INAUGURAL - DISSERTATION
zur
Erlangung der Doktorwürde
der
Naturwissenschaftlich-Mathematischen Gesamtfakultät
der
Rupert - Karls - Universität
Heidelberg

vorgelegt von
Raul Schmidlin Fajardo Silva
aus Joinville, Brasilien

Tag der mündlichen Prüfung:

Contract Testing for Reliable Embedded Systems

Betreuer: Prof. Dr. Reinhard Männer
Prof. Dr. Felix Freiling

Ich versichere, dass ich diese Doktor-Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Abgabedatum:

Acknowledgments

The experience of working on this project was rewarding and enriching. Surprisingly for me, I have learned about relations and communication as much as I have learned about my subject. Nowadays, I feel compelled to be objective, analyze requirements, cost and risk, communicate effectively and abstract. Thanks to many discussions with people from different specific areas of the ECOMODIS project, I learned a lot.

For the possibility of taking part of this work, I want to thank Dr. Phuc Nguyen and Prof. Zwick who recommended me for this project. For believing in my potential and supporting me, I am very thankful to Prof. Jürgen Hesser and Prof. Reinhard Männer.

In the first part of my work, I worked in the group of Prof. Jürgen Hesser at the Institute for Computational Medicine of the University of Mannheim. My first colleagues, Dmitry Maksimov, Dr. Amel Guetat and Dzmitry Stsepankou, helped me better understand research and gave me support when I was less motivated. In the ECOMODIS project, especially talks with Dr. Giovanni Falcone, Monika Jusasz and Dr. Meike Jipp have been a relief in our stressful schedule of reports and workshops.

For my first research successes, I want to thank Prof. Reinhard Männer for his very helpful feedbacks on papers and Prof. Jürgen Hesser for his belief in my results and determination to motivate me. Also, positive feedback and discussions with the OpenCores community have motivated me. Besides my personal project on the platform, I had the privilege of discussing my research work within the community. For this possibility and the interest of the community, I am very thankful.

After I had my first steps in research, I was ready to start a new project that would complement the first one. By now, the Institute of Computer Engineering of the University of Mannheim had been incorporated into the University of Heidelberg. Under the supervision of Prof. Reinhard Männer in the Department for Application Specific Computing, I developed the concept for driver development enclosing architecture, systematization and contracts that allowed me to submit this PhD thesis to the Faculties of Natural Science and Mathematics. Throughout, Dr. Guillermo Marcus has constantly supported this work, and me as a friend. Thanks to Prof. Reinhard Männer, we could refine our ideas to the concept of a complete project, and the feedback of Prof. Felix Freiling steered the project in the right direction.

In the last stages of this project, I was given the chance by Dr. Simon Moore to give a talk at the Computer Architecture Group of the University of Cambridge, Computer Laboratory, thanks to Dr. Jeremy Bennett. This talk provided me with great feedback for the continuation of the project. Besides, it has been an incredible opportunity to meet Dr. Jeremy Bennett and Julius Baxter personally and have great discussions. I want to thank them both for their ideas and interest.

I also want to thank Andreas Heck for his work on an evaluation platform for hardware contracts and my friends and colleagues, Benny Bürger, Pavel Krasnopevtsev, Dzmitry Hlindzich and Dmitry Maksimov, for the uncountable lunches, discussions, excursions and fun.

Finally, I want to give my special thanks to my family for this possibility, their unconditional support, despite the distance, and understanding for my problems. And I want to thank all my friends who have enriched the time of this challenge with love and joy.

Zusammenfassung

Eingebettete Systeme umfassen unterschiedliche Technologien, was ihren Entwurf erschwert. Durch Erstellen eines virtuellen Prototyps vom Zielsystem mittels Electronic System Level (ESL), wird die frühzeitige Analyse von einem System aus Software und Elektronik ermöglicht. Jedoch wird das konkrete Zusammenspiel zwischen Hardwaremodulen beziehungsweise zwischen Hardware und Software erst in den späteren Entwicklungsphasen und in der Prototyperstellung deutlich. Selbst in diesen Entwicklungsphasen liegt das Hauptaugenmerk auf der Systemfunktionalität. Dabei rückt die Sicherstellung des richtigen Zusammenspiels in den Hintergrund, obwohl das für die Systemfunktionalität erforderlich ist.

Während einzelne Komponenten ausführlich getestet werden, wodurch eine gewisse Zuverlässigkeit eingehalten wird, werden ihre Schnittstellen nicht ausführlich genug spezifiziert. Folglich wird weder die korrekte Interaktion zwischen Komponenten noch zwischen einer Komponente und ihrer Umgebung bzw. ihrem Benutzer gewährleistet. Im Normalfall werden sie nur funktional getestet. Das lässt Defekte länger unentdeckt, was höhere Kosten verursacht, denn die Fehlerbeseitigung ist umso teuerer, je später Fehler entdeckt werden. Deshalb zielt diese Arbeit darauf ab, eine richtige Komponenteninteraktion durch die Spezifikation von Schnittstellen, die Testgenerierung und die Testausführung in Echtzeit zu gewährleisten. Die Spezifikation basiert auf dem Design-by-Contract-Ansatz von Software, der die Semantik der Komponenteninteraktion spezifiziert.

Im ersten Teil dieser Arbeit wird eine Spezifikation für das Zusammenspiel zwischen Hardwaremodulen präsentiert. Mit der automatischen Testausführung in Echtzeit können die Vorbedingungen für den richtigen Komponentenbetrieb überprüft werden. Im komponentenbasierten Design werden die Komponenten als fehlerfrei betrachtet. Dennoch weist die Ausführung der Funktionalität Verhaltenseigenschaften auf, durch die Nachbedingungen definiert werden, z.B. zeitversetzte Ergebnislieferung. In einer korrekten Komponentenzusammenstellung ergibt sich ein betriebsfähiges System aus Komponenten, deren Verhalten die Vorbedingungen anderer Komponente einhalten.

Die Spezifikation von Vorbedingungen folgt der Definition von Umgebungseigenschaften, zulässigen Eingangsreihenfolgen für Schnittstellenpins sowie zulässigen Signalparametern, wie Spannungspegeln, Flanken, Verzögerungen und Glitches. Diese Parameter werden während des Betriebs von einer Testschaltung ermittelt. Befinden sie sich außerhalb der definierten Grenzen, werden sie als

fehlerhaft markiert, wodurch ein möglicher Ausfall erkannt wird. Anhand des Beispieles eines Inter-Integrated Circuit (I^2C) Kommunikationssystems wird sein *Contract* definiert und die Parallele zwischen den Verstößen gegen den *Contract*, der Defektkategorisierung und dem Ausfall aufgrund von Fehlereinspeisungen gezeigt.

Um die Arbeit an Hardware-*Contracts* zu vervollständigen, wird die Fehleranalyse mit dem ESL-Design ermöglicht. Die Datenübertragungen zwischen den ESL High-Level-Modellen werden mit den definierten *Contract*-Parametern erweitert. Somit können anhand einer spezifischen Schnittstelle digitale Fehler für Übertragungen mit den vom *Contract* abweichenden Signalparametern generiert werden. Diese Fehler können dann bei der Simulation des ESL-Designs zurückverfolgt werden. Auf diese Weise werden fehlerkorrigierende Maßnahmen für eine synchrone Kommunikation vorgeschlagen und ausgewertet.

Im zweiten Teil der Arbeit wird das Zusammenspiel zwischen Hardware und Software durch spezielle Methoden zur Treiberentwicklung angegangen. Nicht nur wird die Schnittstelle dazwischen spezifiziert, sondern es werden auch die Steuerelemente der Hardware in der Software abgebildet, so dass eine Softwareschnittstelle zum Gerät partiell generiert werden kann. Das ist notwendig, denn die Treiber handhaben Geräte durch Steuerelemente, wie Register, Datenströme und Interrupts, die in der Software nicht vorhanden sind.

Der systematische Aufbau von Treibern vereinfacht die Entwicklung einer Geräteschnittstelle namens *Device Mechanism*. Sie ist die untere Schicht einer Zweischicht-Architektur für die Treiberentwicklung. Der *Device Mechanism* bietet eine reine Softwareschnittstelle zum Gerät an. Im Gegensatz zu einem kompletten Treiber nimmt er nur die Geräteansteuerung vor, ohne selbst die Daten zu verarbeiten. Indem der *Device Mechanism* die Funktionalität mit der Geräteimplementierung teilt, ist er vollständig spezifiziert. Darauf baut eine weitere Schicht namens *Driver Policy* auf. Diese vervollständigt die Datenverarbeitung indem sie den Betriebssystemanforderungen genügt.

Auf der Basis des *Device Mechanisms* werden *Contracts* für die Spezifikation der Schnittstellen eingesetzt. Dafür wird das dynamische Verhalten des Gerätes durch eine erweiterte Zustandsmaschine modelliert. Darauf basierend können die Funktionen des *Device Mechanisms* um Vorbedingungen an den Zustand oder an die Variablen der Zustandsmaschine erweitert werden, die während der Laufzeit überprüft werden. Nach Ausführung einer Funktion wird für die Einhaltung ihrer Nachbedingungen gesorgt. So kann die Einhaltung der Rahmenbedingungen bei der Benutzung von verschiedenen *Driver Policies*, für Betriebssysteme oder Firmwares, gewährleistet werden, die sich einen universellen *Device Mechanism* teilen können. Nach diesem Prinzip wird ein Linux-Treiber für eine Philips Webcam entwickelt. Das Konzept kann Treiberfehler vermeiden, die auf falsche Interpretationen von Gerätedaten oder nicht vorgesehene Bedienungsabläufe zurückzuführen sind.

Abstract

Embedded systems comprise diverse technologies complicating their design. By creating virtual prototypes of the target system, Electronic System Level Design, the early analysis of a system composed by electronics and software is possible. However, the concrete interaction between hardware modules and between hardware and software is left for late development stages and real prototype making. Generally, interaction between components is assumed to be correct. However, it has to be assumed on development implicitly because interaction between components is not considered in the functionality design.

While single components are mostly thoroughly tested and guarantee certain reliability levels, their interaction is based on often underspecified interfaces. Although component usage is mostly specified, operational constraints are often left out. Finally, not only the interaction between components but also with the environment and the user are not ensured. Generally, only functional integration tests are executed and corner-cases are left out, leaving uncovered faults that only manifest as failures later when their cost is higher. Therefore, this work aims at component interaction through specification of interfaces, test generation and real-time test execution. The specification is based on the design-by-contract approach of software that specifies semantics of component interaction in addition to the syntactical definition through functions.

In the first part of this work, a specification for the interaction between hardware modules is given. With the automatic real-time test execution, fulfillment of specified preconditions for correct component operation can be checked. In component-based design, the component is trusted and thus, its functionality is assumed to be correct when certain postconditions are specified. In a correct component assembly, component postconditions fulfill preconditions of other components resulting in an operational system.

The specification of preconditions follows the definition of environmental properties, acceptable input sequences for interfacing pins, as well as acceptable signal parameters, such as voltage levels, slope times, delays and glitches. Postconditions are defined by the description of a functionality accompanying constraints, such as timing. These parameters are automatically determined on operation by a testing circuit. Parameters that violate the specification are signaled by the testing circuit and failure is detected. The chosen parameters can give hint of the reason for the failure being an evidence of a circuit fault. In the example of an Inter-Integrated Circuit (I^2C) communication system, we define contracts

and show comparisons between contract violation, fault categorization and failure occurrence under signal fault injection.

To complete this work, support for fault analysis on the electronic system level design is given. For this, the data transfers between the high-level models used in the design are augmented with the defined contract parameters. With a specific interface, digital faults are generated for transactions with violating signal parameters that can be tracked by the system. This way, recovery mechanisms for synchronous communication are proposed and tested.

In the second part, the interaction between hardware and software is tackled providing special methods for developing device drivers. For this, we do not only specify the interface between hardware and software but also map the hardware control elements to software, partially generating the software interface for a device. This is necessary because drivers handle devices with internal control elements like registers, data streams and interrupts that cannot be represented on software.

This systematic composition of drivers facilitates the development of a device interface called the device mechanism. It is the lowest layer of a two-layer architecture for driver development. The device mechanism carries out the access to the device exporting a pure software interface. This interface is based on the device implementation being, thus, fully specified. Further data processing required for compliance with the operating system or application is carried out in the driver policy, the layer on top of it.

With the definition of a software layer for device control, contracts specifying constraints of this interface are proposed. These contracts are based on implementation constraints of the device and on its dynamic behavior. Therefore, an extended finite state machine models the dynamic behavior of the device. Based on it, functions of the device mechanism can be augmented with preconditions on the state or on state machine variables. These conditions are then checked on runtime. After execution of a function, its postconditions are ensured, such as timing. This guarantees that different driver policies, operating systems or firmwares, use this same device mechanism fulfilling its constraints. On the example of a Philips webcam, we develop the complete driver for Linux based on our architecture, creating contracts for its device mechanism. Following the systematic composition and the contract approach, driver bugs are avoided that otherwise violate allowed values for device data and execution orders of device protocols.

Related Publications

[1] Raul Schmidlin Fajardo Silva and Guillermo Marcus. Device Mechanism: A Structured Device Driver Development Approach. In *Proceedings of the 14th IEEE International High Assurance Systems Engineering Symposium (HASE)*, pages 66–73. IEEE Computer Society, 2012.

[2] Raul Schmidlin Fajardo Silva, Jürgen Hesser, and Reinhard Männer. Contract Specification for Hardware Interoperability Testing and Fault Analysis. *Reliability, IEEE Transactions on*, 60(1):351–362, 2011.

[3] Raul Schmidlin Fajardo Silva, Jürgen Hesser, and Reinhard Männer. Fault Propagation Analysis on the Transaction-Level Model of an Acquisition System with Bus Fallback Modes. In *Proceedings of the Workshop on the Design of Dependable Critical Systems (DDCS)*, page 36. University Heidelberg, 2009.

Contents

List of Figures

List of Tables

ACRONYMS

ADC	Analog-to-Digital Converter
API	Application Programming Interface
BGI	Byte-Granularity Isolation
CAD	Computer Aided Design
CFSM	Co-design Finite State Machine
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
D	Data
DAC	Digital-to-Analog Converter
DC	Direct Current
DDT	Device Driver Testing
DMA	Direct Memory Access
DOM	Document Object Model
DRC	Design Rule Checking
EDA	Electronic Design Automation
EEPROM	Electrically Erasable Programmable Read-Only Memory
EFSM	Extended Finite State Machine
ESL	Electronic System Level
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit

GUI	Graphical User Interface
HDL	Hardware Description Language
HW	Hardware
I²C	Inter-Integrated Circuit
IC	Integrated Circuit
IDL	Interface Description Language
I/O	Input/Output
IOMMU	Input/Output Memory Management Unit
IEEE	Institute of Electrical and Electronics Engineers
IP	Intellectual Property
ISA	Industry Standard Architecture
LAN	Local Area Network
LED	Light-Emitting Diode
LOC	Lines of Code
LTL	Linear Temporal Logic
LUT	Look-up Table
MMU	Memory Management Unit
NAND	Negated AND
NMM	Network-Integrated Multimedia Middleware
NULL	Null Pointer
OpenCL	Open Computing Language
OS	Operating System
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
POSIX	Portable Operating System Interface
PWC	Linux driver for Philips Webcams

R	Reset
RGB	Red, Green, and Blue color model
RTL	Register Transfer Level
S	Set
SAX	Simple API for XML
SCL	Clock Line
SDA	Data Line
SoC	System on Chip
SPI	Serial Peripheral Interface Bus
SR	Set-Reset
SW	Software
TCP	Transmission Control Protocol
TLM	Transaction Level Model
TTL	Transistor-Transitor Logic
UML	Unified Modeling Language
USB	Universal Serial Bus
VHDL	VHSIC hardware description language
WHQL	Windows Hardware Quality Labs
XML	Extensible Markup Language

1 Introduction

In this thesis, the reliability of development methods for embedded systems is analyzed and methods are proposed to improve it. Therefore, the component interaction is targeted as fault source, and requirements and tasks of components are accurately specified resulting in automatic compliance checks. In this chapter, we define the context of embedded systems' development and reliability, our motivations and summarize the scientific contributions.

1.1 Context & Motivation

Current embedded systems comprise mechanical, electronic and software technologies. While this characteristic enables these systems to execute diverse tasks, it also makes them complex [BG06]. Each technology has specialized tools to analyze functionality and quality factors, mostly through simulation. But these tools are not compatible. Hence, the early analysis of the complete system is not possible before prototype making. At this stage, faults are already more expensive requiring redesign.

The Electronic System Level (ESL) design tackles the combined electronics and software design [BMP07]. The design result is a virtual system prototype that models the functional behavior of the target system. It is an executable abstract system description. With the inclusion of the respective visibility details, such as registers and interrupts, it also allows the execution of software on it. Although the system design enables an early software implementation, the hardware implementation still has to occur mostly manually through component selection and Hardware Description Language (HDL) development. Furthermore, interfaces between hardware and software are not designed and the design of hardware interfaces leaves their physical behavior and constraints undefined. The concrete behavior of Central Processing Units (CPUs), peripheral buses, device drivers and operating systems is not available in the unified design. Hence, the design and implementation of these interfaces are critical especially because they are still only evaluated in late design stages.

Also, in every design strategy, interaction of components is assumed to be correct, at most functional tests are performed. In hardware, most part of systems are assembled from established components that undergo many tests ensuring certain reliability levels. However, reliable components do not guarantee a reliable system. The diverse components of a system have to interact correctly. Not

only the sum of every component functionality has to achieve system's goal, but also operational constraints of every component have to be met. Therefore, integration testing is crucial in systems' design. Device drivers, responsible for the interaction between software and hardware, account for up to 70% of operating system failures [GGP06], [Mur04] and the major cause for these failures is an incorrect interaction between driver and device [RCKH09].

In software engineering, component-based design tries to foster component reuse for cost reduction. The literature explains that software is normally not designed with reuse in mind and its reuse holds complications [Gro04]. In particular, missing specification of operational conditions and interface usage, such as function call order, leads to incorrect component interaction and missing system's goal.

Also when usage constraints of components are documented, their assurance remains an implicit part of the design. Operational corner-cases and constraint dependencies are unclear keeping design flaws uncovered throughout design stages. A consistent specification of component interaction and constraints is missing as well as tools that enable their verification, both in real-time and in simulation.

The design-by-contract approach of software proposes a solution to this problem [Mey92]. Its purpose is to specify component interaction explicitly in the form of a contract. Design-by-contract envisions a separation of constraints from functionality programming. It is a systematization of defensive programming, where constraints are checked before function execution, preconditions; and constraints related to the very function are checked after function execution, postconditions.

This thesis makes use of the contract approach to propose a paradigm switch for constraint compliance. Instead of enclosing defensive programming code or tolerance mechanisms deliberately, we support constraint declaration and automatic compliance check through test generation and online test execution.

The strategy is applied to two different cases, the interaction between hardware modules and the interaction between hardware and software. In the former, the interaction between hardware modules is defined through parameters that are tested by a test circuit. For the interaction between hardware and software, an appropriate description of the hardware internals is proposed enabling a seamless access to hardware from the software. Based on it, constraints in the hardware software interaction are defined whose compliance is tested in real-time.

1.2 Hardware Contracts

Hardware has commonly embedded self tests to ensure component functionality. For example, production components include test circuits that can be used to debug them [MRC04], [ASE04], [SLA97]. Although faulty interoperability or external or environmental reasons are responsible for failure, component integration is not ensured under system operation. Although bus monitoring techniques

enable test of components' interaction [PHZ+05], [ARSH05], specification of interaction and corresponding specific testing is not given.

Properties of electronics used to define contracts as component interaction and constraints have been first envisioned by Bunse et al. [BG06]. Kamkin has used extended finite state machines to define the timed behavior of pipelined designs [Kam07] automatically generating functional tests for a model of a microprocessor [Kam08]. However, Kamkin [Kam07] does not address the problem of component interplay and Bunse et al. [BG06] only cover specification.

We specify contracts for hardware interaction also developing the corresponding test circuitry. In component-based design, the component is trusted given that its preconditions are fulfilled, such as compliance with interaction protocol and environmental conditions. Therefore, components are considered black-boxes. In contrast to software, hardware has to care about communication protocols, its signals and especially timing.

Through the usage of bus monitors and categorization circuits for signal properties, levels for signal properties can be defined that allow a reliable communication. Together with the analysis of environmental constraints, component preconditions can be ensured. On the violation of preconditions, component failure can be detected and located.

Compatibility with electronic system level design methodology is provided by extension of the transaction level model to incorporate our defined signal properties for reliable interaction. This way, the propagation of signal faults can be analyzed. In a case study for a synchronous communication protocol, mechanisms to tolerate and recover from these faults are proposed.

1.3 Device Contracts for Drivers

While the contract testing approach allows for a systematic way of testing to ensure operational reliability of the system, in the case of the hardware-software interface, the black-box view of contracts is pointless. The reason lies in the reliance of the software implementation on the hardware's internal structure and behavior, and thus, not only on some specified interface. Therefore, we reinvent driver development towards the definition of a clean device interface which allows for contract testing in return.

Driver synthesis approaches try to overcome the problematic by creating a description for device control structures enabling their access. These are described in a custom language enabling the definition of registers, bit fields and properties such as permission, pre-defined values, and register cross-dependency. The descriptions are synthesized to C macros after a static consistency check [MRC+00], [SYKI05].

We build on this previous work to access device registers, defining an abstraction for the communication system and stipulating rules for the definition of a

device interface that completely wraps the device internals. This interface exposes the hardware module with a pure software interface. For this software interface, a specific way to handle the special access of peripheral devices, interrupts and Direct Memory Accesses (DMAs), is defined with which the software layer above it has to comply.

Based on the device interface, we can specify contracts as operational constraints related to the interaction with the device. These constraints are then synthesized to runtime tests that run together with the device interface preventing faulty usage of the device. Based on it, drivers for different operating systems can be created that cannot misuse the device. Alternatively, the device interface can also be used for firmware.

Our constraints are based on a model of the device through an extended finite state machine. During the model execution, state and variables can be used to evaluate if the device is ready for a specific operation.

1.4 Contributions

This thesis defines a specification for module's interaction in embedded systems comprising both interaction between hardware-hardware and hardware-software. Furthermore, we provide automatic test generation and execution. Instead of repeatedly defining tests, only constraints have to be defined. Specific contributions for the hardware contracts and device contracts for drivers are the following.

- Definition of the specification of hardware component interaction, with categorization of signal faults enabling the discovery of evidence for environment related failures.
- Test circuit for online categorization and detection of constraint violation of a hardware component.
- Extension of the SystemC Transaction Level Model (TLM) for system level simulation of signal faults.
- Recovery mechanisms for synchronous communication after signal faults.
- Two layer architecture for driver development separating device access from data processing. Device access through a pure software interface.
- Framework supporting platform independent access to device from descriptions of a register map and communication. Automatic compliance with special behaviors of register access, such as conflicting register bits.
- Definition of the specification of hardware software interaction for drivers based on Interface Description Language (IDL) and Extended Finite State

Machine (EFSM). Device constraints defined on top of the bottom layer of the proposed driver architecture.

- Automatic constraint compliance for the exported software interface. Extended finite state machine executed together with the interface.

1.5 Thesis Outline

In the next chapter, we review the state of the art in design for reliability, design of embedded systems, component-based design and reliability of device drivers. In the following chapters, we treat component interaction as a source of system faults, providing specification and testing thereof. We distinguish between hardware only interaction and hardware software interaction through drivers. In Fig. 1.1, the hardware and software layers of a system are represented with their corresponding components. The task of every layer is enabled by the interaction of its components. In the hardware layer, interactions rely on communication systems/buses that transfer data through pins and signals, while the software interacts through function calls. The software is integrated in the CPU that typically assumes managing tasks controlling the different hardware components. Therefore, the software needs drivers, i.e. components that know how to control the existing hardware modules. The system's mission is generally implemented in form of algorithms, either in the firmware directly or as processes/applications of an operating system that is ported to run on the CPU and control the underlying hardware.

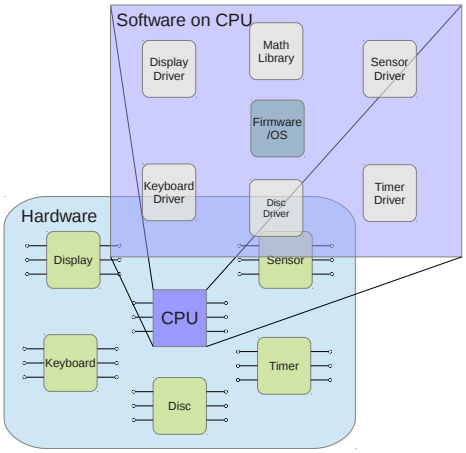

Figure 1.1: Block diagram of an embedded system architecture with a two layered view of its software and electronic parts. Blocks of the same layer are interdependent and associated. Blocks of the software layer are physically associated with the CPU. Drivers are logically associated with their hardware counterparts.

The specification of hardware interaction with contracts and testing os those contracts is described in Chapter 3, followed by the extension of the SystemC TLM to allow the simulation of the defined faults of component interaction. In Chapter 4, recovery methods are presented for a synchronous communication protocol. This completes the specification, simulation and real-time testing of hardware interaction. The following chapters refer to the device driver development as the interface and interaction between hardware and software. Chapter 5 presents our driver architecture defining the rules for the design of a pure software interface for a device. Furthermore, the framework for its systematic development is presented covering the device internal's description, register maps and communication systems, and the generation of access elements. After this, our definition of constraints based on the dynamic behavior of devices is presented in Chapter 6. Its specification and the generation of the dynamic model and the corresponding tests are described. Then, the conclusion of this work is drawn and future work is discussed.

2 Reliability & Design of Embedded Systems

In this chapter, we review the different strategies for reliability design and known methods used in both hardware and driver development. We first discuss reliability in hardware design and generic reliability techniques, particularly in applying reliability techniques to component-based design. Component-based design is well established in electronics but use of a contract strategy as used with software is not. In this work, we apply the concept of contracts for hardware providing both fault avoidance and tolerance. Software is increasingly embedded in hardware systems in which the hardware-software interface plays a key role. Hence, driver development methods and strategies for driver reliability are also discussed.

2.1 Reliability and its Role

The complexity of electronic systems increases continuously. It is impossible to build a competitive system without relying on existing modules. Also, it is not feasible to test new systems exhaustively because of their size. Modules have to be trusted and only the specified functionality can be tested. Examples are found everywhere, from consumer electronics, through laboratory equipment to medical devices. Thousands of modules are used to build them and each one is trusted to some extent. Earlier, telephones, televisions, oscilloscopes (cathod-ray) or electrocardiographs relied purely on hardware modules. Nowadays, software is used in all these examples providing much more functionality. However, this simple fact raises the complexity of these systems immensely. Millions of lines of code are trusted whose behavior is not thoroughly known, together with compilers, processors, peripheral hardware and third-party drivers.

Systems do not fully exercise the functionality of their components. Regularly, they apply components in a specific operational mode for a specific functionality. Operational modes are normally specified and tested. However, because components have requirements on their usage, the component assembly has to be tested with simulation and prototypes as well. This ensures that the system fulfills the component requirements and that the components together follow the system's specification.

The specification defines what the system does and how. Often, user expectations and designer assumptions are not defined in the specification. However,

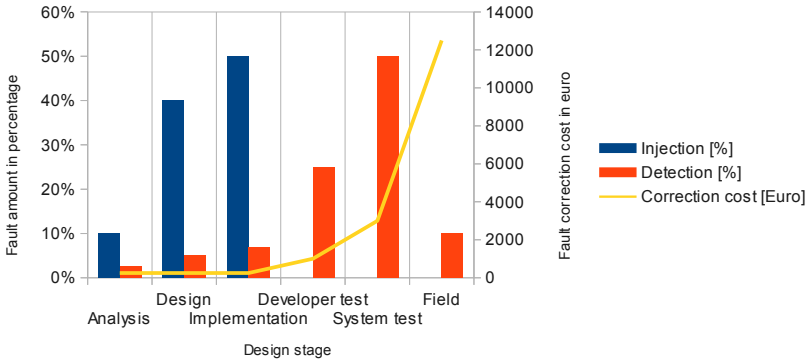

Figure 2.1: Percentage of fault injection and detection over product development stages, and the incurred cost for fault correction depending on the development stage. Analysis made by Möller [M̈96] based on the fault history of several projects.

a fault can only be defined if the resulting behavior goes against the specified functionality or is forbidden. Therefore, the specification has to be as complete as possible. In hardware design, missing specification of components leads to most system failures, while most failures in software occur due to erroneous assumptions about its operating environment [Dub08]. This problem is known as underspecification.

In order to improve system reliability, test cases can expose some flaws in system functionality. However, because the specification is mostly informal, functionality constraints are not well defined. Thus, tests have to be created manually with many unspecified assumptions. That hinders the coverage of the implementation through tests. In agile software development, tests are successively created according to an evolving specification in parallel to—or earlier than—the implementation achieving higher testing coverage of the implementation. In hardware development, the development cycle is much longer and redesign cost is high. However, only recently, mixed-signal simulations have been made available to test hardware more thoroughly prior to prototype production [BCMS05]. Because of the difference in the technologies, digital and analog simulators are optimized differently exhibiting different speeds. Thus, analog and digital electronics are normally simulated individually leaving mixed-signal design flaws uncovered. Furthermore, prototype testing costs more and is less flexible than simulation. In addition, the testing environment might not reproduce the operational environment while the faults discovered might require a new system design.

Finally, the resulting cost of faults is relative to the design stage in which they are solved, see Fig. 2.1. Faults which are uncovered and solved in early design

stages have low cost, while the failure of equipment in field might require a product recall involving high costs. Therefore, industry and academia continuously look for techniques that allow early fault detection. Due to the economic and complexity factors, there is much work involved in reliability research. In particular, the study of the reliability impact of design and implementation methods is very important because most faults can be traced back to these stages and their correction is less expensive in these stages[Jac09].

2.2 Design for Reliability

In contrast to functional testing, reliability techniques have more success in tackling faults. In particular, fault avoidance and removal can be compared to design and testing. But also, fault tolerance and forecasting have played a fundamental role in reliability design. In the following, we first present the reliability threats followed by the four fields of reliability design.

In reliability, threats are categorized into faults, errors and failures. Faults are the origin of all threats. They are defects in a system. However, faults only turn into errors on a fault activation, a system operation that requires the faulty part. Errors are deviations of the intended system behavior that do not compromise the overall system functionality, such as a wrong output value. A failure, on the other hand, is a system behavior not allowed by the specification. A failure has consequences on the system functionality. For instance, interrupted system functionality or necessity of repair or restart.

Because faults are the origins for all threats, design for reliability is categorized by the way they deal with faults. The first technique is fault avoidance. The idea is to ensure correct behavior by design. Faults can be generically dealt with by targeting common flaws in implementation techniques. Approaches are based on requirement specifications, tool-assisted design, enforced design principles, and systematic reuse techniques. Also, formalized implementation and its synthesis promise to completely prevent faults [Lyu07], although, its application scope is limited. Reuse techniques are widely applied in the industry. But, as stated above, the usage of reliable components has to be tested in the deployed system to ensure correct integration. A common example of this problematic is the explosion of the Ariane 5 rocket [Gro04]. The reused piece of code failed when deployed in a different environment.

Although formal methods could completely prevent fault, they can only be applied to software and digital hardware because they disregard the physical environment and can misinterpret system's goal. Thus, strategies to remove faults are equally important. Fault removal techniques are testing and inspection. Different types of testing exist tackling different kind of problems, such as unit, integration or system testing. While unit testing targets a component, integration testing tests if an assembly of components works. Finally, a system testing tests if the

system meets its specified requirements. But tests can also be categorized based on the depth of their checks. While black-box testing only tests the interface of a component, white-box testing has complete information about the component and also tests its internals. In the case of a component with both accessible and non-accessible implementation parts, grey-box testing is possible by mixing black- and white-box testing. Lastly, design inspection can occur both manually and computer assisted. Because computer assisted inspection only finds a limited set of flawed constructs which are pre-programmed in the tool, manual inspection is also required. Examples of assisted inspection are Design Rule Checking (DRC) on layout/Printed Circuit Board (PCB), Computer Aided Design (CAD) systems, simulation, debugging programs and different types of static analysis of program or hardware description. Static analysis is a wide field, it ranges from syntax analysis through semantic analysis [ECCH00] to formal methods.

Formal methods for inspection are known as formal verification. Because regular testing can only prove the existence of a fault, not its absence, the assurance of testing coverage requires more time than used for design and development [Wer09]. On the other hand, formal verification is the only technique that can ensure absence of fault promising complete coverage in reasonable time. It was successfully used to find the Intel's Pentium II bug for example [Kam07]. It uses a mathematical representation of the system for which a specification, defined as mathematical properties, is proven to hold. This proof is given either by an automated theorem prover or by a model checker. In the case of the automated theorem prover, a specified property holds if the theorem prover can formulate mathematical proofs for those properties. The automated theorem prover tries to infer proofs for the specified properties given a set of axioms and the system description [KG99], [Gup92]. The technique of model checkers explores all possible system states of a model. If a defined property is not violated after the exploration of all possible states, the property holds always. If it is violated, a sequence of states leading to it can be defined [Wer09], [KG99]. The difficulty of formal verification lies in the formalization and in the requirement of a model of the system. Currently, some systems work with languages used for implementation. For example CVE, Checkoff-M and RuleBase handle Verilog and VHSIC hardware description language (VHDL), while CBMC and KeY handle ANSI-C and Java respectively. However, a complex system, based on both hardware and software for instance, must use a model for abstraction. Furthermore, the environment is ignored. Thus, either there is a risk of mismatch between model and implementation or uncertainty about the integration remains. Moreover, the formalization of properties for specification can be restrictive and complicate the specification reducing the number of found flaws [ECCH00].

A fault free system requires a high engineering effort. Although a 100 percent test coverage can ensure that all system parts have been exercised, it does not state that the required functionality has been met in all these cases. Even

in a fault free system, environment faults can lead to failures because systems require specific environmental conditions to work that cannot be always guaranteed. Thus, remaining faults should be prevented from manifesting as system failures. This is achieved by fault tolerance which applies redundancy as its basic technique. Redundancy can be achieved by component replication and timed repetition. Uninterruptible power supplies, Transmission Control Protocol (TCP) packet resend, built-in, and online testing are examples. In software, defensive programming checking input ranges and output conditions forbidding illegal operations are also redundancy [Lyu07].

Different mechanisms allow detection, location, containment or recovery of different fault types. Two redundant components can detect and contain a component fault for example. Three redundant components can detect, locate and recover from a component fault. For system specific faults, built-in testing can be developed that allow at least fault detection and generally location. If the tests are executed during operation, built-in testing can be classified as online testing. Upon fault detection/location, recovery can be triggered. Besides triple redundancy, software rollback and hardware reconfiguration are common recovery mechanisms. In software, the program recovers its internal states to a previous point in time. In hardware, the structure of the system is changed isolating the faulty part and enabling a working one. If no replacement is available, the system might work with degraded capability. Recovery mechanisms can be optimized to specific cases. In Field-Programmable Gate Arrays (FPGAs), partial reconfiguration allows timed redundancy in hardware [PHJ06], [ESS00], while virtualization isolates an operating system extending resource control, such as computing power, memory and peripherals [Lev09]. This way, a crashed driver of a specific resource can be recovered while the remaining system is still available, similar to microkernels [HBG+07]. Recovery strategies are diverse and a safe recovery can require complex mechanisms depending on the dependency tree of the system. This is the case for device drivers for instance.

A system is expected to fail. Therefore, a reliable system is a system whose failures, their probabilities and the related operational conditions are known and specified. The specification and achievement of a reliability level is the aim of the fault forecasting strategy. A reliability measurement is only possible given fault models, failure modes, and a specified environment. Naturally, it implies the study of a specific system. For a specific system, failure modes are classified together with their triggering events, such as component faults or environmental conditions. Then, the measure of the reliability follows its description.

Reliability is a measure of the continuous delivery of correct service [ALR01]. It is determined as the probability of correct system operation in an interval $[0, t]$ given that it was operating correctly in time 0 [Dub08], [Lyu07]. A system built of reliable components has many dependencies. Without redundancy, a component failure can lead to system failure. For instance, a reliability of 33.79% is assessed

for a system built of 10,000 non-redundant components, each with reliability of 99.99% [Dub08].

After faults and their possible resulting failures are analyzed, the probability of failure occurrence is evaluated from which the reliability can be derived. There are two approaches to evaluating the reliability of a system; either by a system model in the design phase, or by assessing it by test [Dub08]. The system model used for reliability evaluation is based on probabilistic models that use component level failure rates as determined by their manufacturers for instance. Examples of such models are fault trees, block diagrams, Markov chains [Dub08], Petri-nets [MT95] and Bayesian networks [MT95], [BPMC01]. Although modeling provides an early reliability evaluation, it must be still validated by actual measurements. The test assessment uses test data. Because it works with the system directly, it is more accurate. But its cost is higher as well and occurs later in the project. However, although fault activation is directly related to system operation, hybrid models that simulate the system functional behavior together with their failure modes and fault behavior are not existent.

Finally, given a specified reliability level, the reliability growth process refines it iteratively. The process uses an iteration of testing, failure data collection and system rework, until the specified reliability level is achieved.

Although techniques of design for reliability can be generically applied to enhance system reliability, better results require knowledge about the target system. This implies initially that historic data about system fault and failures exist and later that their relationship is established. With this information, reliability design techniques can be selectively applied achieving higher success.

2.3 Design Methods

Design methods of current electronic systems, especially embedded systems are discussed. The design of these systems comprises board, system-on-chip and software development. An optimal assembly of these components and their integration is the aim of the discussed technology. Starting in the 1990s, hardware/-software co-design has evolved from an integrated circuit to a system level design methodology [Wol03]. Its goal is to increase the predictability of embedded systems by providing performance, power, size analysis already in the design phase. Finally, synthesis methods for development automation are also of interest.

2.3.1 Electronic System Level Design

A lot of terms have been used to describe what is now known as Electronic System-Level design over the years. Electronic System Level Design Automation was described by Doug Fairbairn and Ron Collett in a design abstraction taxonomy in the 1990s. This term evolved later to System Design Automation.

From 1995, the term ESL, introduced by Gary Smith, has been the most used term in the design and Electronic Design Automation (EDA) industry. More recently, Bailey et al. [BMP07] described ESL as "the utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of successful implementation of functionality in a cost-effective manner, while meeting necessary constraints." This description tries to combine other known descriptions of ESL. Specifically, the terms *system*, *abstraction*, and *process* are of particular relevance because ESL is a process to describe a *system* using appropriate *abstractions*. But also, *implementation of functionality* leaves the implementation platform unspecified, hardware or software. This is specially important since ESL has evolved from the hardware/software co-design.

ESL is about design, verification, and handling complexity through abstraction. This complexity is posed by systems composed of hardware and software whose early simulation was not possible leaving actual analysis of functionality, performance and quality factors for late development stages. Thus, the aim of ESL is to enable the analysis of functionality and constraints for these systems in the design stage. The solution is an executable specification based on functional models of components to be implemented later. Component verification occurs through simulation of component implementation and comparison to its high-level model whose functionality is assumed to be correct. Depending on the abstraction level of the functionality and thus its accuracy relating to the implemented system, performance, and quality can be measured based on the executable specification. In contrast to formal verification that verifies system or component independently, ESL verifies an assembly of components for compliance with functionality, constraints and quality factors of the target system. Finally, software to be run on this platform or embedded in the system can be implemented and tested on this same executable specification, a method called Hardware (HW)/Software (SW) co-simulation.

Thus, system design becomes especially compelling for embedded systems with a large amount of hardware dependent software, typically systems with much functionality, smartphones for instance. In these systems, the architectural design of hardware-software partitioning is decisive in the achievement of performance goals. With 250 nm process technology, architecture design and validation were insignificant in comparison to implementation. However, it became considerably important at the 90 nm technology accounting for 25% of the total development time. As well, the requirements for hardware dependent software, such as Operating Systems (OSs), firmwares and drivers, has similarly increased, from 35% to 55% when compared to the complete implementation. At the same time, traditional design with independent hardware and software development has failed; in over 70% of the cases for missing performance expectations by at least 30%; in over 30% of the cases by missing functionality expectations by at least 50%. In addition, 54% of designs missed schedule with an average of 4 months

and about 13% of designs were canceled. The main reason for their failure has been given as limited visibility into the complete system [BMP07].

In addition, ESL aims at architectural exploration. For this, a virtual system prototype is created that models the behavior of the target system. This system model is an architectural proposal for the target system made of component models. These models have to be simple in order to reduce the time taken in their design and to increase simulation speed. This is achieved by abstraction. Only the functional behavior of the target components is described, their internal structure is ignored. These component models communicate with each other through transactions that transport data without applying communication protocol or signals, thus abstracting the communication. Such a model represents a possible architecture for the system imposing a specific task partitioning with corresponding synchronization requirements. Executing this model, its system performance and related quality factors, such as determinism can be estimated. Considering the system goal, a nearly optimal architecture can be found iteratively by the analysis of earlier architectural bottle-necks. Because the resulting model is executable, it can be made available to the software team for software development with the inclusion of the required visibility such as registers and interrupts. In addition, the model can be further refined to include implementation properties of components enabling cost, accurate performance and power consumption analysis. At this stage, hardware-software partitioning also takes place, where the designer decides which tasks will be implemented in hardware components or as software. Fig. 2.2 gives an overview of the ESL design process.

The ESL methodology requires a language for model description, simulators, and different possibilities of model refinements allowing for architecture implementation and performance analysis. In the beginning of ESL and HW/SW co-design, executable C code was used to model the systems. That was sufficient to allow platform description and parallel software development. However, the performance analysis of multiple processes required the appliance of simulators similar to the ones used for network analysis [Wol03]. Moreover, embedded systems have exhibited a tighter control requirement due to loose concurrency, not simply multiple processes in a CPU. At this point, C was replaced by a Co-design Finite State Machine (CFSM) that was able to describe concurrent communicating processes. In order to allow model refinement to implementation, languages as SpecC, SystemVerilog and SystemC have emerged that allow a high-level model description with abstractions but also permit, in the case of SpecC and SystemC, refinement to implementation of both hardware and software. SpecC lost the standards' race to SystemC that has been published as the Institute of Electrical and Electronics Engineers (IEEE) 1666 standard in 2005 [BMP07]. In 2011, an update to the standard has been published. SystemC is currently well supported. Co-simulation is provided by the major simulation tools, as Mentor's ModelSim, Synopsis' VCS, and Cadence's Incisive Design Team simulator. An open source

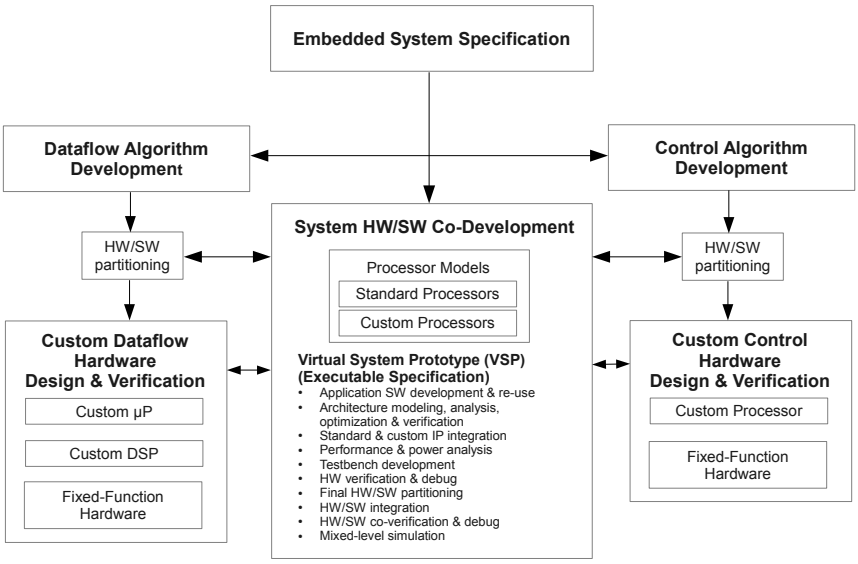

Figure 2.2: Electronic System Level design process leading to an implementation and its architecture. Picture reproduced from [BMP07].

reference simulator for SystemC is available at [Ope] as well.

Besides the language barrier, the biggest barrier to the adoption of ESL has been the automation of chip implementation. Because system-level design adds up to 20% of the design effort, and due to the lack of high-level hardware synthesis, ESL has been seen as negative by the Register Transfer Level (RTL) engineers. Only recently, high-level synthesis tools have appeared like Cynthesizer, CyberWorkBench, BlueSpec, Agility Compiler and Catapult Synthesis. However, these tools have high cost, and require the usage of specific dialects of C/C++ for input complicating the high-level simulation with other vendor's simulators and Intellectual Properties (IPs) hindering a manufacturer independent architectural exploration.

Due to the aforementioned hardware implementation complications and to time-to-market constraints, system design effort concentrates on assembling prefabricated IPs to fulfill system functionality and performance specifications, inhibiting top down design. The component reuse methodology has been long used in engineering. But, the system level design gives the ability to early evaluate the resulting effect of single components with respect to the complete system.

2.3.2 Component-based Design

The software world has taken the opposite path of the electronic system level design. Software has been mostly developed from scratch in a top-down design. In

contrast to hardware, co-simulation and co-design are not a problem for software. In software, different abstraction levels can be simulated seamlessly. However, reuse promises higher quality for less cost. With object-orientation, industry observers thought that reuse would become common-place [ABB+02]. Although, primarily, the extensive runtime environment dependency of software pieces has hampered wider reuse, component models such as Common Object Request Broker Architecture (CORBA), Component Object Model (COM), Enterprise Java Beans or .NET have emerged to solve this problem. However, component-based design is still an emerging software discipline. Despite common runtime environments, wide reuse has not been achieved. Further challenges for component-based design exist and are active research topic. In particular, testing of component-based designs requires different strategies from traditional software testing because the component's implementation is often not available and its deployment environment can differ from the expectations of the component designer.

Component-based design is a subdiscipline of software engineering. Software engineering looks for standard techniques and methods for software development. Similar to traditional engineering, standard methods have to be classified in regard to use, cost and quality. The cost factor is the target of the component-based design. It goes along with the standard assembly of systems in engineering where a system is composed of many prefabricated parts. In electronics, for instance, components of different complexity are available for system's design, such as generic operational amplifiers, logic gates, flip-flops, Digital-to-Analog Converters (DACs), memory, processor. Basic transistors are rarely used and the cost of a microcontroller, under $1.00, cannot be compared to its design cost.

With the rise of object-oriented programming, and especially encapsulation, the design of components became more feasible for the software world. The major advantage of component-based design is reuse which is expected to reduce production costs. In addition, the quality of the end product should improve because well tested components should have less faults than a custom counterpart. However, development for reuse has not been traditionally targeted in software design. Components and their development are not clear concepts in software engineering. In particular, design of components often does not anticipate different contexts of component usage making these components less reusable. The lack of documentation and missing view of possible operational modes contribute further to this.

Therefore, component-based research came up with principles for component design; component composition, clientship, interfaces followed by quality attributes and documentation [Gro04]. Composition means that components can be assembled both hierarchically and in aggregation to create other components. More importantly, clientship and interfaces describe how components work. The clientship defines a client/server relation between components. A component requires services. Given that its requisite services are provided, it provides services

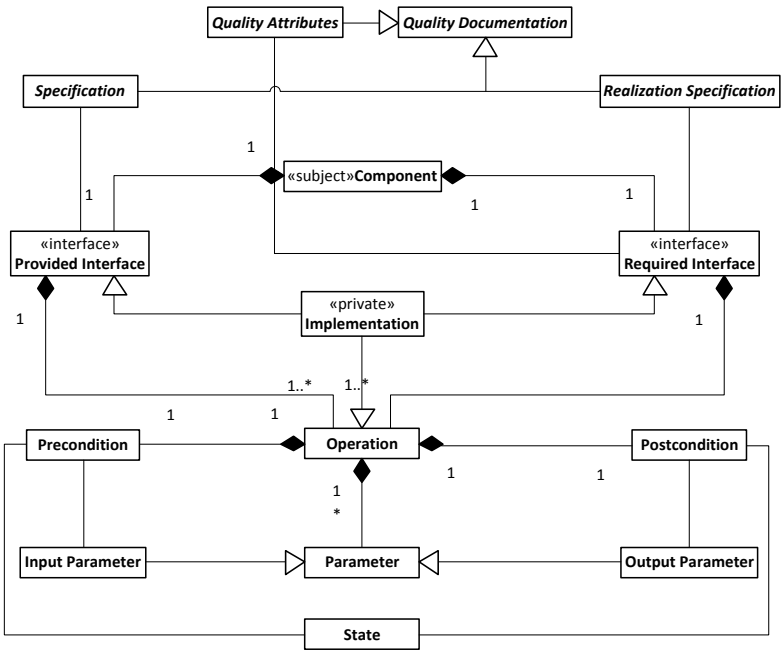

Figure 2.3: Component meta-model that includes component details missing in traditional software development. Picture reproduced from [Gro04].

of its own too. This leads to a definition of component's contract establishing the duties and rights of a component. If any right is not fulfilled, the duties are also not granted. An interface describes a collection of functions that the component provides or requires. Quality aspects are described in quality attributes and documentation. Quality attributes define non-functional aspects of a component such as performance and reliability. The quality documentation augments the functional specification (i.e. description of functions) with operational modes, possible behaviors and how they can be achieved, similar to a datasheet known from electronic parts. A Unified Modeling Language (UML) meta-model of a component with its corresponding composition parts is represented in Fig. 2.3.

Similar to electronic system level design, component-based design cannot fully rely on bottom-up design. This occurs because a system cannot be perfectly decomposed into well matched prefabricated parts. Component-based design occurs iteratively through system decomposition in tasks that can be fulfilled by a component. If a component matches the requirements, it is integrated. If there is no match, the tasks are further refined until a component is found. Otherwise, the task is designed from scratch. Finally, the result is a system built of prefabricated parts and custom implementations.

Different components and requirements can lead to implementations in differ-

ent programming languages using different runtime environments. In addition, a specification is required that defines the system's goals. However, programming languages do not provide a system overview or allow definition of constraints. Thus, as SystemC was created as language for ESL design, UML has bridged the gap as modeling language for software specification, becoming an industry standard [Gro11]. Alongside ESL, synthesis of UML specifications has been studied using the approach of generative programming following the concepts of Model-Driven Architectures [SBF03]. The idea is to generate an executable program based on a UML description. The synthesis of UML can be more or less useful depending on how much more explicit and less time consuming a UML description is in comparison to an algorithmic implementation language.

Component-based testing is technically different from traditional software testing, imposing new challenges. First, components have to be tested in a different context to that in which they have been developed to allow for exercising the component in an untested operational mode. Besides, a prefabricated component cannot be internally tested. Access to its internals is not possible. This requires that black-box testing is used. However, internal state information of the component exposed through the interface is relevant to component test and should be made available by component developers. Finally, a finite number of tests should suffice for system to rely on the component. Otherwise, the cost of testing the new system may exceed the cost advantage of using prefabricated parts [Gro04]. Generically, the operational modes required by the system should be covered. In reality, objective answers to questions about adequate testing, test types, amount and input set, are an open challenge in testing.

Because component reuse may apply components in a different context, integration testing is most relevant. Compared to hardware design, correct operation of applications is verified late in component-based design. Only when the design is ready, at deployment time, can the interactions between components for the final assembly be tested. Component interactions have to be met both syntactically and semantically. The connection of interfaces has to fit and the clientship of the component has to be respected. While a component with the same interface can be connected to the system, the system will fail if its clientship is not respected. Interface compatibility can be ensured by the compiler for example (syntax analysis), while clientship can only be tested in runtime (semantic analysis). Built-in contract testing offers an early and scalable solution for testing component-based systems. Components are equipped with the ability to test their execution environment according to their clientships. In addition, components are made testable by their surrounding components. This way, components validate each other automatically providing system correctness through a fault tolerance strategy where a component's failure is detected early.

Contracts

A contract defines the necessary conditions for a method to be successfully executed. It is a type of defensive programming technique. Meyer defines that a contract is defined by preconditions and postconditions [Mey92]. Preconditions have to be fulfilled for a method to be correctly executed. After method execution, the defined postconditions hold. Preconditions and postconditions are defined in terms of function arguments, object state and return values in object-oriented programming. The aim of contracts is to provide a systematic method for the otherwise ad-hoc defensive programming techniques that reduce visibility of the algorithm and makes it more complex.

Defensive programming has been long used through assertions being a type of built-in testing. Assertions provide a mechanism to verify if an expression is true or false. In the first case, program execution continues normally. If the assertion evaluates to false, some detection or recovery mechanism can be activated. Regularly, a message is printed along with the place where the error has occurred. Built-in tests might run only during development and be automatically disabled by the compiler for deployment, as is the case for assertions. However, they can be also deployed in the field for debugging purposes.

Traditionally, built in testing is used to test the behavior of the component itself. However, exhaustively testing a part of a component that has already been tested will not expose a fault because software does not degrade. On the other hand, a component has requirements on its environment that have to be fulfilled for delivery of correct service. These requirements refer to the system/user and to services of other components that do not know about the requirements of the first and vice versa. These dependencies imply requirements beyond the system and single component specification. The description of these component interactions is possible with contracts.

In the design of applications with component reuse, components have to be trusted. In particular, the internal implementation of third-party components is not accessible but their functionality is normally well tested. However, a working component can fail in the wrong environment. Therefore, component compliance with the environment has to be ensured. Because contracts specify the conditions for every method of components to successfully run and components interact through methods only, they provide the missing specification for component interaction. And thus, contract testing can ensure correct operation of an assembly of reliable components. For this, every component needs a contract for its methods. This contract defines pre- and postconditions for the component's methods in form of argument ranges, internal state and range of return values. If the component's preconditions are fulfilled, it can operate correctly. At the same time, other components have to ensure that their preconditions are fulfilled by its servers' postconditions.

The fulfillment of pre- and postcondition does not cover the interaction between

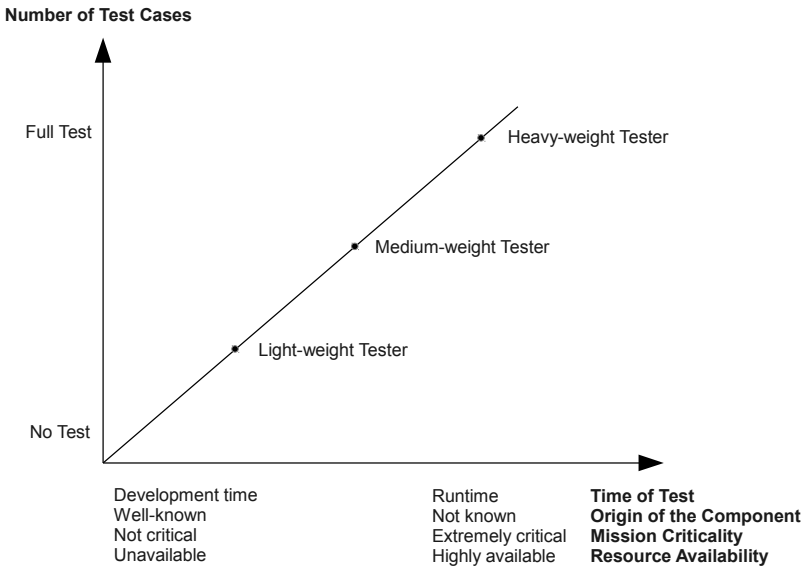

Figure 2.4: Criteria that affect system reliability and the corresponding testing coverage to ensure correct operation. Figured reproduced from [Gro04].

components completely because only the value of the exchanged data is ensured this way. Gross et al. recommend the inclusion of a testing interface that gives access to the global object state [Gro04]. This way, a component can specify the behavior of its servers and verify that they behave accordingly. This implies however that a component tests the implementation of another component with regard to its internal states. Something that the component could not know prior to design completion. This can be useful in the case of a standard such as the Portable Operating System Interface (POSIX) where different components implementing the standard could be tested against a single specification. But regardless of restricted visibility, contract testing remains a meaningful alternative for testing component interaction.

The implementation of contract testing presents further questions such as test deployment and extent. Although every client could possibly test their servers directly, deployment of a separate tester component can still increase reuse. Through combination of tests, functionality can be tested at a higher level embracing many components or even the complete system/application functionality. Furthermore, the time and frequency with which the test is executed can be made dependent on external criteria such as reliability or trust. Fig. 2.4 shows a scheme for test periodicity in relation to component properties.

2.4 Driver Development Methods

A device driver is a software interface (collection of functions) to be run on a CPU that allow the control of hardware modules. This control is enabled by a hardware interface, e.g. bus, between the CPU and the target module that only allows for device access. Proper device control is achieved by switching device registers and electronic data (bytes/bits) of the device made externally visible through this bus. Thus, a device driver needs information not only about the bus and its protocol but also about the device controls in order to translate these to software functions.

Earlier, device control was more complex because it was also achieved through direct signaling of bus level details, such as voltage level and timing control; for example the control of a Direct Current (DC)- or step-motor. Now, devices' actuators are integrated into custom controllers that translate these controls to corresponding electronic data as numbers, such as speed/position, whose access occurs then through a standard bus. Such actuators take over the signaling and timing part releasing the CPU from this task.

Research in HW/SW co-design has given special attention to drivers and communication interfaces. In ESL, software development assumes that drivers will be available for modules' control. Thus, the initial software only interacts with functionality of modules without considering their interface. Existing hardware and software layers that allow communication are not taken into account. While for HW/SW co-simulation, high-level model abstracts the CPU for simulation speed, the communication between hardware modules is also abstracted consisting of transactions without protocol or interface. The refinement of the high-level algorithmic models to implementations requires real hardware and software interfaces. Therefore, the design of CPU buses, including direct memory access, data conversion and buffering is necessary. Similarly, software from basic drivers and Input/Output (I/O) functionality up to sophisticated operating systems and middleware must be ported to the underlying hardware [JW05]. Although ESL and HW/SW co-design have mostly tackled the hardware software partitioning and early software development problems, the co-design of hardware and software interfaces is still a missing link for high-level synthesis. Furthermore, the design of these layers has considerable performance and reliability impact because software and hardware mismatches follow from the separate development strategies.

The choice of the hardware interface has effect on performance, reliability, and other quality factors, such as latency and determinism. Moreover, the Intellectual Property (IP) compatibility on a System on Chip (SoC) is highly dependent on the interface/CPU bus. The use of the same interface for CPU and peripherals allow the CPU to access the peripheral registers as memory addresses. A bus implementation of the interface has to be configured to map different peripherals to different memory areas. If a device has a custom interface, either an

adapter is necessary or its protocol has to be imitated by general purpose I/O, a method commonly known as bit-bang mode. Unfortunately, both techniques involve moderate to high performance penalty.

In the literature, synthesis techniques for interface and driver co-design have been proposed [WB94], [COB95], [BL98], [DRS04], also predicting that network-on-chip would substitute the current master/slave communication systems for SoCs [Jan03]. However, the fixed purpose of a communication system and similar constraints have led the industry towards standardization instead of synthesis. For example, ARM processors use AMBA, MicroBlaze from Xilinx uses CoreConnect, Altera NIOS uses Avalon, and OpenRISC uses Wishbone. The performance and quality factors of the different standards are known and one can be chosen instead of synthesizing. However, performance of interface adapters for compatibility is reduced for example due to protocol differences regarding burst modes.

Access to devices on computers was also standardized and has changed substantially with time. Originally, each peripheral device had a custom interface with its own physical connector, pin connection, signaling and communication protocol. Standard solutions offered more compatibility and reduced cost resulting in advantage for both customer and manufacturer. The initial personal computer standards, like RS-232 [Dep69], parallel port [Com94] and Industry Standard Architecture (ISA) [Cor07], became widely adopted, allowing connectors to be reused and avoiding custom hardware for interface implementations. However, violation of the standard, for example with custom communication protocol for RS-232 and parallel port, was still common. In current personal computers, Universal Serial Bus (USB) and Peripheral Component Interconnect (PCI) have superseded earlier standards providing auto-configuration/enumeration allowing a device to be automatically recognized and configured by the system. In contrast to older standards, no manual configuration of jumpers for addressing or manual driver selection is required. The system recognizes the new device automatically and if a corresponding driver is available, it is automatically loaded. USB and PCI also brought reductions in power consumption through enforcement of suspend modes on peripherals. While also letting devices can on the bus for power control, so simplifying their design.

While the CPU can access registers of devices directly connected to the CPU bus through simple memory accesses, the access to devices connected to external communication systems additionally requires the control of this communication system. In Fig. 2.5, registers of modules connected to the Wishbone bus, in the center, can be accessed by the CPU as memory addresses. Yet, the Serial Peripheral Interface Bus (SPI) Electrically Erasable Programmable Read-Only Memory (EEPROM) connected externally through the SPI module, a communication controller, can only be accessed through SPI transactions that are possible through an SPI driver. To enable SPI transactions, registers of the SPI module have to be set/retrieved. In response, the SPI controller executes the transac-

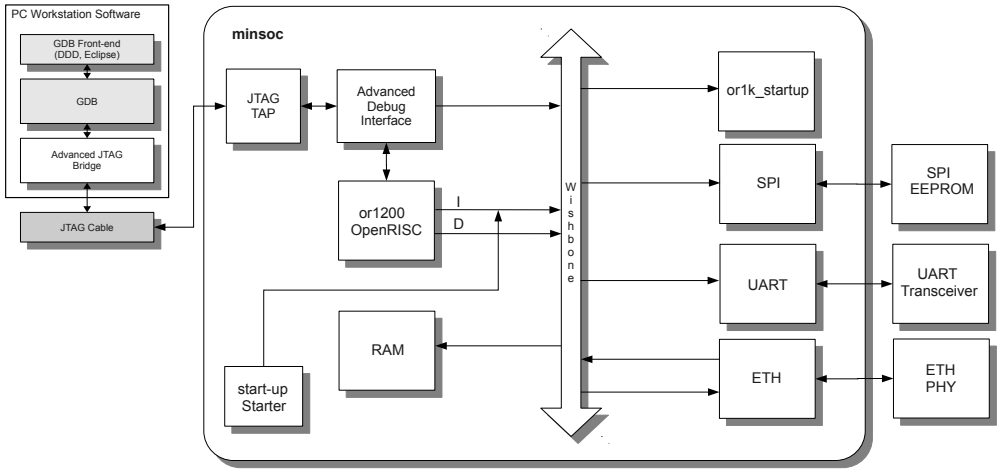

Figure 2.5: Modules' view of MinSoC, OpenRISC based System-on-Chip: arrows leaving a module signalize a module's Wishbone master interface, arrows pointing to a module signalize a Wishbone slave interface. Double-sided arrows are different connections.

tions that finally set/retrieve registers from the device connected to SPI, memory addresses in case of the SPI EEPROM memory of Fig. 2.5. The same occurs for USB and PCI, the most common interfaces in personal computersr. A USB or PCI device can only be accessed through the corresponding interface. Thus, the USB/PCI driver becomes the underlying access method for the device driver. This exemplifies the cascade dependency of system's drivers.

The personal computer industry evolves fast due to its unified design. By contrast, embedded solutions are custom, designed on demand, complicating the development and raising cost. Therefore, an embedded system's development trend is the adoption of parts designed for personal computers. Current SoCs and embedded systems support USB, Ethernet and run operating systems. The enhanced functionality also enables the use of—thanks to the high number of parts produced for them—low cost hardware such as USB flash drives and webcams. In particular, the operating system's notion of a process as abstraction of a task facilitates the software design and enable seamless networking. Furthermore, software applications designed for an operating system are portable. That is not the case for bare-metal firmware where threads are custom designed through the use of purpose specific interrupts and timers. However, as time control is not straightforward for C based bare-metal firmware, it is even more complex for an operating system. The generic solution is to outsource real time constrained tasks

to hardware modules or controllers running a purpose specific firmware, or still balance applications on a real time operating system. Although the adoption of operating systems bring many advantages, they are also complex systems that require well versed specialists.

An operating system fulfills two purposes. It abstracts the hardware providing an extended machine for software use and manages resources [TW09]. Therefore, multiple applications can operate using computing power, memory, disk and printer without knowing about device specifics or caring about resource ownership. The operating system switches the ownership of computing power and devices avoiding conflicts. As the compiler abstracts the internal implementation of the CPU for computation in high-level languages, a central task of operating systems is to allow for multiple processes optionally providing virtually infinite memory. These tasks are enabled by elements of the CPU that the operating system has to control. Current operating systems separate this CPU dependent code from generic code and define fixed interfaces for their implementation known as hardware abstraction layer. Especifically, a timer together with an interrupt controller allow processes to share computing power according to a scheduling algorithm. Processes require their own portion of memory to run. However, it is dangerous to allow that a process inadvertently writes to memory parts which are possibly used by other processes, especially kernel memory. For example, kernel code could be rewritten by a process causing the whole system to fail. This has led manufacturers to build a hardware protection mechanism based on the memory management unit. The main task of the a Memory Management Unit (MMU) is to translate a virtually infinite memory to physical memory addresses. Furthermore, the MMU is able to forbid access to memory pages whose protection level is lower than the protection level of the executing code. Generally, kernel and application protection levels are chosen in a way that the kernel can access any memory but the applications cannot access kernel memory. A standard approach of ensuring encapsulation and thus fault containment on operating systems is to move programs from kernel code to application. This has been extensively done for drivers as described later.

In addition, in an operating system, devices with same purpose can be handled the same way by applications through the definition of generic device interfaces. For example, any disk and Ethernet card can be used for the file system and networking regardless of their different internal controls. However, generic device interfaces incur the cost of restrictive design specifications for device drivers. Drivers for operating system do not only provide the hardware functionality. They have to comply with the generic device interface in order to be seamlessly integrated into the operating system and be compatible with its applications, see Fig. 2.6. Therefore, the generic interface with its specific functions and constraints has to be translated to device register accesses that exhibit the behavior specified by the generic interface. While a driver for bare metal firmware comprises mostly

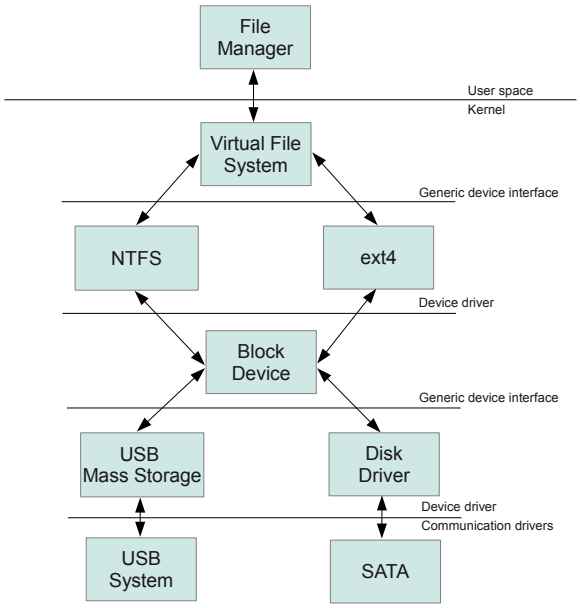

Figure 2.6: Example of driver decomposition for file systems and disk drivers on Linux.

functional specifications, a driver for an operating system also has reentrancy and timing restrictions because processes can request the driver's service concurrently and a single process cannot take over the complete system for too long. Furthermore, driver's memory and synchronization management has to occur according to the operating system's Application Programming Interface (API), as well as initialization, termination and registration. These high number of constraints imposed on driver development have made drivers highly error-prone.

2.4.1 Drivers' Reliability

Root-cause analysis made in the late 1990s has shown that system management was responsible for around 50% of system failures, while software and hardware were responsible for 20% and 10% respectively [GGP06]. The trend of using complex system management as operating system prevails. In operating systems, drivers take almost 70% of their code [CYC+01], and are responsible for up to 70% of operating system failures [Mur04]. In the HW/SW co-design area, this led to approaches for driver development that follow the earlier interface/driver co-design but concentrate on the problem of describing the device controls. Device registers are named memory addresses with a defined bit-width. They can only be accessed as a whole. But certain bits of registers can control special device behavior or inform device status independent of the remaining bits. Drivers are mostly written in the C language that provides neither a description for

these elements nor enable a direct access of bits that respect the semantics of registers [CE04]. Thus, 30% of common driver code consists of error-prone bit-operations [MRC+00]. To tackle these issues, custom languages for register map description have been proposed [MRC+00], [SYKI05]. In addition, O'Nils et al. [OOJ98] and Wang et al. [WMB03] include the description of register access constraints to avoid faulty device access. Finally, Conway et al. [CE04] also proposes the description of the device behavior through state-machines allowing driver inclusion in operating systems by mapping a generic device interface for instance. However, the proposed languages for driver development did not gain much acceptance in the industry possibly because developers struggle to adopt custom languages and because their extension to further communication systems and operating systems can be complex.

Although products must adhere to a tight time-to-market requirement to be competitive, only ad-hoc reuse techniques are applied for the development of device drivers. Device drivers are often updated to allow the control of similar devices, producing unified drivers. They reuse the driver algorithm, updating the functions that control the device. However, this often leads to driver faults. For example, if a device needs to be polled each second instead of each minute, violation of device constraint occurs; if a corner case data flow of the driver algorithm is not allowed by the device, device protocol is violated. On occurrence of these failures, the unified drivers are further adapted, risking non-optimized solutions and further faults.

Due to virtualization, reuse of drivers from foreign operating systems is possible [FHN+04], [Lev09]. This way, an existent driver for an operating system can be directly used in another. For that, a virtual machine that runs the complete donor OS provides every required interface for the driver. When service of the device is required, the driver running on the virtual machine executes the required functions and returns the data to a thin compatibility layer on the host OS that glues the software layers together. Another approach for complete driver reuse, the driver transplantation, is the implementation of a full compatibility layer that imitates a foreign operating system on the host operating system in order to run the foreign driver code. For example, the NdisWrapper project reuses Microsoft Windows networking drivers in Linux. In comparison to the reuse through virtualization, the full compatibility layer has to conform to many interfaces of the donor OS. Due to the flexibility of designs, it can be hard to always find a straightforward translation. Moreover, the OS interfaces also change with time requiring maintenance of the compatibility layer.

Another promising technique from the literature is the complete synthesis of the driver for operating systems [RCK+09]. This approach envisages a three layer architecture of the driver. The central element is the device-class specification. Similar to the generic device interface, it specifies functions generic to same device types that have to be implemented. Given this description, the functionality

described by the device-class specification has to be mapped to both device and the operating system. The behavior of device and operating system is described as state-machines which are the device and OS specifications. The device specification is comparable to the state-machine description described by Conway and Edwards [CE04] that enables the complete description of a driver for an operating system. However, the device specification of [RCK+09] treats the device directly as a software interface that has to be complemented by the earlier work on device internal description. The synthesis approach also includes an OS specification that maps the device-class specification to the behavior of the generic device interface of a target OS, also using state-machines. This tackles the underspecification problem of generic device interfaces and OS APIs in general and enables reuse throughout operating systems [RCKH09]. The problem of underspecification goes beyond the functional specification of the interface because timing and concurrency are also relevant properties for the defined functions. The synthesis approach focuses on the compliance of the driver with the generic device interface and the OS API, not with the device. In order to handle the concurrency problems, a serialization of the driver execution is proposed. Although a network and a disk driver have been synthesized for Linux, more complex drivers on other operating systems still have to be evaluated.

The reliability of drivers has also become a concern for operating system manufacturers. The quality of their products is often measured on how often their systems crash and most system crashes have occurred due to driver faults [Mur04]. Therefore, Microsoft created a certification process based on Windows Hardware Quality Labs (WHQL) testing that ensures hardware compatibility with Windows. Device drivers that pass the tests get a digitally signed certification that allows installation without user confirmation. The use of static checkers has considerably improved the quality of device drivers. In Linux, static checking had modest success because only generic programming errors were modeled to be checked. Nevertheless, Engler et al. [ECCH00] were able to find hundreds of programming bugs in open source operating systems, such as interrupt control, floating point usage and checking pointers before use. In Windows, the SLAM static analysis engine achieved higher success because the SLAM checker models the Windows API and their generic device interfaces uncovering both generic programming bugs as well as violations of these models [BBKL10]. Alternatively, the test of device drivers can also be done dynamically through tests and the usage of real devices, as is possible with Driver Verifier from Microsoft and Device Driver Testing (DDT) [KC10]. This complements the static approach, testing the driver and device interaction on the device environment. Interestingly, DDT also allows testing device drivers dynamically without the real device by providing symbolic hardware that produces symbolic values and interrupts allowing exploration of multiple driver paths. DDT found 14 serious new bugs in 6 device drivers that had already passed the Microsoft certification process [KC10].

Besides design and verification techniques, much attention has been given to the isolation of driver faults. Most techniques concentrate on isolating memory access violations through the memory protection mechanisms of the MMU. Because drivers run in kernel space, they have access to kernel memory and thus the whole system. Microkernels have targeted this issue by making most operating system services, as well as drivers, user-level processes instead of kernel space modules [LCFD+05], [HBG+09]. This way, they run in protected domains without access to kernel memory. This encapsulation ensures that a driver crash does not affect non-related system parts. However, the driver and the related activities still stop working. Similarly, Nooks [SMLE02] creates an encapsulation environment within kernel space called Nook in which the driver runs. They provide a patch of approximately 2,000 lines of code for Linux that creates the memory barrier for drivers. Microdrivers [GRB+08] allow a mixed kernel and user space approach. Most driver tasks are executed in user-space while performance critical parts are kept in kernel space. However, Microdrivers require significant redesign of OS and driver architecture. On the other hand, SUD [BWZ10] creates a Linux kernel environment for user-space allowing kernel drivers to run seamlessly in user-space, while providing memory isolation. It is based on Input/Output Memory Management Unit (IOMMU) hardware, PCI express bridges, and message-signaled interrupts. The system consists of two Linux modules of 4,000 lines of code. Virtualization techniques for driver reuse provide encapsulation and fault containment for drivers [LUSG04], [FHN+04]. Recently, software techniques that do not require a MMU have been proposed. They enable fault checking at runtime through instrumented driver code [CCM+09], [ZCA+06]. SafeDrive [ZCA+06] avoids misuse of OS data structures and buffers by annotating pointers with corresponding data structure sizes and buffer bounds. Instead of annotating the code, Byte-Granularity Isolation (BGI) [CCM+09] maintains tables of access rights for each byte of virtual memory. Every data structure and buffer has an entry in the table. Because the table is updated in runtime, control flow and temporal memory access can be controlled.

While these techniques ensure system reliability, the fault containment mechanism only tackles non-deterministic faults well. If a deterministic fault leads to a driver failure, the fault has to be detected and the driver repaired. Otherwise, every run of the faulty code crashes the driver blocking the system dependent parts permanently. Nevertheless, non-deterministic faults can also lead to component failure. In these cases, a recovery mechanism can restore the operation of the affected part of the system. For deterministic faults, they serve as a compromise if the faults occur seldom and the specific execution can be avoided. Two approaches for recovering a device driver together with its system dependent parts have been proposed. Herder et al. [HBG+07] implemented a reincarnation server for MINIX that can report malfunctioning components, such as unresponsive components or components that violate some kernel API. Upon malfunctioning

report, a policy-driven recovery takes place which defines the dependency tree of the corresponding component and the sequence for recovery. However, these policies have to be defined for recovery candidates and transparent recovery is not available yet. The components are restored to their initial state. Another approach applies a shadow driver that maintains the system functional until it recovers the regular driver [SABL06]. After the driver has been restarted, the shadow driver re-establishes the old state of the driver.

Different operating systems have different generic interfaces and provide different driver frameworks, whereas the driver's task and architecture are mostly the same. Recent driver development frameworks, such as Mac OS's IOKit framework [App07] and Windows Driver Foundation [Mic06], offer generic synchronization facilities that shift some of the driver's synchronization requirements to the framework facilitating driver development. Most industry work towards driver reliability has focused on verification techniques. Verification has the advantage of no performance penalty. However, it has been shown that faults find their way into the field. The adoption of development techniques developed in academia has not been significant. Although new development strategies can provide solutions to common development problems and tackle compatibility, they require extensive tool support. In particular, compatibility with operating systems without the support of the operating system's manufacturers can be very expensive. Generically, new development techniques are also met with skepticism by developers and do not provide an instant solution because maintenance of old APIs and code is still required. Developers and manufacturers are commonly against any approach which reduces performance hampering many techniques. For example, the synthesis technique raises the implementation level but reduces performance due to lack of control. Fault tolerance and recovery techniques impose an even higher performance penalty and have beet met with apathy from the operating system's side. Nevertheless, manufacturers of devices without critical performance requirements are already willing to accept special reuse techniques, such as standard drivers (e.g. Jungo's WinDriver), OS transplantation and virtualization. Finally, the study of drivers has recently become more popular. Eventually, developers are going to look for development techniques with reduced complexity for enhanced reliability. Approaches still have room for improvement and new techniques are still going to be developed. For example, the correctness of the device/driver interaction has not been strongly targeted yet, and is a missing link for drivers' reliability.

2.5 Summary

In this chapter, we have discussed techniques used to achieve more reliable systems. We first discussed the economic and practical roles of reliability today that are drivers for reliability engineering. Also, reliability threats, faults, errors and

failures, have been defined followed by the categorization of techniques to avoid, remove, tolerate and forecast faults. It has been made clear, that knowledge about a system's goal, structure and historical failure data is necessary in order to measure reliability and forecast faults. Historical fault/failure data allow techniques to concentrate on relevant failure modes instead of generic technological faults, such as electronic or software development flaws.

Every fault has its origin in the design and development stages. However, it is difficult to foresee a system's flaws during design, where fault correction costs the least. In particular, complex system composed of hardware and software can not be well simulated, hindering early visualization of design complications. ESL proposes a way to simulate early designs also allowing performance and quality factors to be estimated. The approach proposes a top-down design in contrast to the bottom-up design common in electronics. The idea is that a verified design made of high-level models be gradually refined to real components. Either existent components should be reused or new components developed from scratch. After an initial partitioning, software can be simulated together with the hardware in a process called co-simulation that is enabled by including the CPU internals required for software execution. An obstacle for ESL is the missing link between high-level modeling and RTL design that should be bridged by high-level synthesis tools. However, current tools do not have wide adoption yet, so most embedded system designs are based on prefabricated IPs.

Top-down design is more common in software. However, component reuse promises reduction of development cost and also enhancement of reliability, considering that the reused parts are already tested. But experience has shown that software components are seldom developed for reuse. Therefore, component-based design proposes rules for component development that requires the explicit definition of operational modes and environment requirements. In software reuse, integration testing is crucial because components are reused in environments that may not have been anticipated by its developer. Testing methodologies target mostly coverage and functionality whereas integration testing is performed mostly ad-hoc. Therefore, contracts are proposed as specification of component interaction while testing thereof tackles integration.

The hardware/software co-design area has evolved to the ESL methodology. However, a gradual refinement of the communication abstraction of the high-level design of ESL is difficult. Although synthesis approaches for interface/driver have been proposed, SoCs are largely based on standard interfaces. In addition, the underlying software layers that enable the execution of the software of the high-level models have to be developed, so I/O, drivers and operating system have to be ported to the underlying hardware for software compatibility. In particular, drivers have been recognized as the major source of operating systems' failures. Techniques ranging from custom languages for device description, driver synthesis, memory protection to driver's recovery have been proposed. The verification

of device drivers has been augmented with models of the operating system interfaces allowing the detection of many driver bugs. Some techniques for reuse also exist and have been well accepted for devices that are not performance critical. Finally, although the description of the internal control of devices has been proposed, the semantics of device/driver interaction have not been well targeted yet.

This work is concerned with systems and their reliability as an assembly of reliable components. We investigate how the contract testing methodology can be transferred to hardware and subsequently to drivers and which advantages it has. In the next chapter, we present an approach to transfer the contract testing technique to hardware components.

3 Contract Specification for Hardware Interoperability Testing and Fault Analysis

3.1 Introduction

While hardware systems are becoming increasingly complex, short time to market implies both the use of prefabricated components and a high degree of flexibility. In particular safety critical tasks increasingly depend on complex hardware, posing additional reliability constraints. This is addressed by a bottom-up design [BG06] using reliable components. Furthermore, extensive functional testing [IKK+07], and reliability modeling [Pd05] in the design phase confirm a certain level of fault-free operation. In the case of safety critical systems, in real-time behavioral testing becomes necessary to avoid hazard behavior [KDK06]. Although experience shows that generally external or environmental reasons like noise, component aging, and faulty interoperability are responsible for failure, they are not explicitly tested [STZ08]. In addition, while failures are typically detected, fault localization requires the analysis of error propagation which is often not provided [SS04].

In software, similar questions of reliability often arise. The strategy of contract testing is presented as a suitable method for component interplay, and for external service compliance [Gro04]. In a component contract, interoperability conditions are defined under which the component service is guaranteed. Through constant monitoring of those conditions, a component can detect faulty usage. Moreover, this leads to the localization of the fault agent, a faulty component.

In this chapter, we investigate the transfer of this methodology to hardware systems, formalizing component specification. We show that, similar to software systems, we are able to identify faulty component assemblies using hardware monitors. Furthermore, we categorize signal faults allowing the discovery of evidence for environment related failures, tackling the fault localization problem. For an example of a communication system, we specifically define contracts, and show comparisons between contract violation, fault categorization, and failure occurrence under signal fault injection.

In the next section, we give a survey of the related work on which the work in this chapter is based. In Section 3.3, we explain the general contract, and contract

testing methods, followed by the specification of hardware contracts. After that, the requirements for contract testing and fault categorization are defined. Our case study of a communication system with its results is presented in Section 3.6. Discussion of method is carried out in Section 3.7, after which the conclusion follows.

3.2 Related Work

Contracts are a formal specification to develop reliable systems from reliable components following the principles of software engineering [Mey92]. A contract specifies component interaction. They manifest as clauses called pre- and postconditions. Preconditions define what the component requires for correct operation, while postconditions define what a component guarantees to deliver. The contract testing methodology spans the contract specification, and its tests.

Contract specification has been embraced by software specification methodologies such as KobrA [ABB+02], [Gro04], [SPB+06]. First attempts to extend this concept to hardware assumed that hardware can be abstracted to enable the specification of contracts in a software way, not really addressing the hardware perspective of contracts nor their testing [BG06], [Jan03]. Hardware contracts were first formalized by [Kam07], addressing the timing problem of pipelined designs. Based on its specification, they automatically generate functional tests for a model of a microprocessor [Kam08]. However, they do not address correctness of component interplay, which is the aim of the contract testing methodology [Gro02].

Contracts assume a black-box view of components whose service is guaranteed by fulfillment of their preconditions. In addition to software contract preconditions, like the order of data input, hardware has to care about communication protocols, its signals, and especially timing. Therefore, contract testing requires the verification of communication protocols.

Bus monitoring methods access the link information for analysis of protocol faults. Some methods apply behavioral simulation allowing for flexibility of test cases and fault injection. However, modeling the environment is very difficult for simulation methods. Moreover, to detect and isolate faults caused by the physical environment or sporadic component misbehavior, hardware interplay has to be monitored in real-time. For that, usually the analysis of frames is performed to check for protocol conformity [CPC03], [BGF03], [ASH+04]. A higher fault coverage is achieved by analyzing every bit of a transmission which allows detection of binary faults such as glitches [PHZ+05], [ARSH05]. Even higher fault coverage is achieved by mixed-signal verification because complicated interactions between analog and digital components can be then verified [BCMS05]. However, this technique is only applied in simulations. A comparable monitoring implementation generates too much data to be processed for fault detection.

A solution for the amount of monitoring data is pre analysis of that data with categorization of signal faults as deviations of a defined signal behavior (physical level). Moreover, categorized signal faults give evidence of the fault cause. Crossman et al. [CGMC03] relate specific behaviors of sensor signals to specific faults for example. These signal features or faults are abnormal magnitudes, rolling, noise, and dependency faults.

To specify hardware contracts, we adapt the contract definition of [Gro04] to include common hardware requirements. We focus on interoperability testing by monitoring component communication as well as tackling the analog behavior, using mixed-signal verification. This focus allows us to measure divergences of the component input from its input specification, permitting fault localization by considering electrical faults from conductor, external noise, and component degradation. We categorize these faults based on signal faults, similar to the categorization from Crossman et al. [CGMC03]. However, dependency faults are not relevant for communication, and noise is not singularly characterized by our model. Instead, bit delays and glitches are considered.

3.3 Contract Testing

Contract testing is developed from the idea that the system mission is only accomplished if all components operate correctly together [Gro04]. A hardware system is assembled from many prefabricated components, glue logic, and some custom modules. Correctness of integrated components is trusted in a component-based design. Therefore, components are considered black-boxes. Instead of testing their inner behavior, contract testing verifies the compatibility of a component with its system by monitoring fulfillment of component requirements.

Often an integrated component does not fit completely to system requirements. It requires inclusion of glue logic to adapt to the target implementation, which restricts the operational range of the resulting system. In corner cases, the component requirements might be barely ensured, resulting in the delivery of unusable services. A deficient service can manifest as failures throughout the system. In the contract testing method, the component monitors its specified requirements. In addition, other components interacting with it, monitor if its services comply with their requirements. This way, faulty components can be localized, and faulty services isolated.

The specification of component usage and requirements is summarized in a contract. It formalizes the operation between components. Every component offers a number of functions or services for which a contract is specified with the conditions for use of the service. For every service, two roles are differentiated: the client, and the server. Clients have to comply with certain conditions to acquire a service; in a contract, these are called preconditions. In addition, a contract specifies under which conditions the service is delivered. It describes not

only the functionality of the service itself but also how the service is delivered; these are postconditions. Pre- and postconditions of a service build its contract. A component's contract is the compilation of the contracts for every service it delivers.

In the following, we define a contract formally.

- a *contract* C is a set of *conditions* C_i:
  - $C = \{C_i \mid i \in I\}$ that can be subdivided into:
    - *Preconditions*: $C_{pre} = \{C_i \mid i \in I_{pre}\}$
    - *Postconditions*: $C_{pos} = \{C_j \mid j \in I_{pos}\}$
- Each *condition* C_i is a clause that is true or false:
  - $C_i \rightarrow \{true, false\}$

For a *system function* $F \subset I_{pre} \times I_{pos}$:

- the correct operation of a *function* $f_i(C_{ipre}, C_{ipos})$ has as necessary *conditions* that $C_{ipre} \wedge C_{ipos} = true$.

Given this contract, interoperability can be defined by considering the contract of a *system* S, which consists of *components* $S(i), i \in I_s$.

- For this system, the component interaction $R \subset I_s \times I_s$ defines a component set-up.
- For a component set-up $S(i,j) \in R$, the set-up function $F_{ij} = F_i \cup F_j$ is defined.
- For the component set-up the contract between the two components, $C_{ij} = (C_{ipos} \cap C_{jpre}) \cup (C_{ipre} \cap C_{jpos})$ is defined.

In the example of a counter $S(i)$ with its outputs connected to a display's inputs $S(j)$, the component set-up $S(i,j)$ is given, where $S(i)$ provides service to $S(j)$ (i.e. counter drives signals of the display's input). For this set-up, the contract $C_{ij} = C_{ipos} \cap C_{jpre}$ is valid if the preconditions of component $S(j)$ are fulfilled by the postconditions of component $S(i)$, i.e. $C_{jpre} \supseteq C_{ipos}$. This result shows a case where only one part depends on the other, simplifying the interoperation contract. This result is valid for the example because the counter does not depend on services from the display, and we neglect the power supply requirements of the counter.

In addition, in case of a hardware system, the hardware components can lose functionality due to aging. This behavior asks for a continuous measurement of the contracts. In particular unequal component aging leads to fault propagation because of degraded service constraining compliance with preconditions of clients. Contract testing avoids this, and localizes faulty components (i.e. old components).

3.4 Hardware Contract

To define the contract conditions for a component, information regarding its interface behavior and conditions of operation has to be gathered. In the case of hardware, product datasheets contain detailed information about components. For every possible operating condition, component behavior is described. Despite datasheets containing sufficient information, contract testing requires a common format for contracts to allow monitoring and testing. To specify such a contract, we categorize operability constraints from hardware to systematically characterize a precondition table in a standard format.

Digital components have analog implementation, which implies operating conditions for correct behavior. We categorize the four following constrained types for digital components: ***environment*** requirements, ***input levels*** of signals, ***timing,*** and ***logic set*** restrictions. Electronic components have different operating points, depending on their supply, temperature, or input ranges of signals. Furthermore, gain compensation of high frequencies leads to slew rates, creating propagation delays for digital outputs. Some components have undefined behavior for certain input sets. For example, a Set-Reset (SR) flip-flop should never have both Set (S) and Reset (R) inputs set at the same time, and communication systems only accept certain data frames ignoring everything else.

A systematic definition of the contract results from the analysis of the constrained types. Environmental conditions are defined first, and then input and timing restrictions, because both timing and input restrictions are environment dependent. Input levels and timing constraints are interdependent, and associated with each other in the contract. That is, timing information determines when input restrictions have to apply. The subset of the input combinations, which is accepted by the component, composes its logic set. This restriction applies only when all other constraints are held.

3.4.1 Hardware Contract Constraints

- **Environment** restrictions: voltage supply, current sink, and temperature determine the operating point of the analog implementation of a digital component. Outside of the specified values, the behavior of the digital component cannot be guaranteed.

- **Input levels** of signals are restricted by the logic levels. Depending on the technology, certain ranges are considered to be logic 0 or 1. Values outside of the ranges propagate through the system, potentially leading to a failure.

- **Timing** restrictions can be established based on 2 premises.

  - Logic components have propagation times (Fig. 3.1). This demands that the component input be stable for the propagation time so that

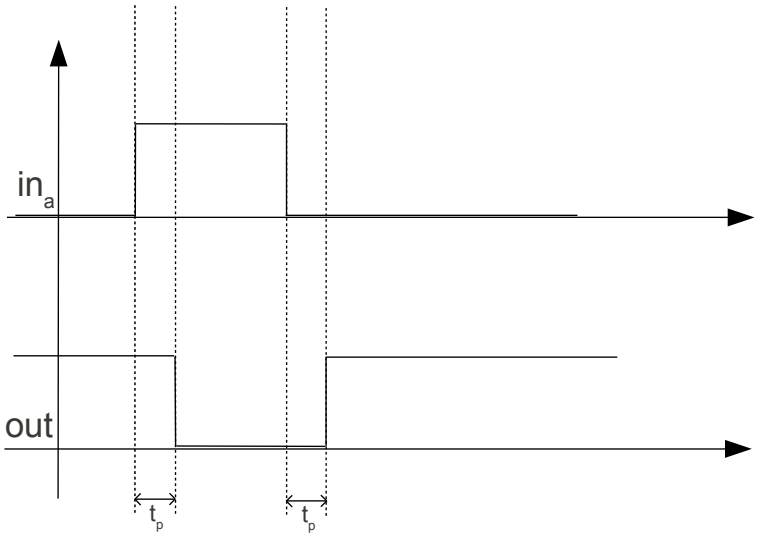

Figure 3.1: Propagation delay of an inverter.

its output can be fully driven. Stable means no logic level violation by the corresponding input; thus the component can successfully complete operation.

- Some inputs of memory elements define time windows, for which the other inputs are accepted (Fig. 3.2). Inside of time windows, inputs have to remain stable so that the circuit can complete operation.

- **Logic set** restrictions have to be defined for components which have undefined behavior for certain input sets. For example,
  - SR flip-flop should never have both S and R inputs set together; and
  - the communication system, which defines a specific frame with a specific length and characterization of start and end of frame, might not accept different frames.

3.4.2 Hardware Contract Example

To provide a concrete example of a hardware contract, we are going to carry out a contract specification based on component behavior, and its datasheet information. We will describe contracts for a Negated AND (NAND) gate, and a Data (D) flip-flop, showing how contracts scale for different complexity levels. The 4^{th} contract constraint category, logic sets, does not apply for these examples, because their output does not depend on different sequences, and will thus be treated later in our case study. The contracts for the NAND Gate, and the D flip-flop are defined in the Tables 3.1, and 3.2 respectively.

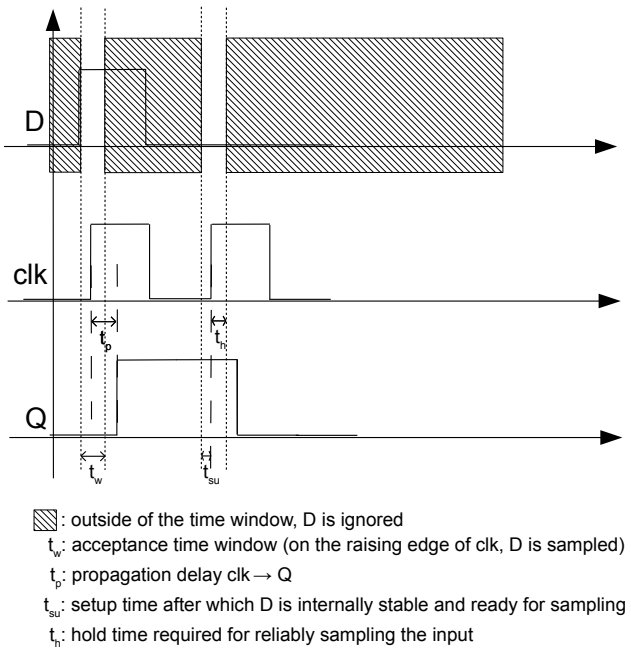

Figure 3.2: Acceptance time window t_w and propagation delay t_p of a D flip-flop in contrast to setup t_{su} and hold times t_h. D must be held at a constant logic level throughout the acceptance time window.

A NAND gate ideally compares its inputs instantaneously, producing a negated output of the binary product of its inputs. In the contract, this behavior is affected by a propagation time, and is expressed in the form of a postcondition, which considers this time. The preconditions are expressed in form of the input and timing information extracted for one mode of operation, which is represented by environmental constraints. The timing constraint of the NAND gate's inputs is dependent on the propagation time only. Combinational logic, i.e. components without memory, does not form time windows, thus the timing constraints of every input are independent from the signal behavior of any other input. On the other hand, because the D flip-flop is composed of sequential logic, the time constraint of input D is dependent on the signal behavior of the input *clk*. This forms a acceptance time window that has to be specified by a a time reference and a corresponding duration. Specifically for the D flip-flop, the rising edge of the signal *clk* depicts the time window for which the D signal is processed by the circuit. Therefore, the timing constraint of the input D has to be defined in reference to the *clk* input, as in Fig. 3.2. Finally, the timing constraint of the input *clk* depends on the propagation time only, as for every NAND gate input.

In the D flip-flop contract, Table 3.2, the acceptance time window of signal D is derived from the times t_{su} and t_h. While the acceptance time window is here no different than the well known setup and hold times, we also represent signal

NAND gate 74LCX00	
Datasheet	**Contract**
Behavior: $o(t) = \neg(a(t) \wedge b(t))$	Postconditions: $o(t) = \neg(a(t - 5.2ns) \wedge b(t - 5.2ns))$
Constraints: -temperature range: -40 *to* $85°C$ -Power consumption: $< 0.55mA$ -voltage/current supply: $2.7 - 3.6V$ -Analog condition for bit = 0: $0 - 0.8V$ -Analog condition for bit = 1: $2V - V_{cc}$ -propagation delay of $5.2ns$	Preconditions: **ENVIRONMENT:** -temperature range: -40 *to* $85°C$ -Power consumption: $< 0.55mA$ -voltage/current supply: $2.7 - 3.6V$ **INPUT LEVELS:** -Analog condition for bit = 0: $0 - 0.8V$ -Analog condition for bit = 1: $2V - V_{cc}$ **TIME:** how long: -input has to be stable for minimum $5.2ns$ when: -at all times

Table 3.1: Contract specification for a NAND gate.

dependencies of communication systems/protocols with it that are unrelated to setup and hold times, in our case study for instance. Thus, we apply the name acceptance time window to cover a broader view of signal interdependency.

3.5 Contract Testing in Hardware

Contract testing identifies contract violations, reporting potential system interoperability problems. Every component has to ensure its operability by checking the fulfillment of its preconditions. If a precondition is not fulfilled, then the provider of that service has an interoperability issue with the checking component. Contract testing includes the process of checking the preconditions of a component's contract. To enable contract testing for a component, its contract specification and special monitoring circuits are built in the component. In this way, the component accesses the external parameters, and compares them with its contract, identifying violations.

Because our approach aims at execution of tests in real-time, real-time measurement, and real-time processing are essential. Each categorized hardware constraint applies different requirements to measuring techniques. Environment variables like temperature, voltage supply, or current drain do not change fast; thus acquisition with low sampling frequency is acceptable, compared to the system clock. Because the propagation times and the time windows are short, input levels and timing requirements need a high sampling frequency which is typically

D flip-flop 74HCT74	
Datasheet	**Contract**
Behavior: a D flip-flop just accept input when the clk input switches from 0 to 1 $o(t) = \begin{cases} i(t), & \frac{\partial clk(t)}{\partial t} > 0 \\ o(t), & \text{otherwise} \end{cases}$	Postconditions: $o(t) = \begin{cases} i(t - t_{pr}), & \frac{\partial clk(t-t_{pr})}{\partial t} > 0 \\ o(t), & \text{otherwise} \end{cases}$
Constraints: -temperature range: -40 *to* $125°C$ -Power consumption: $< 0.57mA$ -voltage/current supply: $4.5 - 5.5V$ -Analog condition for bit = 0: $0 - 0.8V$ -Analog condition for bit = 1: $2V - V_{cc}$ -t_{pr} propagation delay of $53ns$ -t_{su} set-up time of $18ns$: D to clock -t_h hold time of $3ns$ clock to D -maximum pulse frequency $18MHz$ -clock pulse width of $27ns$	Preconditions: **ENVIRONMENT:** -temperature range: -40 *to* $125°C$ -Power consumption: $< 0.57mA$ -voltage/current supply: $4.5 - 5.5V$ **INPUT LEVELS:** -Analog condition for bit = 0: $0 - 0.8V$ -Analog condition for bit = 1: $2V - V_{cc}$ **TIME:** data: how long: -minimum $t_{su} + t_h$ when: -t_{su} before clock switch from 0 to 1 and t_h after clock switch clock: how long: -minimum $27ns$ high and low when: -at all times

Table 3.2: Contract specification for a D flip-flop.

higher than the signal clock because one wants to identify the slope behavior of the signal. On the other hand, logic sets do not have to be monitored analogically. Instead, a logic test as for software is sufficient, and possible using logic gates.

The environmental characteristics, temperature, supplying voltage, and power consumption can be compared to the contract upon acquisition, and deviations can be reported. The amount of monitored data of time windows and logic levels is huge, and has thus to be compressed to guarantee real-time feedback time. This is realized by a triggering and filtering solution. As we are only interested in the active data transfer, we trigger the acquisition only when data are transmitted. Filtering then processes these data, and has the option to categorize faults.

3.5.1 Fault Categorization

In digital systems, data are clocked; and for every period, there is a time window where the digital inputs have to be stable. Although stability guarantees correct interpretation, it does not ensure correct transmission of information. On the other hand, if the signal changes between time windows are evaluated, e.g. signal evaluation of the complete frame transmissions, faulty signal behavior affecting bit sampling can be recognized. If changes occur much ahead of the time window, and if its voltage level is stable afterwards, the probability of correct transmission is high.

Hardware components interact based on interface protocols defined by the constraints, logic set, timing, and input levels described earlier. Techniques to en-

hance realiability of data transmission for protocols have been investigated, such as monitoring. Prototype based communication monitoring techniques [PHZ+05], [ARSH05] categorize bit faults, glitches, and delays. Crossman et al. [CGMC03] define possible signal faults of sensors, as abnormal magnitudes (voltage levels), rolling (slope times), and noise and dependency faults (context dependent). We categorize signal and bit faults related to digital hardware communication as a combination of both. A digital signal is ideally represented by two voltage levels, with instantaneous switching between both levels. For a real electronic component to drive its output from one logic level to the other, the resulting signal has a ***slope time***, which can be measured as a time degraded behavior. The same applies for ***delays***, which represent the response time of a component. Degraded ***voltage levels*** are variations of the output voltages for the logic levels approaching its boundaries, while ***glitches*** are voltage pulses of short duration resulting from interferences from outside. These four signal and bit faults of digital signals are used as quality measurement of the operation of a component interface.

- Reduced **voltage levels** indicate degraded component supply or high load on the line.
- **Slope times** measure the ability of the circuit to work up to a certain frequency.
- **Delays** are a measure of the response time of the component.
- **Glitches** gives a measure of the interference or noise on the system.

These are also direct effects of circuit faults, and can thus be used to find the causes of failure. In addition, they can be detected with the same measurements needed for contract testing.

3.5.2 Fault Diagnosis

Although contract violations indicate malfunction of the system, finding the cause thereof is typically not provided. Assuming correct logic operation of the system, violations of contracts happen due to external influences like temperature, power supply fluctuation, and underlying structure or interference changes. Upon aging, components fail more often. In that case, the operation is not ensured, i.e. the postconditions might not be held even on fulfillment of all preconditions. These are reasons for a circuit fault. They have to be first located so that they can later be isolated and repaired.

A faulty component can be found by elimination. On a sequence of components tested for contract, the component not fulfilling the precondition of a client is faulty given fulfillment of its precondition. Furthermore, circuit faults mentioned above also relate to the environment, temperature, power supply, PCB condition,

Causes/Circuit faults	**Effects/Parameters**	**Cause detection by contract-testing**
Supply Problem	Voltage & Current Supply	yes
Excessive Load	Voltage Level	yes
Temperature out of Operating Range	Temperature	yes
Excessive Wire Load/Exceeded Fan-Out	Slope Times	yes
Faulty Conductor		no
Faulty Component	Delay	no
Faulty Conductor		no
External Interference/Noise	Glitches	no
Faulty Conductor		no

Table 3.3: Fault diagnosis: Fault causes, its effects, and detection by contract testing.

cables, etc. The environment can influence components to misbehave. Thus, detection of these circuit faults is especially relevant for system recovery.

Considering a generic electronic component equipped with contract testing connected to a system/other components through wires or tracks of a PCB, we relate our measurable effects to a set of external factors leading to generic circuit faults in the Table 3.3. Component fault is also listed, because component misbehavior is categorizable with our fault set. Half of the defined external factors are measured by contract testing: temperature, power supply, and load. For the external causes measured indirectly that lead to a fault, the parameters of fault categorization serve as indicative factors although the relation of the parameters to external causes leaves uncertainty. For example, deviations in delay, slope times or glitches can be the effect of multiple causes. Moreover, a faulty conductor can result in glitches, signal absence, increased slope times or delays or a combination of them all. Because we consider a generic electronic component, we have only associated possible causes to the effects that we can measure, without defining combination of effects leading to causes or quantifying probabilities of causes to measured effects. Table 3.3 only provides an approximated diagnosis for generic electronics using contract testing. Nevertheless, this approximated diagnosis already enables reactions to contract violations because it narrows down their possible causes. Furthermore, more precise diagnosis is possible if we consider a specific system. A system fault model can be created based on historic fault data and failure modes analysis. Failure mode analysis reduces possible fault outcomes due to system construction, while statistical data further rate the most probable causes. However, historic data and the analysis of failure modes require a specific system and its long term evaluation which we do not pursue in this work. Nevertheless, the measurable effects presented on the table serve as evidence nodes of these system reliability models.

3.6 Case Study

Our case study is based on an I^2C bus which is extensively used for Integrated Circuit (IC) communication on PCB. It allows masters to set and retrieve register values on components connected to the bus. Correctness of component interplay on PCB, as in embedded systems, is seldom assured. The protocols used in its communication, such as I^2C, have neither error correction nor detection codes. Moreover, protocol compliance is not verified. However, we address constraints of hardware interaction based on communication protocols in this chapter. With them, every communication standard can be verified for compliance.

The contract specification for this system consists of two steps. First, the I^2C standard is analyzed to define the rules for the inputs and outputs of the bus from and to the system. Then the product datasheets of the components, bus repeater (i.e. line driver), and bus controller are interpreted for further constraints.

For the case study, we develop built-in circuits to allow contract testing, i.e. monitoring of contract conditions and fault categorization. The difficulty in implementating testing circuits is variable depending on the speed of the monitored protocol. Our implemented circuits are not optimized. They serve however as proof-of-concept. Applying contract testing with the built-in circuits, we perform fault isolation and fault categorization for two different tests.

The first test shows the capability of the circuit to detect contract violations, and categorize errors. Here, we explicitly test compliance with component requirements detecting contract violations. Contract violations are typical interoperation failures, such as the fault effects defined for communication by Sosnowski et al. [STZ08]. For our definition, we focus especially on the following failures. Components, working at their limits, lead to a behavior incompatible with the preconditions of the clients. PCB conditions lead commonly to contract violation. For example, the communication medium fails because it is overloaded, and protocol violations occur due to high bus usage or inter-component conflicts. External influences are also a typical case of protocol failure, such as the noise generating glitches being targeted here.

In addition, the first test validates the fault categorization. The categorized faults are the fundamental effects leading to faults on communication. Other approaches only detect bit error, wrong frame composition, or failed transmissions. While that enables failure detection and isolation, it does not analyze the effects leading to them, preventing fault diagnosis. The categorization of errors which occurred indicates possible circuit faults. This assists system maintenance, accelerating fault removal.

The second test demonstrates fault isolation. Erroneous client behavior that certainly leads to wrong data delivery, and thus system failure, is emulated. The data interpreted by the bus controller are logged for both cases: (1) without contract testing; and (2) with contract violation assessment and service denial.

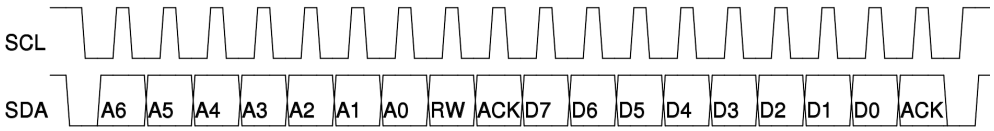

Figure 3.3: I²C Transmission — SCL: Clock signal — SDA: Data signal.

3.6.1 I²C Contract Specification

I²C Bus

I²C is a communication standard [Phi00] which specifies how the communication between nodes is established. An I²C bus is synchronous. Its communication is a 2 bit wide serial packet composed of a Clock Line (SCL), and a Data Line (SDA). The specification [Phi00] consists of a transmission frame (Fig. 3.3), along with bit transmission rules.

I²C Master Role

The I²C controller is configured as a bus master. The master starts every transmission, which can be write or read requests. Because the contract is made between the I²C master and the bus, and the controller operation is not affected by write requests, write transmissions from the master do not concern this contract. A write request requires acknowledgement of the addressed node, but this acknowledgement does not influence controller operation. That is, the transmission takes place normally, only that acknowledgement miss is asserted. On the other hand, read requests require input data, which has to be interpreted by the controller so that it fulfills its operation of receiving data. Here, the contract testing has to assure that the data bit transmission rules are being followed. The I²C standard defines that the data line shall not be toggled while the clock line is high. This bit transmission rule can be directly translated to a timing constraint in our contract as a time window.

I²C Communication System [1]	
Datasheet	**Contract**
Behavior: **Component:** The I²C communication controller receive the data sent through the bus, which attain to the I²C standard. It transmits a waveform, as in Fig. 3.3, where the SCL has a high to low rate of 3/2. On data reception, position 11th to 18th of the frame, it samples the data at t = 1/5 of SCL period to rising edge of SCL. -SCL is set to 100kHz -Samples of data are sampled 2 s after rising edge of SCL **Communication standard:** Given an valid input set, correct data can be acquired on the high level of the SCL signal.	Postcondition: **Component:** The output data correspond to the data on the bus to the sampled times. **Communication standard and Component:** The output data correspond to the sent data.
Constraints:	Preconditions:

[1] Table continues on the next page.

Constraints: **Component:** -temperature range: 0 *to* 85°C -Power consumption: $< 10W$ -voltage/current supply A: 1.14 *to* 1.26V -voltage/current supply B: 2.375 *to* 2.625V -voltage/current supply C: 1.140 *to* 3.465V IOB configured as LVTTL -Analog condition for bit = 0: $0 - 0.8V$ -Analog condition for bit = 1: $2 - 3.6V$ SDA: -tsu, 0.738 ns before clock edge -th, 2.762 ns after clock edge SCL: -tpr, 7.042 ns Communication standard: SDA is valid for the high level of the SCL signal Input set: S = {ST: start bit SP: stop bit 1: data = 1 0: data = 0} $x \in 1, 0$ Read sequence = {ST, x, x, x, x, x, x, x, 1, x, x, x, x, x, x, x, x, x, x, SP} Write sequence = {ST, x, x, x, x, x, x, x, 0, x, x, x, x, x, x, x, x, x, x, SP}	Preconditions: **Component:** **ENVIRONMENT:** -temperature range: 0 *to* 85°C -Power consumption:< $10W$ -voltage/current supply A: 1.14 *to* 1.26V -voltage/current supply B: 2.375 *to* 2.625V -voltage/current supply C: 1.140 *to* 3.465V **INPUT LEVELS:** -Analog condition for bit = 0: $0 - 0.8V$ -Analog condition for bit = 1: $2 - 3.6V$ **TIME:** Given a read sequence, positions 11th to 18th in the sequence have the following time constraints: SDA: how long: -minimum $t_{su} + t_h$ when: -t_{su} before sample trigger -t_h after sample trigger -Sample trigger: $2s - t_{pr}$ after rising edge of SCL **Communication standard and Component:** **ENVIRONMENT:** Component **INPUT LEVELS:** Component **TIME:** Given a read sequence, positions 11th to 18th in the sequence have the following time constraints: SDA: how long: -minimum time $6s$: SCL at high level when: -starting from SCL rising edge **LOGIC SET** -not required, since these are generated by the own component

Table 3.4: Contract for a I^2C master node.

I²C Communication System

To define the contract specification of the system, we have to analyze how it is implemented, and get the contract information out of each of its composing parts. The example system uses the I²C communication controller from OpenCores [Her01], implemented on a Xilinx FPGA Spartan3A DSP 1800A. Therefore, environment requirements and input levels are derived from the FPGA specification. The placement and routing of an FPGA design determines the propagation delays of the circuit on the configured FPGA. These delays are reported to the user by the place & route software. The time windows for sampling the inputs are given, first, by the IP core itself when it occurs (for the start of the window), and then by the same FPGA integrated flip-flops (for the end of the window) because they determine how long it has to be sampled. Finally, the accepted logic sets depend only on the IP core implementation.

Because we aim at communication correctness, the communication contract has to be added to the component contract to assure it. The component contract claims that, given its preconditions, its output data accord to the sampled data of the input. On the other hand, the communication contract affirms that, given its preconditions, the read data are considered to be the sent data. With this acknowledgement, we can say that, if communication and component contracts are fulfilled, the output data can be considered to be the sent data. Table 3.4 shows the contract for this communication system, distinguishing the component contract from the communication system contract where relevant.

3.6.2 Built-in Contract Testing

Contract testing is composed of the measurement part, and the comparison against the contract. In our case, an FPGA implementation is used due to the flexibility of building dedicated hardware, and the real-time requirement to process the fast sampled data. Analog-to-Digital Converters (ADCs), and filters are stand-alone components on the board, which are connected to the FPGA, inside which the processing units are implemented as dedicated hardware. Fig. 3.4 contains an overview of the architecture.

The use of an open source communication controller allows us to modify it to inform the acquisition module about frame transmission. This is used to trigger the system to filter out irrelevant data on the bus, when no transmission is taking place.

Measuring & Comparison

Two different acquisition and comparison strategies are used, one for environment, and another for input level and timing variables. For the environment variables, a single IC with a temperature sensor, and an eight-channel ADC is used. This IC is

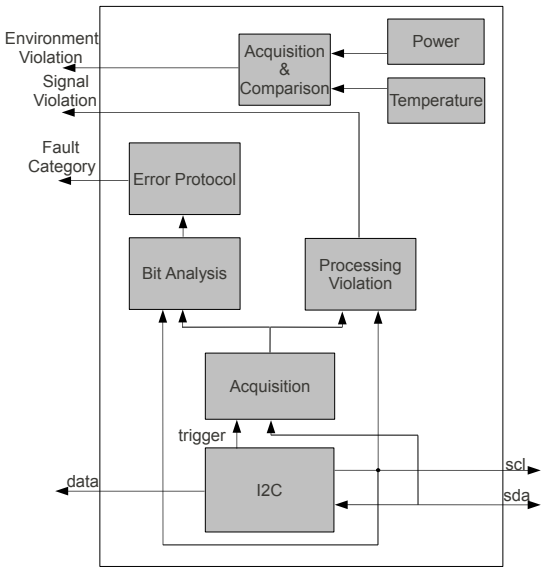

Figure 3.4: Block diagram of the I^2C communication system with contract testing and fault categorization.

initialized to run each second in a round-robin fashion, acquiring the temperature and data from all its inputs. Three different voltage supplies together with a measure of the total drained current are connected to this ADC. Due to the high time constants related to these values, no anti-aliasing filtering is necessary. Comparisons of the temperature and the power supply with the contracts are made with the same frequency with which the data are acquired. Each second, a module fetches the measures from the stand-alone ADC and temperature sensor, and compares them with the contract values informing physical violations.

For the time and input level contract parameters, dedicated signal acquisition hardware controls a 32MHz 8-bit ADC. Furthermore, an anti-aliasing filter is used, as well as voltage adaptation to the ADC input range, so that the bus signal can be acquired. Having the ADC driver inside the FPGA IC, the trigger signal of the communication controller is directly connected to it. In doing so, the modules, processing violation, and bit analysis have information about when a bit begins and ends. This approach enables the detection of the sequence position by counting clock toggling, which is needed by the time and input level contract clauses. When the processing violation module starts to receive data from the acquisition module, it first counts the rising edges of the SCL. As defined in contract, from count values of 10 to 17 while SCL is high, the input data are monitored not to trespass its input level. On trespassing of allowed input level, signal violation is asserted.

Categorization of Fault

On top of the acquisition module, a bit analysis module is designed to extract our categorized signal parameters. These are called faults or signal faults in the following sections. These extracted parameters are compared with contract limits on the error protocol module. If the parameters exceed the limits, an error is asserted in a given protocol. This error is forwarded to the component user as output of the component.

Bit Analysis The bit analysis module extracts the signal faults from the digital signal converted by the ADC. The output of this module, for every transmitted bit, consists of the values for the previously categorized parameters: voltage level, delay, rise time or fall time, and glitches.

The recognition of the bit boundaries, start and end of bit transmission, is necessary after the signal acquisition. To find the bit boundaries, a clock signal is used. In the synchronous case of the I^2C controller, its own clock signal is used. It is connected to the processing violation and bit analysis modules of our design.

Besides the information of the bit boundaries, we detect bit flip and glitch occurrences. Having this information, the parameters of Fig. 3.5 can be calculated from the time stamp relations, and the digitalized signal levels. Rise and fall times can be derived from the time it takes for the bit flip to fully complete. The delay value is assumed to be the time it takes for the bit to start flipping after bit middle (clock bit flip divided by sampling time). The voltage level is the voltage average after flipping, or after bit middle, in case of no bit flip. To be able to determine a glitch, we store its time stamp and voltage level upon occurrence. As we cannot know how many glitches occur, a glitch count is present to give an idea of how much environmental interference there is on the bus.

Glitch and bit flips are detected in a sub-module of the bit analysis module, independent of the bit boundaries. Bit flip detection is based on two voltage limits: a lower, and an upper. On initialization, an arbitrary bit level 0 or 1 is assumed. On bit level 0, a flip is detected when the voltage exceeds the lower limit. It is then confirmed if it trespass the upper one. It works vice-versa for the bit level 1. Glitch detection uses minimum voltage tangent, and maximum step value to filter out small voltage variations and bit flips, which also are a sudden change of voltage level. The input signal is first derived, and a glitch start is signalized upon higher tangent than the given parameter. When the tangent value is again lower than the defined minimum, the voltage step between these two time points is compared with the maximum defined step. If the step was lower than it, a glitch is confirmed.

During the glitch evaluation, the time between glitch start signaling and glitch confirmation, the voltage sample with the maximum voltage difference to the voltage at glitch start is stored. Glitch duration is counted as the number of samples which belonged to the glitch.

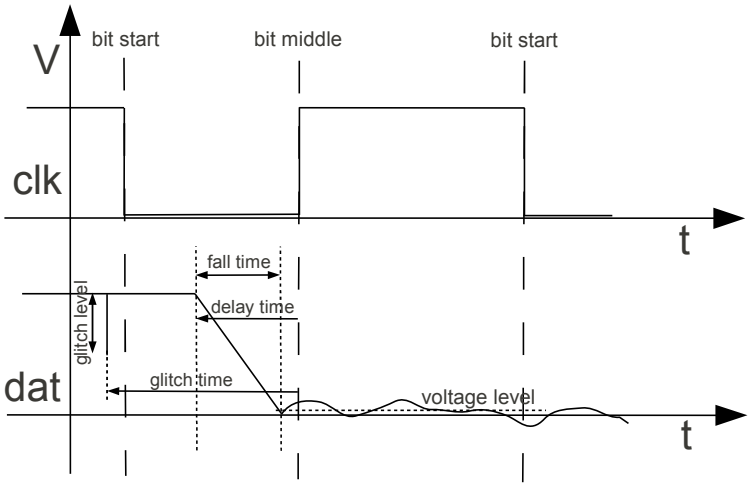

Figure 3.5: Signal parameters of a bit window categorizable by our fault analysis.

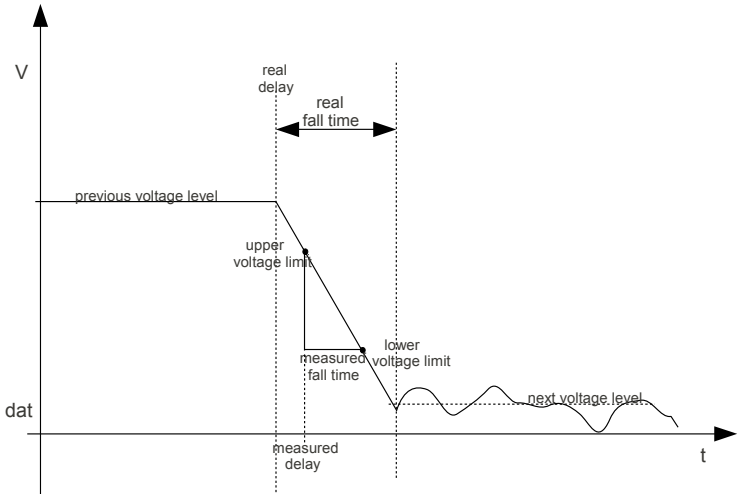

Figure 3.6: Linear approximation of the bit flip based on threshold measured times.

Based on the stored glitch information, and the detected flips, the time parameters are measured in relation to the bit boundaries. At bit start, all previous parameter outputs and counters are reset. A counter starts again from zero, and several time stamps are marked, on arrival of the bit middle, at the start of a later confirmed flip, on the first confirmed glitch, and at the end of the bit. Between flip recognition and flip confirmation, another counter is started and stopped for the slope interval count. If flip is confirmed, the counter value is saved to the slope interval output parameter.

Still inside the bit scope, the voltage level is stored as the sum of all the voltage levels either after clock or after flip, with priority to flip. Together with this sum, a count of how many elements have been added is also stored. The complete output of the bit analysis module is then composed of four items.

- **Voltage level:** voltage sum and number of added elements.
- **Delay:** count mark for start of confirmed flip, and count mark for bit middle.
- **Slope time:** count of the interval between start and confirmation of a flip, and total bit window count (i.e. amount of samples in a bit window).
- **Glitches:** first glitch start mark, glitch level, glitch interval count, and glitch count.

The output of the bit analysis module is moved to the error protocol module where the signal fault degrees are compared against time and voltage limits.

Error Protocol In this module, we treat all bit fault information of a transmission. Each bit fault information is compared with fault limits. If any violation is found, the error protocol is filled with the found fault. This error protocol is then forwarded to the output of the component.

For comparison, the module's input data have to be preprocessed. We divide the voltage level sum by the voltage level count to compare the average voltage level of the transmitted bit with the accepted logic levels. The rise and fall times are retrieved from the slope interval input. But to know if it has been a rise or a fall time, the voltage level of the previous bit is used.

We compare the time input values in a normalized manner. The absolute discrete time (counter values), stored by the bit analysis module, changes with the bus operating frequency, and the ADC operating frequency. Thus, we normalize it dividing time marks by a time mark corresponding to the bit length, allowing us to compare time values independently of acquisition rules or bus operation frequency. Similarly, the delay is normalized. Relative to the time to the bit middle, as in Fig. 3.5.

Fault category	Assumed standard value	Violating value
Voltage level		
Low	$0V$	> 0.8
High	$3.3V$	< 2.0
Delay	$-20,00\%$	$> 0\%$
Slope time	$1,00\%$	$> 26.4\%$
Glitch $[0\% - 60\%]$	$0 - 20\%$	$> 25\%$

Table 3.5: I^2C communication standard and contract violating values for signal parameters.

We apply corrections to delay and slope time. Slope time is measured only between the limit parameters for high and low bit. Delay time is also marked on start of flip recognition being also the upper or the lower limit for the flip, instead of last voltage level as represented on Fig. 3.5. We assume a linear bit flip, and calculate its gradient by dividing the difference between the voltage boundaries for the flip recognition by the measured slope interval (Fig. 3.6). Then, the previous, and next voltage levels are divided by this gradient to assess the real slope interval. A delay correction is calculated by dividing the difference between previous voltage level and upper voltage limit by the gradient. The real delay is assessed then by the subtraction of the delay correction from the measured delay.

Finally, the error protocol is summarized in 5 bits. They indicate which faults occurred in a transmission. Each bit stands for a categorized fault. As the error protocol is asserted together with the data in each transmission, there is continuous information of which faults are occurring in the communication system. The values for the standard conditions compared to the processed fault data are listed in Table 3.5. These comparison values can be adapted depending on the application. For our case study, they are extracted from the level leading to a contract violation, when the other signal parameters are at their standard levels.

Circuit Performance

The voltage level detection of the bus monitoring module is accurate enough to detect voltage differences on the order of 0.01 V. A 100kHz bus operating frequency requires a sampling rate of the ADC of approximately 21 MHz, resulting in 215 samples for each bit window. In testing, the times measured diverge from the inserted ones by 2 samples. We assume that the anti-aliasing filter and voltage adaptation used for the analog digital conversion are responsible for those divergences because the propagation delays of the digital circuit are negligible in comparison to these, and nothing else influences the signal/data propagation of the circuit.

Fault category	Deviation level [%]	Error protocol	Contract violation	Failure occurrence
Low bit Voltage Level	0.5 V			
	1 V	Low voltage level	yes	yes
High bit Voltage level	3.0 V			
	1.8 V	High voltage level	yes	yes
Delay	0 %			
	+10%	Delay	yes	
	+25%	Delay	yes	yes
Slope time	20%			
	30%	Slope time	yes	
	50%	Slope time	yes	
Glitch 20%	40%	Glitches	yes	yes
	50%	Glitches	yes	yes

Table 3.6: Detection of contract violation and signal fault in contrast to failure occurrence on I^2C communication.

3.6.3 Test Cases

To test the categorization and contract violation detection of our monitoring system, we needed a function generator to inject signal level faults. To be able to analyze the faults, the communication controller had to receive data. A protocol specific frame was created as a waveform for the function generator. After a standard waveform was created, it was possible to distort the waveform to inject signal level faults.

Test Case 1: Detection of Contract Violation, and Error Categorization

The first test consists of adding the specified signal faults to the payload individually, checking failure on the received data, contract violation, and fault categories assertion. The operating frequency was 100kHz. Emulated faults like voltage level faults are described by their actual voltage level. The other faults use normalized parameters as in the error protocol module. For rise time and fall time, it represents how much of the bit time (i.e. clock period) in percent was required for linearly sweeping from one voltage level to another. For the delays, the same percentage of bit time is used with reference to the middle of the bit, ranging thus from -50% to 50%. The glitch values mean how much the voltage level of the glitch represents the other logic voltage level. But for recognition of a glitch it is necessary to say where it occurred. So, we choose the glitch at 20%, because the component samples its inputs on that time most probably leading to failures. The fault categorization output, violation detection, and failure occurrence are listed in Table 3.6.

Test Case 2: Fault Isolation

Although contract violations already indicate an emerging problem in hardware, sporadic interferences or external events which lead to an instant violation of con-

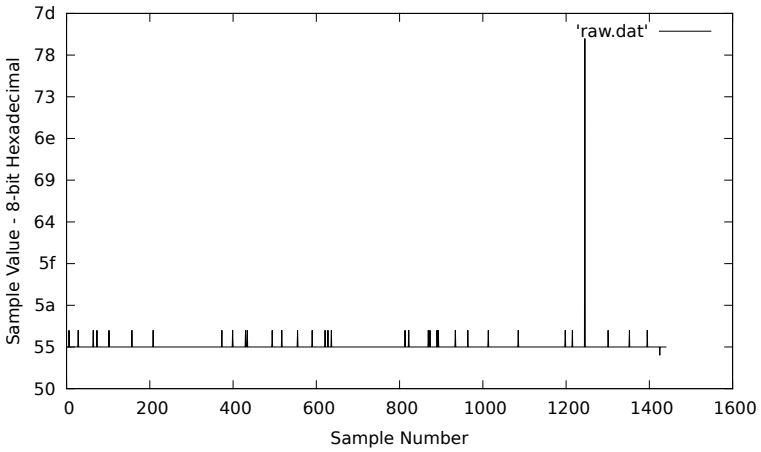

Figure 3.7: Raw data acquired by the I²C communication controller under the faults that have occurred in test case 2.

tract or even failure have to be successfully recognized by this monitor. Although contract violations are asserted, nothing assures that they really occurred. On the other hand, data failure occurrence is a proof of a contract violation, which validates the detection and categorization of the monitor. Therefore, we inject signal faults which lead to failure in order to effectively measure the capability of the monitor to detect them. Because we rely on a static waveform to inject faults in our system, we had to use signal faults which are most influenced by the environment as delay or slope times. Therefore, our function generator was configured to return the byte, 55 hexadecimal, with signal slope times of 16%, and delay of 16% for 1 hour. Due to analog parametrical changes over time, cables, and inner circuit operation of the generator, these parameters float within +-5%. Because the I²C controller implementation samples the data on 60% of the clock period, or 20% after the clock as in our normalization, delay faults leading to data failure randomly occur.

Adjusting the monitor parameters to only assert delay violations of 20%, and slope times above 25%, we were able to filter out 100% of the sporadic failure occurrences. The filtering process uses the information about fault occurrence to filter out data with faulty signal parameters. The acquired data without filtering can be seen in Fig. 3.7. The filtered acquisition has no failures, thus only with the value hex 55. The test has been run for 3593 seconds, 1441 frames of data have been received, and 758 frames identified as faulted. All faults have been categorized as delay faults. Through this experiment, we were able to analyze the dynamic performance of the monitor.

3.6.4 Generic Results

The built-in test circuit consists of 2 ADCs, 2 quadruple operational amplifiers, and 1 instrumentation amplifier. In our proof-of-concept case, the total cost of the IC is $14.16. In addition, the logic part is a synthesizable Verilog code requiring development time of about 2 person months. With the developed code, there is no cost for further implementation. The logic implementation requires 1,548 Look-up Tables (LUTs), and 376 flip-flops of the Spartan3A DSP 1800A, which costs around $65.00. Considering that the target application requires an FPGA, the cost of contract testing implementation would consist only of the external ICs.

On a developed application using the contract testing strategy for the I^2C communication, the FPGA design comprises an OpenRISC CPU controlling the I^2C communication controller to acquire data from external sensors. Without the developed monitor logic, the FPGA design used 12,192 LUTs, and 5,125 flip-flops. Thus, the monitor consumes 12.7% LUTs, and 7.3% flip-flops in comparison to the functional FPGA design.

In exchange to this overhead, failures of the data acquisition resulting from cable faults, external interference, or protocol misbehavior of sensors can be detected and isolated. In comparison to the use of error detecting codes used in different protocols, we categorize the errors which occurred, enabling the inference of the circuit fault.

However, the maximal sample rate of our monitor is of 32 MHz due to the chosen ADC. This reduces the applicability of our monitor to interfaces running at a maximum of 3.2 MHz to still allow enough samples for error characterization.

Contrary to the area overhead and cost factor, the insertion of the monitoring circuits does not influence the component behavior. Because the monitor inputs are buffered using an operational amplifier, their input impedance is very high, representing no load to the input signal. Furthermore, the execution of the contract testing can be done in parallel to system operation, providing fault isolation. Unfortunately, on fault isolation, a failure of the monitor leads to component failure. Therefore, the monitor must be highly reliable to not negatively affect the component.

The contract testing method is developed for component-based design. How meaningful the appliance of the built-in testing circuits is depends on the component granularity. Monitoring the behavior of every flip flop in a system is neither efficient nor feasible. However, monitoring the behavior of critical components of the target application can considerably raise the overall system reliability. The implementation of contract testing for singular critical components on the design limits the localization of faulty components. However when contract testing is implemented for a level of granularity of system functionality, misbehaving system functions can still be localized.

3.7 Discussion

Hardware contracts are easy to define because hardware components and standards are well documented. The contract information has only to be extracted from the existing documentation.

Furthermore, hardware components age, degrade, and are directly influenced by the environment. Thus, the environment situation and component use change over time. That is, contract testing for hardware is always meaningful, while it may only be reasonable for software in a modularized system where components are steadily replaced.

With the assessment of signal faults and environment violations by the application, information about the possible causes of the faults can be inferred. If the cause is located, the application is able to repair or isolate it.

The results of our case study show that a fault detection and categorization is possible for slow speed systems. With the monitor approach, both hardware errors and sporadic faults can be detected and categorized. On the other hand, the algorithm we use to calculate the signal fault responsible for the contract violation does not categorize contract violations which occur due to the combined effect of many signal faults.

Moreover, in our case study, contract violations are asserted for signal fault levels, which still allow normal operation. This happens because of the divergence between specification and implementation. This implementation does not violate the I^2C communication specification because it is a subset of it. However, regardless of no system failure, the communication contract is being violated; only the component contract is not. So, the behavior is reliable for the component, but not for the communication system.

Due to the speed requirements of the signal acquisition to execute contract testing, the testing implementation can be expensive. This is especially the case for simple systems, where the cost of the components required for contract testing is higher than for the system being tested. Another constraint of the implementation is the assumption that the input signals of the component can be measured. Even constraining the contract testing to only acquire binary levels, the implementation is based on oversampling. Thus, for all but the slowest systems, the implementation might be infeasible because the maximum acquiring speed is limited, and some systems will also sample their inputs in a similar speed. However, the circuit shows the strengths and the feasibility of the approach. A substantial complexity reduction can be achieved by applying analog circuits to measure the signal faults. Such optimization is beyond the scope of this work, which focuses on the approach, the specification and the test method, not its implementation.

3.8 Conclusion, and Future Work

The presented method represents a way of specifying and testing. The contract specification for software aims to ease component reuse. In hardware design, component reuse is standard. Therefore, hardware components already possess detailed documentation from which contracts can be extracted. A contract is defined in two steps. First, the requirements for component operability have to be defined, which are called preconditions. Later, the results of component operation and their related constraints are the postconditions, to which the component is committed.

To formalize the contract for hardware, the essential requirements for hardware have been pointed out. Requirements for environment, input levels, time, and logic set are considered to be common to every hardware component. This information represents the preconditions of a hardware contract. With the regular structure of the contracts, a machine is able to interpret and monitor it.

Contract testing tests the fulfillment of the contract preconditions. For this, real data have to be acquired, and compared to the contract. The environment variables have slow time constants, and can thus be sampled with low frequencies, while the timing restrictions for the input levels must be very short times, requiring high sampling frequencies. The comparison of the acquired data to the contract preconditions can be made directly for the environment variables, while high speed input data require data compression and post-processing.

The information about contract violation is enough to detect an operation fault. But to locate the cause of the fault we have detected, that fault categorization is needed. With more information about the digital signal on the inputs of a component, the causes for contract violation can be better determined. The parameters' voltage levels, delays, slope times, and glitches are chosen to further categorize contract violations, and thus enable detection of causes.

Finally, we show how to describe a contract for a master node of an I^2C communication system. Based on this, a contract testing circuit was built to detect contract violations and categorize them. The contract testing circuit was able to detect contract violations, and in addition categorization of the faults has been carried out. In a second experiment, the validation of the contract testing circuit against occurrences of communication failures have shown 100% failure recognition. However, contract violations can be caused by combined faults, inhibiting our fault categorization.

A contract represents an easy way to formalize specification, and to test in real-time. The advantage of contract testing becomes evident when the aging and environment dependent nature of hardware is taken into account. However, contract testing turns out to be a considerable overhead for simple systems. For safety critical systems, the overhead is justified because critical operations can be constantly verified for their operability.

For the continuation of this project, we are first going to optimize the categorization mechanism to allow the categorization of combined signal faults. In addition, we plan to support fault localization for contract testing. The current idea is to apply a Bayesian network as a fault analysis model using the hardware contracts, and categorized faults as evidence nodes. Furthermore, we are planning different approaches to execute contract testing faster, and in a less expensive way.

4 Model of Hardware Contracts and Violations on Transaction Level: A Fault Propagation Analysis

4.1 Introduction

Safety critical systems have to operate safely even on the presence of faults. It means that malfunctioning components have to be located and their faults isolated, so that they do not propagate to the user. The behavior of the system in the presence of faults can be analyzed using a model of the system. For that, faults and methods for localization, isolation and correction have to be modeled. In the case of an acquisition system, the communication buses connected to the sensors are influenced externally by the environment and by each communicating node, being a critical point of the design.

The early design of complex hardware systems, including software and hardware parts interfacing with the real world and user, is aided nowadays by system design methodologies. System design [BMP07] abstracts the behavior of the system components only specifying their function. With it, different system architectures can be explored to reason about performance, cost and dependability. As the number of components raises, their communication begins to be crucial to the design of the system. However, the model of communication is strongly restricted in classical system modeling methods. For instance, in a C model, the communication between components is abstracted to function calls hindering performance and dependability analysis. On the other hand, the RTL, used in hardware design, does not abstract communication hardening the exploration of different bus architectures.

In order to effectively design system communication, the Transaction Level Model (TLM) has been developed [Ope]. It allows the design of the communication to be independent from the components or architecture design. Furthermore, the detail of the model can span from function calls to pin signaling. Because TLM enables the incremental design of communication systems, the early analysis of communication faults on TLM designs is of special interest. In particular, modeling of the mixed-signal behavior of communication systems has not been

targeted yet.

In this chapter, we analyze the propagation of signal faults through a synchronous bus in a Transaction-Level model of an acquisition system. This system is composed by multiple sensors connected to a bus, a bus master and a CPU that polls the data. First, the bus, its modes and operating characteristics are modeled. The selection algorithm of fallback mode is placed on the communication controller, the bus master. For the fault injection, probability distributions are defined for the characteristics of the signal: Delay, slope level, voltage level and glitches, thus statistically generating faults. These faults are traced by the model so that their propagation results can be evaluated later.

The next section outlines the related work of this chapter. Section 4.3 explains the bus model, its operating modes, the fault analysis and fault processing modules. Section 4.4 presents the acquisition architecture simulated in this chapter, the fallback selection algorithm and the fault generation, followed by the simulation results. In Section 4.5, the conclusion of the work is presented.

4.2 Related Work

Monitoring techniques are especially used for the verification of buses. The monitoring methods, based on simulation, monitor faults which are simulated by testbenches as a virtual faulty environment. Generally, digital fault is injected in the behavior model of a communication controller described in traditional HDL languages, such as Verilog or VHDL [DLMSS08]. But novel mixed-signal verification approaches model the analog behavior of transceivers as well, allowing the use of analog inputs in order to analyze both transceiver and communication controller faults [BCMS05]. They claim that the fault coverage of a mixed-signal system is often decreased due to the complex interaction between analog and digital verification methodologies. Their verification method opens the possibility of covering these propagating signal level faults early in the development.

The use of the TLM methodology for fault analysis has been derived from the use of SystemC [Ope] as a high level language for system design and verification. It allows the abstraction of behavior, structure and communication [BMP07]. However, to achieve measures of performance and cost and reason about system architecture, SystemC is able to gradually refine the specification of the system. One key development for this is the TLM library, which tackles the problem of the design of communication systems with gradual refinement of implementation details. The use of SystemC for verification raises the question of the design dependability. To reason about dependability, the previous work has focused on the reliability assessment of the design through fault injection and analysis on simulation. Because SystemC has several abstraction levels, Beltrame et al. [BBM09] carry out an analysis of how RTL faults, digital faults in ports and registers, can be modeled in the abstraction levels of TLM (e.g. approximately-

timed and loosely-timed). Chen et al. [CWP08] developed a framework for the event triggered injection of bit faults in a TLM design. The event is here a communication functionality (e.g. burst-read).

The classic verification methods evolve towards the inclusion of complex analog digital interactions to the model for higher fault coverage. Although SystemC AMS covers mixed-signal simulation, it works at RTL not TLM level. Therefore, our work aims at the verification of mixed-signal systems for the TLM design methodology taking advantage of early system level design and high speed simulation, while enabling the analysis of the impact of mixed-signal faults. This complements the work of Beltrame et al. [BBM09] and Cheng et al. [CWP08] extending the verification of TLM designs to the signal level. For that, signal characteristics: Delay, slope level, voltage level and glitches are defined for transactions. These characteristics are interpreted and corresponding signal faults are assigned if operating limits of the communication system are exceeded. Assigned signal faults are then translated to digital faults.

4.3 Bus Model

In order to model a signal fault aware bus and its fallback modes, the TLM library is used. The current standard considers performance issues related to the communication, but does not include operating characteristics that affect the communication reliability. The standard comprehends standard blocking and non-blocking transport interfaces and defines a standard payload[1] which includes performance characteristics, such as delay and latency [Ope]. We extend this standard payload to include the signal quality factors: Delay, slope level, voltage level and glitches, which are directly related to communication faults and the bus operation mode. In addition, the model of the synchronous bus also holds its operation mode: Operating frequency, clock phase, logic levels and connected nodes.

Payload extensions contain both the value of signal characteristics of the transmission and a record of the violation of their limit (i.e. signal failure). When forwarding read calls, the extension is ignored, the signal characteristics are set by the callee. When the callee sends back the payload, the bus analyzes its signals, assigning the corresponding signal failure if the bus operation limits are exceeded. In addition, the bus modifies the payload transmitted data according to the signal failures which have occurred. Finally, the complete payload is sent to the caller. Based on the failure record, the caller can then decide to change the operation mode in order to avoid further failure. On write calls, the bus analyzes the signal characteristics and processes the data first, then forwards these to the

[1] An instance of the standard payload corresponds to a packet, when modeling a regular communication protocol.

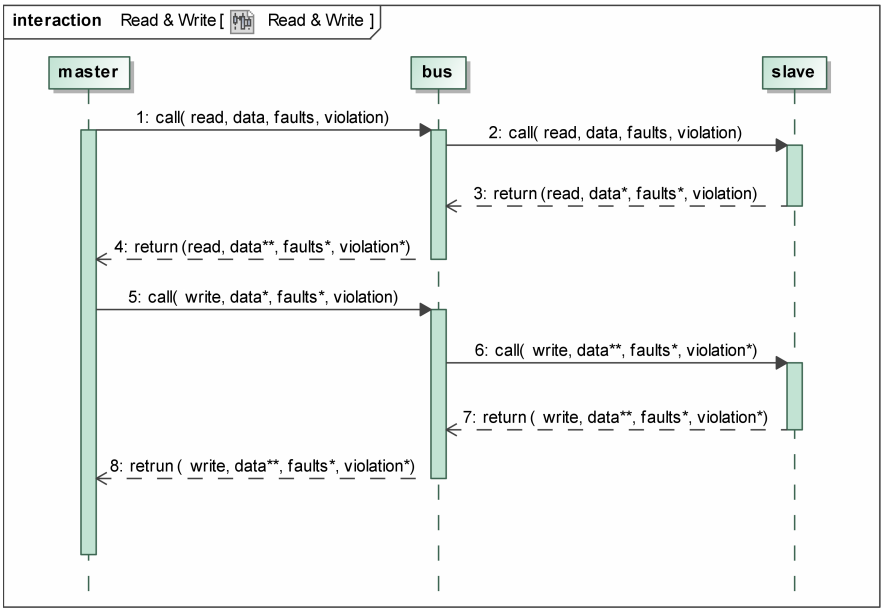

Figure 4.1: UML sequence diagram for write and read calls to the modeled bus. * represent that the variable has been set, ** modified.

callee, ignoring the extension on return. The UML sequence diagrams for these transactions are shown in Fig. 4.1.

4.3.1 Modeling Signal Faults

Prototype based communication monitoring techniques of Pallierer et al. [PHZ+05] and Armengaud et al. [ARSH05] categorize bit faults, glitches and delays. Crossman et al. [CGMC03] define possible signal faults of car sensors, as abnormal magnitudes (voltage levels), rolling (slope times), noise and dependency faults (context dependent). We categorize signal and bit faults related to digital hardware communication in a combined manner. A digital signal is ideally represented by two voltage levels, with instantaneous switching between both levels. For a real electronic component to drive its output from one logic level to the other, the resulting signal has a ***slope time***, which can be measured as a time degraded behavior. The same applies for ***delays***, which represent the response time of a component. Degraded ***voltage levels*** are variations of the output voltages for the logic levels approaching its boundaries, while ***glitches*** are voltage pulses of short duration resulting from interferences from outside. These four signal and bit faults of digital signals (Fig. 4.2) are used as quality measurement of a transmission. They are described in the payload extension of a data frame as: Both high and low bits voltage level; rise and fall time; delay; glitch time, level and count.

Signal Conditions	**Signal Failure Detection**	**Digital Fault Generation**
$V_H < 2.0V$	High bit Voltage Level	All 1s to 0s
$V_L > 0.8V$	Low bit Voltage level	All 0s to 1s
$\phi_d > \phi_s$	Delay	Rotate data to the right
$\phi_d + \phi_r > \phi_s$	Rise time	$y[n] = \begin{cases} 0, x[n-1] = 0 \\ x[n], \text{otherwise} \end{cases}$
$\phi_d + \phi_f > \phi_s$	Fall time	$y[n] = \begin{cases} 1, x[n-1] = 1 \\ x[n], \text{otherwise} \end{cases}$
Glitch count > 0 $\phi_s - 18° < \phi_g < \phi_s + 18°$	**Glitch time**	Nothing
$V_H - \Delta V_g < 2.0V$	Glitch high level	In combination with **glitch time** All 1s to 0s
$\Delta V_g + V_L > 0.8V$	Glitch low level	In combination with **glitch time** All 0s to 1s

Table 4.1: Signal conditions for signal failure detection (limit for bus operation) and digital fault generation according to detected signal failure. The phases ϕ of the Table are defined in Fig. 4.2. Moreover, we define x[n] as a bit series of the output data of the transmitter, while y[n] is the bit series of the data arriving at the receiver. The index represents the bit position of the data.

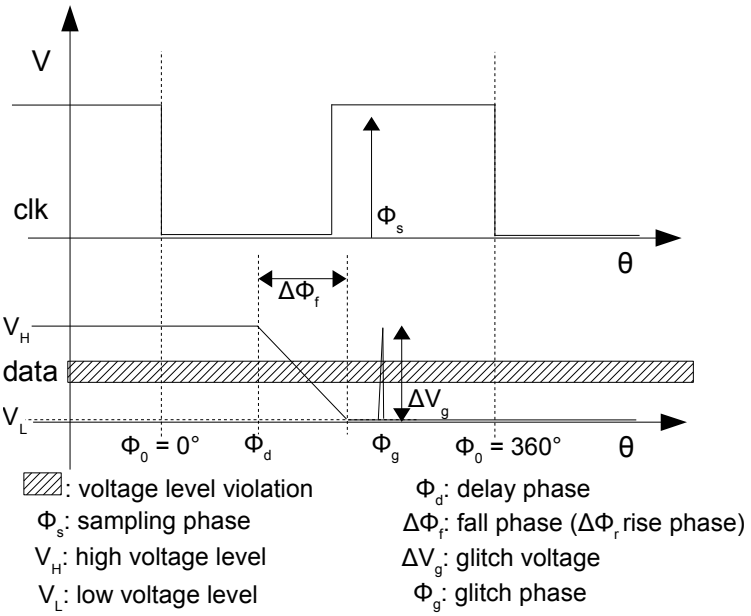

Figure 4.2: Time normalized signal characteristics. Times are multiplied by the operating frequency resulting in phase values.

4.3.2 Fault Analysis and Digital Fault Generation

The signal conditions of the data being transmitted through the bus are analyzed and digital faults are generated accordingly. The data sender sets the signal characteristics for the transmission. These are then compared to the conditions of Table 4.1 to detect signal failures. The listed conditions are based on the limits imposed by the operation of the bus. For comparison, the *timing* signal characteristics are normalized to *phase* signal characteristics accounting for the bus frequency. The bus operation conditions, sample time and clock phase are merged to the ϕ_s sampling phase. Logic levels and sample time depend on the specification and are thus constant, not influencing the relationship between operation mode and violation limits. Signal failures lead to data failure. In order to model this, each detected violation generates a digital fault according to the generation schemes described in Table 4.1.

4.4 Acquisition Architecture

In order to avoid asynchronous complexity and cope with the earliest design stages, the architecture modules, acquisition CPU, bus master and sensors are modeled in SystemC using the loosely-timed coding style of the Transaction-Level Model, calling thus blocking transport only. While the approximately-timed style is able to model communication handshaking through non-blocking interfaces, it

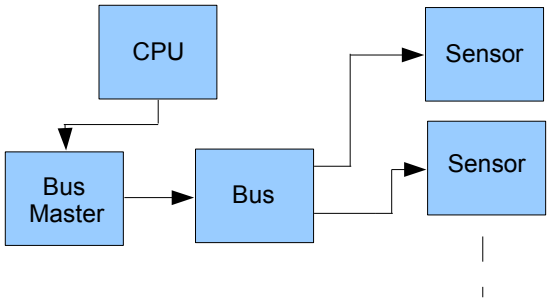

Figure 4.3: Block diagram of the modeled acquisition system.

also requires accurate models and complicates the simulation.

The architecture connects the acquisition CPU to the bus master, which is connected to the sensors through the previously modeled TLM bus, Fig. 4.3. In the model of the acquisition CPU, only the acquisition polling function is modeled. The bus master contains a thread safe buffer implementation, which is accessed by the CPU. To the other side it interfaces with the bus, executing two tasks. First, it request the data of every sensor. Then, if the bus detected a signal failure, the bus master may change the operation mode of the bus and retry transmission. In addition, the operation mode of the bus can be periodically reset to raise bus performance. This also reconnects previously isolated nodes, which might have been faulty only for a short period of time.

Each sensor continuously reads data from a different input file that can be accessed by calls to the blocking transport method. Upon each sensor access, the signal characteristics of the TLM extended payload are set. Apart from glitch count, these signal characteristics follow a Gauss distribution. The values for the mean and the standard deviation of the distributions can be set on sensor instantiation. The glitch count is modeled by a geometric distribution instead whose initialization value is equal to the chance of no glitch occurrences in a bit. For this configuration, the statistic variable x_k represented by the distribution is the number of sequential bits requested for a glitch occurrence. In order, then, to calculate the statistic variable glitch count for a data frame, the formula $framebits/x_k$ is applied.

4.4.1 Recovery Mechanism

As recovery mechanism, we implement fallback modes for our bus system that allow for correct functionality in the presence of faults by the usage of an alternative operation mode. In the bus master, the fallback mode selection algorithm of Fig. 4.4 can be activated. The bus master gets the information about signal failure occurrences from the bus instance. If the algorithm is activated and any failure occurs, a selected fallback mode is assigned to the bus by the bus master.

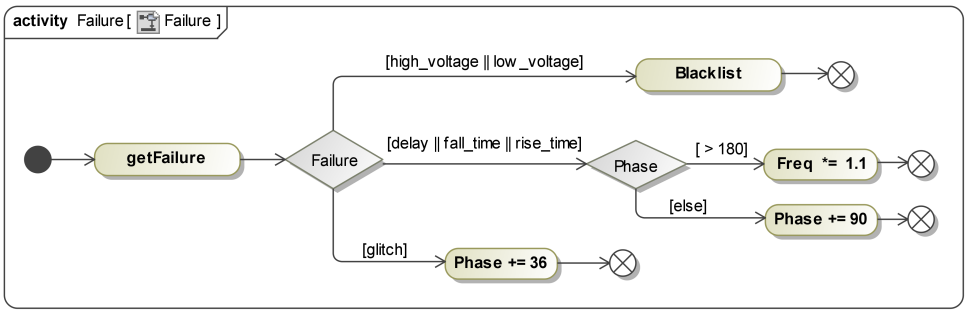

Figure 4.4: UML activity diagram of the algorithm for fallback mode selection.

Directly after mode change, a single transmission retry is carried out, for which neither fallback mode nor further retries are activated. After this transmission is completed, fallback modes can continue to be assigned.

Because we model a synchronous bus, we can circumvent delays and slope times by postponing the sampling of a bit. This strategy is followed until we reach a maximum of 180° phase between the data and clock signals because higher degrees would imply in a transmission delay interfering with the transmissions to follow. If failures still occur due to delays or slope times despite the appliance of this strategy, we reduce the operation frequency of the bus. We also deal with adverse voltage levels by disconnecting the nodes responsible for driving the bus incorrectly. Moreover, this also helps in cases where the bus fan-out has been exceeded. Finally, on the occurrence of successive glitches on bit sampling, we again increase the phase between the data and clock signals to avoid sampling on glitch times.

4.4.2 Results

During the simulation of the model, all data is accepted by the acquisition CPU. Faulty data is marked on simulation and counted, if faults are detected, information about isolation or correction is logged, otherwise failure occurrence is asserted. With these data, fault propagation analysis can be made, producing statistics about the robustness of the model against the environment modeled by the probability distributions.

An environment is defined in the Table 4.2. The bus works with a 100kHz clock frequency, the sampling phase of the implementation is of 216°, and the logic levels are Transistor-Transitor Logic (TTL) (bit 0: 0.8 V/bit 1: 2.0 V). For a simulation of a bus in this configuration, the values of total system faults (signal failures), fault isolation, fault recovery and failure occurrence are compared for 2 modes: fallback without periodic mode reset (Reset OFF) and fallback with periodic reset (Reset ON). The corresponding results are presented in Table 4.3.

Signal Characteristic	Mean	Standard Deviation
High bit Voltage Level	3	0.35
Low bit Voltage level	0	0.3
Delay	$4\mu s$	1.2 μs
Rise time	$2\mu s$	$0.1\mu s$
Fall time	$2\mu s$	$0.1\mu s$
Glitch time	$4\mu s$	$0.5\mu s$
Glitch level	0.5	0.1

Table 4.2: Parameters of the normal distribution used to model the signal characteristics of the sensor transfers. The unlisted glitch count parameter follows a geometric distribution with initialization value of 0.8. That is an 80% chance of glitch free bit.

We do not list the results for a bus without fallback because it neither isolates nor recovers any fault. Thus, the same test for such a bus produces the same number of failures as arisen faults. On the example of a transmission of data through this bus corresponding to a sinus signal, a graphic of the data received by the CPU for a simulation with bus fallback turned off and on can be seen on Fig. 4.5. The simulated fallback mode includes the periodic mode reset.

Applying signal fault detection and adapting the bus operation mode according to Section 4.4.1, 98% of the total transmissions have been blocked. The reason for this behavior lies in the blacklist fallback that disconnects a sensor from the bus on the delivery of wrong logic levels. This way, faults do not occur and the bus is free for correct data transmission. But sporadic faults result in permanent exclusion of the sensor's data. In order to avoid permanent sensor exclusion and ever increasing degradation of bus performance, we apply a periodic operation mode reset that copes with sporadic fault occurrence. This results in the recovery of 97% of the generated faults. The remaining 3% of the faults turned into failures because they have occurred on the retry transmission after fallback mode set. In this situation, no further measures are taken, otherwise the communication process with other nodes is interrupted and an endless degradation of bus performance can occur.

4.5 Conclusion

The verification using classic hardware description languages evolves towards applying mixed-signal verification to reduce uncertainty about the interoperability between analog and digital systems. Faults in the different abstraction levels of

Number of	Reset OFF	Reset ON
Transmissions	40000	40000
Transmission Retries	6	2833
Blocked Transmissions	39198	457
Faults	8	5516
Isolated Faults	4	239
Recovered Faults	8	5143
Failures	0	134

Table 4.3: Test results for fallback without periodic mode reset (Reset OFF) and with it (Reset ON).

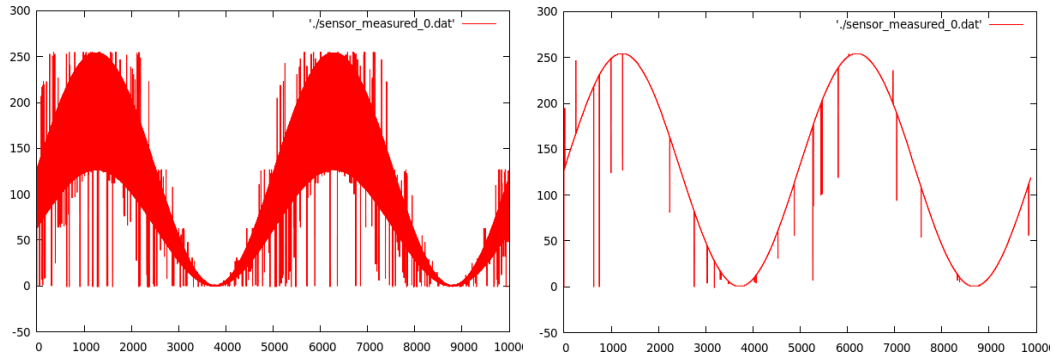

Figure 4.5: Graphics of the data received by the master through a bus with fallback modes (left) and without (right). Sensors send digital words of the amplitude of a sinusoidal signal together with signal parameters for the transmission. On transmission, faulty signal parameters generate digital faults according to Table 4.1 that distort the received data.

TLM have not been yet completely modelled. In this chapter we have introduced a mixed-signal verification strategy for TLM models, which profits from early verification of system design.

In order to process and analyze signal faults created in the system, we first developed a signal fault model, based on standard signal quality characteristics. Afterwards, an algorithm for detecting these faults based on operating properties of the same bus was created. Similarly, the same bus processes the transmitting data generating data failures according to the detected signal faults.

Then, we inserted the developed bus in a TLM model of an acquisition system to reason about fault propagation through a bus with fallback modes. Here, a bus master is implemented, which controls the bus, providing the bus with different operation modes. Faults have not been directly injected in the system. Instead, probability distributions have been assigned to the different signal characteristics of the sensors, building the environment of the system, which statistically generates faults.

The description of the signal characteristics of the sensors is realistic and can be easily adapted to different conditions. The online adaptation of the operation modes of the bus is able to isolate and correct almost every fault by sacrificing performance. In future work, we plan to compare this results with the fault tolerance of communication protocols with error correcting codes and error detecting codes with retries.

5 Device Mechanism: Structured Device Driver Development

5.1 Introduction

Device drivers are error-prone especially because the development methods of hardware and software are independent. Interaction between software and hardware is informally specified, and interface testing is rarely designed. This situation is illustrated by device drivers accounting for up to 70% of the failures on operating systems [GGP06], [Mur04]. In an operating system, the device driver has not only to comply with the device interface but also with a possible device class interface, and the kernel specific API. The major problem lies in the flexibility of these interfaces. The operating system interfaces may be non-standard or not fully specified, and the device may have a custom interface. In the categorization of six-years of bugs for Linux kernel drivers, Ryzhyk et al. [RCKH09] found out that, 38% of driver bugs are related to violations of device protocol, and 20% to violations of OS protocol.

In addition, the steady rise of new devices requires fast development of device drivers. Thus, drivers are re-used creating unified drivers. A unified driver, such as the USB class driver, serves a multitude of devices with a similar device interface. These approaches imply that the upcoming devices have to maintain the old device interface to some extent. Less certain, however, is the compliance of the new devices with the non-explicit device interface constraints that the driver expects. Finally, registers used for device control cannot be natively represented by general purpose programming languages. Hence, up to 30% of driver code deals with error-prone bit operations [MRC+00].

In this chapter, we propose a new architecture for device drivers based on the separation of concerns. Most specifically, we envision a device centric software interface for device access, the device mechanism. Instead of configuring a device operation mode for a system specific implementation, this interface enables full device control while providing a universal means of correctly operating the device. This way, implementations of various device abstractions, different operating systems or device classes, can take advantage of the same device access interface. This idea matches the concept of separation of mechanism and policy widespread in the operating system design [TW09]. Despite the architectural focus on cohesion of device access, the bit operations are error-prone, and the

device access through a communication system is complex and operating system dependent. Thus, we provide a framework for the systematic development of the device mechanism giving semantics to device registers and communication. The semantics allows for register and communication description and their seamless software access providing correct access behavior and operating system independence.

The limitations of our device mechanism abstraction and the effort for porting the framework to another communication or operating system are evaluated. We show the feasibility of our architecture by re-engineering the Linux driver for Philips webcams. After describing the internal device structure, we create a device mechanism interface for the camera using it in both a user and a kernel space driver. Differences in code and memory size between our driver and the original driver are presented. Based on an analysis of the Linux Git logs for the respective driver, we show known and unknown bugs which could have been avoided.

The next section reviews the state of the art on driver development and reliability methods. Section 5.3 fully describes the device mechanism, its interface, implementation rules, and specification of Direct Memory Access (DMA) and Interrupt functions. After that, the implementation of the framework for the systematic composition of the device mechanism is explained. Limitations of the device mechanism abstraction and the implemented framework are presented in Section 5.5. Our proof-of-concept Philips webcam example is presented together with its results in Section 5.6. Discussion about the method is presented in Section 5.7, followed by the conclusion.

5.2 Related Work

Device abstraction is the goal of device drivers on operating systems. To accomplish this, operating systems envision the architectural decomposition of drivers into a generic device interface and device specific implementations. A generic device interface embraces a device type presenting a generic interface to any device of that type for the applications of the operating system [TW09]. The remaining driver maps the abstract interface to device commands using any structure of device control (i.e. addresses of device registers and communication). Given that the specific underlying device has a driver implementation for the operating system, keyboards, pointing devices, disks, network cards, webcams, and others can be used by applications in a unified way through the generic device interfaces.

Although the driver architecture in operating systems is useful, a driver remains a complex software component mainly because its task spans multiple operational scopes, such as controlling the device, administrating resources, synchronizing, and coping with the generic device interface. Recent research on device driver synthesis proposes a generic device interface as a central driver specification to

which two separate description mappings have to be provided, mapping to operating system, and to a specific device [RCK+09]. With the driver specification and the two descriptions, the driver is synthesized to a specific device, and a specific operating system. By exchanging the corresponding description, another operating system or device is supported. In addition, this reduces the complexity of the driver by reducing the scope of each description.

Earlier work on driver synthesis complements the architectural approach providing semantics to device control structures. Otherwise, these devices structures are mapped through error-prone bit operations. Earlier synthesizers generate register accessor functions in the C language, and interface hardware in HDL, automatically resolving visibility of module registers through the interconnect, and complying with the interface protocol of the modules [COB95], [WB94]. Later work targets the description and access of device registers. These are described in a custom language enabling the definition of properties such as permission, bit fields, pre-defined values, and register cross-dependency. The descriptions are synthesized to C macros after a static consistency check [MRC+00], [SYKI05].

Beside the avoidance of programming mistakes by the inclusion of device semantics, further work moves towards the inclusion of runtime checks of assignable values, and allowed events. The former is enabled by embedding assignable values to the register description in the custom language. The latter requires the description of either a call order [OOJ98] or a state machine [WMB03]. Driver synthesis is already able to leverage the driver development on operating systems. For example, Conway et al. [CE04] proposes the implementation of a device specific implementation through a state machine description. However, the requirements of using a custom language and the restricted access modes of memory mapped and port I/O discourages its adoption.

Fault tolerance and removal approaches have also been used to enhance driver reliability. Fault tolerance through driver encapsulation targets non-deterministic faults well. Most encapsulation techniques isolate memory access violations [TW09], [LCFD+05], [SMLE02]. Based on the detection of a fault, some systems recover to a known state. Herder et al. [HBG+07] provides a policy controlled recovery system for unresponsive drivers, while Swift et al. [SABL06] uses a shadow driver for mimicking the real driver behavior during recovery. Unfortunately, fault isolation is not suitable for solving design or implementation faults. It is able to avoid the failures but their deterministic behavior continuously activates the fault isolation mechanism and the fault is not corrected. To find programming faults, model checking tools have been successfully used [ECCH00]. Extensive violations of OS protocol have been exposed by model checking tools including a model of the OS protocol behavior [BBKL10], [BR02]. However, a formal specification of the device protocol, whose violation is the highest source of driver faults, is rarely available resulting in a modest success on the removal of driver programming errors. Therefore, recent research moves towards new driver architectures for fault

prevention specially dealing with the lack of formal device protocol specifications.

Contribution

Our device mechanism architecture proposes a software interface for device access. The device access embodies the mechanism part of the driver, allowing for the software control of the device, while the remaining driver implements a policy mapping the generic device interface, the device class, to the device mechanism. The device mechanism is device centric, directly extracted from the hardware functionality, providing a stateless software image of the device in contrast to the generic device interface proposed as central driver specification by Ryzhyk et al. [RCK+09]. The device mechanism is fully specified because it is based on the hardware device implementation making it easy to associate constraints with the interface. This helps in filling the device protocol specification gap.

In addition, the device mechanism completely abstracts the device control structures, device registers and communication to the driver policy. In order to efficiently control these elements, we provide a framework for the description and access of registers and communication. In comparison to earlier work on the mapping of device registers [MRC+00], [SYKI05], we define an abstraction for communication systems for the transport of stream data, and the definition and usage of interrupts and DMAs. The communication abstraction can also be bound to the register map for custom register access, complementing the traditional memory mapped and port I/O access types.

5.3 Device Mechanism

In order to correctly explain the task of the device mechanism, we give an overview of what devices do, and how they are integrated into operating systems. From the point of view of a computer system, devices provide new functionality, generally being a means to interact with the environment. As an example, Fig. 5.1 shows the integration of a pointing device in an operating system. The device captures movements on a plane which are reproduced with a graphical cursor on the computer's screen. Generically speaking, devices translate physical dimensions to electronic data. The position of the cursor on the computer screen, i.e. the designed functionality, is provided by the operating system to applications through a generic device interface.

Put in this way, it does not seem that there is much left for the driver to accomplish. However, the functionality implemented in the device need not be directly related with the designed functionality. As a result, the electronic data of a device is not necessarily the data required by the generic device interface. For example, a graphics tablet exports the position of the pen on the table, while

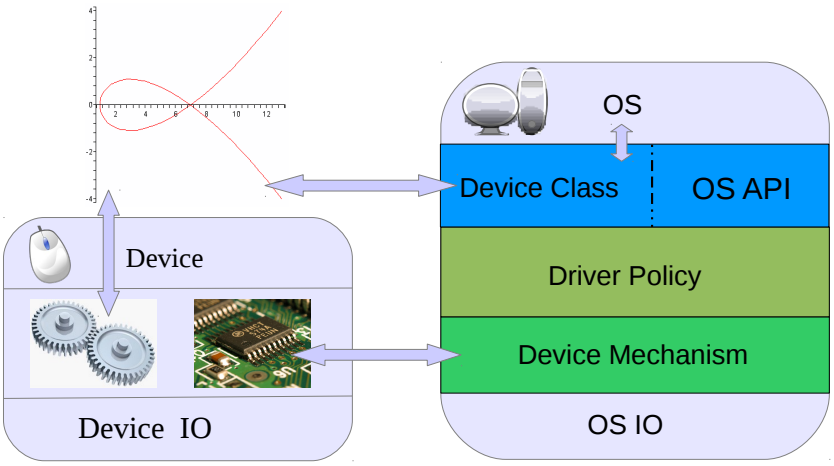

Figure 5.1: Hardware and software architectural layers involved in integrating device functionality in an operating system. Overview of the driver's interfaces in an operating system.

a regular mouse provides the displacement of the mouse in an elapsed time. In the case of a mouse, the driver is responsible of calculating the coordinates, to be compatible with the operating system requirement on input of pointing devices. This interface mismatch is exploited by vendors to produce more affordable devices by leaving some functionality to be implemented by the driver.

Besides the mapping of designed functionality to the functionality implemented by the device, the driver has to access the device data through a communication system. Although native access or memory mapped I/O might be available, the data might not be natively representable in programming languages. Thus, the driver has to handle the internal device structure, addresses, ports, registers and their bit fields in order to acquire the correct information. Furthermore, in case of non-native access, the access sequence has to be programmed, such as the access of device registers through transmission of USB setup packets. Therefore, the driver needs the complete information about the device structure and the communication properties. In addition, configuration of external controllers must also be available to the driver for implementation of specialized functions, such as DMAs and interrupts which require joint configuration of device and controller.

In our approach, we separate the driver in two software layers. The device mechanism provides access to the device data exporting the implemented functionality as a software interface. The driver policy interfaces with the device mechanism mapping the designed functionality to the functionality implemented by the device. In this work, we focus on the device mechanism, whose task is to handle the internal device structure, addresses, ports, registers, and control of DMA and interrupts controllers for the driver, as well as the communication

system to access them. Therefore, it needs complete information about the device structure and the communication properties. In return, the device data and control are exposed to the driver as a software interface.

We aim to export the state machine of the device to be handled directly by the driver. Hence, the device mechanism must stay a clean static interface. A stateless interface reduces the risk of increased complexity due to state explosion. The device mechanism is not meant to adapt the device logic to comply with the operating system interface, which is ensured by defining that the device mechanism must not modify or process any data from or to the device. The compliance with the operating system interface is left to the driver policy using the device mechanism to control the device.

5.3.1 Interface Design

The device mechanism interfaces with the driver policy, and with the device through the communication system and the internal device structures. We define here the interface to the driver policy. The interface to the driver policy is the external interface of the device mechanism and belongs to its definition. On the other hand, the interface to the device allows for the control of the device requiring a mapping of the device structures and a configuration of the communication system. The definition of how to specify and implement them can influence the final reliability of the device mechanism. However, it is an implementation aspect and will be dealt with under the Section 5.4.

The interface to the driver policy is a pure software interface. That means that no device structure and communication configuration is required to use it. The software interface is represented as a collection of software functions. In addition to conventional functions, the device mechanism has to support DMA transfers and interrupts.

Interrupts are supported by passing a function pointer as an argument to a registry function of the interface. The passed function can optionally accept values of device data, in which case also a data buffer has to be registered for this use. This way, data related to an interrupt event can be passed back to the driver policy right after interrupt occurrence, in the case of completion of a DMA transfer for example. In addition, it is mandatory for an interrupt or DMA transfer to be automatically disabled after completion to release the device mechanism from the task of managing buffer ownership over time, and thus, to avoid states in the device mechanism. The buffer used by the interrupt or DMA transfer is held by the device mechanism until completion thereof because it will be automatically filled with device data by the device mechanism. Upon completion, the buffer is released back to the driver policy for data processing.

5.3.2 Formal Definition of Device Mechanism

Formalizing the device mechanism, we give an unambiguous understanding of the task of this software layer. Moreover, the formal system allows us to mathematically define implementation rules for it that can be used to ensure correct architectural design.

We first define the device data S as being the tuple of sets for the possible values for register field ϕ, register ρ and stream σ. Then, there is a device structure X which is the tuple of sets for the specific structures for the access to field Φ, register P and stream Σ. These structures hold properties regarding the amount of information that can be held, where the information is held, and how it can be accessed.

1. $S = \langle \phi, \rho, \sigma \rangle$
2. $X = \langle \Phi, P, \Sigma \rangle$

Given both, we define a device D as their union. This is true from the point of view of the device mechanism which sees the device as data input or output disregarding its behavior. The device must be accessed through a communication system Ω. Data are accessible through a communication system given a device structure that contains the access data (e.g. address). Thus, a communication system is a relation between the device structure and data.

3. $D \subseteq X \cup S$
4. $\Omega \subseteq X \times S$

Device data are only accessible given that there is a communication system relation between device structure and data and the corresponding device structure exists. Thus, accessible data S' are the elements of S that can be accessed by the communication system Ω and for which there is a corresponding access structure x.

5. $s \in S' \leftrightarrow \forall s \exists (x, s) \in \Omega \wedge x \neq \varnothing$

Moreover, certain structures are only internally available. Therefore, we must differentiate between externally X' and internally visible device structures X. Normally, every device structure is visible from within the device whereas only a subset of it is exported. Given that Ω is the external device communication system, we can define that the externally visible device structures are the ones for which the external device communication system can find data. Thus:

6. $x \in X' \leftrightarrow \forall x \exists (x, s) \in \Omega \wedge s \neq \varnothing$

Finally, the relation between externally visible device structures X' and the external communication system is the device mechanism, M. And its interface, I, is a subset of the relation between a device mechanism and the externally visible data, S'.

7. $M \subseteq X' \times \Omega$
8. $I \subseteq M \times S'$

In addition, we can express the liveness property below that every request completes. For that, we use temporal logic, and assume the system M to have an arbitrary number of states. Given an interface method $i \models f(m) = s \in I$ that is a call to the device mechanism, we separate the call into two steps in time, the request R and the completion C.

9. $R \rightarrow \diamond C$

5.3.3 Implementation Rules

Based on the device mechanism formal system, we define rules which further constrain the implementation of the device mechanism. These rules ensure a valid architectural behavior of the device mechanism. They enforce that it does not further process any data or add states to the device. However, the functionality exported by the mechanism is device dependent, as well as the device structure and communication mapping. The driver correctness still depends on them but we cannot create generic rules for them.

The rules have been defined with the nature of the device mechanism in mind, to control a device and transport data in both directions. The idea is to restrict anything else to be done in this software layer, specially avoiding states and data processing. Some special cases, as the implementation of callbacks and the grouping of register accesses in a function have been specifically considered. The latter is required by register cross dependencies or functions controlled by multiple registers, such as enabling a DMA address space.

Argument Rules

The interface functions of the device mechanism may have arguments only as values of types register field, register or stream of bytes. They may have no further arguments.

1. $S' \subseteq S = \langle \phi, \rho, \sigma \rangle$

In addition, the device mechanism might temporarily hold stream of bytes in form of driver buffers. But it must release them and issue a callback call.

2. C R S'

Variable Rules

The device mechanism holds no static state variables of the device.

1. $M \cap S = \varnothing$

The device mechanism holds variables with definitions of the device structure or communication system. These can be addresses, ports and registers, endpoint addresses, interrupt lines, etc.

2. $M \subseteq X' \times \Omega$

Calling Rules

The device mechanism is allowed to group several device functions in a single driver function.

1. $g_n(x'_n) = s'_n \in M \mid n \in \mathbb{N}$

2. $i \models \{f(m) = s \mid m = \bigcup_{n=1}^{p}(g_n(x'_n) = s'_n) \in M\} \in I \mid p \in \mathbb{N}^+$

Calls from the device mechanism to the driver are only allowed after being previously enabled by the driver through a non-blocking call. After completion of the upward call (callback to the driver), the upward call is disabled until another request re-enables it.

3. $C \rightarrow \neg C \ \mathrm{U} \ R$

Processing Rules

The device mechanism must not modify the value of function arguments. They must be forwarded through from the driver to the device and from the device to the driver.

1. $\forall f(m) = s \in I \exists g(x') = s' \in M \mid s' = s$

A single device mechanism function can access multiple device registers/data.

2. $i \models \{f(m) = s \mid m = \bigcup_{n=1}^{p}(g_n(x'_n) = s'_n) \in M\} \in I \mid p \in \mathbb{N}^+$

Likewise, these data can be omitted from arguments of the device mechanism function.

3. $i \models \{f(m) = s \mid m = \bigcup_{n=1}^{p}(g_n(x'_n) = s'_n) \in M \mid \exists s'_n \neq s\} \in I \mid p \in \mathbb{N}^+$

5.3.4 Specification of Specialized Functions

Besides every call to the device mechanism interface being a request to the device mechanism, the device mechanism can acknowledge the receipt of a request instead of immediately completing it. However, it must complete the request at some point in time after that. This way, we can register a buffer for stream data for the device to fill it, receive the acknowledgement, and gather the data at a later point when the completion arrives. The above stated rules still hold for this case.

In addition, this allows modeling both DMAs and interrupts. However, the above defined rules impose that every completion of either interrupt or DMA transfer automatically disables them. That restricts the automation of the device mechanism leaving the automation and control of the device totally to the driver policy. This goes along with the goals of our approach because we want the device mechanism to be mostly static. As an outcome, a driver interfacing with it has to re-issue DMA transfers and re-enable interrupts after they complete.

5.4 Systematic Composition

The device mechanism is the relation between the device structure and the communication system $M \subseteq X' \times \Omega$. A major difficulty of its implementation lies in the absence of semantics for these hardware entities in general purpose programming languages. In other words, there is no standard way of handling device registers or its fields, whose information exchange control the device. Moreover, the communication system is dealt with as a software interface providing mechanism to access addresses. However, current communication systems, like USB and PCI, are controlled through a device driver by themselves. As such, they include new structural elements which have to be handled as well.

With a consistent description of the device structure and communication, and a corresponding interpretation for them, we provide the semantics missing in current programming languages. A consistent description defines the context of the hardware entities and defines their elements, subelements, properties, and interrelations. The correct interpretation allows handling these elements and subelements while respecting the given properties and interrelations. As an example, the consistent description of a register embraces its access information, properties, meaningful bit fields, and names for the access of the register and the bit fields. The interpretation then enables the access to them through the defined names also avoiding bit operations.

The explicitly described access information allows the access to be engineered in a standard way instead of arbitrarily defining disconnected parameters and access functions. This enhances the cohesion of these elements. Hence, a malfunction of an element implies necessarily that it has been wrongly described narrowing the

search for the fault. While in conventional access, every access to the same register is independent from each other and the parameters used may vary. Furthermore, elements, its subelements and properties can be checked for correctness in their context. This way, overlapping registers, assignment of invalid values or conflicts between memory and register definitions can be detected.

Moreover, the specialized functions, DMA and interrupts, are special hardware operations. Because they are also controlled through registers, and the communication system, they do not interfere with their description. However, a first half interrupt handler has to be implemented by the device mechanism. It acquires the interrupt data, and queues the second half interrupt handler, a registered callback function of the driver policy, to be processed later. However, it must not re-enable the interrupt as stated in the Device Mechanism Section.

5.4.1 Design

The choice of a language to describe the hardware entities, and to implement the interpretation of the hardware semantics depends especially on its descriptive power and user acceptance. Another important aspect is the impact of this choice on the implementation effort for the device mechanism and the underlying framework. Currently, only custom languages are available. NDL [CE04] and Wang et al. [WMB03] define a language for the complete driver development requiring support for the whole driver control, (e.g. synchronization, communication, etc.), while Devil [MRC+00] and Hail [SYKI05] generate C macros for the access of the defined registers. However, macros are only suitable for memory mapped or port I/O.

Moreover, custom languages and the absence of support tools discourage the user from adopting a new technology. Besides, only the description of the hardware entities is a problem for conventional programming languages, not their access. Thus, we separate description and access allowing for hardware semantics also in standard languages. In addition, access in multiple platforms only requires a new access API, keeping the description.

For the description of the hardware entities, we consider the Extensible Markup Language (XML) appropriate due to its efficiency and wide adoption. It does not ease the data input but keeps it generic, machine/human readable, and can be extended. Besides, metadata does not require a specific syntax. Instead, it defines type and data explicitly in a hierarchical way. Moreover, it can be used as output of a custom language or of a Graphical User Interface (GUI) for the definition of the data, and editors for XML are also available. Finally, existing XML parsers can be used for the translators.

For the access of the hardware entities, we use a C API. As parameters, it accepts C structures, generically called records, containing data from the XML documents. In contrast to C macros, this allows for the implementation of run-

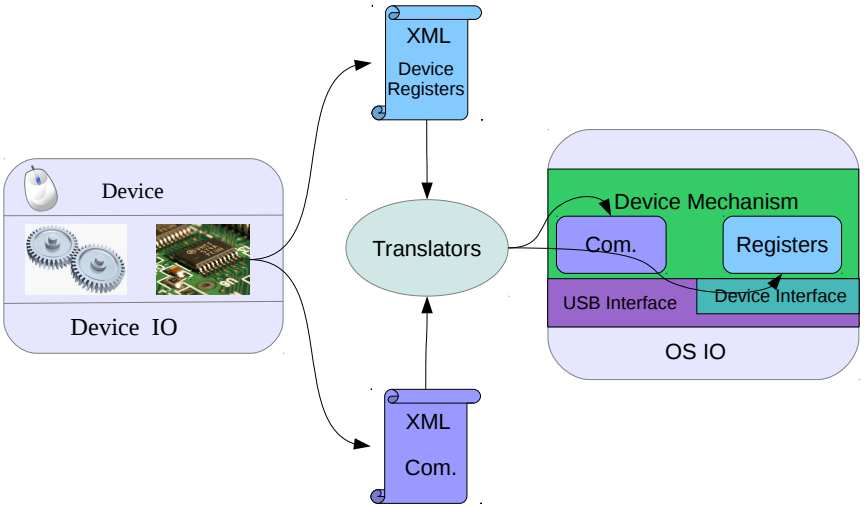

Figure 5.2: Systematic composition workflow. Register map and communication descriptions are translated to software elements that can be managed through the existing device and USB interfaces facilitating the implementation of the device mechanism.

time checks, such as permission, constrained write values, and access behaviors. C is also the most popular programming language, leveraging its maintenance and extension. Furthermore, only drivers implemented in C run on commodity operating systems avoiding the necessity of translation of its code. Also, custom register access through diverse bus systems is then possible by the access to the respective drivers in C. Moreover, the user can use this API natively on his driver code and directly profit from the operating system API by, for instance, using the conventional synchronization primitives.

5.4.2 Implementation

The workflow of the systematic composition starts with the definition of XML documents for the device registers and the communication. Each XML document is processed by a translator producing named instances of C structures for the corresponding XML elements. These C structures contain the same fields as the XML element, and some fields for internal use of the APIs. The next step is the implementation of the device mechanism. For that, the developer can access the device data by simply issuing API calls with the generated C structures as argument. The Fig. 5.2 gives an overview of the workflow.

The input of the translators are XML documents based on a specific XML Schema. Hence, the structure, and the content of the input document are fixed. From an input document, two files are generated, a C header and a source. The C

Listing 5.1: Register map structure

```
extern struct pwcblock {
    Reg * brightness;
    Reg * contrast;
    ...
    Reg * agcMode;
    RegField * agcMode_agcMode;
    ...
    Mem * cache;
        struct { //agcAuto register file
        Reg * agc;
        Mem * cache;

    } agcAuto;

    struct { //agcFixed register file
        Reg * agc;
        Mem * cache;
    } agcFixed;

} pwcblock;
```

header contains one main C structure for the complete XML document enclosing named pointers to instances for all elements which allow the access to electronic data. The hierarchy of XML is mimicked by nested C structures containing further elements. As an example, registers of a specific register file are grouped into a C structure as shown in Listing 5.1 and are accessible through the register file's given name and the '.' operator.

The data types of the elements declared in the generated header file are defined by the C APIs for device access. Together with their definition, helper functions for allocation, removal, and association of instances are provided. The generated source files define functions for initializing and removing an interface, register map or communication. Their initialization consists of allocating all instances, populating them with the data from the XML document, and associating them with related elements. The removal of the interface gives back the memory to the system removing the instances.

Context Definition

IP-XACT is a standard for integration and reuse of IP [IEE10]. They provide XML Schemas which also contain a detailed definition of a register map. Instead of creating our own register map definition, we use IP-XACT. Because our implementation, translators and APIs fully support the IP-XACT memory map Schema, we profit from IP-XACT being able to directly integrate third-party descriptions into our workflow,

The IP-XACT register map description includes the access to registers and their bit fields, the description of permission, constrained write values and volatile read behavior. Access rules are available in the form of register files and alternate

registers. Registers can be described inside register files and can belong to an alternate group, in which case, the alternate group or register file can be enabled or disabled blocking the access to the related registers. Furthermore, there are bit fields whose access triggers an action that can collide with other actions. These bit fields should be described with a triggering of actions property that is not available in the standard. Taking advantage of the metadata description, we extend the standard, naming the property `strobe read` or `strobe write` because registers which exhibit this behavior are called strobe registers. However, we do not introduce more advanced access rules in this layer. We think this can be better handled on top of the device mechanism with the inclusion of device constraints.

In addition, we create a context for the communication system. Our XML Schema allows the specification of the standard descriptors defined in the USB specification. Furthermore, we extend them with setup packets for access of status/registers. Generally, configurations, interfaces, endpoints and setup packets are treated simply as integers by the system API. Instead, we define these elements with all corresponding properties and dependencies. For instance, the interface does not simply have a corresponding number but also belongs to a configuration.

Translators

The translators are implemented in Python with the Lxml library for XML parsing. We chose Python for its simplicity, also allowing for rapid prototyping thanks to its memory management, metaprogramming features and dynamic typing. The Lxml library has a native Python API and enables parsing in similar way to the Document Object Model (DOM) but requires less memory and can traverse a tree similar to the Simple API for XML (SAX) while freeing not used elements.

The translators are divided in three modules, the parser, the resolver, and the code generator. The parser defines native data types for the corresponding data of the XML document. Upon the document being read, these objects are instantiated and the XML data is imported. That generates a tree with different objects containing the XML data. The tree represents the XML hierarchical structure. It is then input to the resolver that resolves cross references of the tree elements. After that, the objects of the resulting tree contain data independent of inherited data and default values. Moreover, only for data of the IP-XACT translator, the following context rules are proven.

- Register width fits in the specified memory width
- Register address is within the defined memory range
- Listed values for the registers can be represented with its bit width

- Declared bit fields do not surpass the register width
- Listed values for the bit fields fit in

Finally, the code generator generates the header and source files in the C language based on the tree processed by the resolver. For that, it also takes into consideration that no naming conflicts occur. If multiple elements in a namespace have the same name, this is detected and averted.

Semantics

The defined contexts, register map and USB communication, get a unique interpretation with the implementation of C APIs, device interface and USB interface respectively. These APIs provide the access to the values of the S' set through the C structures generated by the translators. This enables the control and forwarding of data from and to the device. In the register map context, operations to transfer data to bit fields and registers are provided, while transfers of data streams are available on the USB interface.

Device Interface

To access registers and bit fields, the device interface offers `set` and `get` functions. The interface maintains a cached value of every register. At startup, the cache is invalid, so every first read or write to a register activates the cache. If a register is not volatile and the cache is active, the value of the cache is supplied on a `get` function. Every `set` function updates the cache and writes to the device. We use the denominations `read` and `write` when referring to the device access, while `get` and `set` are the API operations.

Accesses to bit fields are dependent on the access to the register. Hence, to set or get a bit field, the parent register has to be set or get. The bit fields of a register share the cache of the register. Thus, on set or get to bit fields, the register cache is used. The bit operations required to retrieve a bit field value based on a register value or vice-versa are independent of updating the cache or accessing the device. In case of a bit field setting, the register value is updated and then the full register is written to the device. On a bit field get, the cached value of the register is retrieved and the bit field value is calculated based on it. Depending on multiple rules, the cached value is updated with the device data prior to bit field calculation.

In addition, different access modes are available for setting and getting registers and fields. Because a constrained write value can be set to `write as read`, we support functions for getting and setting a register or a bit field at once. Also, functions for delayed access exist. These functions are useful for setting multiple bit fields of a register one by one and only writing the register value to the device later. Finally, the device interface has functions for getting registers or bit fields

with forced behavior. Using these functions, the developer can force the interface to return the cached value of a register or bit field or force a read from device.

In order to temporarily store the values assigned to bit fields in the delayed mode, we use a linked list whose elements are the temporary bit field values and a pointer to the corresponding bit field. We also keep track of cached bit fields through a dirty flag as a field in the bit field C structure. This helps on the quantification of the dirty fields required to determine the access behavior.

With multiple access properties (volatile, permission, strobe) and the combination of multiple access modes (get/set, forced, delayed), the rules defining the access behavior become complex. Moreover, bit fields can have properties on their own, influencing the behavior of the register. The Table 5.1 summarizes the possible access behaviors regarding the possible register and bit field properties for access in standard mode. To begin with, we compile the required queries for every type of access listed in the second column of the table. Every query is executed as its corresponding predicate depending on register's and bit field's properties as defined. We use P and Φ as relations of register's and bit fields' properties and these properties' values. Notice that some properties' values are already either true or false. Moreover, there can be multiple bit fields for a register. Thus, we use quantification operators to define the evaluation of their properties. Depending on the result of the query, actions of the Table 5.2 are undertaken.

The last implemented functionality of the device interface are register files and alternate registers. These are defined by the IP-XACT XML Schema for extended register access control. Our implementation does not fix the control conditions. Instead, we allow the developer to freely enable or disable a register file or an alternate group. There is no difference in the implementation of a register or an alternate register. A register holds a list for its register files and another for alternate groups that enable an alternate register. On every device access, if any listed register file is not active, the access fails. If the access does not fail for the register file, and any alternate group in the list is active, the access is granted. Alternate groups and register files are generated by the translators, and are also found in the main C structure in the generated header file.

USB Interface

The USB interface can be selected as wrapper to either Libusb or to the Linux USB subsystem. Thus, our USB drivers can also be implemented in user or kernel space in a similar manner. In comparison to the functions of the wrapped subsystems, our interface expects the typed communication entities as arguments for its functions. The main functions of the interface are functions to transfer data. Again, `get` and `set` functions are provided. They expect the input of an endpoint, a buffer, and length as arguments. The endpoint is defined by the XML Schema and is a named pointer in the main C structure of the corresponding

Access	Query	Predicate
getReg	Access	$P(access) = read \wedge \forall\Phi(access) = read$
	Need Transfer	$P(volatile) \vee P(strobe\ read) \vee \exists\Phi(volatile) \vee \exists\Phi(read\ action) \vee \exists\Phi(strobe\ read)$
	Allowed Content	$\neg\exists\Phi(strobe\ write) \vee \exists!\Phi(strobe\ write)$
	Need Modify Content	$\exists\Phi(read\ action)$
getField Φ_t	Access	$\Phi_t(access) = read$
	Need Transfer	$\Phi_t(volatile) \vee \Phi_t(read\ action) \vee \Phi_t(strobe\ read)$
	Allowed Content	$\neg\exists\Phi(strobe\ write) \vee \exists!\Phi(strobe\ write)$
	Need Modify Content	$\exists\Phi(read\ action)$
setReg	Access	$\Phi(access) = write \wedge \forall\Phi(access) = write$
	Need Transfer	$true$
	Need Read before Write	$\exists\Phi(write\ value\ constraint) = writeAsRead \vee \exists\Phi(volatile) \wedge \neg\exists\Phi(read\ action) \wedge \neg\exists\Phi(strobe\ read)$
	Allowed Content	$\forall\Phi_{constraint}(\phi) \wedge (\neg\exists\Phi(strobe\ write) \vee \exists!\Phi(strobe\ write))$
	Need Modify Content	$\exists\Phi(modified\ write\ value) \vee \exists\Phi(strobe\ write) \mid \neg\Phi(volatile)$
setField Φ_t	Access	$\Phi_t(access) = write$
	Need Transfer	$true$
	Need Read before Write	$(\exists\Phi(write\ value\ constraint) = writeAsRead \mid \Phi \neq \Phi_t) \vee (\exists\Phi(volatile) \mid \Phi \neq \Phi_t) \wedge \neg\exists\Phi(read\ action) \wedge \neg\exists\Phi(strobe\ read)$
	Need Adapt Content	$\exists\Phi(modified\ write\ value) \mid \Phi \neq \Phi_t \wedge \exists\Phi(strobe\ write) \mid \Phi \neq \Phi_t$
	Allowed Content	$\forall\Phi_{constraint}(\phi) \wedge \Phi_t(write\ value\ constraint) \neq writeAsRead \wedge (\neg\exists\Phi(strobe\ write) \vee \exists!\Phi(strobe\ write))$
	Need Modify Content	$\neg\Phi_t(volatile) \wedge \Phi_t(modified\ write\ value) \wedge \Phi_t(strobe\ write)$

Table 5.1: Access behaviors: queries based on register and bit field properties that affect their access. The access procedure follows the listed queries in the top down direction. Depending on the queries' outcomes that are defined in Table 5.2, access procedure might continue, fail, data be managed or a cache value be returned.

Query	True	False
Access	Continue	Fail
Need Transfer	Continue	Return Cache
Need Read before Write	Fail	Continue
Need Adapt Content	Adapt Content	Continue
Allowed Content	Continue	Fail
Need Modify Content	Modify Content	Finish

Table 5.2: Queries' Outcomes dictating the processing of the procedures of Table 5.1.

generated header file. Because the endpoint is a typed entity, endpoints belonging to different interfaces or alternate interfaces are different even if they have the same number. This way, attempts to use the device on the wrong configuration can be blocked by the USB interface. In order to keep control of the current device configuration and alternate interface, the device and interface configuration are linked to the endpoint record through pointers.

In the XML Schema, setup packets have been defined for the access of device status or registers. A transfer function can be used with a setup packet, buffer, and length as arguments to simply transfer a packet and read or write the corresponding data. More interesting, however, is binding a register to a pair of setup packets for read and write from/to the device. This has to be done in three steps.

1. The registers have to define an element `complexType` with a `recvPacket` and `sendPacket` under the `vendor- Extension` element of a register in an IP-XACT XML document. The elements `recvPacket` and `sendPacket` must contain the name of a setup packet defined in the USB document.

2. Along with the initialization function for the device interface, that translator will generate a bind function which expects a pointer of type `struct * setup_packets` as argument. This function must be called with the corresponding pointer which is generated by the USB translator.

3. The device interface needs to bind to the USB interface, so that the device interface calls the USB interface for device access. For that, the USB interface provides a bind function which requires the pointers to the device and the USB interface respectively.

Their implementation is straightforward. The translator generates a bind function, which assigns the `complex_if` field of the C structure of a register with pointers to the named setup packets. The assigned setup packets have the same name under the `struct * setup packets` structure of the main C structure of the generated USB header file. At last, the bind function of the USB interface, re-assigns the function pointers `readIf` and `writeIf` of the device interface. This way, the device interface will call these functions for accessing the device along with it. It also forwards the `complex_if` pointer that contains the setup packet for either read or write. This implementation can be easily adapted to other custom access types.

5.4.3 Implementation of Specialized Functions

Interrupt and DMA are supported by the USB interface. According to the rules defined for the device mechanism, the implementation is based on a callback function and buffer registering. Moreover, the callback must be disabled after each run. The USB subsystem already works like this, urb transfers submitted

with callback functions are disabled after completion. To provide the improved semantics, we wrap the subsystem calls with our own interface. We combine the instantiation of a transfer with the buffer registration, only requiring an endpoint, buffer, and length as arguments. Then, a callback function can be registered. It is called by the system with the buffer and its transferred length as arguments. At last, we define a function unlock through which the interrupt can be re-enabled.

In this approach, we combine DMA with an interrupt for signaling DMA completion. Generally, the DMA only requires the registration of a buffer, while the interrupt only registers a callback. In USB, from the driver developer view, there is only a behavioral difference between interrupt and isochronous (DMA) endpoints because they actually both use the same mechanisms. The interrupt occurs sporadically, and therefore, does not require multiple transfers to be queued, while the isochronous transfers complete at regular intervals and at least one transfer has to be constantly pending to avoid data loss.

The USB interrupt transfer issues the callback upon receipt of one packet filling the registered buffer. This looks different for the isochronous transfers which contain multiple packets. The callback function is only issued after all packets have been processed. Furthermore, the registered buffer can have scattered data because it is divided equally and assigned for every packet on fixed offsets. However, these implementation details are not relevant to the access of the data. Following the systematic composition concept, this is an implementation detail which has to be handled by the underlying framework. Therefore, the buffer registration function initializes the urb transfers along with its multiple packets. Before calling the registered callback, a function of the USB interface moves the data in the buffer to ensure that the data are contiguous. The registered callback function is then called with the buffer, and the transferred length. Moreover, the USB interface also forwards an array containing the length of every received packet, and the number of received packets. This detail can be necessary to find the end of a data stream, data with no higher level protocol.

5.5 Evaluation

We analyze our method in two different ways. First, we discuss the generic limitations of our device formal definition with respect to different device types. Then, we evaluate the effort required to port our strategy to another operating system or communication system.

5.5.1 Limitations of Device Mechanism

Peripheral devices offer functionality to a system. They are special purpose transducers transforming physical information into electronic data or vice-versa. There are all kinds of devices hosted through diverse communication systems. However,

the communication systems are simply a medium to transport the internal data from and into the device. But the types of internal data made available from devices are the concern of the device mechanism abstraction and also its limitation.

We have defined three electronic data types, registers, bit fields and streams. Registers are the smallest accessible data being a collection of bits. Bit fields are register parts ranging from a single bit to the whole register width that play a configuration role on their own. They are therefore semantically meaningful but are only accessible through their registers. Streams are data collections in a specific context, such as time, destination, origin or physical interpretation. These data types are conventionally used for digital electronic design being the standard way of controlling peripheral devices. Therefore, these same data types are used for the control of a digital temperature sensor or an ADC or even a graphics card.

There are exceptions. Our abstraction is inappropriate for interfaces below the register level, as analog and unregistered bit level signaling. One example is an older printer on the parallel port. Here, the time to hold a bit level on a specific parallel port pin has to be controlled. Although this could still be managed by the bit fields, it is not its design goal. Moreover, analog signals are completely out of the scope of our approach.

However, the trend of the electronic development is to control analog signals with transceivers, integrated circuits for specific communication. They establish the signal level protocol, taking care of the tightest time schedules as well. For example, USB, the serial port and Ethernet controllers use transceivers. Although time is a dimension of interest in the real world, real time control using operating systems is not trivial. Therefore, real time control is custom provided, generally already integrated in the device. In commodity operating systems, time control is coarse grained using events/interrupts. We also take this approach specifying the interrupt usage for the device mechanism.

Finally, the device mechanism is not designed for devices exporting more than data. For instance, a General Purpose - Graphics Processing Unit (GPU) that exports computing power is going to use a framework like Open Computing Language (OpenCL) to deal with the hardware. On the other hand, the device mechanism is suitable to more specific computing devices. Coprocessors in general can export a device mechanism interface that is suitable to their computational model. The device mechanism serves as an interface to input the data and acquire the computed result. This model may change with the coprocessor. Every model matches another device mechanism.

5.5.2 System and Communication Compatibility

The current code size of the implementations of the device and USB translators and interfaces has been measured with the CLOC tool from the SourceForge

Tool	Lines of Code (LOC)
Device Translator	1,676
Device Interface	2,986
USB Translator	1,130
USB Interface	1,860

Table 5.3: Code sizes of the elements of the device mechanism framework.

repository. Its result is shown in Table 5.3. The device and USB translators are written in Python and are therefore platform independent. Their output uses C structures and initialization code defined in the device and USB interfaces respectively. The output code uses custom macros that wrap platform dependent implementations eliminating thus the translators' dependencies.

The device and USB interfaces are written in C. The device interface depends on the C standard library for special types, fixed width types, Null Pointer (NULL), boolean, and for dynamic memory. In user space, it is therefore platform independent. In kernel space, it needs the counterparts of these declarations which are defined for the Linux kernel. These declarations contain 10 lines of code in a header file of the device interface. Therefore, the device interface is nearly platform independent.

The USB interface is dependent on the USB access system. We have implemented it for the Linux USB subsystem and for Libusb applying macros for conditional compilation. Extracting clean implementations of both, the interface for the kernel contains 1,379 lines of code while the Libusb one 1,455. A patch file created with the diff command from the user to the kernel port accuses that 227 lines have been removed and 149 added[1]. This small difference results from the Libusb design that is based on the same Linux USB subsystem. Nevertheless, the code interfacing with the USB subsystem mostly makes calls to functions specified in the USB standard which should be mapped for any operating system.

The greatest effort to adapt the framework to another operating system lies in the development of a new communication interface. In the USB case, only the USB interface needs a new implementation. Its translator can remain, as the defined structures should remain also for internal compatibility. A different communication system, however, requires both a new communication interface and a new translator.

The requirements on the communication interface and its binding to the device interface depends on the following device mechanism assumptions:

1. Registers have an accessor function.
2. Streams have an access structure Σ.
3. Interrupts and DMAs' completion are signaled through a callback.

[1]The sum differs probably because CLOC ignores comments or blank lines.

The synchronization and access methods may vary. But from the software point of view, they are abstracted by callbacks and functions respectively. If the communication system does not standardize them in this way, the communication interface provided by the framework has to provide it.

For example, the specialized functions interrupt and DMA transfers are directly supported by the USB framework. On the other hand, the access of device registers requires the definition of the setup packets. Still, a vendor could come up with another way of register access, using bulk transfers for instance. Since the current USB interface only allows register access through setup packets, it would have to be extended with the vendor specific register accessors. For PCI, device access is made through mapped memory avoiding this issue.

However, a PCI port requires the extension of the input XML to define the purposes of the respective memory mapped areas, either device registers or stream. Furthermore, DMA streams have to be described with a link to an interrupt line to signal completion, as specified by the device mechanism formalism in 5.3.4.

5.6 Philips Webcam - Case Study

In our case study, we re-engineered the Linux driver for Philips Webcams (PWC) according to the device mechanism development. In contrast to network, disk or graphics devices, the generic device abstraction for media video devices is less mature [Rub06]. This puts a higher burden on the driver policy because it has to cope with incomplete specifications and implement special behaviors without help of upper layers. This allows for a more thorough evaluation of difficulties of the driver policy design that we intend to perform in our future work.

Linux allows for the generic management of video capturing devices through the Video4Linux generic device interface. Every compatible device has to implement a video stream with a supported codec, and controls of image properties, e.g. contrast or brightness. In exchange, the device can be used by every Video4Linux compatible application.

In the device mechanism, we define the registers and the USB communication in XML files. These are translated to C structures for API usage. Using these structures, we create the device mechanism as a collection of functions for device access. The resulting functions follow the device mechanism rules having only device data as arguments. In the following paragraphs, we explain how these functions differ from the functions they replace.

The image properties of the Philips webcam are adjusted by setting registers which are accessed by sending setup packets. PWC has a collection of functions for image property control which execute bit operations, control ranges, and transfer the data along with a setup packet. We substitute them by the access to a semantically defined register which has been linked to the setup packets for writing and reading the value to or from the device.

Listing 5.2: Data definition on PWC

```
#define SET_CHROM_CTL   0x03
#define PRESET_MANUAL_BLUE_GAIN_FORMATTER 0x1400
```

More significantly, the driver has to decompress the incoming image data to a sequence of Red, Green, and Blue color model (RGB) or YUV pictures, as the Video4Linux interface expects, a non trivial task. The current code acquires the data from the camera storing it in buffers. When a complete frame has been stored, the corresponding buffer is marked full and another buffer begins to be filled. When the Video4Linux interface requests an image from the device, the driver decompresses the data of a full buffer and forwards the image to the application.

The data acquisition is initiated with the instantiation of urb transfers (i.e. Linux kernel USB transfers) that are associated with a buffer and a callback function. The callback function compacts the scattered data and analyzes the size of each transferred packet to find the end of the stream. We substitute the initialization with our USB interface using registration functions for buffer and interrupt. In comparison to the regular driver, our callback function does not have to compact the buffer.

When a programmer starts to develop a driver, he needs the address, width and fields of the registers. Generally, this information is gathered in precompiler macros. A macro is named as the register containing its address while its bit fields are defined as masks. This is not mandatory; as drivers can arbitrarily define the device data. The PWC driver does not define bit masks for example. Because it has to transfer register values through USB packets, various setup packet parameters for access of different registers are defined. Generically speaking, this is the current code guideline. In our approach however, these parameters are defined in a context, which also gives the developer a workflow to follow. The description of register and communication are fixed by XML Schemas. If any information is missing or inconsistent, the translators will generate corresponding warnings. Listings 5.3 and 5.4 show how we define our data in comparison to 5.2.

In PWC, the register's access is custom made throughout multiple functions, which narrow down the 5 required parameters for the setup packet step by step. The access of the data in the device mechanism is standard and only requires one parameter, a hardware defined type. Besides, the hardware defined type has a meaningful name, blueGain, and is accessed by a function named setBlueGain providing clarity, as shown in Listing 5.5. By contrast, the PWC access only nearly references a blue gain by the macro PRESET_MANUAL_BLUE_GAIN_FORMATTER throughout the extensive procedure `send_control_msg`. An example of an access on PWC can be seen on Listing 5.6.

Listing 5.3: Register data definition on device mechanism

```
<spirit:register>
  <spirit:name>blueGain</spirit:name>
  <spirit:addressOffset>1</spirit:addressOffset>
  <spirit:size>8</spirit:size>
  <spirit:vendorExtensions>
    <spirit:complexType>
      <spirit:recvPacket>getBlueGainManual</spirit:recvPacket>
      <spirit:sendPacket>setBlueGainManual</spirit:sendPacket>
    </spirit:complexType>
  </spirit:vendorExtensions>
</spirit:register>
```

Listing 5.4: USB data definition on device mechanism

```
<udev:request>
  <udev:name>setChromCtl</udev:name>
  <udev:bRequest>03</udev:bRequest>
</udev:request>
<udev:setupPacket>
  <udev:name>setBlueGainManual</udev:name>
  <udev:bmRequestType>
    <udev:direction>HostToDevice</udev:direction>
    <udev:type>Vendor</udev:type>
    <udev:recipient>Device</udev:recipient>
  </udev:bmRequestType>
  <udev:requestName>setChromCtl</udev:requestName>
  <udev:wValue>1400</udev:wValue>
  <udev:wIndex>0003</udev:wIndex>
  <udev:wLength>1</udev:wLength>
</udev:setupPacket>
```

Listing 5.5: Access on device mechanism via custom register access

```
enum transfer_error setBlueGain(uint8_t value)
{
        enum transfer_error ret;
        ret = setReg(devcom, pwcblock.awbOthers.blueGain, value);
        PWCMECH_DEBUG_MECH("Setting BlueGain register to: 0x%x.\n", value);
        return ret;
}
```

Listing 5.6: Setting blue gain on PWC Linux-2.6.31

```
int pwc_set_blue_gain(struct pwc_device *pdev, int value)
{
        unsigned char buf;

        if (value < 0)
                value = 0;
        if (value > 0xffff)
                value = 0xffff;
        /* only the msb is considered */
        buf = value >> 8;
        return send_control_msg(pdev,
                SET_CHROM_CTL, PRESET_MANUAL_BLUE_GAIN_FORMATTER,
                &buf, sizeof(buf));
}
```

5.6.1 Results

We successfully used the device mechanism for the Philips webcam to implement a kernel and a user space driver. The implemented kernel module is divided into a device mechanism module called pwcmech and a driver policy module, pwc. The pwcmech module connects to the device and USB interfaces which are also kernel modules called devif and usbif. In kernel mode, the USB interface connects to the Linux USB subsystem.

In the user space driver, the Video4Linux interface calls are transferred to user space by a video loopback module (vloopback) enclosed in the Motion application. For the user space driver, the device mechanism and interfaces appear as libraries, pwcmech, devif and usbif. The user space driver is an application connected to one end of the vloopback module. In order to decompress the video stream, we completely reused the decompression code of PWC. Finally, the USB interface is linked to the Libusb library transferring USB data directly from user space.

We built a bug database for the PWC driver by analyzing the Linux Git logs for the PWC driver from the period from 04/2006 to 02/2012. We only evaluated changes providing bug fixes in comments or patches. From a total of 141 commits, 28 bugs have been found. We categorized them according to Rhyzhyk et al. [RCKH09], see Table 5.4. For this driver, the distribution of bugs in the categories diverges from the average. In particular, the 53.57% of operating system protocol violations show that the Video4Linux interface is still under development and lacks a concise specification. The device protocol violations which our approach avoids comprehend 4 bugs, 14.29% of the found bugs.

Ryzhyk et al. [RCKH09] categorize device protocol violations as following: (1) value; (2) ordering; (3) timing and (4) data race violations. In Table 5.5, we depict the device protocol violations of value type found in our Git repository analysis. In the repository, only fixes of value defect and access ordering types are present, 3 and 1 respectively. Fixes of timing and data races violations have not been found. From the three value defects found, two would not have occurred in

Types of faults	Number	Percentage
Operating System	15	53.57%
Device	4	14.29%
Concurrency	2	7.14%
Programming	7	25%

Table 5.4: Bug analysis of Philips webcam driver. Bug categorization with corresponding percentage of total bug count.

Date of fix	Type	Avoided
04/24/2006	Value shifts	Yes
02/11/2010	USB transfer size	Yes
06/26/2011	Per device missing configuration	

Table 5.5: Value defect violations of device protocol and avoidance with an implementation based on the device mechanism.

our approach. The third bug fix is related to a per device missing configuration which was previously a global configuration, a static variable. This bug could have occurred in our approach because it is related to the driver policy. However, it is improbable because the missing configuration would be generated as a per device configuration.

Furthermore, we have come across another bug which was not noticed before. In the custom access implemented by the driver for the Linux kernel versions 2.6.28-39, the function `pwc_set_saturation` sets a random value for the saturation because it failed to assign the buffer to the value, see Listing 5.7. This is less probable to happen in our approach because there is a clear register context whose access requires fewer parameters. Finally, the `send_control_msg` and its receive counterpart have passed through a cosmetic change in 2006 and have been completely rewritten in 2011 to be adapted for the second version of Video4Linux. This would not have been necessary if standard accessing methods were available, as the device mechanism.

In our implementation, we found two bugs, one of them was critical. In the XML specification of the communication, we defined the setup packet `getDynamicNoise` as a packet to send data and not to receive due to a bit flip in the bmRequestType value. Because our device translator does not verify the direction bit of the linked `sendPacket` and `recvPacket`, the wrong code was built into the driver. Unfortunately, the driver initialization grabs the value of the dynamic noise configuration, and the data wrongly sent to the camera breaks it after a single transfer. Normally, the camera firmware should ignore this packet but we were unlucky. Only after new cameras arrived and the code was carefully debugged, we found out the issue. The only remaining issue was less critical.

Listing 5.7: PWC random value register write PWC Linux-2.6.31

```
int pwc_set_saturation(struct pwc_device *pdev, int value)
{
        char buf;
        int saturation_register;

        if (pdev->type < 675)
                return -EINVAL;
        if (value < -100)
                value = -100;
        if (value > 100)
                value = 100;
        if (pdev->type < 730)
                saturation_register = SATURATION_MODE_FORMATTER2;
        else
                saturation_register = SATURATION_MODE_FORMATTER1;
        return send_control_msg(pdev,
                SET_CHROM_CTL, saturation_register, &buf, sizeof(buf));
}
```

The function which performs control transfers always used an internal buffer, reserved for register data, instead of the buffer provided by its argument. When transferring other data, the data were, thus, incorrect.

In the stream handling, our USB interface removes the necessity of dealing with urb transfers and the USB subsystem directly. Instead, we simply deal with callbacks and buffers. Also the returned data is already contiguous and can be directly stored. This greatly simplifies the task of the driver policy. It has only to look for the end of frame, mark frame completion and gather a new buffer. We found a bug in the buffer handling of our earlier standalone user space driver while adapting it to the device mechanism. If the end of frame was found in a different packet than the last of a transmission, the remaining packets were not attached to any buffer leading to a frame loss. We only found it out because the moves of the scattered data were removed from the code, allowing the other behaviors of the function to become clearer.

The code size for logic is reduced because the access APIs do not belong to the driver any longer. However, the XML increases the code lines for the definition of the access information. But there is considerably more information defined in it too, which guarantees that the correct behavior for register access is implemented. While the PWC driver has 5,896 lines of C code, its driver policy has 5,575. To the driver policy numbers, we still have to add the device mechanism interface and the XML files, respectively 437 and 1,840 lines of code. For the logic part, driver policy and device mechanism, there is a 2% lines of code overhead. While 1,840 lines of code for the hardware description is much, it results in a complete and clear access information of the device which does not exist in conventional drivers. Besides, a lines of code comparison does not consider code clarity. A function call with 5 arguments amounts to the same amount of code as a single line single parameter function call which is less code.

In the modified driver, we did not see considerable performance drawbacks. While the calculation of a register value takes longer by the multiple behavior tests carried out by the device interface, the transfer of the data to the register occurs mostly within a much longer period, 125 μs for high-speed USB. Moreover, low latency and real time sensible data generally use interrupt or DMA functions. These functions are implemented in our method without any overhead. On the other hand, the memory footprint of the conventional driver is 81,519 bytes, while the sum of the pwc and pwcmech modules is 123,668. The difference results from the dynamic allocation of the whole register parameters and communication elements. Moreover, the device and USB interfaces occupy 26,688 and 13,291 bytes. These values have been measured through the lsmod command.

5.7 Discussion

The device mechanism interface can be used to structure the device driver development, while its described systematic composition provides an efficient development strategy. If device mechanism interfaces cannot be trusted, a formal verification for its architectural behavior could be created based on its specification.

The description of the register map and the communication system using XML is very verbose. Its advantage is its generality. While XML can be used directly as front-end, other front-ends can use an XML Schema as back-end as well. A XML Schema simply defines the required document content and its hierarchical structure imposing type safety for the data input. For example, the SPIRIT Consortium plans to offer IP-XACT generation from register descriptions in the SystemRDL language. In case no front-end is available, the XML verbosity can be managed by the use of XML editors.

Due to the separation of the hardware description from its access, the device mechanism is platform independent. It only contains the information required for the device access while the access is implemented by a matched interface. Therefore, another system can be supported through the implementation of an interface for the access mechanism. In contrast, conventional access code is neither reusable nor platform independent.

Besides the enhanced API accessing the hardware seamlessly, avoiding error-prone bit operations, the enforcement of described access behaviors can further improve the device driver reliability. These behaviors complete the description of the hardware types by defining their access permissions, allowed values and bit fields dependency, Table 5.1. Without them, the driver developer has to take care not to violate these rules, whereas a value being passed from the application directly to the device has to be necessarily checked.

Naturally, there is a performance penalty involved in the enforcement of the access rules. In systems running commodity operating systems, interfacing with

devices over a communication system, the very execution time of the I/O operation is expected to be higher than execution time of the rules check. For embedded devices with restricted computation power and real time constraints, the enforcement can be disabled. In this case, the device interface simply executes the necessary bit operations to set or retrieve the bit field information.

5.8 Conclusion & Future Work

We have proposed a new separation of concerns in the device driver development process by modeling policy and mechanism separately. A device mechanism is a universal interface that exposes the functionality implemented by the device using a consistent interface for the system. The policy uses the mechanism to implement the features required by the operating system, effectively matching both interfaces. The device mechanism together with its proposed systematic composition enhances code reuse by coherently defining a device interface which can be used throughout platforms.

Another advantage of the systematic composition is that it provides the hardware semantics missing in program languages, i.e. named registers are provided for access instead of register addresses and bit operations. The actual mapping between the semantics and the physical layer is done using an XML file conformant with the IP-XACT standard. This also provides the capability of linking the hardware and sofware design teams to share the register map definitions. Besides, we define an XML Schema for the description of USB communication which completes a device's access by providing a description of the communication layer. Furthermore, we extended the IP-XACT description and our underlying framework to support custom register access through which register accesses can be bound to USB packets.

The device mechanism is a useful abstraction for a wide range of devices. However, it is not designed to comply with interfaces below the register level, as analog or unregistered interfaces. Also, devices that export generic computing power do not cope with this abstraction. On the other hand, specific coprocessors with a defined computing model can be abstracted by a device mechanism interface. Moreover, to port the framework to another operating system, new interfaces for the required communication systems have to be implemented. The adaptation of new communication systems to the framework requires that device mechanism assumptions about the access of internal device structures not already implemented by the communication system be completed by the framework's communication interface.

As a prototype for this methodology, we wrote the USB interface mappings and ported a USB webcam driver to use the separation of policy and device mechanism. The same resulting device mechanism has been used to create a kernel and a user space driver, showing a reuse case. Moreover, by using our approach, 2 of

4 bugs related to device access could have been avoided, as well as two complete changes to the device access mechanism. Additionally, an unknown bug has been uncovered. For no visible performance overhead, we provide rule checks for register accesses. However, the access description of our implementation is lengthier and the memory footprint of the loaded driver is also 50% bigger than for the conventional approach.

Our next goal is to better investigate the performance impact of our framework. For this, we will implement a network device for which most benchmark tools are available. Moreover, we plan to port PCI communication and implement a driver for our FPGA based coprocessors whose drivers have served as base for this work. Besides, we are going to implement a checker for the correct direction of the linked `sendPacket` and `recvPacket` of a register. In addition, we plan to investigate constraints, whose enforcement would further increase the reliability of a device driver, and match well on top of the device mechanism interface. Later, in accordance with this work, we intend to analyze good alternative descriptions for driver policies in order to avoid development faults.

6 Device Contracts for Drivers

6.1 Introduction

In commodity operating systems, drivers are the major contributor of system crashes [GGP06]. Traditionally, computer and hardware manufacturers support their hardware correcting faulty drivers. However, by the time when drivers would become more reliable, the hardware parts are already outdated. In order to provide high code coverage in conceivable time, static checkers have had substantial success being able to check driver conformity with kernel level programming paradigms and the OS API [BBKL10], [KC10]. Other solutions use runtime techniques to avoid memory corruption [LCFD+05], [HBG+09], [SABL06], bad use of OS data structures [ZCA+06], [CCM+09] and synchronization issues [App07], [Mic06]. But most drivers errors are actually related to violations of the device protocol that is not formally specified or tested [RCKH09].

Furthermore, the same device requires multiple drivers for different operating systems. With strategies for separation of drivers in device and OS specific parts, as described in the Chapter 5, the reuse of the device part poses non-functional requirements (e.g. timing, device protocol) that the driver has to comply with in its functional implementation. Although every device is specific, the nature of their non-functional requirements is the same. Therefore, they can be summarized in contracts. Software contracts enable the description of non-functional requirements targeting the separation of constraint checks from the functional code. In addition, its constant check can signal faulty behavior enabling recovery mechanisms.

Operating systems provide different runtime environments for drivers that cause different interactions with the same device. In driver development, this results in more fault sources and increased porting effort. Interface Description Languages (IDLs) serve as a universal interface specification whose implementation can be used throughout different runtime environments. For that, the IDL specification is translated to the target runtime environment. In this work, we target reuse of the device part of the device driver as a software interface to the device by defining an IDL to export the functions of the device part of the driver. We also define a contract based on these functions specifying device constraints explicitly as pre- and postconditions. Constraint checking is then automatically generated for a target platform. In addition, we define the global state view of the device with an Extended Finite State Machine (EFSM) to enhance the vis-

ibility of the device for tests, also creating a formal description of the device. Based on the defined states, functions' preconditions can be defined detecting ordering defects of device protocol. These faults account for 28% of device protocol violations being the second source of errors after incorrect interpretation of device value [RCKH09]. In our case study, we show the feasibility of our method and violations of component requirement due to the interaction with different environment and components.

In the next section, we discuss the work related with this chapter. Section 6.3 introduces our framework followed by the definition of our Interface Description Language (IDL). Section 6.5 defines the device contracts for drivers and the description of the Extended Finite State Machine (EFSM) for modeling the global dynamic behavior of the device. After that, we explain the translation of the interface definition, EFSM and the contracts to a software interface and enforcement of constraints based on the EFSM description. The proof-of-concept case study based on the Philips webcam Linux driver is presented in Section 6.7 followed by an evaluation of the method. A discussion is carried out in Section 5.7, after which the conclusion follows.

6.2 Related Work

A contract specifies the semantics of component interaction in software. While the syntax of software interfaces comprehends function signature, arguments and return value, contracts define their semantics providing meaning to the parameters and defining related non-functional behavior such as timing constraints and allowed function call order. Bunse et al. [BG06] envisions a unified specification of a hardware component embracing both an informal description of electronics and a software interface with a contract. However, they do not outline this contract nor define how its testing should be done. Our work completes the contracts for the HW/SW interface by defining the contracts and providing a method to test them.

The HW/SW co-design research field proposed synthesis of drivers from the specification of the communicating interface and related software control tackling mostly electric protocol and timing [WB94], [COB95], [BL98], [DRS04], [Jan03]. These systems did not target non-functional requirements of the driver, only the functional part of the driver is synthesized. Further concerns about software reuse are not considered because these systems do not target operating systems. In their environment, the developer of the embedded software is responsible for fulfillment of constraints.

Languages to describe registers, the control elements of the device, have been proposed as well. They also allow the description of special device functions such as interrupts or Direct Memory Access (DMA) [MRC+00], [SYKI05]. Through the definition of state machines [WMB03], the device behavior associated with

register accesses can be further defined. Similarly, Ryzhyk et al. [RCK+09] synthesize a driver from the state machine descriptions of the device and OS ensuring that these behaviors are followed. We define a state machine based on the global view of the device. The device control is not derived from this description. Thus, only constraint relevant behavior has to be modeled. The resulting EFSM is comparable simpler. Still, it enhances the visibility of the device allowing more tests that can be applied to any function. Following the contract testing methodology, we define non-functional requirements as preconditions for device functionality based on the global states of the device. Furthermore, our system allows parallelism, while previous work serializes the complete driver by using a `spinlock` around every driver function [WMB03], [RCKH09].

For verification, Kudlugi et al. [KHSP01] proposes a hardware/software cosimulation technique that can early debug a driver implementation with regard to the target hardware. Similarly, Ryzhyk et al. [RKM+10] propose merged verification process where the testbench is designed with driver verification in mind. Their intention is to avoid multiple descriptions of the device behavior for testing. However, both testbench and high-level simulation are based on functional description languages. By contrast, we propose to define non-functional requirements explicitly and synthesize runtime tests.

6.3 Framework

The framework for contract implementation is based on the driver architecture proposed in Chapter 5. The implementation of a driver is divided in a bottom layer, the device mechanism, and a top layer, the driver policy. The architecture aims at the separation of the device access from data processing. The device mechanism exports the function implemented by the device as software functions to the driver policy that implements the behavior expected by a specific OS. This way, the hardware is handled by the device mechanism and the OS by the driver policy.

Devices imply non-functional (e.g. timing and ordering of device protocol) constraints that are not handled in the device mechanism. This is enforced by this framework as an extension of the device mechanism. The framework contains a specification language to define non-functional requirements explicitly. The functions of the device mechanism are then exported while equipped with constraint enforcement. The result is a device mechanism with control of its software protocol. Violations of these constraints result in the device mechanism denying the device access and returning an error code. The driver policy is then responsible for handling the error properly. The idea is to avoid device failure and ensure correct functionality. Furthermore, it separates constraint specification from test generation and execution avoiding repeated design of tests and clutter of functional and defensive programming code.

Constraints are defined based on an interface description for the device mechanism. The software functions to be exported are defined in XML and related non-functional requirements are included. From the defined interface and constraints, code generators produce a function augmented with code for constraint enforcement. Some constraints are based on states of an EFSM that is described along with the interface and implemented by the code generators. Because the generated code is intended to run on different platforms, macros map the usage of synchronization entities and function export mechanism to the different platforms.

6.4 Interface Description Language

Interface Description Languages (IDLs) allow for encapsulation because they are the only communication path of the described interface. Following the separation of device mechanism and driver policy, we define a language for the description of the device mechanism interface. The interface has to be understood by client and server equally, regardless of their platform. For that, types for arguments and return values are translated and the function prototype is exported.

The device mechanism is platform and communication independent. It is described with device registers and endpoints. Through code generation and special interfaces for register map and communication, software can access the device registers and endpoints directly without the need of special functions or bit operations. Interrupt and DMAs are abstracted by a callback mechanism. Registers and endpoints are then wrapped in functions that require as arguments only the data to be transferred. In the device mechanism, these data are designed to be of C standard type.

Through the device mechanism interface, only data from registers, bit fields and streams are passed from/to the device. Therefore, we define our interface using these data and translate them to C which is used for both the device mechanism and the driver policy modules. Moreover, callback and context are added to these types because they are necessary to define interrupt handlers. These are the same types defined for the device mechanism. Their translation is summarized in the Table 6.1. The `context` defines a pointer to a structure that contains data to be made available for the callback function. Callbacks must be registered together with a `context` before their use. The `context` is defined generically because it is defined and used by the driver policy, not the device mechanism. The stream is defined using the fixed width integer `uint8_t` instead of `unsigned int` because USB transports data in 8 bits and we only support USB currently. Callbacks' return value indicates if the callback will be reactivated or finished. Their parameters are the `context` defined on registration, a stream, and two USB transport parameters that can be required to identify frames in some simple protocols.

IDL Type	Device Mechanism Type	C translation
Regiter/Field Value	ρ ϕ	unsigned int (in) unsigned int * (out)
Stream	σ	uin8_t * , size_t
Callback		enum cb_ret (*)(void * context, uint8_t * buf, size_t len, size_t * p_packet_len, size_t nr_of_packets)
Context		void *

Table 6.1: Argument type translation.

6.5 Device Contracts

A contract specifies duties and rights in component interaction. In a component-based approach, components are thoroughly tested and trusted. However, the assembly of reliable components does not guarantee a reliable system because components' interaction can be faulty. A component can interact with other components, the runtime environment and also with the physical environment to some extent. For instance, it is crucial that the size of variables accords to the physical range of the unit they hold. In the interaction of software components, the corresponding software protocols have to be mutually fulfilled for correct common behavior.

This is specified with regard to the client/server principle. Every component can assume the role of a server and a client. It is a client when it requests services and a server when it offers. Every server/client relation is based on a service that is defined as a function in software. The contract defines properties that have to hold for correct function execution, and the properties that are guaranteed after execution. While the function prototype is a syntactic definition of a service, the contract specification provide the semantics of the service covering the non-functional properties related to the function execution.

Contracts specify non-functional requirements for services as pre- and post-conditions. Preconditions define the requirements for correct function execution, while postconditions specify the constraints of this execution. Preconditions are associated with a component's configuration and argument ranges, while the post-conditions define properties of the execution and return values.

The device interface is always custom. On the other hand, the nature of devices and drivers imposes specific non-functional requirements/constraints that can be summarized in contracts. Drivers have to handle the device configuration, allocate memory for device data and, when an asynchronous data model is not possible, wait for completion of commands. We summarize these properties in pre- and postconditions in Table 6.2. Current preconditions envisage states and state machine variables whose update can be attached to functions and their

Contract Property	Pre-Condition	Post-Condition
EFSM variables (e.g. Buffer Size)	X	
EFSM State	X	
Execution Time		X

Table 6.2: Properties used as contract's conditions.

arguments. An example is its usage to control buffer sizes. Postconditions can be timing constraints related to the execution of the function. It is related to the time the device takes to complete the execution of the function in spite of possible non-blocking return of commands. We force the function to wait for commands' completion, only updating the EFSM after the time has elapsed. In a multithreading environment, that does not compromise performance because other tasks can be executed meanwhile. Because device data (i.e. registers and endpoints) require special handling, they are defined and their value ranges are enforced by the device mechanism. Thus, they are disregarded here.

6.5.1 Device State View

Software components depend on external services and function calls have to follow the function prototype providing the arguments in correct order. Furthermore, components can have different operational modes, and exhibit a dynamic behavior in which functions have to be executed in a specific order to provide the expected functionality. There are different ways to describe these dynamic behaviors of a component. In particular, a state machine representation is widespread and provide an event-based model. In a state, events to which the machine reacts are described. Other events are rejected not having any effect on the machine.

We describe the behavior of the device mechanism with an extended finite state machine. Because the device mechanism is designed to be an image of the device itself, this description corresponds to the device behavior. With the execution of the state machine, the state of the device can be assessed. With the definition and enforcement of a state precondition for function call, the device access is only performed if the device is in the required state. This way, correct usage of the device in conformance with its dynamic behavior can be ensured for different driver policies.

6.5.2 State Machine Description

The state machine is generated from the description of (1) states; (2) variables; (3) guards; (4) events; and (5) transitions. Variables hold global values which transitions may depend on. Guards evaluate relevant variables and allow or block a transition. The transitions are defined by an origin state, a target state,

Macro	Kernel	User Space
EXPORT(FUNC)	EXPORT_SYMBOL_GPL(FUNC)	
MSLEEP(MSEC)	msleep(MSEC)	usleep(1000*MSEC)
STATEMUTEX	struct rw_semaphore	pthread_mutex_t
INIT_MUTEX(MUTEX)	init_rwsem(MUTEX)	pthread_mutex_init(MUTEX, NULL)
DEST_MUTEX(MUTEX)		pthread_mutex_destroy(MUTEX)
DOWN_READ(MUTEX)	down_read(MUTEX)	pthread_mutex_lock(MUTEX)
DOWN_WRITE(MUTEX)	down_write(MUTEX)	
UP_READ(MUTEX)	up_read(MUTEX)	pthread_mutex_unlock(MUTEX)
UP_WRITE(MUTEX)	up_write(MUTEX)	

Table 6.3: Macro definitions for the implementation of contracts in Linux, in kernel and user space.

triggering events and a guard. If (1) a triggering event occurs; (2) the current state is the defined origin state; and (3) the guards evaluate true, the state is updated.

Variables are updated independently or by the state machine transitions. An update is described by a value assignment. Two mechanisms are provided. Either value assignments are associated with events or with successful transitions.

The device mechanism provides events and variable values to the EFSM. Its functions are used as events and their arguments can be used as values for variable updates.

6.6 Translation of Constraint Description to Checks

The framework targets the Linux kernel and user spaces. From the XML description of the interface, state machine and functions with constraint enforcement are generated. The generator is written in Python using the Lxml library with the Objectify API. The generated code handles synchronization by using macros that are defined for both kernel and user space.

In the Linux kernel, function of modules can be made available to other modules through the macro `EXPORT_SYMBOL_GPL`. Furthermore, C standard types are used for the arguments of the functions of the module interface. This enables us to export functions from the device mechanism module to the driver policy module and interpret types for passed arguments and return values. In user space, library functions do not have to be exported.

More complex is the definition of the synchronization entities. Semaphores have to be defined, initialized, locked and unlocked by macros. In user space, POSIX mutex can be used while we use `rw_semaphore` in kernel space. For timing control, the user space relies on `usleep`, while kernel uses `msleep` to avoid busy waiting. Table 6.3 summarizes the framework defined macros.

Listing 6.1: XML description of a function with contract conditions to be exported kernel wide.

```
<operation name="registerVideoCallback">
  <parameters>
    <in name="func" type="callback"/>
    <in name="context" type="context"/>
  </parameters>
  <return type="idlError"/>
  <internal>_registerVideoCallback</internal>
  <preCondition mode="state">ready</preCondition>
  <preCondition mode="variable">videoBufferLevel >= 2*10*devmech->com->
  standard_descriptors.highspeed.cameraStream.Video->wMaxPacketSize</preCondition>
</operation>
```

Pre- and postconditions are defined in the context of an IDL operation, see Listing 6.1. From the IDL operation, a C function is generated with a function prototype corresponding to the operation name, parameters and return type. The device access is executed by a C function of the device mechanism declared under the `internal` tag.

We define a state machine in XML whose implementation follows the rules of Subsection 6.5.2. Listing 6.2 shows the description of the XML state machine for the device mechanism of a Philips webcam. Fig. 6.1 shows the UML representation of this machine.

The state machine is run in parallel to the device mechanism and does not necessarily affect it. Functions of the device mechanism are the events of the state machine and state transitions may result from them. Also, a function can be made dependent on a state through the definition of a state precondition. In that case, the device mechanism function declared as `internal` is not executed if the current state does not accord with the defined precondition denying the access to the device.

When an IDL function is called, it is executed in the following order. First, preconditions are evaluated. If every precondition is fulfilled, device access occurs. If the device access is successful, variables associated with it are set. After that, if the current state is equal to the origin state and the guard evaluates to true, the variable updates of the transition take place followed by the update of the machine state. The C code translation of the XML operation is shown in Listing 6.3. If the function updates the state or a variable, it has to acquire a writer lock. Functions that do not update the state machine but have preconditions on state or variables acquire a reader lock. Functions unrelated to the EFSM do not acquire any lock.

To the main module C structure of the device mechanism, we add a pointer to the C structure `efsm` generated for the state machine, see Listing 6.4. The main module structure is used by the generated code to access the `efsm` structure, evaluate the current state and update it. The `efsm` structure already contains

Listing 6.2: XML description of the extended finite state machine of the device mechanism for the Philips webcam.

```
<extendedFiniteStateMachine>
  <declaration>
    <states>
      <state>idle</state>
      ...
    </states>
    <variables>
      <variable>videoSet</variable>
      ...
    </variables>
  </declaration>
  <behavior>
    <assignments>
      <assignment>
        <event>setVideoMode</event>
        <cmd>videoSet = true</cmd>
      </assignment>
      <assignment>
        <event>assignVideoBuffer</event>
        <cmd>videoBufferLevel += len</cmd>
      </assignment>
      ...
    </assignments>
    <transitions>
      <transition from="ready" to="streaming">
        <name>configure</name>
        <trigger>
          <event>assignVideoBuffer</event>
          <event>registerVideoCallback</event>
          <event>setPower</event>
        </trigger>
        <guard>videoBufferAssigned and videoCallbackAssigned and power == 0</guard>
      </transition>
      <transition from="ready" to="idle">
        <name>finish</name>
        <trigger>
          <event>unassignVideoBuffers</event>
        </trigger>
        <update>
          <cmd>videoBufferAssigned = false</cmd>
          <cmd>videoSet = false</cmd>
          <cmd>commandSent = false</cmd>
          <cmd>videoBufferLevel = 0</cmd>
        </update>
      </transition>
      ...
    </transitions>
  </behavior>
  ...
</extendedFiniteStateMachine>
```

Listing 6.3: Kernel wide exported C function generated from IDL operation description.

```
enum idl_error registerVideoCallback(DevMech * devmech, enum cb_ret(*func)(void * context, uint8_t *
      buf, size_t len, size_t * p_packet_len, size_t nr_of_packets), void * context) {
    enum idl_error ret;
    DOWN_WRITE(&devmech->efsm->mutex);
    if ( devmech->efsm->state != ready )
    {
        DEVCONTRACT_DEBUG("registerVideoCallback(): state preCondition failed! Expected state,
            ready, differs from current state: %u\n", devmech->efsm->state);
        UP_WRITE(&devmech->efsm->mutex);
        return INVALID_STATE;
    }
    if ( !( devmech->efsm->videoBufferLevel >= 2*10*devmech->com->standard_descriptors.highspeed.
        cameraStream.Video->wMaxPacketSize ) )
    {
        DEVCONTRACT_DEBUG("registerVideoCallback(): variable preCondition failed! Expected
            videoBufferLevel value: 2*10*devmech->com->standard_descriptors.highspeed.cameraStream.
            Video->wMaxPacketSize, differs from current value: %u\n", devmech->efsm->
            videoBufferLevel);
        UP_WRITE(&devmech->efsm->mutex);
        return INVALID_STATE;
    }
    ret = _registerVideoCallback(devmech, func, context);
    if ( ret == 0 )
    {
        DEVCONTRACT_DEBUG(">> registerVideoCallback(): updating state machine variables\n");
        devmech->efsm->videoCallbackAssigned = true;
        DEVCONTRACT_DEBUG("videoCallbackAssigned <- %x\n", devmech->efsm->
            videoCallbackAssigned);
        DEVCONTRACT_DEBUG("<< registerVideoCallback(): state machine variables updated\n");
        transitionConfigure(devmech);
    }
    UP_WRITE(&devmech->efsm->mutex);
    return ret;
}
EXPORT_SYMBOL_GPL(registerVideoCallback);
```

Listing 6.4: C state machine generated from the XML state machine description.

```
struct efsm * init_efsm(void)
{
    struct efsm * efsm = ALLOC(sizeof(struct efsm));
    efsm->state = idle;
    efsm->videoSet = false;
    efsm->commandSent = false;
    efsm->videoBufferAssigned = false;
    efsm->videoCallbackAssigned = false;
    efsm->power = 0xFF;
    efsm->videoBufferLevel = 0;
    INIT_MUTEX(&efsm->mutex);
    return efsm;
}

void stop_efsm(struct efsm * efsm)
{
    DEST_MUTEX(&efsm->mutex);
    FREE(efsm);
}
...
void transitionRelease(DevMech * devmech)
{
    DEVCONTRACT_DEBUG(">> transitionRelease() called!\n");
    if (devmech->efsm->state == streaming)
    {
        devmech->efsm->videoCallbackAssigned = false;
        DEVCONTRACT_DEBUG("<< transitionRelease(): state updated to ready\n");
        devmech->efsm->state = ready;
    }
    else
    {
        DEVCONTRACT_DEBUG("transitionRelease(): only valid from state streaming but current state is
            %u\n", devmech->efsm->state);
        DEVCONTRACT_DEBUG("<< transitionRelease() skipped\n");
    }
}
```

Listing 6.5: Generated efsm structure used by IDL functions for EFSM and contract execution.

```
struct efsm
{
    enum states state;

    unsigned int videoSet;
    unsigned int commandSent;
    unsigned int videoBufferAssigned;
    unsigned int videoCallbackAssigned;
    unsigned int power;
    unsigned int videoBufferLevel;

    STATEMUTEX mutex;
};
```

the state, variables and required mutexes. Listing 6.5 shows the declaration of the `efsm` structure for our example.

Finally, because the Linux kernel does not have an exception mechanism, we also add debugging information to the generated code. Through the definition of a macro, the debugging information is enabled and misuse of the device mechanism can be followed by logs.

6.7 Case Study

Our case study is a proof-of-concept of our method showing its feasibility. We implement a driver for the Philips webcam based on the case study of the Chapter 5. The Philips webcam driver is divided in two modules, the device mechanism, device part, and the driver policy, different for each platform. In the earlier case study, the device mechanism contained only the functional requirements for device control. Its correct usage depended on the interaction with the driver policy. In this case study, we describe the expected interaction between device mechanism and driver policy through an EFSM. Moreover, we describe non-functional timing, memory size and protocol ordering requirements that constrain correct device usage.

The dynamic behavior of the camera is specified on top of its device mechanism. While most device mechanism functions can be arbitrarily called, some functions change the device/device mechanism state and others can only be called from specific states.

A webcam outputs the video data stream that has been acquired by an image sensor. The driver configures the frame mode (picture size and frame rate) and optic values (brightness, contast, etc.). After a frame mode has been selected, the driver can receive the data stream, generally through DMA. In parallel, optic values can be changed to modify picture properties such as brightness or contrast.

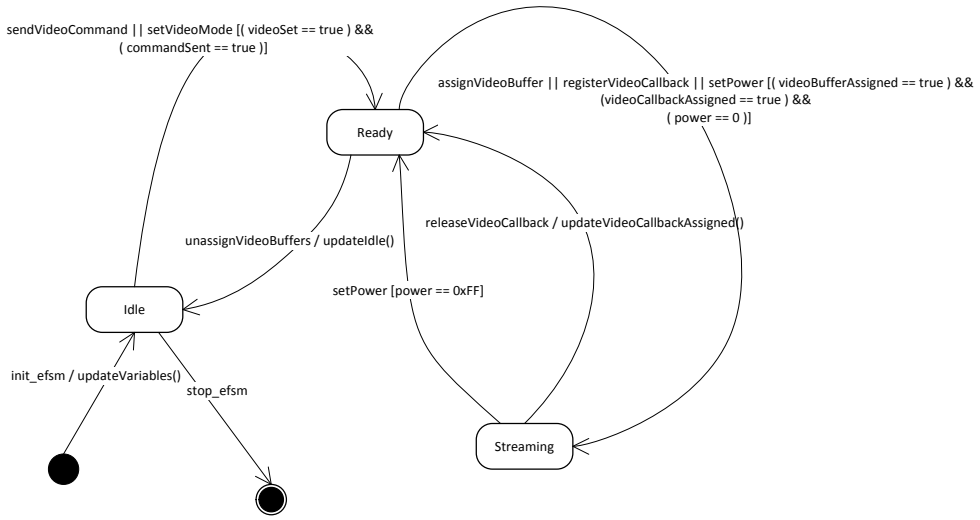

Figure 6.1: UML state machine is describing the dynamic behavior of the device mechanism of the Philips webcam.

Upon device connection or module instantiation, the state machine variables are updated and the machine initialized to the idle state. The state ready is assigned when a specific frame configuration has been selected for the device. With the configuration of buffers, callback and device power, the device can advance to the streaming state. In this state, after acknowledgement, buffers are filled and callback functions called. The dynamic behavior of device and its device mechanism with respect to the functions calls can be seen as the UML state machine diagram of Fig. 6.1.

We describe the interface of the device mechanism using XML. The defined functions are used as events for the state machine that is described in the same XML document. We map the UML state machine above to our IDL description manually. With the description of preconditions, functions are made dependent on the state and state variables. This enforces that certain functions follow the dynamic behavior of the device.

Table 6.4 summarizes the defined conditions for the Philips webcam contracts. Besides the state conditions, the `buffer_level` precondition ensures that at least 10 ms of pending transfers can be buffered. This is a heuristic USB recom-

Function	Pre-Condition	Post-Condition
sendVideoCommand setVideoMode	state: idle	
assignVideoBuffer unassignVideoBuffers	state: ready	
registerVideoCallback	state: ready buffer_level $\geq$ $\frac{2 \cdot 10\ ms}{2 \cdot \frac{10 \cdot (7 \cdot high\_speed + 1) \cdot packet\_size}{2^{bInterval-1}}\ bytes}$	
acknowledgeVideoCallback	state: streaming	
setMotor resetMotor		5,000 s

Table 6.4: Philips webcam contract's conditions.

mendation to avoid data loss. This condition was simplified to `buffer_level` $\geq 2 \cdot 10 \cdot packet\_size\ bytes$ because the camera only works at full speed and $bInterval$ is always 1, see Listing 6.1. Furthermore, timing postconditions are described for camera movement such as tilt and pan configurations. The function blocks until the expected result is achieved.

6.7.1 Results

The XML file for the contract definition of the Philips webcam device mechanism has 526 lines of code measured with the CLOC tool from sourceforge. This file holds the state machine description, the description of the functions exported from device mechanism and their contract conditions. The IDL generates a header and a source file of respectively 84 and 639 lines of code.

The state and variables of the state machine have to be updated atomically to avoid race conditions. We use the Linux `rw_semaphore` for that which implements a readers-writer lock. Because the device mechanism callbacks are executed by `workqueues`, in the context of a kernel process, our state machine does not require a `spinlock`[1]. This way, any function that does not modify a state machine variable or state can execute in parallel. Functions that update the state machine variables or state execute alone and block any other function dependent on these variables. The callback functions do not update state by design (see Chapter 5) and can thus be executed in parallel. Since these are the performance critical functions of the driver, the driver performance is not compromised.

We analyzed the Git logs of the Philips webcam driver of the Linux kernel from 24/04/2006 to 02/02/2012. In this case, only 4 bugs, 14.29% of the defects, are related to the device protocol. From these bugs, there is one ordering defect because the Light-Emitting Diode (LED) control was being called before the

[1]Interrupts and callbacks in interrupt context have to block with a spinlock (busy waiting) if they must, because they run in the kernel thread that cannot be preempted.

input arguments of the function were initialized. This fault could be detected by our approach and its failure avoided.

The device mechanism with automatic constraint check is a Linux module that can be used by two driver policies, a Linux kernel driver and a user space driver. The user space driver is an application with computing time that performs the data acquisition by itself. The video application only configures and requests complete frames. The kernel driver is an interface without computing time. It only reacts to calls and consumes computing time of the callee. Due to its strong coupling, it became apparent that different applications handle the driver differently. Using the Cheese application, a NULL pointer dereference occurred, while video worked with Ekiga. A new buffer request function returned NULL on the absence of free buffers, while it used to return the oldest full buffer earlier. The buffer pointer was thus not checked for NULL before usage. The failure occurred more frequently with the Cheese application because it uses three frame buffers while Ekiga only two. This is a failure of an interaction between application/-driver and kernel, while we focus on the driver/device interface. But it shows that a component can fail if used in an unspecified way, also when it previously worked.

6.8 Evaluation

We first analyze the portability issues related to our approach. Later, we show which kind of bugs can be avoided using this approach and how representative they are for drivers.

6.8.1 Portability

The device contracts strategy requires a code generator that translates the state machine description to an implementation and constraints to tests, also providing debug code. Our current implementation of the IDL/constraint generator has 592 lines of code. Moreover, the macros required to support a target platform are minimal only taking 42 lines.

Because operating systems modules are programmed in C, the same target language of the code generator, the effort to port the system to another operating system is minimal. For that, the header file with the nine macros required for the synchronization has to be extended with corresponding implementations for the target system.

In order to have a race-free state machine, access to state variables have to be linearized. Generally, this is achieved by the use of a `spinlock` that allows waiting in interrupt context and impede that any task is fulfilled in parallel. We use the readers-writers-algorithm for mutexes instead that allows multiple functions to read the data but only one to write it. This allows for a multithreaded

Drivers	Value defects	Ordering defects	Timing defects	Data races
Broad analysis [RCKH09]	61%	28%	8%	3%
Philips webcam	75%	25%	0%	0%
USB network devices	64.29%	26.79%	7.14%	1.79%

Table 6.5: Categorization of violations of device protocol for different drivers and respective percentage of total violations.

driver. However, this requires that the operating system has an implementation of the readers-writers-algorithm in form of a special mutex. A further special consideration is that callbacks are not called in interrupt context to avoid the necessity of a `spinlock`. This requires that the target operating system has a mechanism to postpone tasks to be run in context of a dedicated kernel process.

6.8.2 Error Analysis

In the root-cause analysis of driver defects by Ryzhyk et al. [RCKH09], device protocol violations are responsible for 38% of the defects. The majority of faults are related to value defects, 61%, followed by ordering defects 28%. In our method, the device mechanism layer is already responsible for avoiding value defects, see Chapter 5. Through the implementation of a state machine view of the device and definition of state related preconditions, we intend to target ordering defects too.

From 24/04/2006 to 02/02/2012, we analyzed the Git logs of the USB network devices. These are Ethernet, modem or wireless Local Area Network (LAN) devices that are connected to the computer through USB. The defect distribution for these devices are close to the broader distribution found by Ryzhyk [RCKH09]. From the device violations, 64.29% were related to value defects, followed by 26.79% of ordering defects. More details can be seen in Table 6.5.

Many ordering defects occur because the driver is not aware of the global device configuration. For example, 12 of 56 total USB network bugs occurred because initialization or power resume failed, sometimes due to incomplete termination or suspend code. Further five bugs occurred because the configuration of the data transmission deviated from the driver's assumption; 3 times related to data transfer disregarding link configuration; and 2 times related to the configuration USB transfer lengths.

In our approach, we are able to detect these faults and avoid their failures because the driver is bound to the device configuration that is represented by the state machine variables or states. If the preconditions are correctly assigned, a device function that depends on a state or configuration will not be executed and the occurrence logged instead of leading to unexpected behavior. Moreover, timing is enforced by delays provided the device mechanism function has an according timing postcondition.

6.9 Discussion

The contract approach for device drivers enables the separation of functional implementation from tests for non-functional requirements. Instead of manually ensuring that function implementation fulfills non-functional requirements implicitly, we define constraints to these functions for which tests are automatically generated. Non-functional requirements are generally grasped by documentation. With our approach, their fulfillment is enforced automatically also avoiding multiple design of tests.

Constraints are often related to device configuration and device's dynamic behavior. We describe this behavior through an EFSM. EFSM is a sound, well known model whose description is possible with various tools. It also allows for a reduced state number due to the inclusion of state variables and guards.

The device mechanism's functions are appropriate events for the state machine because they mirror concrete device operations mapping the device functionality. These functions do not have to necessarily cause a state change but any function can be designated for this purpose. The description of a global dynamic model of the device gives an insight into complex functionality/configuration dependencies. The dependencies can be then directly declared as function's preconditions in an easy and explicit way. Furthermore, recovery procedures can be extracted from the global state machine description by calculating which functions have to be called to reach the required state.

Generally, state machine implementations rely on a `spinlock` impeding parallel execution of the driver. Because we use the readers-writer lock, the parallel execution of functions that do not update the state machine is possible.

The usage of XML is intentional because we do not want to specify a description language. Our work addresses the specification of requirements and the implementation of constraint tests while using standard descriptions. XML enables us to define the necessary data in a context. The XML document can be used as backend for different description languages or visual description of an EFSM, such as UML state diagrams for example.

6.10 Conclusion & Future Work

We define device contracts for drivers to specify non-functional requirements on top of a functionality layer, the device mechanism, and allow its online testing automatically through a framework. The framework is based on an IDL and an EFSM description of the device, a code generator and underlying platform descriptions. Non-functional requirements are summarized in pre- and postconditions based on EFSM states, variables and timing. These conditions are associated with functions of the device mechanism whose declaration occur through an interface description language. Then, a code generator exports the functions

and generates constraint tests from the interface description. The generated code is based on macros to facilitate porting to other platforms.

The proposed IDL is adapted to the software data types used in the device mechanism, register/bit field values, streams, callbacks and contexts. A described interface is translated to a C header and source exporting functions with a standard C translation of the data types for software use.

In the IDL, pre- and postconditions are associated with the exported functions. Postconditions describe the execution time of the device functionality. Preconditions define a specific device configuration required for correct function execution. The device configuration is declared as the state of an EFSM or a range value of an EFSM variable. Tests for fulfillment of these conditions are automatically generated together with an implementation of an EFSM that describes the dynamic behavior of the device. The EFSM's events are defined as the device mechanism's functions. By executing the EFSM in parallel to the device mechanism, the device configuration is mapped to the EFSM states and variables.

The implementation of an EFSM, timing control, export of functions and data types translation requires a target platform. Required functions are summarized in macros that need an implementation on the different target platforms. By using `workqueues` for callback implementation, we can avoid `spinlocks` and busy waiting altogether, suspending the requesting process if necessary. Semaphores for synchronization and timers for delaying execution have been translated to Linux Kernel using readers-writer lock and scheduled delay. In user space, POSIX mutexes and a standard sleep routine have been used. The proposed approach can be ported to other platforms easily by creating the required macros.

We developed a Philips webcam driver as a proof-of-concept of our method. One known protocol ordering defect could have been detected by our approach. Furthermore, a fault in our Linux kernel driver policy can depict the challenges of component-based design imposed by the complexity of current systems. The Cheese application acquired more buffers than expected by our driver policy leading to a NULL pointer dereference. Our work targets the device/driver interaction and not the application/driver interaction. This violation was thus not automatically detected. But it exemplifies that runtime tests for component interaction is crucial because different applications using the same components exercise them in different ways, which can lead to a failure due to an unexpected corner case execution.

In our approach, configuration and function dependencies can be efficiently described. Ordering defects correspond broadly to 28% of device protocol errors and 10% of all driver defects [RCKH09]. These defects occur due to misunderstanding or missing specification of device configurations and arise from an unexisting overview of the device behavior in drivers. Because our approach covers this lack, these faults can be detected and their failures avoided.

A possible optimization for this method would be to disable constraint tests

that can be proven by static model checkers. For example, this could be done for execution paths that do not depend on user input. Wider adoption of the method could be achieved with support for automatic generation of our XML documents from UML tools. Also, an IDL is an interesting concept as basis for constraints. In the future, constraints for standard libraries for drivers could automatically enforce certain interface protocols this way. Furthermore, other constraint types could be considered. The inclusion of Linear Temporal Logic (LTL) constraints [HR04] can enrich the testing coverage. Also, constraint sets for specific execution modes could be envisoned, such as reentrancy and interrupt context. With techniques for driver implementation in user space and the use of IDLs, the driver policy could be implemented in safe languages and still communicate with the C runtime environment.

7 Conclusions

Current development methods take advantage of reliable components to build reliable systems. But the interaction between these components is only informally specified, not simulated and remain untested leaving room for future failures. Therefore, this thesis aims at the specification, test generation and runtime test of the interaction between electronic components and between software and electronics.

We have described a suitable specification for hardware and for hardware-software interaction, and demonstrated the feasibility of our testing approach through case studies avoiding interaction failures. In this chapter, we review our achievements, giving a comparison to current methods where possible, and depicting limitations and difficulties of our method. Then, we outline the future work of this thesis generally. Future work related to the specific chapters is described in each chapter individually.

7.1 Contributions

7.1.1 Hardware Contracts

In Chapter 3, we have presented a method to specify and test hardware interaction through hardware contracts. Requirements for component operability are defined as environmental parameters and compliance with communication protocols. On their compliance, component functionality and its constraints are assured. In order to test these, circuits have been proposed. In the case of the communication protocols, the high speed input data require special circuits for signal categorization in order to evaluate errors. Through this categorization, evidences for circuit faults can be evaluated.

We extended the contract specification from software to hardware enclosing hardware specific requirements and constraints. A similar hardware specification method does not exist. Similar test methods have been proposed, such as bus monitors that are able to analyze frame composition and bit faults of communication systems [PHZ+05], [ARSH05]. Crossman et al. [CGMC03] relate specific behaviors of sensor signals to specific faults using signal parameters. We extended the bus monitor approaches by the categorization of the signal reducing the amount of data to enable the test of a corresponding specification. Our generic specification of signal parameters for hardware modules is similar to the

one of Crossman et al. [CGMC03]. This endorses the potential of the approach to give evidence of circuit faults.

Limitations of our approach are related to our test circuit. It is not appropriate for high speed communication protocols. Furthermore, it represents a considerable overhead for simple systems. However, the overhead is well justified for safety critical applications because critical operations can be constantly verified.

In Chapter 4, we extended the generic payload of the Transaction Level Model to comprise the signal characteristics defined in the hardware contracts in order to allow signal fault simulation within the system level design. Based on this extension, we simulate an acquisition system with connected sensors that inject signal faults in their data transmission based on probability distributions. Signal characteristics are interpreted and corresponding signal faults are assigned if operating limits of the communication system are exceeded. Assigned signal faults are then translated to digital faults. For this model, we have analyzed fault propagation through a bus system with fallback modes enabling fault recovery. This work complements the work of Beltrame et al. [BBM09] and Chen et al. [CWP08] extending the verification of TLM designs to the signal level.

7.1.2 Device Contracts for Drivers

The subsequent chapters consider the interaction between hardware and software. Software interfaces with hardware through a device driver that requires information about internal hardware elements. This shifts the black-box view of component-based design to a grey-box.

In order to allow for a contract specification of a device, we designed a driver architecture that wraps the access to internal device elements exporting its functionality through a software interface, see Chapter 5. We define rules to avoid that device data be processed by the device interface leaving this task for the upper driver layer. Moreover, the description of the internal device elements and accessing them is a non-trivial task in software because these elements cannot be handled directly by the programming language. Therefore, we propose an XML description with declaration of a register map and communication system that is translated to C structures. Through access interfaces, these C structures can be used to access the device directly respecting the access behavior of the internal elements.

Our architecture is a new approach that hides the internal parts of the device separating access from processing. It enables the description of the device based on the device implementation instead of a desired functionality. Ryzhyk et al. [RCK+09] propose that the device specification–the lowest layer of their architecture–complies with a generic device interface called device class. However, this architectural layer is not very different from a regular driver because OS drivers have to comply with a generic device interface too. The advantage of

basing the lowest architectural layer on the device implementation instead is its completeness with regard to constraints. By contrast, non-functional constraints are hard to be provided for an abstract interface, as a generic device interface.

The systematic development bases on earlier work of register map declaration [MRC+00], [SYKI05], [WMB03], [CE04], providing further runtime checks for register consistency. Besides the register map declaration, we define the communication interface with the device that is responsible for interrupt and DMA functionality. Furthermore, we enable the access to registers through custom functions, such as through USB setup packet transmission. This allows for register access on absence of memory mapped or port I/O. Our systematic development also allows for platform independence because register map and communication descriptions and their generated C structures are platform independent.

The proposed device interface abstraction is limited to peripheral devices. Devices that export generic computing power do not cope with this abstraction. Also, devices with interfaces below the register level, such as analog or unregistered interfaces are not compatible. The effort to port the systematic development framework to another operating system lies in the implementation of a communication system interface. The communication system interface has to comply with requirements of the framework. The current Linux USB interface comprises approximately 2 thousand lines of code.

The device interface provides a clean access to the device. In Chapter 6, we define device contracts for drivers through the description of the dynamic behavior of the device, using an Extended Finite State Machine (EFSM), and by associating constraints to the functions of its device interface. These constraints are based on timing and on the EFSM. An IDL declares the device interface, and describes its constraints and the EFSM. The device functions are treated as events that can trigger state or variable changes. The description is carried out in an XML document from which a software interface is generated respecting the described behaviors and constraints. Also, logging facilities are generated to allow tracking of faulty usage. The proposed contracts are able to avoid roughly 10% of driver failures. These are related to ordering defects of the device protocol, being the second major cause of driver failures after device value defects, 23% [RCKH09], that are targeted by the device mechanism.

By defining contracts for the device interface, we augment the device access with runtime checks for non-functional constraints. Early work on contracts for embedded systems only grasped possible constraint types for hardware [BG06]. We have defined constraint types, a specification description and developed automatic tests. In the HW/SW codesign research field, state machines have been proposed for the description of the dynamic behavior of the device [WMB03], [RCK+09]. In contrast to these descriptions, our model allows for a simplified view of the specific device because the functionality does not depend on it. Thus, only constraint relevant configuration has to be modeled. Moreover, because this

layer is based on the device implementation, it is fully specified allowing for constraints to be defined naturally. In contrast, Wang et al. [WMB03] interleave the dynamic behavior with register accesses and Ryzhyk et al. [RCK+09] target a generic device interface without specially considering device access. Our system allows parallelism by automatically applying the readers-writer-lock algorithm instead of using a `spinlock` around the complete driver.

The effort to port the system to another operating system is minimal because the code generator outputs C code which is the language operating systems' modules are programmed in. Nine macros are required for this porting. However, a multithreaded driver can only be achieved if the operating system supports the readers-writer-algorithm. Alternatively, a regular mutex can be used that would again serialize the driver execution.

7.2 Future Work

Runtime tests involve performance penalty. But, in contrast to static tests, they are able to check external influence on systems, the user input of a software for instance. Tests of internal interaction can occur statically or be formally verified ensuring that certain constraints are always met. In such cases, runtime tests can be skipped without performance or reliability penalty. Therefore, a smart balance of runtime checks for external dependencies and static checks for internals provides an appealing future solution.

Also, a framework that understands specified interfaces and their constraints could provide the assembly of systems as easy as connecting blocks in a diagram. In order to provide a reliable system, interfaces and constraints compliance have to be statically checked and inconsistencies detected. Otherwise, new modules may fulfill the interface but violate its constraints. This is currently the case for SoC design. IP providers' tools offer system assembly as in a block diagram by relying on standard interfaces. But the interface verification is left to the user, who does not know the constraints well.

Finally, operating systems have generic interfaces to control all devices of a specific type in the same manner. Devices of a supported type could embed driver implementations for different operating systems enabling their usage even if the OS did not have a driver for it. An appropriate interface for gathering the driver and a framework for automatically instantiating it enable a new way of connecting peripheral devices. In unified drivers, such as USB classes, the device has to comply with a specific driver implementation, here the driver complies with the device. Thus, the driver remains flexible and the device design free of adverse constraints. Because the device mechanism is platform independent, our driver architecture could provide a base for this by making driver policies that are as platform independent as possible reducing amount of driver code to be embedded.

Bibliography

[ABB+02] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wust, and Jorg Zettel. *Component-Based Product-Line Engineering with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[ALR01] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of dependability. Technical report, Computer Science Department, University of California, Los Angeles, USA, 2001.

[App07] Apple. I/O Kit Fundamentals. Technical report, Apple Inc., 2007.

[ARSH05] E. Armengaud, F. Rothensteiner, A. Steininger, and M. Horauer. A method for bit level test and diagnosis of communication services. In *Proc. of the 8th International IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems*, Sopron, Hungary, 2005. IEEE.

[ASE04] Miron Abramovici, Charles Stroud, and John M. Emmert. Online BIST and BIST-Based Diagnosis of FPGA Logic Blocks. *IEEE Trans. Very Large Scale Integr. Syst.*, 12(12):1284–1294, 2004.

[ASH+04] Eric Armengaud, Andreas Steininger, Martin Horauer, Roman Pallierer, and Hannes Friedl. A Monitoring Concept for an Automotive Distributed Network-The FlexRay Example. In *Proceedings of the 7th IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2004)*, pages 173–178, Slovakia, 2004. IEEE.

[BBKL10] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. SLAM2: static driver verification with under 4% false alarms. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 35–42, Lugano, Switzerland, 2010. IEEE.

[BBM09] Giovanni Beltrame, Cristiana Bolchini, and Antonio Miele. Multi-level fault modeling for transaction-level specifications. In *Proceed-*

ings of the 19th ACM Great Lakes symposium on VLSI - GLSVLSI '09, pages 87–92, New York, NY, USA, 2009. ACM.

[BCMS05] G. Bonfini, M. Chiavacci, R. Mariani, and R. Saletti. A new verification approach for mixed-signal systems. In *Proceedings of the IEEE International Behavioral Modeling and Simulation Conference (BMAS 2005)*, pages 22–23, San Jose, California, USA, 2005.

[BG06] Christian Bunse and Hans-Gerhard Gross. Unifying hardware and software components for embedded system development. In *Proceedings of the 2004 international conference on Architecting Systems with Trustworthy Components*, pages 120–136, Heidelberg, Germany, 2006. Springer-Verlag.

[BGF03] J. W. Bruce, M. A. Gray, and R. F. Follett. Personal digital assistant (PDA) based I2C bus analysis. *IEEE Transactions on Consumer Electronics*, 49(4):1482–1487, November 2003.

[BL98] Gaetano Borriello and Luciano Lavagno. Interface synthesis: a vertical slice from digital logic to software components. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 8–12, New York, NY, USA, 1998. ACM.

[BMP07] Brian Bailey, Grant Martin, and Andrew Piziali. *ESL Design and Verification a Prescription for Electronic System-Level Methodology.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[BPMC01] A. Bobbio, L. Portinale, M. Minichino, and E. Ciancamerla. Improving the analysis of dependable systems by mapping fault trees into Bayesian networks. *Reliability Engineering & System Safety*, 71(3):249–260, March 2001.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 37, pages 1–3, New York, NY, USA, 2002. ACM.

[BWZ10] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in Linux. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, Boston, MA, 2010. USENIX Association.

[CCM+09] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In

Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09, pages 45–58, New York, New York, USA, 2009. ACM Press.

[CE04] Christopher L. Conway and Stephen A. Edwards. NDL: a domain-specific language for device drivers. *ACM SIGPLAN Notices*, 39(7):30–36, 2004.

[CGMC03] Jacob A. Crossman, Hong Guo, Yi Lu Murphey, and John Cardillo. Automotive signal fault diagnostics. I. Signal fault analysis, signal segmentation, feature extraction and quasi-optimal feature selection. *IEEE Transactions on Vehicular Technology*, 52(4):1063–1075, July 2003.

[COB95] Pai H. Chou, Ross B. Ortega, and Gaetano Borriello. The Chinook hardware/software co-synthesis system. In *Proceedings of the 8th international symposium on System synthesis*, pages 22–27, New York, NY, USA, 1995. IEEE Comput. Soc. Press.

[Com94] Microcomputer Standards Committee. IEEE Standard 1284 - Standard signaling method for a bidirectional parallel peripheral interface for personal computers, 1994.

[Cor07] Intel Corporation. Implementing Industry Standard Architecture (ISA) with Intel Express Chipsets. 2007.

[CPC03] José Carvalho, Paulo Portugal, and Adriano Carvalho. A framework for dependability evaluation of PROFIBUS networks. In *Proceedings of the 2003 IEEE International Symposium on Industrial Electronics*, volume 1, pages 466–471, Rio de Janeiro, Brasil, 2003. IEEE.

[CWP08] Yung-Yuan Chen, Yi-Chiang Wang, and Jian-Min Peng. SoC-level fault injection methodology in SystemC design platform. In *Proceedings of the 2008 Asia Simulation Conference - 7th International Conference on System Simulation and Scientific Computing*, pages 680–687. IEEE, October 2008.

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. *ACM SIGOPS Operating Systems Review*, 35(5):73–88, December 2001.

[Dep69] Electronic Industries Association. Engineering Dept. Interface between data terminal equipment and data communication equipment employing serial binary data interchange, 1969.

[DLMSS08] Mehdi Dehbashi, Vahid Lari, Seyed Ghassem Miremadi, and Mohammad Shokrollah-Shirazi. Fault Effects in FlexRay-Based Networks with Hybrid Topology. In *Proceedings of the 2008 3rd International Conference on Availability, Reliability and Security*, pages 491–496, Washington, DC, USA, 2008. IEEE Computer Society.

[DRS04] Vijay D'silva, S. Ramesh, and Arcot Sowmya. Bridge over troubled wrappers:automated interface synthesis. In *Proceedings of the 17th International Conference on VLSI Design*, pages 189–194, Washington, DC, USA, 2004. IEEE Comput. Soc.

[Dub08] Elena Dubrova. *Fault tolerant design: An introduction*. Kluwer Academic Publishers, 2008.

[ECCH00] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, Berkeley, CA, USA, 2000. USENIX Association.

[ESS00] John Emmert, Charles Stroud, and Brandon Skaggs. Dynamic fault tolerance in FPGAs via partial reconfiguration. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 165–174, Washington, DC, USA, 2000. IEEE Computer Society.

[FHN+04] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*. Citeseer, 2004.

[GGP06] Archana Ganapathi, Viji Ganapathi, and David Patterson. Windows XP kernel crash analysis. In *Proceedings of the 20th conference on Large Installation System Administration*, pages 149–160, Berkeley, CA, USA, 2006. USENIX Association.

[GRB+08] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The design and implementation of microdrivers. *SIGARCH Comput. Archit. News*, 36(1):168–178, 2008.

[Gro02] H. G. Groß. Built-in contract testing in component-based application engineering. In *Proceedings of the Workshop on Component-based Development*, pages 87–100, Madrid, 2002.

[Gro04] Hans-Gerhard Gross. *Component-Based Software Testing with UML*. SpringerVerlag, 2004.

[Gro11] Object Management Group. Unified Modeling Language (UML), 2011.

[Gup92] Aarti Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2-3):151–238, October 1992.

[HBG+07] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure Resilience for Device Drivers. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 41–50, Washington, DC, USA, June 2007. IEEE Computer Society.

[HBG+09] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Fault isolation for device drivers. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 33–42. IEEE, June 2009.

[Her01] Richard Herveille. I2C Controller Core, 2001.

[HR04] Klaus Havelund and G. Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):158–173, 2004.

[IEE10] IEEE Standard for IP-XACT, Standard Structure for Packaging Integrating, and Reusing IP within Tool Flows. *IEEE Std 1685-2009*, 2010.

[IKK+07] V. P. Ivannikov, A. S. Kamkin, A. S. Kossatchev, V. V. Kuliamin, and A. K. Petrenko. The use of contract specifications for representing requirements and for functional testing of hardware models. *Programming and Computer Software*, 33(5):272–282, September 2007.

[Jac09] Daniel Jackson. A direct path to dependable software. *Communications of the ACM*, 52(4):78–88, 2009.

[Jan03] Axel Jantsch. NoCs: a new contract between hardware and software. In *Proceedings of the Euromicro Symposium on Digital System Design*, pages 10–16, Washington, DC, USA, 2003. IEEE Computer Society.

[JW05] Ahmed A. Jerraya and Wayne Wolf. Hardware/software interface codesign for embedded systems. *Computer*, 38(2):63–69, 2005.

[Kam07] Alexander Kamkin. Contract Specification of Pipelined Designs: Application to Testbench Automation. In *Proceedings of the 1st Spring Young Researchers Colloquium on Software Engineering (SYRCoSE'2007)*, 2007.

[Kam08] Alexander Kamkin. Coverage-Directed Verification of Microprocessor Units Based on Cycle-Accurate Contract Specifications. In *Proceedings of the East-West Design & Test Symposium (EWDTS)*, pages 84–87. IEEE, 2008.

[KC10] Volodymyr Kuznetsov and Vitaly Chipounov. Testing closed-source binary device drivers with DDT. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 12–25, Berkeley, CA, USA, 2010. USENIX Association.

[KDK06] Pavel Kubalik, Radek Dobiáš, and Hana Kubátová. Dependable Design for FPGA Based on Duplex System and Reconfiguration. In *Proceedings of the 9th EUROMICRO Conference on Digital System Design (DSD'06)*, pages 139–145, Washington, DC, USA, 2006. IEEE Computer Society.

[KG99] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, April 1999.

[KHSP01] Murali Kudlugi, Soha Hassoun, Charles Selvidge, and Duaine Pryor. A Transaction-Based Unified Simulation / Emulation Architecture for Functional Verification. In *Proceedings of the 38th annual Design Automation Conference*, pages 623–628, New York, NY, USA, 2001. ACM.

[LCFD+05] Ben Leslie, Peter Chubb, N Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting Shen, Kevin Elphinstone, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20:1–17, 2005.

[Lev09] Joshua Thomas Levasseur. *Device driver reuse via virtual machines*. Phd thesis, University of New South Wales, 2009.

[LUSG04] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation-Volume 6*, pages 2–17, San Francisco, CA, 2004. USENIX Association.

[Lyu07] Michael R. Lyu. Software reliability engineering: A roadmap. In *Proceedings of the 2007 Future of Software Engineering*, pages 153–170, Washington, DC, USA, 2007. IEEE Computer Society.

[MÖ96] Karl-Heinz Möller. Ausgangsdaten für Qualitätsmetriken - Eine Fundgrube für Analysen. In *Softwaremetriken in der Praxis*. Springer, 1996.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[Mic06] Microsoft. Architecture of the Kernel-Mode Driver Framework. Technical report, Microsoft Corporation, 2006.

[MRC+00] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for hardware programming. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, pages 2–15, Berkeley, CA, USA, 2000. USENIX Association.

[MRC04] S. Mir, L. Rufer, and B. Courtois. On-chip testing of embedded transducers. In *Proceedings of the 17th International Conference on VLSI Design*, pages 463–472, Washington, DC, USA, 2004. IEEE Computer Society.

[MT95] Manish Malhotra and Kishor S. Trivedi. Dependability modeling using Petri-nets. *IEEE Transactions on Reliability*, 44(3):428–440, 1995.

[Mur04] Brendan Murphy. Automating software failure reporting. *Queue*, 2(8):42–48, 2004.

[OOJ98] Mattias O'Nils, Johnny Öberg, and Axel Jantsch. Grammar based modelling and synthesis of device drivers and bus interfaces. In *Proceedings of the 24th Conference on EUROMICRO - Volume 1*, pages 55–58, Washington, DC, USA, 1998. IEEE Computer Society.

[Ope] Open SystemC Initiative. http://www.systemc.org.

[Pd05] Paulo José Portugal and Adriano da Silva Carvalho. A Simulation Model based on a Stochastic Petri Net Approach for Dependability Evaluation of PROFIBUS-DP Networks. In *Proceedings of the 2005 IEEE Conference on Emerging Technologies and Factory Automation*, pages 485–494. IEEE, 2005.

[Phi00] Philips Semiconductor. I2C Bus Specification, 2000.

[PHJ06] Katarina Paulsson, M Hubner, and Markus Jung. Methods for run-time failure recognition and recovery in dynamic and partial reconfigurable systems based on Xilinx Virtex-II Pro FPGAs. In *Proceedings of the Emerging VLSI Technologies and Architectures*, pages 159–164, Washington, DC, USA, 2006. IEEE Computer Society.

[PHZ+05] Roman Pallierer, Martin Horauer, Martin Zauner, Andreas Steininger, Eric Armengaud, and Florian Rothensteiner. A generic tool for systematic tests in embedded automotive communication systems. In *Proc. of the Embedded World Conference*, pages 42–49, Nürnberg, Germany, 2005.

[RCK+09] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with termite. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 73–86, New York, New York, USA, 2009. ACM Press.

[RCKH09] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 275–288, New York, NY, USA, 2009. ACM.

[RKM+10] Leonid Ryzhyk, John Keys, Balachandra Mirla, Arun Raghunath, Mona Vij, and Gernot Heiser. Improved device driver reliability through verification reuse. In *Proceedings of the Sixth international conference on Hot topics in system dependability*, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association.

[Rub06] Martin Rubli. *Building a Webcam Infrastructure for GNU / Linux*. Master thesis, EPFL, Switzerland, 2006.

[SABL06] Michal M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4):333–360, November 2006.

[SBF03] Bernd Steinbach, Thomas Beierlein, and Dominik Fröhlich. UML-based co-design for run-time reconfigurable architectures. In *Languages for system specification*, pages 5–19. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[SHM09] Raul Schmidlin Fajardo Silva, Jürgen Hesser, and Reinhard Männer. Fault Propagation Analysis on the Transaction-Level Model of an Acquisition System with Bus Fallback Modes. In *Proceedings of the Workshop on the Design of Dependable Critical Systems (DDCS)*, page 36. University Heidelberg, 2009.

[SHM11] Raul Schmidlin Fajardo Silva, Jürgen Hesser, and Reinhard Männer. Contract Specification for Hardware Interoperability Testing and Fault Analysis. *Reliability, IEEE Transactions on*, 60(1):351–362, 2011.

[SLA97] Charles Stroud, Eric Lee, and Miron Abramovici. BIST-Based Diagnostics of FPGA Logic Blocks. In *Proceedings of the 1997 IEEE International Test Conference*, pages 539–547, Washington, DC, USA, 1997. IEEE Computer Society.

[SM12] Raul Schmidlin Fajardo Silva and Guillermo Marcus. Device Mechanism: A Structured Device Driver Development Approach. In *Proceedings of the 14th IEEE International High Assurance Systems Engineering Symposium (HASE)*, pages 66–73. IEEE Computer Society, 2012.

[SMLE02] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: an architecture for reliable device drivers. In *Proceedings of the 10th ACM SIGOPS European Workshop*, pages 102–107, New York, NY, USA, 2002. ACM.

[SPB+06] Dima Suliman, Barbara Paech, Lars Borner, Colin Atkinson, Daniel Brenner, Matthias Merdes, and Rainer Malaka. The MORABIT Approach to Runtime Component Testing. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, pages 171–176, Washington, DC, USA, 2006. IEEE Computer Society.

[SS04] Małgorzata Steinder and Adarshpal S. Sethi. Probabilistic Fault Localization in Communication Systems Using Belief Networks. *IEEE/ACM Transactions on Networking*, 12(5):809–822, October 2004.

[STZ08] Janusz Sosnowski, Dawid Trawczyński, and Janusz Zalewski. Safety Issues in Modern Bus Standards. *Computer*, 41(1):97–99, January 2008.

[SYKI05] Jun Sun, Wanghong Yuan, Mahesh Kallahalla, and Nayeem Islam. HAIL: a language for easy and correct device access. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 1–9, New York, NY, USA, 2005. ACM.

[TW09] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating systems: design and implementation*. Pearson, Upper Saddle River, 3rd edition, 2009.

[WB94] Elizabeth A. Walkup and Gaetano Borriello. Automatic Synthesis of Device Drivers for Hardware/Software Codesign. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA, 1994.

[Wer09] Frank Werner. *Applied Formal Methods in Wireless Sensor Networks.* PhD thesis, Universität Fridericiana zu Karlsruhe (TH), 2009.

[WMB03] Shaojie Wang, Sharad Malik, and Reinaldo A. Bergamaschi. Modeling and integration of peripheral devices in embedded systems. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, pages 136–141, Washington, DC, USA, 2003. IEEE Computer Society.

[Wol03] Wayne Wolf. A Decade of Hardware/Software Codesign. *Computer*, 36(4):38–43, 2003.

[ZCA+06] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 45–60, Berkeley, CA, USA, 2006. USENIX Association.