applied with schedule lengths of 0.5 ms, 1 ms, and 2 ms. However, for start-up tests, the schedule length only influences the initial timeout of each NC. The majority of start-up execution times is located below its respective initial timeout of 0.5 ms, 1 ms or 2 ms. Overall, all observed execution times don't provide extremes that indicate a high occurrence of logical collisions or an improper selection of timeout values. Results for the network that consists of four routers are given in Figure 7.4. It doesn't provide significant differences to the single router configuration. However, it is noticeable that some kind of borders of start-up execution times are present throughout all configurations. This can be clearly seen inside Figure 7.3 for three start-up NCs and a 1 ms schedule at start-up execution time value of approximately 1.1 ms. This happens for specific NC enable arrangements. In these cases, the last NC required to enable a measurement is activated and able to begin and complete the start-up sequence without interruption.

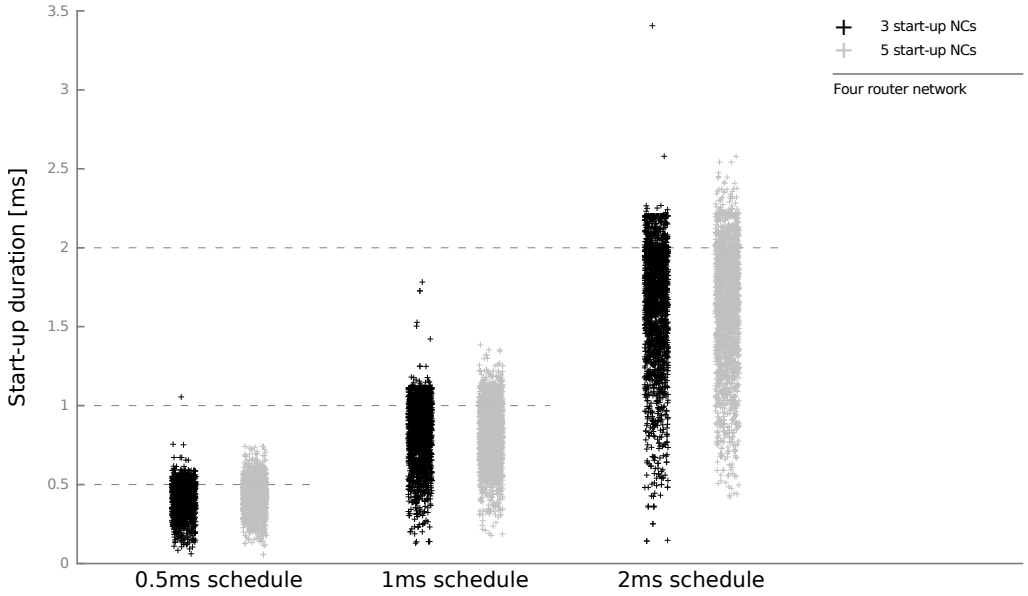

Figure 7.4: Tracked start-up durations for a network consisting of four routers.

7.3.2 Logical Collisions

Logical collisions are present whenever NCs receive a not acknowledge as a response on their SUFs as introduced in Section 5.4.1. The probability of logical collisions increases, the more NCs are allowed to participate in start-up processes. Additionally, the probabilities are influenced by timeout durations that are executed by NCs initially or after a logical collision is detected. The number of logical collisions for each executed start-up was captured by the UVM environment with its results given in Figure 7.5. The distribution of observed logical collisions is given for each system configuration separately. It can be seen that the network structure is another system property that affects the occurrence of logical collisions. The overall distribution within configurations that consists of a single router is much lower compared to the four router configuration. Additionally, a significant occurrence of four collisions is present for five NCs operating at the four router network.

Every initiated start-up was finished successfully, although up to eight logical collisions were observed. However, this test is not a full proof that

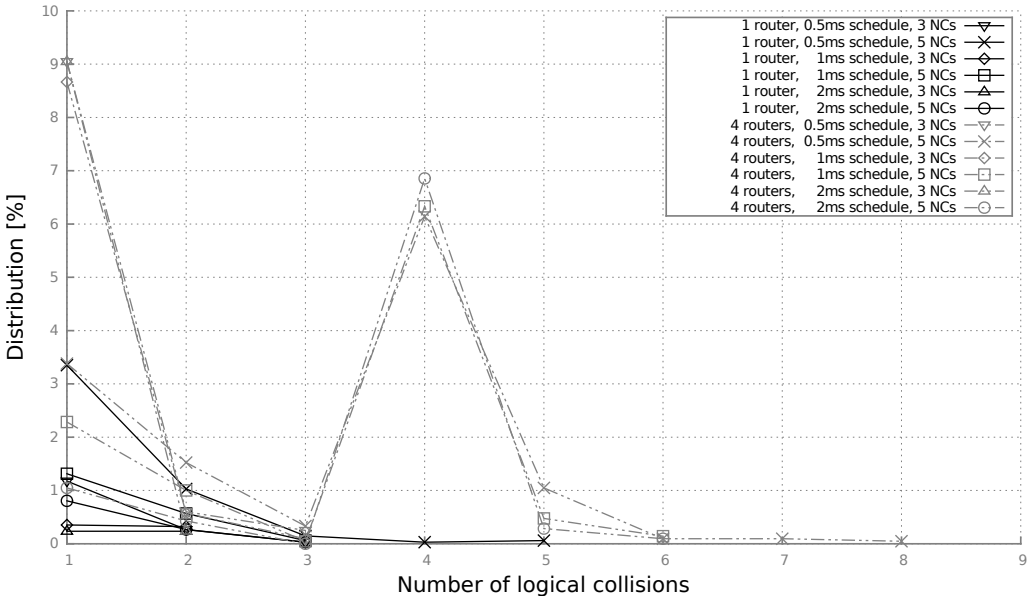

Figure 7.5: Distribution of encountered logical collisions for all system configurations.

every start-up under the applied parameters can finish because the analysis is based on functional simulation. There still might be input combinations that lead to periodic logical collisions which preventing start-up phases to be finished. This problem must be investigated by the use of formal verification as a continuation of this work.

7.4 Distributed Clock Analysis

Clock deviations were measured for all defined system configurations. This includes three different schedule lengths and two network configurations. Additionally, two different SpaceWire interface implementations were used. One implementation behaves as defined in its related standard [ESA19, p. 84]. The second implementation contains the jitter reduction capabilities introduced in Section 5.5 and investigated in Section 6.4.

The general test setup was already introduced in Figure 7.1. All executed tests were performed with eight NCs to exchange and control their clock in-

formation. All NCs are enabled in parallel at the beginning of each test, whereas only two NCs are selected for a start-up process to establish a synchronous system operation. All remaining NCs integrate initially over time, as introduced in Section 5.4.2. Each NC is driven by a separated oscillator, which is simulated in a way that uncertainties of +100 ppm or −100 ppm are applied. These values are considered as reasonable based on the experience gathered during space-related projects.

The overall evaluation consists of multiple executed tests for all system configurations. Each test contains 118 schedule cycles with an overall executed slot number of 3894, 7906, or 15694 depending on the selected schedule. In total, 549880 slots were executed and its clock deviations observed across all tests. Oscillator uncertainties are applied randomly to all NCs at the beginning of each test and kept stable throughout the whole test execution. However, the randomization of uncertainties is further constrained in a way that three times +100 ppm, three times −100 ppm, and two times 0 ppm uncertainties are applied. This is required to prevent that eight equal uncertainties are applied. This, in turn, would lead to full removals of clock and schedule drifts between all NCs. The nominal main frequency of NCs is set to 50 MHz. Two additional oscillators, with nominal frequencies of 120 MHz, are used inside each SpaceWire interface for serial data transmissions and receptions, whereas all available oscillator inputs are asynchronous to each other and individually controlled.

7.4.1 Deviation Over Time

The system evaluation is based on a repetition of test cases with different static configurations and randomized parameters. The result of two single test runs is given in Figure 7.6 to show possible clock deviations as a function of executed schedule slots. Clock deviations are measured for each slot change as an absolute value. Due to clock deviations, it is likely that some NC start its slot change before other NCs. This results into n slot changes[1], performed in an arbitrary order, within a defined time window. The first slot

[1]Value n is defined by the number of synchronized NCs.

change triggers a time measurement, whereas the last slot change completes the measurement by determining the temporal difference. The given traces are observed in a four router network and a schedule length of 2 ms (15694 slots). Arbitrary NC deactivations followed by reactivations are performed during synchronous operation to force ongoing integration throughout a test. However, at least three NCs are kept active as they are required to maintain a synchronous operation for all executed tests. The ongoing reintegration of NCs causes further deviations alongside to the deviations introduced by DIRQ jitter.

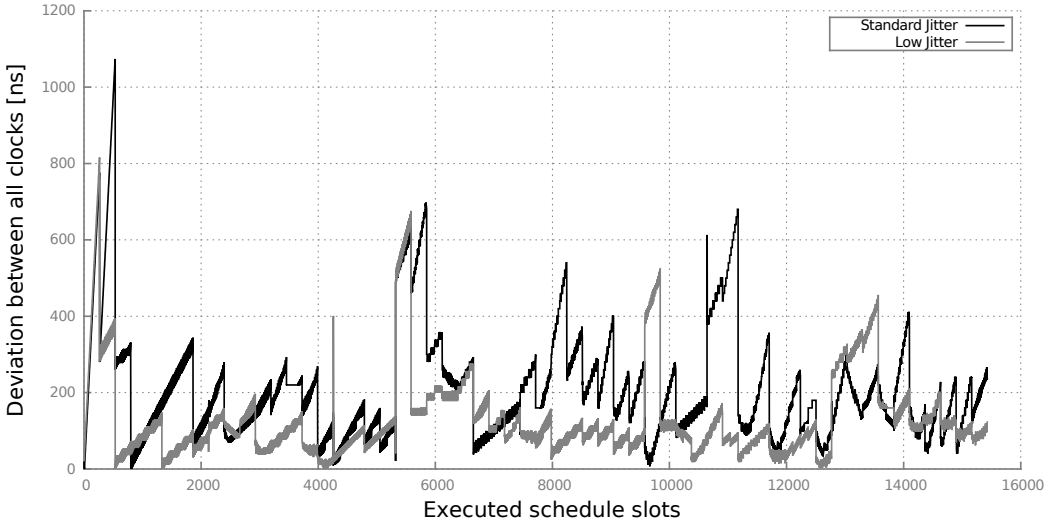

Figure 7.6: Deviation of single traces as a function of executed schedule slots for a 2ms schedule and a four router network.

The initial spike of both traces represents a particular problem. As introduced in Section 5.4.3, rate and state corrections are applied after the execution of two schedule cycles. This allows integrating NCs to apply both corrections multiple times before entering synchronous operation with an improper synchronized clock. However, NCs that complete start-up processes successfully transfer immediately into synchronous operation without any possibility of applying a correction. Thus, clocks of these NCs deviate without compensation, depending on their oscillator differences. The length of executed schedules additionally affects the problem because clocks have more time to

deviate until initial corrections are ready for application.

The overall deviation for the low jitter SpaceWire interface implementation can be treated as much lower compared to the standard jitter implementation. However, the given traces only represent a subset of all available traces. Thus, graphs that provide distribution of collected deviations is the convenient way to interpret available results and given in the following Section 7.4.2.

7.4.2 Distribution

Throughout all executed tests, a set of observed deviations was collected for each static configuration. A static configuration is identified by its network (one router, four routers), schedule length (0.5 ms, 1 ms, 2 ms), and the kind of SpaceWire interface implementation (standard jitter, low jitter). Hence, 12 different static configurations are investigated in this work.

The effect of both SpaceWire implementations for all network and schedules configurations is given in Figure 7.7 and Figure 7.8. Measured clock deviations are grouped into ranges of 50 ns on the x-axis. This size is considered as suitable to illustrate and compare the overall behavior between all tests. A total number of deviations are measured for each static configuration. These deviations are located inside its related range to calculate the occurrence, which is represented as distribution on the y-axis.

The advantage of the low jitter implementation is obvious. The distribution is larger within all static configurations for low clock deviations up to 99 ns, with a steep decrease for higher clock deviation values. In contrast to that, standard jitter implementations provide a broader percentage of higher clock deviations starting from 100 ns.

The effects of networks and different schedule lengths are illustrated in Figure 7.9. It is expected that larger networks introduce more uncertainties because of an increased number of SpaceWire interfaces that need to be traversed. This, in turn, leads to larger DIRQ jitters with a direct impact on the overall clock synchronization quality. This assumption is confirmed for the standard jitter implementation. All single router configurations provide a higher distribution up to 149 ns compared to the four router setups. A

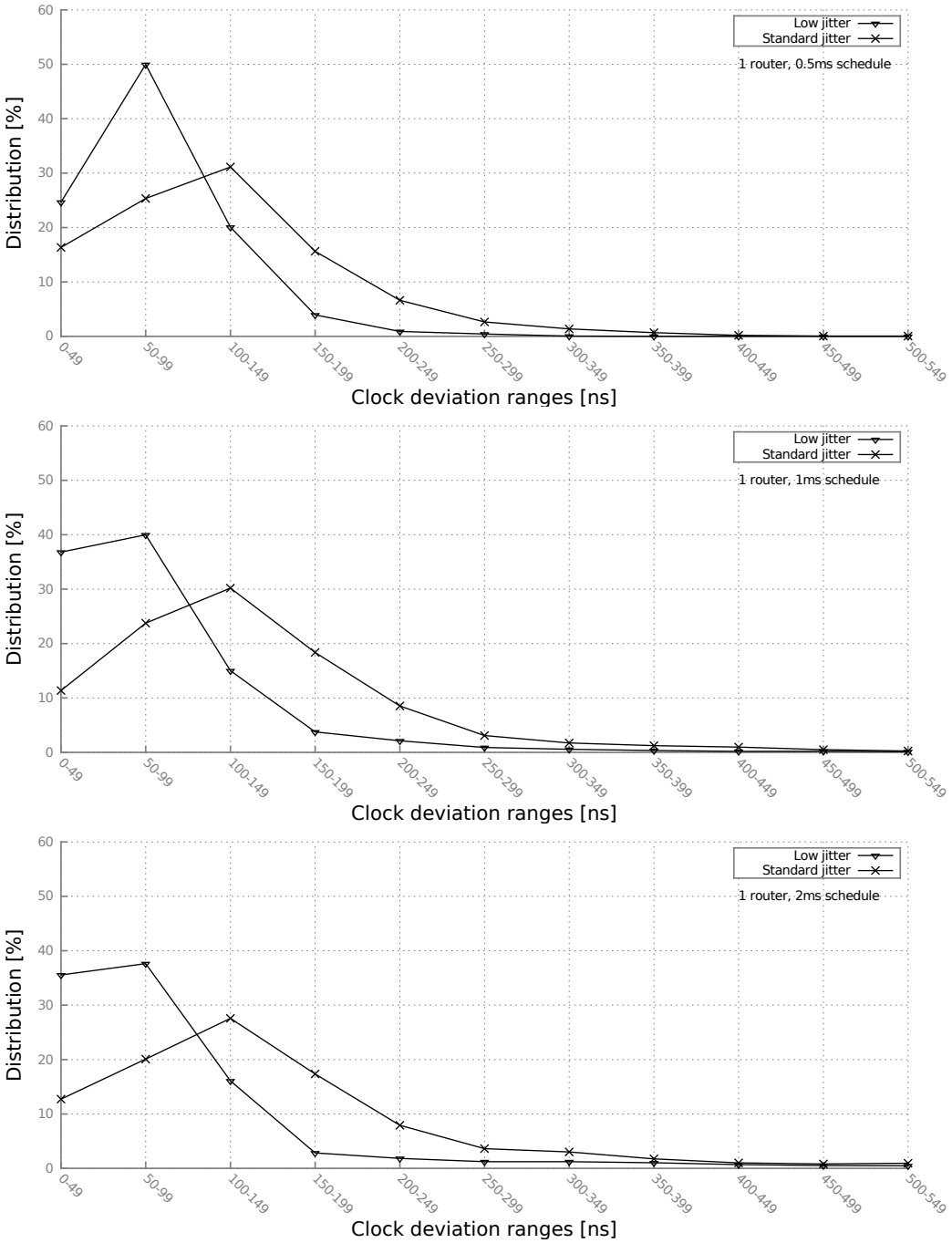

Figure 7.7: Clock deviations for a one router network between low jitter and standard SpaceWire interface implementations.

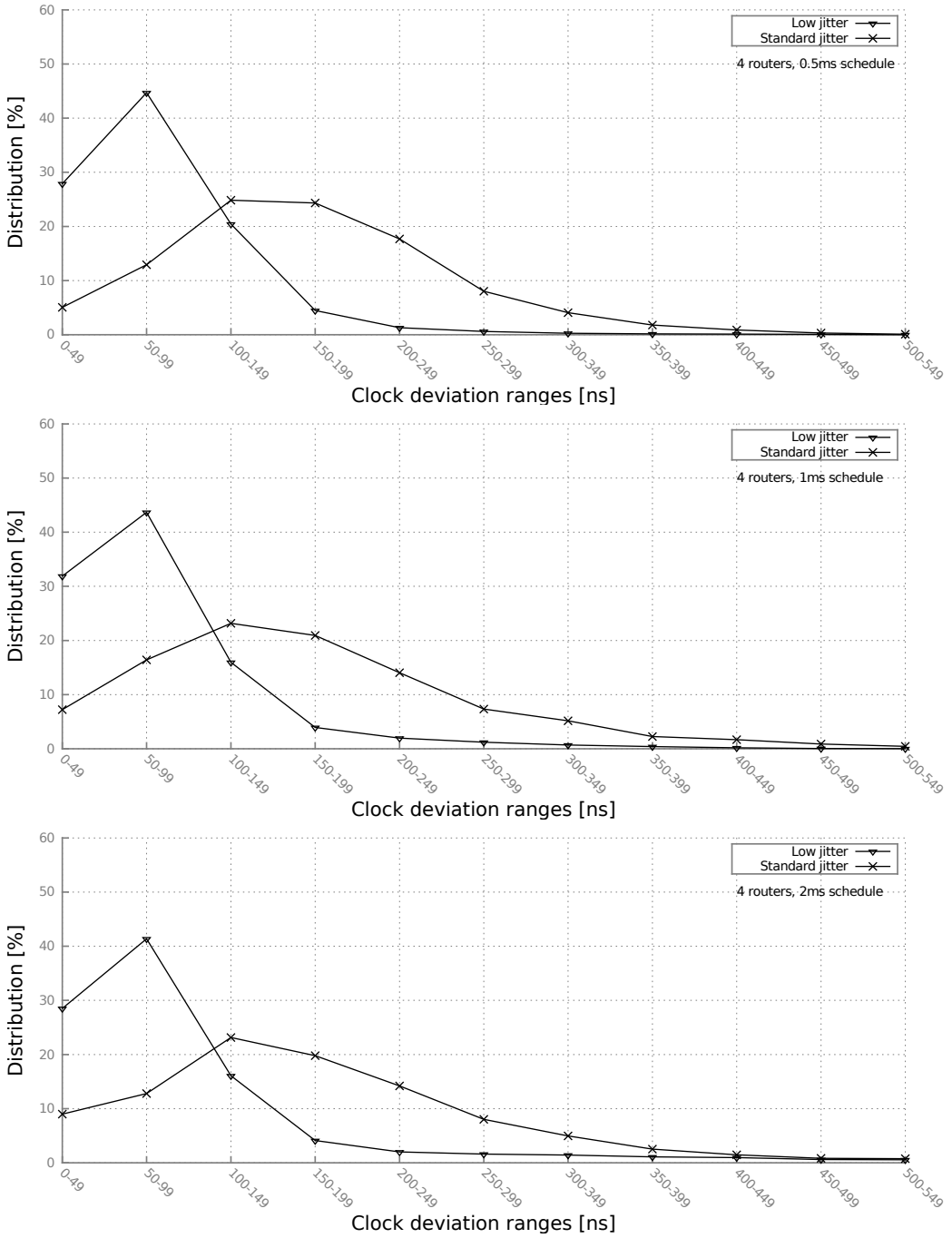

Figure 7.8: Clock deviations for four router network between low jitter and standard SpaceWire implementation.

different situation is present for the low jitter implementation. A closer arrangement of all traces is expected. It is because of the overall reduced jitter that correlates to the clock synchronization quality. However, a clear advantage for the single router configuration can't be determined for the available set of data. This may be corrected by an increased number of executed tests and must be further evaluated.

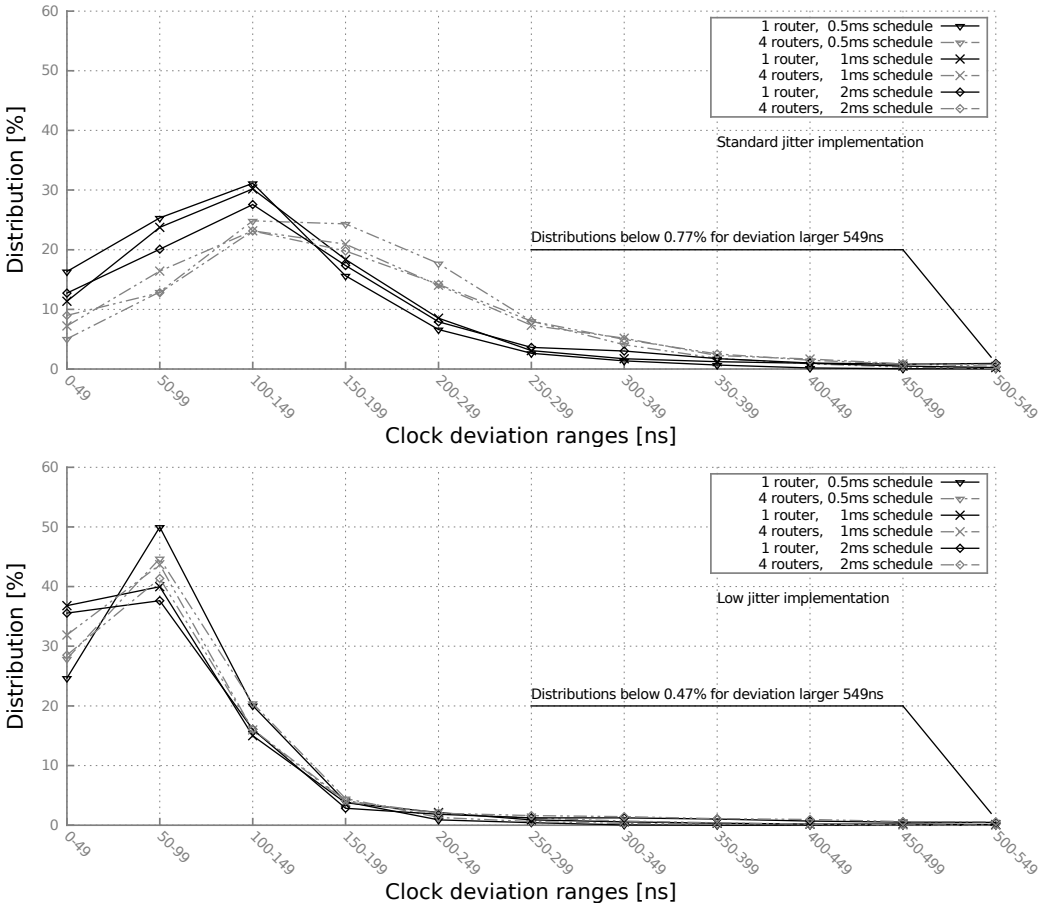

Figure 7.9: Effect of clock deviations for different networks and schedules.

Different schedule lengths are also of interest because they define the frequency of rate and state correction application, as explained in Section 5.4.3. It can be expected that higher correction frequencies achieve better synchronization results than lower ones. The reason for this is that clocks increase their deviation over time, whereas an increased synchronization frequency

decreases this duration.

This expectation is typically illustrated by a higher distribution of low deviation ranges for low schedule duration. However, the differences between schedules are not that distinctive for the observed values. Especially the differences between 1 ms and 2 ms schedules are minor. The expected behavior is illustrated at best for the standard jitter implementation with a single router network. The 0.5 ms provides the highest distribution of up to 149 ns. After that, the distribution decreases below the 1 ms and 2 ms schedules until they finally converge. Although not all traces exactly match, it clearly shows a trend in the direction of the expectation, which may become more visible for a larger set of overall observed deviations.

The given deviation ranges with a limit of 500 - 549 don't represent the maximum observed value. The maximum values are given in Table 7.1 for each static configuration. It is noticeable that maximum deviations of low

Table 7.1: Maximum deviations for all static evaluated configurations.

Router	Schedule [ms]	Implementation	Max. Deviation [ns]
1	0.5	standard	540
1	0.5	low	350
1	1	standard	683
1	1	low	540
1	2	standard	1134
1	2	low	1094
4	0.5	standard	617
4	0.5	low	496
4	1	standard	758
4	1	low	564
4	2	standard	1154
4	2	low	833

jitter implementations are always lower than standard jitter implementations for the same router and schedule configuration. Additionally, maximum deviation values increase for larger schedules with equal implementation and

router configurations. Finally, four router configurations for a given schedule and implementation always provide larger deviations compared to the same configuration of a single router.

7.5 Target Utilization

A single NC was synthesized for several FPGA devices to evaluate the resource utilization and the maximum operating frequencies. All results are based on synthesis without applied Triple Modular Redundancy (TMR) and given in Table 7.2. Two Microsemi FPGAs (RTG4, ProASIC3L) are given inside

Table 7.2: Synthesis results for a single NC with different target devices.

FPGA device	Utilization [%]		Est. frequencies [MHz]		
	LUT	FF	Main	TX	RX
RTG4[1]	6	4	53.8	276.2	271.8
ProASIC3L[2]	42	9	35.3	196.6	174.3
Virtex-4QV[3]	17	12	77.5	332.7	322.0
Virtex-5QV[4]	8	7	114.8	389.5	614.3
Zynq-7000[5]	2	1	149.9	704.9	739.3
Kintex UltraScale[6]	2	1	193.3	956.6	1145.2

[1] rt4g150cg1657-1
[2] a3pe3000lfbga324-1
[3] xqr4vfx60cf1144-10
[4] xqr5vfx130cf1752-1
[5] xc7z100ffg1156-1
[6] xcku060-ffva1517-1

the utilization results, whereas both are available as radiation-hardened or radiation-tolerant parts. All remaining FPGAs are manufactured by Xilinx. Virtex-4QV and Virtex-5QV currently represent the only available space-grade devices. The Kintex UltraScale is part of the evaluation because it is selected to be the next space-grade Xilinx device [Elf18]. The Zynq-7000 is a System-on-Chip (SoC), consisting of two CPUs connected to a FPGA and located inside a single chip. This part has recently been subject to COTS

based on-board computing systems to be used in the space domain [WG18; Tre+18].

The results table provides the consumption of LUTs and FFs in order to allow a rough comparison between devices and vendors. However, it must be considered that LUTs can differ between technologies in their number of inputs which leads to different capabilities. Additionally, there might be restrictions in using LUTs and FFs for specific situations. For instance, ProASIC3L devices restrict parallel usage of LUTs and FFs in case they are located in the same VersaTile. FF and LUT consumptions are very low for most parts and provides sufficient remaining resources for additional Intellectual Property (IP) Cores alongside NCs on each FPGA.

All estimated frequencies are sufficient to allow SpaceWire transfer rates of at least 120 MBits/s for the used IP. However, frequency estimations based on synthesis are often reduced throughout place and route processes. This could lead to a transfer rate reduction down to 100 MBits/s for ProASIC3L devices. Optimizations of IPs to shorten combinational paths are typical tasks to solve this problem and could be applied to the given prototype as well.

Block RAM utilization significantly depends on the schedule size and should be shifted into external memory for future developments. Hence, BRAM utilization is not added here. This shift is vital for smaller FPGAs with fewer BRAM blocks to provide sufficient resources to IPs that might be instantiated in parallel to NCs.

7.6 Conclusion

The evaluation demonstrates all the capabilities of the introduced approach. Clock synchronization is performed without centralized time distribution and doesn't require time accumulation inside the network. Additionally, a global time is established decentralized throughout a dedicated start-up sequence.

The evaluation of the start-up behavior shows several important aspects. Each applied start-up sequence was successfully completed for all given sys-

tem configurations. The number of five start-up participating NCs should provide sufficient fault tolerance for most applications. The low number of observed logical collisions indicates a very low probability of having start-ups that never succeed. For the given timeout and system configurations, all start-ups were completed within 3.5 ms, with most of them are finished in less than 2 ms. However, the start-up should be executed ideally only once throughout system operation. Thus, it is essential that start-ups succeed at all rather than focusing on the maximum execution time.

A synchronization between eight distributed clocks was performed throughout the evaluation for multiple system configurations. A modified SpaceWire interface was created to improve the overall synchronization quality by reducing the jitter of broadcast codes. This modification also represents a part of all used system configurations. Overall, a maximum clock deviation of 1154 ms was observed for the longest schedule, a four router network, and the standard SpaceWire implementation. The impact of synchronization quality, depending on the network size, is clearly represented by the distribution graphs for standard SpaceWire interfaces. The dependency between synchronization quality and network size is reduced for the modified SpaceWire interface. The correlation between schedule length and synchronization quality is much lower than expected for both SpaceWire implementations. The modified SpaceWire interface causes the most noticeable improvement of synchronization quality. Its usage provides much better clock synchronization for all schedules and network configurations compared to the standard SpaceWire interface. The permanent applied NC disconnects did not lead to a full loss of synchronization, which demonstrates the system tolerance to fail-stop, crash, or omission failures.

All metrics were tracked by functional simulation inside a UVM test environment. Its main advantage is the unlimited access to all relevant prototype internal values required to create the statistics presented in this work. However, the execution of all tests requires multiple weeks to be finished. Thus, emulation based verification should be considered to extend the test coverage drastically.

Chapter 8

Outlook

Formal broadcast code latency evaluation suffers from the complexity introduced by the end-to-end checks. These end-to-end evaluations typically have a massive cone of influence for the given properties that are analyzed. Additionally, formal property checks on data paths generally require a larger temporal space compared to control paths, which complicates evaluations further. A single SpaceWire interface should be analyzed isolated in order to target full proofs. This reduces the overall logic and the number of clks that need to be modeled. However, this requires direct interaction on the data/strobe interface, which leads to more complex assumptions and properties. Clk modeling was identified as a major complexity issue that caused a massive proof radius reduction. Static clk offsets could be applied by the use of scripts with all possible phase relations are evaluated in a separated verification run. This probably leads to an increase of verification time but allows proof radii in the area of 100 with more evaluated phase relations compared to the actual used approach.

Formal property checking has not been applied to the complete system because of complexity concerns that came up after the formal broadcast code evaluation. However, it should be possible to select specific sub-functionalities of the complete system to be checked by formal tools. Mainly the start-up behavior should be targeted by formal verification because of its vast system

importance. A more abstract model checking approach with UPPAAL[1] could be an alternative if formal property checking on the given RTL is not possible.

Several tasks should be addressed in order to apply the system to real applications. The executed schedules in this work were defined manually, which is a sophisticated and error-prone process. Thus, an automated schedule synthesis needs to be developed. Additionally, a larger system run-time should be evaluated. This can be achieved by emulation, which provides access to system internal signals by default. However, emulation generally requires a significant financial investment. Hence, an FPGA-based prototype with access to system internal parameters could be a cost-efficient alternative.

A practical use case of the given clock synchronization approach will be evaluated for the successor of the DLR internal compact satellite Eu:Cropis [Kot+18]. A utilization for secondary payloads might be possible to keep risks moderate that originate by the open tasks.

[1] Tool to model systems as timed automata in order to apply model checking on it.

Part IV

Appendix

Appendix A

Notations

A.1 Introduction

System properties or definitions are given by several representations. In the following, a short listing of EBNF and Unified Modeling Language (UML) is given due to its usage inside this thesis.

A.2 UML State Diagrams

UML provides a wide range of graphical notions in order to describe and design software systems. However, specific UML diagram types (e.g. state machine or timing diagrams) are very useful do describe behavior of hardware systems. Figure A.1 shows a subset of available elements used for state machine diagrams inside this work.

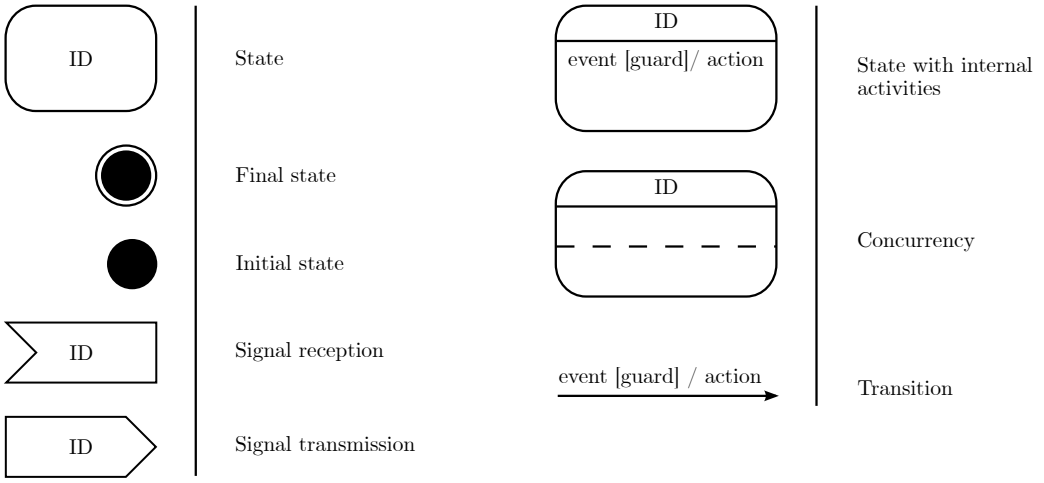

Figure A.1: Subset of UML state diagram elements.

A.3 Extended Backus-Naur Form

The Extended Backus-Naur Form (EBNF) is a syntactic metalanguage used to formally define a specific syntax. A well known use case is the definition of programming languages. The EBNF is standardized in [Int96] and provides the following operators.

Table A.1: EBNF operators.

Operator	Usage
*	Repetition
-	Exception
,	Concatenation
|	Alternative
=	Definition
;	Terminator
’	Quotation 1
”	Quotation 2
(* *)	Comment
( )	Group
[ ]	Option
{ }	Optional repetition
?	Special sequence

Bibliography

[Agg+11] P. Aggarwal, D. Chu, V. Kadamby, and V. Singhal. "End-to-End Formal using Abstractions to Maximize Coverage". In: *Formal Methods in Computer Aided Design (FMCAD)*. 2011.

[AK07] A. Ademaj and H. Kopetz. "Time-Triggered Ethernet and IEEE 1588 Clock Synchronization". In: *International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*. 2007.

[AP98] E. Anceaume and I. Puaut. *Performance Evaluation of Clock Synchronization Algorithms*. Research Report RR-3526. Institut National de Recherche en Informatique et en Automatique, 1998.

[Bag+10] M. Bagatin, S. Gerardin, A. Paccagnella, G. Cellere, A Visconti, and M. Bonanomi. "Increase in the Heavy-Ion Upset Cross Section of Floating Gate Cells Previously Exposed to TID". In: *IEEE Transactions on Nuclear Science* 57.6 (Nov. 2010), pp. 3407–3413.

[Ber+19] H. Bergeron, L. C. Sinclair, W. C. Swann, I. Khader, K. C. Cossel, M. Cermak, J. D. Deschenes, and N. R. Newbury. "Femtosecond time synchronization of optical clocks off of a flying quadcopter". In: *Nature Communications* 10.1 (Apr. 2019).

[BMD19] K. Borchers, S. Montenegro, and F. Dannemann. "Volatile Register Handling for FPGA Verification Based on SVAs Incorporated into UVM Environments". In: *IEEE Aerospace Conference*. Mar. 2019.

[Bor+18] K. Borchers, G. Fey, D. Lüdtke, and S. Montenegro. "Time-Triggered Data Transfers over SpaceWire for Distributed Systems". In: *IEEE Aerospace Conference*. Mar. 2018.

[Bor+19] K. Borchers, D. Lüdtke, S. Montenegro, and F. Dannemann. "Performance and Utilization Results for Time-Triggered Data Transfers over SpaceWire". In: *IEEE Aerospace Conference*. Mar. 2019.

[Boy+16] M. Boyer, H. Daigmorte, N. Navet, and J. Migge. "Performance impact of the interactions between time-triggered and rate-constrained transmissions in TTEthernet". In: *Embedded Real-Time Software and Systems (ERTS)*. 2016.

[BSV11] N. Battezzati, L. Sterpone, and M. Violante. *Reconfigurable Field Programmable Gate Arrays for Mission-Critical Applications*. Springer, 2011. ISBN: 978-1-4419-7595-9.

[Caf+02] M. Caffrey, P. Graham, E. Johnson, and M. Wirthlin. *Single-Event Upsets in SRAM FPGAs*. Research Report LA-UR-02-3005. Los Alamos National Laboratory, 2002.

[Cen+13] G. Cena, I. C. Bertolotti, S. Scanzio, A. Valenzano, and C. Zunino. "Synchronize Your Watches: Part II: Special-Purpose Solutions for Distributed Real-Time Control". In: *IEEE Industrial Electronics Magazine* 7.2 (June 2013), pp. 27–39.

[Cer+10] E. Cerny, S. Dudani, J. Havlicek, and D. Korchemny. *The Power of Assertions in SystemVerilog*. Springer, 2010. ISBN: 978-1-4419-6599-8.

[CH17] A. Choudhury and A. Hari. "Accelerating CDC Verification Closure on Gate-Level Designs". In: *Design and Verification Conference & Exhibition (DVCon)*. 2017.

[CLS04] V. Claesson, H. Loenn, and N. Suri. "An efficient TDMA start-up and restart synchronization approach for distributed embedded systems". In: *IEEE Transactions on Parallel and Distributed Systems* 15.8 (2004), pp. 725–739.

[Coh+16] B. Cohen, A. Kumari, L. Piper, and S. Venkataramanan. *SystemVerilog Assertions Handbook, 4th Edition: ... for Dynamic and Formal Verification*. 4th ed. VhdlCohen Publishing, 2016. ISBN: 978-1518681448.

[Coh13] B. Cohen. "SVA in a UVM Class-based Environment". In: *Verification Horizons* 9.1 (Feb. 2013), pp. 24–28.

[Cou+09] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. "An Introduction to High-Level Synthesis". In: *IEEE Design & Test of Computers* 26.4 (Aug. 2009), pp. 8–17.

[Cum08] C. Cummings. "Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog". In: *Synopsys Users Group (SNUG)*. 2008.

[DH02] R. Drechsler and S. Höreth. "Gatecomp: Equivalence Checking of Digital Circuits in an Industrial Environment". In: *International Workshop on Boolean Problems*. 2002, pp. 195–200.

[Dod+10] P. E. Dodd, M. R. Shaneyfelt, J. R. Schwank, and J. A. Felix. "Current and Future Challenges in Radiation Effects on CMOS Electronics". In: *IEEE Transactions on Nuclear Science* 57.4 (Aug. 2010), pp. 1747–1763.

[Dol+83] D. Dolev, N. A. Lynch, S. Pinter, E. W. Stark, and W. E. Weihl. "Reaching approximate agreement in the presence of faults". In: *IEEE Symposium on Reliability in Distributed Software and Database Systems*. 1983, pp. 145–154.

[DS01] R. Drechsler and D. Sieling. "Binary decision diagrams in theory and practice". In: *International Journal on Software Tools for Technology Transfer* 3.2 (May 2001), pp. 112–136.

[DS07] S. Disch and C. Scholl. "Combinational Equivalence Checking Using Incremental SAT Solving, Output Ordering, and Resets". In: *Asia and South Pacific Design Automation Conference*. 2007.

[Dut+12] B. Dutertre, A. Easwaran, B. Hall, and W. Steiner. "Model-based analysis of Timed-Triggered Ethernet". In: *Digital Avionics Systems Conference (DASC)*. 2012.

[Duz05] S. Duzellier. "Radiation effects on electronic devices in space". In: *Aerospace Science and Technology* 9.1 (Jan. 2005), pp. 93–99.

[EDN04] R. Edwards, C. Dyer, and E. Normand. "Technical standard for atmospheric radiation single event effects, (SEE) on avionics electronics". In: *IEEE Radiation Effects Data Workshop*. 2004.

[Elf18] D. Elftmann. *Xilinx On-Orbit Reconfigurable Kintex UltraScale FPGA Technology for Space*. 2018. URL: `https://indico.esa.int/event/232/contributions/2161/` (visited on 03/21/2019).

[ESA08] ESA Requirements and Standards Division ESTEC. *SpaceWire - links, nodes, routers and networks*. ECSS-E-ST-50-12C. European Space Agency. 2008.

[ESA19] ESA Requirements and Standards Division ESTEC. *SpaceWire - links, nodes, routers and networks*. ECSS-E-ST-50-12C Rev.1. Version Rev. 1. European Space Agency. 2019.

[EY19] M. Eslinger and P. Yeung. "Formal Bug Hunting with 'River Fishing' Techniques". In: *Design and Verification Conference and Exhibition (DVCon)*. 2019.

[Fle10] FlexRay Consortium. *FlexRay Communications System Protocol Specification*. 3.01. 2010.

[FLS15] A. Ferrara, P. Liberatore, and M. Schaerf. "The Size of BDDs and Other Data Structures in Temporal Logics Model Checking". In: *IEEE Transactions on Computers* 65.10 (Oct. 2015), pp. 3148–3156.

[Fos+07] H. Foster, L. Loh, B. Rabii, and V. Singhal. "Guidelines for creating a formal verification testplan". In: *Design and Verification Conference and Exhibition (DVCon)*. 2007.

[Fos18a] H. Foster. *Part 1 - FPGA Design Trends*. 2018. URL: https://blogs.mentor.com/verificationhorizons/blog/2018/11/19/part-1-the-2018-wilson-research-group-functional-verification-study/ (visited on 05/05/2019).

[Fos18b] H. Foster. *Part 10 - IC/ASIC Language and Library Adoption Trends*. 2018. URL: https://blogs.mentor.com/verificationhorizons/blog/2019/02/14/part-10-the-2018-wilson-research-group-functional-verification-study/ (visited on 05/05/2019).

[Fos18c] H. Foster. *Part 6 - FPGA Verification Language and Library Adoption Trends*. 2018. URL: https://blogs.mentor.com/verificationhorizons/blog/2019/01/15/part-6-the-2018-wilson-research-group-functional-verification-study/ (visited on 05/06/2019).

[Fos18d] H. Foster. *Part 7 - IC/ASIC Design Trends*. 2018. URL: https://blogs.mentor.com/verificationhorizons/blog/2019/01/22/part-7-the-2018-wilson-research-group-functional-verification-study/ (visited on 05/05/2019).

[Füh+01] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, M. Walther, and R. Bosch. *Time Triggered Communication on CAN*. Research Report. Robert Bosch GmbH, 2001.

[Gib+16] D. Gibson, S. Parkes, C. McClements, and S. Mills. "SpaceWire-D Prototype and Demonstration System". In: *International SpaceWire Conference*. Networks & Protocols. 2016.

[Gra19] Mentor Graphics. *Questa Sequential Logic Equivalence Check (SLEC) App*. 2019. URL: http://s3.mentor.com/public_documents/datasheet/products/fv/questa-slec-ds.pdf (visited on 09/26/2019).

[Hab+13] S. Habinc, A. Sakthivel, J. Ekergarn, A. Bjorkengren, R. Pender, S. Landstrom, F. Cordero, J. Mendes, T. M. Ho,

and K. Stohlmann. “MASCOT On-Board Computer Based on GR712RC”. In: *Data Systems in Aerospace (DASIA)*. 2013.

[Han06] A. Hanzlik. *A Case Study of Clock Synchronization in FlexRay*. Research Report 31/2006. Treitlstr. 1-3/182-1, 1040 Vienna, Austria: Technische Universität Wien, Institut für Technische Informatik, 2006.

[HD92] K. Hoyme and K. Driscoll. “SAFEbus”. In: *Digital Avionics Systems Conference*. 1992.

[IEE06] IEEE. *IEEE Standard for Verilog Hardware Description Language*. 1364-2005. IEEE standards association. Apr. 2006.

[IEE10] IEEE. *Property Specification Language (PSL)*. 1850-2010. Apr. 2010.

[IEE96] IEEE. *1355-1995 - IEEE Standard for Heterogeneous InterConnect (HIC), (Low-Cost, Low-Latency Scalable Serial Interconnect for Parallel System Construction)*. 1355-1995. June 1996.

[Int96] International Electrotechnical Commission International Organization for Standardization. *Information technology - Syntactic metalanguage - Extended BNF*. Test. Dec. 1996.

[IS08] IEEE Instrumentation and Measurement Society. *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. 1588-2008. IEEE standards association. 2008.

[KAH04] H. Kopetz, A. Ademaj, and A. Hanzlikm. “Integration of internal and external clock synchronization by the combination of clock-state and clock-rate correction in fault-tolerant distributed systems”. In: *International Real-Time Systems Symposium*. 2004.

[Kai+09] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodova, C. Taylor, V. Frolov, E. Reeber, and A. Naik. “Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation”. In: *In-*

ternational Conference on Computer Aided Verification. 2009, pp. 414–429.

[KB03] H. Kopetz and G. Bauer. "The time-triggered architecture". In: *Proceedings of the IEEE* 91.1 (2003), pp. 112–126.

[KB06] F. Kesel and Bartholomä. *Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs*. Oldenbourg Wissenschaftsverlag, 2006. ISBN: 978-3486575569.

[KCR06] F. L. Kastensmidt, L. Carro, and R. Reis. *Fault-Tolerance Techniques for SRAM-Based FPGAs*. 1st ed. Springer, 2006. ISBN: 978-0387310688.

[KHE00] H. Kopetz, M. Holzmann, and W. Elmenreich. "A universal smart transducer interface: TTP/A". In: *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. 2000.

[KO87] H. Kopetz and W. Ochsenreiter. "Clock Synchronization in Distributed Real-Time Systems". In: *IEEE Transactions on Computers* C-36.8 (1987), pp. 933–940.

[Kog+97] R. Koga, S. H. Penzin, K. B. Crawford, and W. R. Crain. "Single event functional interrupt (SEFI) sensitivity in microcircuits". In: *European Conference on Radiation and its Effects on Components and Systems (RADECS)*. 1997.

[Kop03] H. Kopetz. "Fault containment and error detection in the time-triggered architecture". In: *International Symposium on Autonomous Decentralized Systems*. 2003.

[Kop11] H. Kopetz. *Real-Time Systems. Design Principles for Distributed Embedded Applications*. Springer, 2011. ISBN: 978-1-4419-8237-7.

[Kot+18] S. Kottmeier, C. F. Hobbie, F. Orlowski-Feldhusen, F. Nohka, T. Delovski, G. Morfill, L. Grillmayer, C. Philpot, and H. Müller. "The Eu:Cropis Assembly, Integration and Verification Cam-

paigns: Building the first DLR Compact Satellite". In: *International Astronautical Congress IAC*. 2018.

[KR07] I. Kuon and J. Rose. "Measuring the Gap Between FPGAs and ASICs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (Feb. 2007), pp. 203–215.

[KTR08] I. Kuon, R. Tessier, and J. Rose. "FPGA Architecture: Survey and Challenges". In: *Foundations and Trends in Electronic Design Automation* 2.2 (2008), pp. 135–253.

[Lam78] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565.

[LEG12] P. Loschmidt, R. Exel, and G. Gaderer. "Highly Accurate Timestamping for Ethernet-Based Clock Synchronization". In: *Journal of Computer Networks and Communications* (Jan. 2012). DOI: 10.1155/2012/152071.

[Lis91] B. Liskov. "Practical uses of synchronized clocks in distributed systems". In: *ACM symposium on Principles of distributed computing*. 1991.

[Lit13] M. Litterick. "SVA Encapsulation in UVM". In: *Design and Verification Conference and Exhibition (DVCon)*. 2013.

[Lit14] M. Litterick. "Advanced UVM Register Modeling - There's More Than One Way to Skin A Reg". In: *Design and Verification Conference and Exhibition (DVCon)*. 2014.

[LL88] J. Lundelius Welch and N. Lynch. "A New Fault-Tolerant Algorithm for Clock Synchronization". In: *Information and Computation* (1988), pp. 1–36.

[LM85] L. Lamport and P. M. Melliar-Smith. "Synchronizing Clocks in the Presence of Faults". In: *Journal of the Association for Computing Machinery* 32.1 (Jan. 1985), pp. 52–78.

[Loe99] H. Loenn. "Initial synchronization of TDMA communication in distributed real-time systems". In: *IEEE International Conference on Distributed Computing Systems*. 1999.

[LP97] H. Loenn and P. Pettersson. "Formal verification of a TDMA protocol start-up mechanism". In: *Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems*. 1997.

[Lüd+14] D. Lüdtke, K. Westerdorff, K. Stohlmann, A. Börner, O. Maibaum, Ting Peng, B. Weps, G. Fey, and A. Gerndt. "OBC-NG: Towards a reconfigurable on-board computing architecture for spacecraft". In: *Aerospace Conference, 2014 IEEE*. Mar. 2014. DOI: 10.1109/AERO.2014.6836179.

[Man+08] A. Manuzzato, S. Gerardin, A. Paccagnella, L. Sterpone, and M. Violante. "On the Static Cross Section of SRAM-Based FPGAs". In: *IEEE Radiation Effects Data Workshop*. 2008.

[Mau+08] R. Maurer, M. Fraeman, M. Martin, and D. Roth. "Harsh Environments: Space Radiation Environment, Effects, and Mitigation". In: *Johns Hopkins APL Technical Digest* 28.1 (Jan. 2008).

[MBS18] A. K. V. Madhunapantula, A. A. Bharadwaj, and B. S. Singanamalli. "An Efficient and Modular Approach for Formally Verifying Cache Implementations". In: *Design and Verification Conference and Exhibition (DVCon)*. 2018.

[McG+19] W. F. McGrew, X. Zhang, H. Leopardi, R. J. Fasano, D. Nicolodi, K. Beloy, J. Yao, J. A. Sherman, S. A. Schäffer, J. Savory, R. C. Brown, S. Römisch, C. W. Oates, T. E. Parker, T. M. Fortier, and A. D. Ludlow. "Towards the optical second: verifying optical clocks at the SI limit". In: *Optica* 6.4 (Apr. 2019), pp. 448–454.

[Meh18] A. B. Mehta. *ASIC/SoC Functional Design Verification. A Comprehensive Guide to Technologies and Methodologies*. Springer, 2018. ISBN: 978-3-319-59417-0.

[Mic+16] W. Mich, K. Romanowski, P. Tyczka, R. Renk, V. D. Kollias, and N. Pogkas. "Implementation and validation of the SpaceWire-R protocol. SpaceWire networks and protocols, short paper". In: *International SpaceWire Conference*. 2016.

[Mic15] Microsemi. *ProASIC3L Low Power Flash FPGAs*. DS0100. June 2015.

[Mil91] D. L. Mills. "Internet Time Synchronization: The Network Time Protocol". In: *IEEE Transactions on Communications* 39.10 (Oct. 1991), pp. 1482–1493.

[MM04] P. Molitor and J. Mohnke. *Equivalence Checking of Digital Circuits: Fundamentals, Principles, Methods*. Springer US, 2004. ISBN: 978-1-4020-7725-8.

[MS05] M. N. Mneimneh and K. A. Sakallah. "Principles of sequential-equivalence verification". In: *IEEE Design & Test of Computers* 22.3 (May 2005), pp. 248–257.

[MS09] G. Martin and G. Smith. "High-Level Synthesis: Past, Present, and Future". In: *IEEE Design & Test of Computers* 26.4 (Aug. 2009), pp. 18–25.

[Nan+16] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. "A Survey and Evaluation of FPGA High-Level Synthesis Tools". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (Oct. 2016), pp. 1591–1604.

[NM93] L. M. Ni and P. K. McKinley. "A survey of wormhole routing techniques in direct networks". In: *Computer* 26.2 (Feb. 1993), pp. 62–76.

[Obe11] R. Obermaisser. *Time-Triggered Communication*. CRC Press Taylor & Francis Group, 2011. ISBN: 978-1-4398-4661-2.

[Pen+16] T. Peng, B. Weps, K. Höflinger, K. Borchers, D. Lüdtke, and A. Gerndt. "A New SpaceWire Protocol for Reconfigurable Distributed On-Board Computers". In: *International SpaceWire Conference*. 2016.

[PMM15] V. Pangracious, Z. Marrakchi, and H. Mehrez. *Three-Dimensional Design Methodologies for Tree-based FPGA Architecture*. Springer, 2015. ISBN: 978-3-319-19173-7.

[Pnu77] A. Pnueli. "The Temporal Logic of Programs". In: *Annual Symposium on Foundations of Computer Science*. 1977, pp. 46–57.

[PR09] L. Piga and S. Rigo. "Comparing RTL and high-level synthesis methodologies in the design of a theora video decoder IP core". In: *Southern Conference on Programmable Logic (SPL)*. 2009.

[RA16] Z. Ren and H. Al-Asaad. "Overview of Assertion-Based Verification and its Applications". In: *International Conference on Embedded Systems, Cyber-physical Systems, and Applications*. 2016.

[Rau07] M. Rausch. *FlexRay - Grundlagen, Funktionsweise, Anwendung*. Carl Hanser Verlag GmbH & Co. KG, 2007. ISBN: 978-3-446-41249-1.

[RE10] ESA Requirements and Standards Division ESTEC. *SpaceWire protocol identification*. ECSS-E-ST-50-51C. European Space Agency. Feb. 2010.

[Rom+16] K. Romanowski, P. Tyczka, W. Holubowicz, R. Renk, V. D. Kollias, N. Pogkas, and D. Jameux. "SpaceWire network management using network discovery and configuration protocol. SpaceWire networks and protocols, short paper". In: *International SpaceWire Conference*. 2016.

[Sak+14] A. Sakthivel, J. Ekergarn, D. Hellstrom, S. Habinc, and M. Suess. "Spacewire time distribution protocol implementation and results". In: *International SpaceWire Conference*. 2014, pp. 19–25.

[Sal+16] L. Salvy, A. Varotsou, A. Samaras, B. Vandevelde, L. Gouyet, A. Rousset, L. Azema, C. Sarrau, N. Chatry, and M. Poizat. "Total ionizing dose influence on the single event effect sensitivity of active EEE components". In: *European Conference on Radiation and Its Effects on Components and Systems (RADECS)*. 2016.

[Sch86] F. B. Schneider. *A Paradigm for Reliable Clock Synchronization*. Technical Report. Ithaca, NY, USA: Cornell University, 1986.

[SD11] W. Steiner and B. Dutertre. "Automated Formal Verification of the TTEthernet Synchronization Quality". In: *NASA Formal Methods Conference (NFM 2011)*. 2011.

[SFL14] K. Stohlmann, G. Fey, and D. Lüdtke. "Automatic Performance Tracking of a SpaceWire Network". In: *International SpaceWire Conference*. 2014.

[SG17] IEEE Computer Society and IEEE Standards Association Corporate Advisory Group. *IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language*. 1800-2017. IEEE standards association. 2017.

[Sim15] P. A. Simpson. *FPGA Design. Best Practices for Team-based Reuse*. 2nd ed. Springer, 2015. ISBN: 978-3-319-17923-0.

[SK06] W. Steiner and H. Kopetz. "The Startup Problem in Fault-Tolerant Time-Triggered Communication". In: *International Conference on Dependable Systems and Networks (DSN'06)*. 2006.

[Soc08] IEEE Computer Society. *IEEE Standard VHDL Language Reference Manual*. 1076-2008. IEEE standards association. 2008.

[Soc17] IEEE Computer Society. *IEEE Standard for Universal Verification Methodology Language Reference Manual*. 1800.2-2017. IEEE standards association. 2017.

[SP02] W. Steiner and M. Paulitsch. "The transition from asynchronous to synchronous system operation: an approach for distributed fault-tolerant systems". In: *International Conference on Distributed Computing Systems*. 2002.

[SRR16] I. Saha, S. Roy, and S. Ramesh. "Formal Verification of Fault-Tolerant Startup Algorithms for Time-Triggered Architectures: A Survey". In: *Proceedings of the IEEE* 104.5 (2016), pp. 904–922.

[SSK15] E. Seligman, T. Schubert, and M V A. K. Kumar. *Formal Verification. An Essential Toolkit for Modern VLSI Design*. Morgan Kaufmann, 2015. ISBN: 978-0128007273.

[ST12] C. Spear and G. Tumbush. *SystemVerilog for Verification*. 3rd ed. Springer, 2012. ISBN: 978-1-4614-0714-0.

[Ste+06] K. Steinhammer, P. Grillinger, A. Ademaj, and H. Kopetz. "A Time-Triggered Ethernet (TTE) Switch". In: *Proceedings of the Design Automation & Test in Europe Conference*. 2006.

[Ste+09] W. Steiner, G. Bauer, B. Hall, M. Paulitsch, and S. Varadarajan. "TTEthernet Dataflow Concept". In: *IEEE International Symposium on Network Computing and Applications*. 2009.

[Ste09] W. Steiner. "TTEthernet: Time-triggered services for Ethernet networks". In: *Digital Avionics Systems Conference*. 2009.

[TIM16] F. Torelli, J. Ilstad, and G. Magistrati. "JUICE time distribution protocol. SpaceWire networks and protocols, short paper". In: *International SpaceWire Conference*. 2016.

[Tre+14] C. Treudler, J.-C. Schröder, F. Greif, K. Stohlmann, G. Aydos, and G. Fey. "Scalability of a base level design for an on-board-computer for scientific missions". In: *Data Systems In Aerospace (DASIA)*. Jan. 2014.

[Tre+18] C. Treudler, H. Benninghof, K. Borchers, B. Brunner, J. Cremer, M. Dumke, T. Gärtner, K. Höflinger, J. Langwald, D. Lüdtke, T. Peng, E. Risse, K. Schwenk, M. Stelzer, M. Ulmer, S. Vellas, and K. Westerdorff. "ScOSA - Scalable On-Board Computing for Space Avionics". In: *International Astronautical Congress*. 2018.

[TTT02] TTTech. *Time-Triggered Protocol TTP/C High-Level Specification Document*. D-032-S-10-028. 2002.

[TTT11] TTTech. *Time-Triggered Ethernet*. AS6802. SAE Aerospace. 2011.

[Urb+13] C. Urbina-Ortega, G. Furano, G. Magistrati, K. Marinis, A. Menicucci, and D. Merodio-Codinachs. "Flash-based FPGAs in space, design guidelines and trade-off for critical applications". In: *European Conference on Radiation and Its Effects on Components and Systems (RADECS)*. 2013.

[Vec17] Vectron. *OS-68338 Hi-Rel Clock Oscillator Series*. 2017.

[Wan+15] G. Wang, H. Lam, A. George, and G. Edwards. "Performance and productivity evaluation of hybrid-threading HLS versus HDLs". In: *IEEE High Performance Extreme Computing Conference (HPEC)*. 2015.

[WG18] C. Wilson and A. George. "CSP Hybrid Space Computing". In: *Journal of Aerospace Information Systems* (2018), pp. 215–227.

[Xil18a] Xilinx. *Design Flows Overview*. UG892. Dec. 2018. URL: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug892-vivado-design-flows-overview.pdf`.

[Xil18b] Xilinx. *High-Level Synthesis*. UG902. Dec. 2018. URL: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf`.

[Y+00] Abarbanel Y., Beer I., Gluhovsky L., Keidar S., and Wolfsthal Y. "FoCs - Automatic Generation of Simulation Checkers from Formal Specifications". In: *International Conference on Computer Aided Verification*. 2000.

Own Publications

[BMD19] K. Borchers, S. Montenegro, and F. Dannemann. “Volatile Register Handling for FPGA Verification Based on SVAs Incorporated into UVM Environments”. In: *IEEE Aerospace Conference*. Mar. 2019.

[Bor+18] K. Borchers, G. Fey, D. Lüdtke, and S. Montenegro. “Time-Triggered Data Transfers over SpaceWire for Distributed Systems”. In: *IEEE Aerospace Conference*. Mar. 2018.

[Bor+19] K. Borchers, D. Lüdtke, S. Montenegro, and F. Dannemann. “Performance and Utilization Results for Time-Triggered Data Transfers over SpaceWire”. In: *IEEE Aerospace Conference*. Mar. 2019.

[Hab+13] S. Habinc, A. Sakthivel, J. Ekergarn, A. Bjorkengren, R. Pender, S. Landstrom, F. Cordero, J. Mendes, T. M. Ho, and K. Stohlmann. “MASCOT On-Board Computer Based on GR712RC”. In: *Data Systems in Aerospace (DASIA)*. 2013.

[Lüd+14] D. Lüdtke, K. Westerdorff, K. Stohlmann, A. Börner, O. Maibaum, Ting Peng, B. Weps, G. Fey, and A. Gerndt. “OBC-NG: Towards a reconfigurable on-board computing architecture for spacecraft”. In: *Aerospace Conference, 2014 IEEE*. Mar. 2014. DOI: 10.1109/AERO.2014.6836179.

[Pen+16] T. Peng, B. Weps, K. Höflinger, K. Borchers, D. Lüdtke, and A. Gerndt. “A New SpaceWire Protocol for Reconfigurable Dis-

tributed On-Board Computers". In: *International SpaceWire Conference*. 2016.

[SFL14] K. Stohlmann, G. Fey, and D. Lüdtke. "Automatic Performance Tracking of a SpaceWire Network". In: *International SpaceWire Conference*. 2014.

[Tre+14] C. Treudler, J.-C. Schröder, F. Greif, K. Stohlmann, G. Aydos, and G. Fey. "Scalability of a base level design for an on-board-computer for scientific missions". In: *Data Systems In Aerospace (DASIA)*. Jan. 2014.

[Tre+18] C. Treudler, H. Benninghof, K. Borchers, B. Brunner, J. Cremer, M. Dumke, T. Gärtner, K. Höflinger, J. Langwald, D. Lüdtke, T. Peng, E. Risse, K. Schwenk, M. Stelzer, M. Ulmer, S. Vellas, and K. Westerdorff. "ScOSA - Scalable On-Board Computing for Space Avionics". In: *International Astronautical Congress*. 2018.

Typeset February 10, 2020