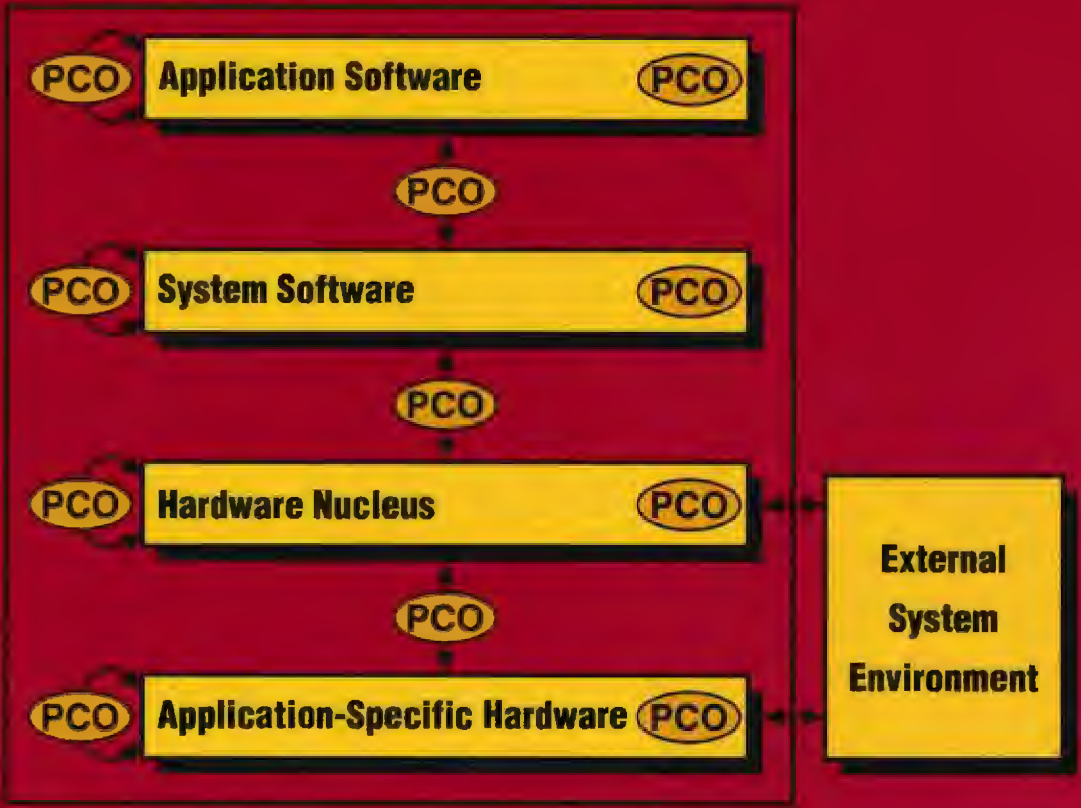
Design for test & debug in hardware/software systems

DOI:
10.6100/IR510839

Document status and date:
Published: 01/01/1998

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Design For Test & Debug in Hardware/Software Systems



H.P.E. Vranken

Design For Test & Debug in Hardware/Software Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. M. Rem, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 2 juni 1998 om 16.00 uur

door

Hendrikus Petrus Elisabeth Vranken

geboren te Maastricht

Dit proefschrift is goedgekeurd door de promotoren:

prof.ir. M.T.M. Segers
en
prof.ir. M.P.J. Stevens

Copromotor:
dr.ir. J.P.M. Voeten

Druk: Universiteitsdrukkerij Technische Universiteit Eindhoven

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Vranken, Hendrikus P.E.

Design for test and debug in hardware/software systems / by Hendrikus P.E.
Vranken. - Eindhoven : Technische Universiteit Eindhoven, 1998.
Proefschrift. - ISBN 90-386-0440-8
NUGI 832
Trefw.: foutendetectie ; computers / debugging / geintegreerde
schakelingen ; testen / automatische testmethoden.
Subject headings: design for testability / computer debugging /
computer testing / computer architecture.

Samenvatting

Dit proefschrift beschrijft onderzoek naar de ontwikkeling van een generieke methode voor 'design for test & debug' in hardware/software systemen. Deze methode verschaft oplossingen voor de complexe problemen gerelateerd aan testen en debuggen op systeem niveau.

Het doel van testen op systeem niveau is te verifiëren of het gedrag van de hardware/software implementatie van een systeem overeenkomt met het gespecificeerde systeemgedrag. Debuggen is nodig om de exacte oorzaak te achterhalen van de fouten die door testen aan het licht zijn gebracht.

Een groot probleem bij het testen en debuggen van hardware/software systemen is de beperkte zichtbaarheid van het interne systeemgedrag. Sommige aspecten van het interne systeemgedrag zijn bijzonder moeilijk te observeren en te controleren in de externe omgeving van het systeem, zoals de volgorde van events in het systeem, de wisselende executie van processen, tijd afhankelijkheden en non-deterministisch gedrag. In onze 'design for test & debug' methode worden deze problemen opgelost door het verbeteren van de controleerbaarheid en observeerbaarheid van het interne systeemgedrag.

Wij geven diverse classificaties voor fouten in hardware/software systemen. We concentreren ons op foutieve communicatie en synchronisatie protocollen, foutieve wederzijdse exclusieve toegang tot gezamenlijke gegevens of gezamenlijke middelen, foutieve executie volgordes van processen, deadlocks, race condities en foutieve interrupt afhandeling. Deze fouten doen zich typisch voor als tijdelijke fouten gedurende de executie van een systeem en ze kunnen vaak niet gereproduceerd worden tijdens debuggen.

In dit proefschrift definiëren we een ontwerp proces bestaande uit zes stappen: opstellen van systeem eisen, systeem specificatie, architectuur onderzoek, architectuur verfijning, synthese en hardware/software integratie. Onze 'design for test & debug' methode is volledig geïntegreerd in dit ontwerp proces.

Wij definiëren een generiek architectuur-model voor hardware/software systemen, bestaande uit applicatie software, systeem software, hardware nucleus, applicatie-specifieke hardware en communicatie interfaces. Onze 'design for test & debug' methode is gericht op het verbeteren van de zichtbaarheid van de communicatie interfaces in het architectuur-model. Tevens verschaft onze 'design for test & debug' methode zichtbaarheid in de toestand van software processen en hardware componenten.

Het belangrijkste element van onze methode is het invoegen van Punten van Controle en Observatie (PCOs) in de systeem specificatie. PCOs verschaffen controle en observatie van communicatie interfaces en de toestand informatie van processen. We nemen de PCOs vervolgens op in de systeem architectuur en tenslotte implementeren we de PCOs in hardware en/of software. Onze methode voorziet erin dat de essentiële informatie over het interne systeemgedrag kan worden gecontroleerd en geobserveerd. Onze methode impliceert tevens dat de effecten van PCOs op de systeem architectuur van tevoren kunnen worden voorspeld en dat gepaste maatregelen kunnen

worden genomen om onduldbare neveneffecten te vermijden.
We beantwoorden twee belangrijke vragen aangaande het invoegen van PCOs in de systeem specificatie: waar moeten PCOs worden ingevoegd in de systeem specificatie en wat zijn de effecten van het invoegen van PCOs op het systeemgedrag. We introduceren testbaarheids-analyse gebaseerd op scenario's om het invoegen van PCOs in goede banen te leiden. Deze analyse methode identificeert allereerst de essentiële informatie in een systeem en vervolgens hoe goed deze essentiële informatie kan worden gecontroleerd en/of geobserveerd in de systeem omgeving. We geven een mathematische analyse van de interferentie van PCOs in het systeemgedrag, gebruik makende van proces algebra. We definiëren een aantal - mathematisch bewezen - transformatie functies voor het invoegen van PCOs. Deze transformatie functies garanderen dat het extern observeerbare systeemgedrag behouden blijft gedurende het invoegen van PCOs.
We lichten de implementatie van PCOs toe in hardware en software. We laten zien dat PCOs efficiënt kunnen worden geïmplementeerd in hardware door hergebruik van DFT en DFD faciliteiten op IC niveau zoals 'scan paden'. De test bussen, test interfaces en test controllers op PCB niveau en systeem niveau kunnen een infrastructuur verschaffen om toegang tot PCOs te verkrijgen. Verdere verbetering van de controleerbaarheid en observeerbaarheid van het interne systeemgedrag kan worden verkregen door gebruik te maken van speciale software monitoren, hardware monitoren of hybride monitoren.
We passen onze 'design for test & debug' methode toe op de specificatie en de implementatie van een liftbesturingssysteem. Diverse experimenten demonstreren het gebruik en het nut van PCOs voor testen en debuggen op systeem niveau.

Summary

This thesis describes our research efforts to develop a generic design for test & debug method for hardware/software systems. Our method provides solutions for the complex problems related to system-level testing and debugging.

The goal of system-level testing is to verify whether the behavior of a system's hardware/software implementation conforms to the specified system behavior. Whenever testing reveals the presence of an error, debugging is required to determine the exact fault mechanism in the system that caused the error.

A major problem when testing and debugging hardware/software systems, is the limited visibility into the internal operation of a system. In particular, aspects of system behavior such as the ordering of events in the system, the interleaved execution of processes, timing dependencies and non-determinism are difficult to observe and control in the system environment. In our design for test & debug method, we deal with these problems by improving the controllability and observability of the internal system behavior.

We give various classifications for faults in hardware/software systems. We concentrate on faulty communication and synchronization protocols, faulty mutual exclusive access to shared data or shared resources, faulty process scheduling, deadlocks, race conditions and faulty interrupt handling. These faults typically appear as temporary faults at run-time and they often cannot be reproduced during debugging.

In this thesis, we define a co-design flow consisting of six steps: system requirements capture, system specification, architecture exploration, architecture refinement, synthesis and hardware/software integration. Our design for test & debug method is fully integrated into this co-design flow.

We also define a generic architectural model for hardware/software systems, consisting of application software, system software, hardware nucleus, application-specific hardware and communication interfaces. Our design for test & debug method aims at providing visibility into the communication interfaces in our architectural model. In addition, our design for test & debug method provides visibility into the state information of software processes and hardware components.

The key element of our method is the insertion of Points of Control and Observation (PCOs) in the system specification. PCOs provide control and observation of communication interfaces and process state information. Subsequently, we incorporate the PCOs in the system architecture and finally we implement them in hardware and/or software. Our method provides that the essential information on the internal system behavior can be controlled and observed. Our method also implies that the effects of PCOs on the system architecture can be predicted in advance and appropriate measures can be taken to avoid intolerable side effects, such as performance degradation and the probe effect.

We discuss two key questions related to the insertion of PCOs in the system specification: where should PCOs be inserted in the system specification and what are the effects of PCO insertion on the system behavior. We propose scenario-based testability analysis to guide PCO insertion. This analysis method identifies the essential information in a system and analyzes how well this essential information can be controlled and/or observed in the system environment. We provide a formal analysis on the interference of PCOs in the system behavior using process algebra. We define a set of mathematically proven transformation functions for PCO insertion. These transformation functions guarantee that the externally observable system behavior is preserved during PCO insertion.

We discuss the implementation of PCOs in hardware and/or software. We show that PCOs can be implemented efficiently in hardware by reusing IC-level DFT and DFD facilities such as scan paths. The PCB-level and system-level test buses, test interfaces and test controllers provide an infrastructure that can be used to access PCOs. Improved control and observation into the internal system operation can be obtained by using dedicated software monitors, hardware monitors or hybrid monitors.

We apply our design for test & debug method on the specification and implementation of an elevator control system. Several experiments demonstrate the use and the benefits of PCOs for system-level testing and debugging.

Acknowledgements

During the past five years I had the privilege of working as a research assistant in the Information and Communication Systems group at the Eindhoven University of Technology. I look back on a very pleasant time and I would like to thank all people who contributed to this.

In the first place I would like to thank my promotors prof. René Segers and prof. Mario Stevens for their supervision and for giving me the opportunity to perform my Ph.D. research. They offered me the freedom to choose my own research directions, while their valuable suggestions guided me to take the right paths. Likewise, I would like to thank my copromotor Jeroen Voeten for the many fruitful discussions and in particular for his contribution to the formal theory and the formal proofs in this thesis.

Furthermore, I would like to thank all members and ex-members of the Information and Communication Systems group for providing a stimulating and pleasant working environment. In particular I would like to thank Lennart Benschop, Maarten Boersma, Max Bonsel, Ad Chamboné, Artur Chojnacki, Koen van Eijk, Rian van Gaalen, Marc Geilen, Fons Geurts, Theo Gransier, Ton van Hekezen, prof. Jochen Jess, Lech Jóźwiak, Marcel Kolsteren, Piet van der Putten, Ilja van Rhee, prof. Hans de Stigter, Ad Verschueren, Gert-Jan de Vos, Bart Vostermans and Rinus van Weert.

Many people in industrial research laboratories have contributed to this thesis by discussing and commenting on my work. I would like to thank Marc Witteman and Ronald van Wuijtswinkel at KPN Research in Leidschendam. Their work on protocol conformance testing provided a major source of inspiration in the first years of my research. In the period from November 1994 to January 1995 I had the opportunity to stay with the Advanced Development group of Philips Media Systems in Eindhoven, for which I am particularly grateful to Cor Luijks. Studying the compatibility problems in CD-i systems considerably improved my understanding of faults in hardware/software systems. I would like to thank Frank Bouwman, Erik Jan Marinissen, Gert Jan van Rootselaar and Math Verstraelen at Philips Research for their valuable comments on my work during the last few years. Furthermore, I would like to thank Philips Electronic Design & Tools/Test for their financial support.

Last but not least, I would like to thank my family and my friends for their support and for providing a marvelous social environment. Most of all, I would like to thank my parents, my sister and Jeannine for their continuous support, their understanding and their patience. I would like to dedicate this thesis to them.

Contents

List of Figures

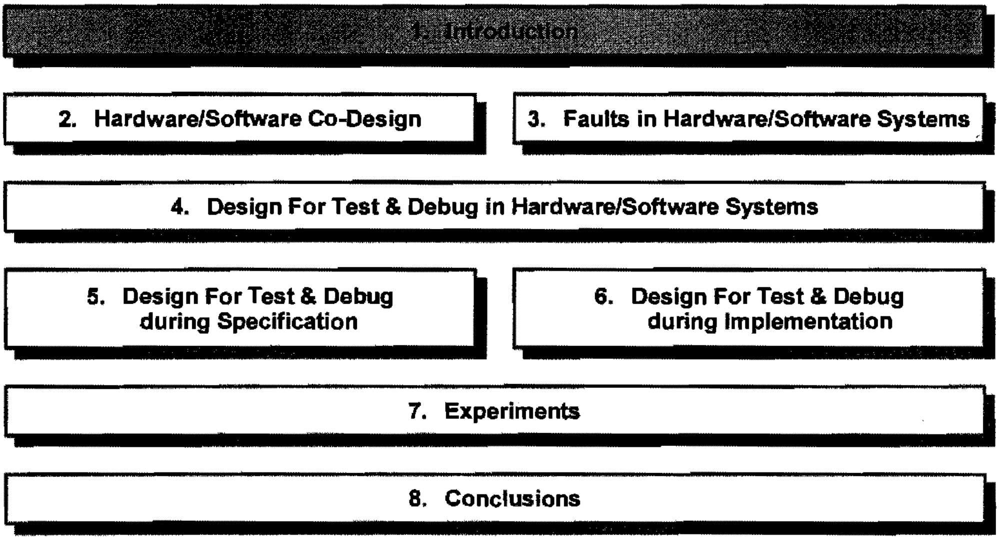
List of Acronyms

ACM	Association for Computing Machinery
A/D	Analog/Digital
ALU	Arithmetic and Logic Unit
ANSI	American National Standards Institute
API	Application Program Interface
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction Processor
ASP	Abstract Service Primitive
ATM	Asynchronous Transfer Mode
ATPG	Automatic Test Pattern Generator
BIST	Built-In Self-Test
CCS	Calculus of Communicating Systems
CFG	Control-Flow Graph
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSP	Communicating Sequential Processes
D/A	Digital/Analog
DFD	Design-For-Debug
DFG	Data-Flow Graph
DFT	Design-For-Test
DMA	Direct Memory Access
DSP	Digital Signal Processor
ECS	Elevator Control System
EDA	Electronic Design Automation
EEPROM	Electrically Erasable Programmable Read Only Memory
EFSM	Extended Finite-State Machine
FCFS	First-Come First-Served
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
FSM	Finite-State Machine
GUI	Graphical User Interface
HW	Hardware
IC	Integrated Circuit
ICE	In-Circuit Emulator
ICT	In-Circuit Testing
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers

IFIP	International Federation for Information Processing
I/O	Input/Output
ISA	Instruction-Set Architecture
ISDN	Integrated Services Digital Network
ISO	International Organization for Standardization
ITG	Information Transfer Graph
ITU	International Telecommunication Union
IUT	Implementation Under Test
LFSR	Linear Feedback Shift Register
LIFO	Last-In First-Out
LT	Lower Tester
MCM	Multi-Chip Module
MHz	Mega Hertz
MPEG	Moving Pictures Experts Group
MTBF	Mean Time Between Failures
MTM	Module Test and Maintenance
MTTR	Mean Time To Repair
MVS	Multiple Virtual Storage
OOAD	Object-Oriented Analysis and Design
OMT	Object Modeling Technique
OOSE	Object-Oriented Software Engineering
OSI	Open Systems Interconnection
PC	Point of Control
PCB	Printed Circuit Board
PCO	Point of Control and Observation
PDU	Protocol Data Unit
PO	Point of Observation
POOSL	Parallel Object-Oriented Specification Language
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RPC	Remote Procedure Call
RTL	Register-Transfer Level
SAD	Structured Analysis and Design
SAP	Service Access Point
SDL	Specification and Description Language
SHE	Software/Hardware Engineering
SMT	Surface-Mount Technology
SUT	System Under Test
SW	Software
TAP	Test Access Port
TCP	Test Coordination Procedure
TIE	Test Interface Element
TMF	Test Management Function
TTCN	Tree and Tabular Combined Notation
UT	Upper Tester
VHDL	Very High Speed Integrated Circuit (VHSIC) Hardware Description Language
VLSI	Very Large Scale Integration

Chapter 1

Introduction


1. Introduction
2. Hardware/Software Co-Design
3. Faults in Hardware/Software Systems
4. Design For Test & Debug in Hardware/Software Systems
5. Design For Test & Debug during Specification
6. Design For Test & Debug during Implementation
7. Experiments
8. Conclusions

This chapter provides an introduction to this thesis. We define the problems that are subject of this thesis, we outline our objectives and we give an overview of the contents of this thesis.

1.1 Introduction

This thesis describes our research efforts for developing a method towards **design-for-test** and **design-for-debug** in hardware/software systems. The terms design-for-test and design-for-debug indicate that we consider testing and debugging of hardware/software systems already in the system design process. We will use the term **design for test & debug** to comprise both design-for-test and design-for-debug. This thesis deals with hardware/software systems, which are systems that contain both hardware components and embedded software. Such systems are typically found in application domains such as tele and data communication systems, consumer electronic products, industrial control systems, automotive systems and aerospace systems. In general these systems contain one or more processors to execute software and dedicated hardware components like ASICs (Application-Specific Integrated Circuits) and FPGAs (Field Programmable Gate Arrays).

The objective of this thesis is to provide solutions for problems that are related to testing and debugging of hardware/software systems. Since the introduction of IC technology and software programmable devices, we are facing an exponential increase in the complexity of hardware/software systems. Gordon Moore, co-founder of Intel, predicted in 1965 that the density of transistors on ICs such as memory chips and microprocessors would double roughly every 18 months. This prediction has been proven to be valid until today and it is likely to remain valid in the near future. Also the amount of embedded software in systems is increasing rapidly. For instance, an exponential increase in the amount of embedded software has been reported for Philips' consumer products such as TV sets, VCRs and stereo equipment [RAvG96]. Handling this increasing complexity in hardware/software systems demands improvement of all technologies, tools and methods required for the design, implementation and verification of these systems. Unfortunately, the difficulty of design and verification increases even faster than the exponential growth in the number of transistors on ICs or the amount of embedded software in systems. Consequently, there is a growing complexity gap between hardware and software technology and the designer's ability to design and verify complex hardware/software systems. The key issue nowadays is to design and verify systems while meeting time-to-market constraints [SIA94].

The task of **verification** is to check the correctness of a system. Various techniques for verification are used in the successive steps of the design process, such as simulation or formal verification of system models and testing of the system implementation. The focus of this thesis is on system-level testing and debugging. We define **system-level testing** as verifying the correctness of a system by applying test stimuli to the hardware/software implementation of the system and observing the responses. System-level testing implies that we verify the correctness of a system as a whole, constituted of all its hardware and software components. Hence, we do not address traditional hardware testing, which mainly aims at detecting physical defects in hardware components, nor do we address traditional software testing, which mainly aims at detecting coding errors in software components. Instead, we focus on verifying whether the behavior of the entire hardware/software implementation conforms to the specified system behavior. System-level testing should yield a verdict on the correctness of a system. Whenever testing reveals the presence of an error, **debugging** is required to determine the exact fault mechanism in the system that caused the error.

At present, system-level testing and debugging are major bottlenecks when developing complex hardware/software systems. In [MCC96] is stated that 30–40% of the total development costs and time for hardware/software systems are spent on hardware/software testing and debugging. Hence, an improved method for system-level testing and debugging will directly result in reduced development costs and a shorter time-to-market.

The increasing difficulty of system-level testing and debugging is closely related to the increasing complexity of hardware/software systems. First of all, exhaustive testing of a hardware/software system is generally impossible to achieve because the required number of test cases is astronomical. Second, testing and debugging a system through its external interfaces does usually not provide sufficient visibility into the internal operation of a system. Consequently, it may be very difficult to observe and control some specific parts of the system, such as the interaction between hardware and software components. Debugging a complex system often corresponds to finding a microscopic needle in a gigantic haystack.

In this thesis we deal with the complexity of hardware/software systems by taking a system-level view and by adopting a multi-disciplinary approach. Taking a system-level view implies that we reason about a system at a high level of abstraction, concentrating on issues like concurrency, communication and distribution. Adopting a multi-disciplinary approach implies that we consider and combine various disciplines related to specification, architecture design and implementation of both hardware and software. In our opinion, a system-level view and a multi-disciplinary approach are prerequisites for developing a method towards design for test & debug in hardware/software systems. However, taking a system-level view and a multi-disciplinary approach requires a profound knowledge of many disciplines, which is definitely not easy to obtain.

1.2 Objectives

The objective of this thesis is to develop a generic method towards design for test & debug that deals with the problems of system-level testing and debugging in hardware/software systems. We primarily concentrate on design for test & debug by improving visibility into the internal system operation. We pay less attention to problems related to test case generation.

The key element of our method is to improve the design process for hardware/software systems in such a way that testing and debugging of the system implementation is facilitated. Hence, instead of being confronted with the hardware/software implementation of a system and trying to generate ad hoc solutions to testing and debugging, we provide a structured solution for testing and debugging in advance by modifying the system design. The basic principle of our method is improving accessibility to the internal operation of a hardware/software system. This should provide that an external tester/debugger in the system environment can observe and control the internal operation of the system for testing and debugging purposes.

In our view, any research initiative on design for test & debug in hardware/software systems should be based on a thorough understanding of the design process and on a thorough understanding of the faults that are typically encountered during system-level testing and debugging. We meet both requirements in this thesis by first analyzing the current design methods for hardware/software systems and pointing out their shortcomings with respect to testing, debugging and design for test & debug. Next, we characterize faults in hardware/software systems while concentrating on basic concepts such as hardware/software architecture, communication interfaces and parallelism.

Any method to design for test & debug should make a clear distinction between activities related to system specification and activities related to system implementation. During system specification, the focus is on defining the functional behavior of a system which is generally implementation-independent. A method to design for test & debug should answer questions related to why and how to perform design for test & debug during system specification. Furthermore, a method should consider the impact of design for test & debug features on a system. During system implementation, the focus is on realizing a system with hardware and software components. A method to design for test & debug should now deal with the implementation of test & debug facilities in hardware and software.

Finally, any theory should be proven by experimental results obtained from applying the theory into practice. We meet this requirement in this thesis by describing a case study of an elevator control system.

In summary, this thesis describes a generic method towards design for test & debug in hardware/software systems which should fulfill the following objectives:

- The method should provide a solution to handle the complexity of system-level testing and debugging in hardware/software systems.
- The method should be fully integrated into the hardware/software co-design process.
- We should describe precisely the faults that are typically encountered during system-level testing and debugging. The method should support testing and debugging for these faults.
- The method should provide means to design for test & debug during system specification, answering questions on why and how to perform design for test & debug.
- The method should provide means to design for test & debug during implementation, addressing test & debug facilities in hardware and software. Furthermore, the method should consider the transition from specification to implementation.
- The method should take into account and quantify the effects of design for test & debug on a system. In particular, the insertion of test & debug facilities in the system specification and the system implementation should not lead to incorrect system behavior.
- We should demonstrate the method in practice and provide experimental evidence that the method indeed improves system-level testing and debugging of hardware/software systems.

This thesis contributes to advancing the state-of-the-art on design for test & debug in several ways. First of all, our method comprises some new techniques, where each technique covers some specific part in the area of system-level testing and debugging. The added value of this thesis however is in the combination of concepts and theories from many different fields, including hardware/software co-design, formal specification, formal verification, and traditional approaches to design-for-test and design-for-debug in hardware and software. We integrate these concepts and theories into a coherent method towards system-level design for test & debug.

1.3 Thesis Organization

This thesis is divided into eight chapters and one appendix. The organization of the thesis is outlined in figure 1.1.

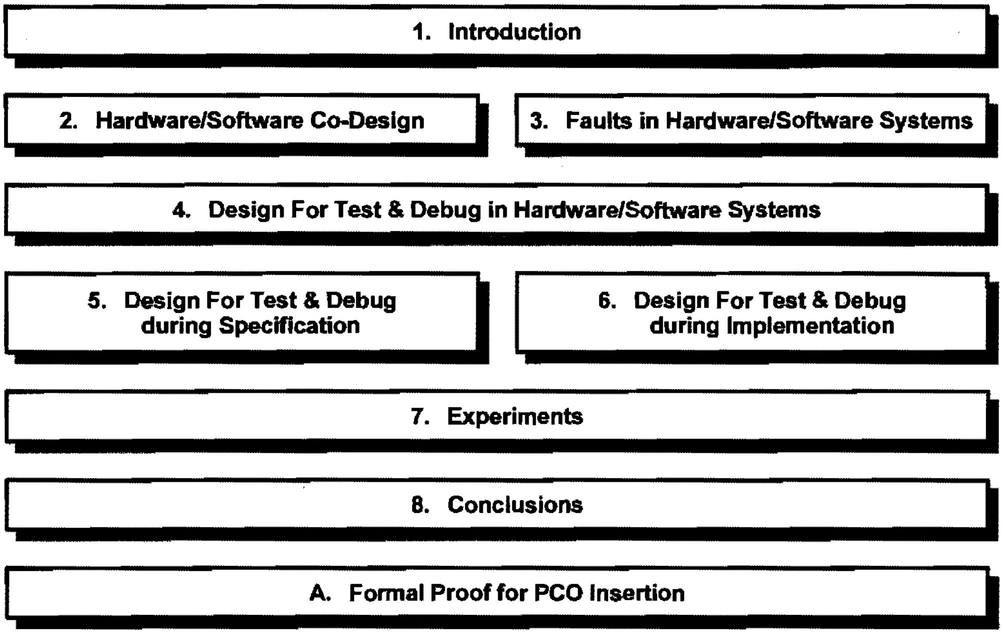

1. Introduction
2. Hardware/Software Co-Design
3. Faults in Hardware/Software Systems
4. Design For Test & Debug in Hardware/Software Systems
5. Design For Test & Debug during Specification
6. Design For Test & Debug during Implementation
7. Experiments
8. Conclusions
A. Formal Proof for PCO Insertion

Figure 1.1 *Outline of thesis*

In **chapter 2** we describe the state-of-the-art on hardware/software co-design. We define the successive steps in the co-design flow while paying special attention to validation and verification activities such as co-simulation and testing. The first objective of this chapter is to show the improvements brought by hardware/software co-design methods over traditional design methods. The second objective is to uncover the shortcomings of co-design methods with respect to design-for-test and design-for-debug.

In **chapter 3** we elaborate on faults in hardware/software systems. The objective of this chapter is to examine and classify the faults in hardware/software systems that are typically encountered during hardware/software integration testing and system testing. This chapter also concentrates on the architecture of hardware/software systems and communication interfaces.

In **chapter 4** we present our approach to design for test & debug in hardware/software systems. Our approach is founded on two fundamental ideas. First, we are convinced that design for test & debug should be integrated into hardware/software co-design, as described in chapter 2. Second, we feel that design for test & debug should aim at detecting interfacing faults and system-level faults, as described in chapter 3. In chapter 4 we discuss the basic principles of our design for test & debug approach. Furthermore, we introduce the concept of Point of Control and Observation (PCO), which forms the key element of our approach. We detail our approach in chapter 5 and chapter 6, where we concentrate on design for test & debug in the specification and the implementation of hardware/software systems.

In **chapter 5** we elaborate on design for test & debug during system specification. The focus is on answering two key questions: where should PCOs be inserted in a system specification, and what are the effects of PCO insertion on the system behavior.

In **chapter 6** we discuss design for test & debug during system implementation. We outline the current design-for-test and design-for-debug techniques for both hardware and software. Next, we discuss how these techniques can be used to implement PCOs and to implement the infrastructure for accessing PCOs from the external environment.

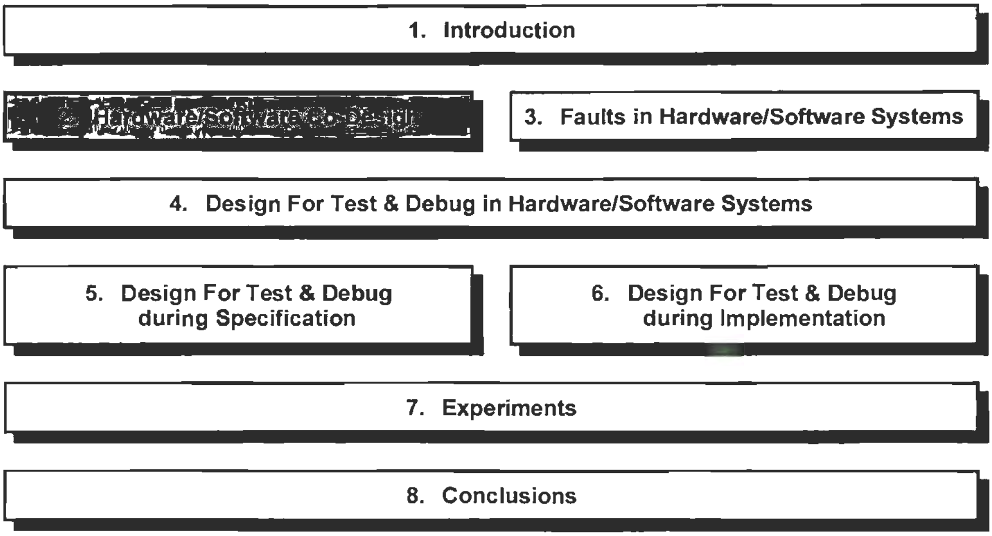
In **chapter 7** we present experiments on an elevator control system. We elaborate on the requirements, on the formal specification, on the implementation, and on applying our design for test & debug approach in the elevator control system. We discuss our experiences and we illustrate the strength of our design for test & debug approach.

In **chapter 8** we summarize the conclusions of this thesis and we recommend directions for future research.

Finally, in **appendix A** we provide a formal proof for inserting PCOs while preserving the externally observable system behavior. The formal theory in this appendix is discussed informally in chapter 5 and it is applied to the case study in chapter 7.

Chapter 2

Hardware/Software Co-Design


1. Introduction
2. Hardware/Software Co-Design
3. Faults in Hardware/Software Systems
4. Design For Test & Debug in Hardware/Software Systems
5. Design For Test & Debug during Specification
6. Design For Test & Debug during Implementation
7. Experiments
8. Conclusions

This chapter describes the state-of-the-art on hardware/software co-design. The various co-design methods for embedded, heterogeneous hardware/software systems are classified and the successive steps in the co-design flow are elaborated. Special attention is paid to verification and validation activities like co-simulation and testing. The goal of this chapter is to examine the improvements brought by hardware/software co-design methods over traditional design methods, focusing on hardware/software verification issues. In addition, this chapter highlights the shortcomings of co-design methods with respect to testing and debugging.

2.1 Introduction

Hardware/software systems are heterogeneous systems that contain both hardware and software components. The increasing complexity of designing hardware/software systems requires new design methods in which synergy between the hardware and software design flows is established. These new hardware/software co-design methods, in which hardware and software design proceed concurrently, should reduce design time and costs.

Hardware/software co-design mainly concentrates on the design of embedded systems. An embedded system is used to control a larger heterogeneous system. The functionality of an embedded system is usually fixed and is determined primarily by the interactions of the system with its environment. An embedded system often has various modes of operation. Typically, an embedded system is a real-time reactive system, which implies that the system has to react on events in the environment considering timing constraints. To achieve this, embedded systems usually posses a great deal of concurrency, constituted by the parallel operation of hardware and software components. This parallelism is the primary cause of the vast complexity of hardware/software systems.

Hardware/software co-design is not a new idea. Designers have been developing heterogeneous hardware/software systems ever since the introduction of software programmable integrated circuits. In the past, traditional hardware/software design methods relied heavily upon experienced system designers. These experienced designers were able to handle complexity by abstracting from implementation details and obtaining a system-level view. Implementation choices and hardware/software trade-offs were made very early in the design process, based upon the designer's experience and knowledge. After this a priori, premature, hardware/software partitioning, hardware design and software design were performed as two rather independent activities. The interaction between hardware and software components was not verified until the actual prototype hardware was available. The integration of hardware and software components and the corresponding testing and debugging activities often formed the major bottleneck in the design process. Correcting errors required long iteration cycles in the design process, as shown in figure 2.1.

The problems of hardware/software integration are well-documented in literature. In [MCC96] is stated that 30–40% of the total development costs and time for embedded systems are spent on hardware/software integration, testing and debugging. In [Bou90, Sch93a] is stated that although 90% of the ASIC (Application-Specific Integrated Circuit) prototypes work fine when tested in isolation, 50% fail when integrated in the system. These failures are due to errors in the interactions between the ASIC and other hardware and software components. In [TY95] is reported that 50–70% of the costs for developing distributed, real-time systems are spent on testing and debugging, mainly due to timing errors. The obvious conclusion is that hardware/software integration and the subsequent re-design loops are the major bottlenecks in the traditional design methods. However, these traditional design methods were common practice and they generally sufficed until the late 1980's.

In the late 1980's, the progress in VLSI technology enabled to build a system from catalogue ICs, custom hardware components and software components. Custom hardware components such as ASICs and FPGAs (Field Programmable Gate Arrays) allow to implement algorithms directly into hardware with high performance. Software components can be executed on general-purpose

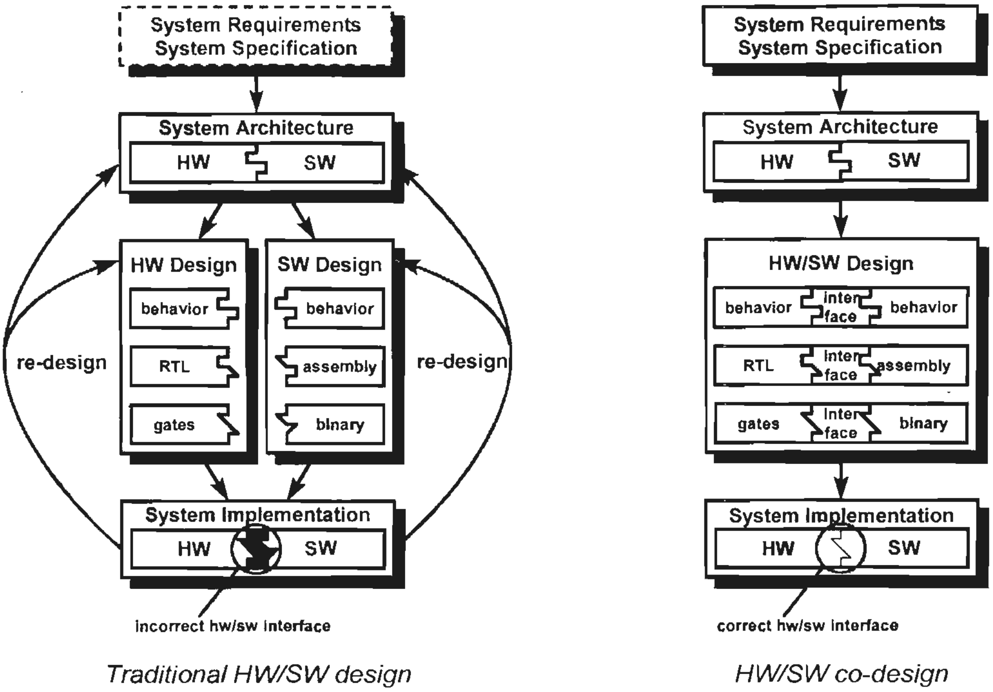


Figure 2.1 Traditional design versus co-design

microprocessors, microcontrollers, domain-specific processors like DSPs (Digital Signal Processors), or ASIPs (Application-Specific Instruction Processors). As a result, the design space of potential hardware/software implementations has increased significantly. It is becoming more and more difficult for designers to make a balanced hardware/software partitioning that meets all performance constraints at minimal costs. The current technological trend is to implement system parts which require intensive computations and high performance in hardware, e.g. in an ASIC. Other system parts, such as user interfaces and less computation-intensive functions, are implemented in software to reduce costs and to provide flexibility.

The advance in VLSI technology also permits to integrate more and more functionality on a single chip: yesterday's systems are today's chips. Consequently, more and more functionality and parallelism is integrated into products such as consumer electronics and telecommunication systems. However, there is a growing complexity gap between the number of transistors on an IC and the designers' ability to design complex, parallel systems using these ICs. Due to this increasing complexity it is becoming very difficult for designers to prospect a system-level view and to make a balanced hardware/software partitioning.

Re-use of hardware and software components is gradually becoming more and more important. Re-using components that have been developed earlier as part of other systems, can shorten design time and costs. Hardware re-use is mainly driven by the increasing number of hardware cores that can be purchased from many providers. Cores can be fixed cores like floating-point units or MPEG decoders, or programmable cores like DSP or processor cores.

Flexibility is another emerging criterion for modern hardware/software systems. Flexibility is mainly achieved by software, because adapting and upgrading software is much cheaper than replacing hardware. Examples of software flexibility are incorporating changes in software late in the design cycle, for instance due to changes in the system specification. Software flexibility also allows to develop various slightly different or personalized versions of a system, to satisfy the diversity of consumer demands, to deal with regional differences on the current global markets, and to make several generations of products using the same hardware architecture.

The result of these developments is that new design methods for heterogeneous hardware/software systems are required. The traditional, ad hoc design methods are no longer sufficient and hardware/software co-design has become an economic necessity for reducing design time and costs. As shown in figure 2.1, the basic problems of the past design methods are the lack of a well-defined design flow, the lack of well-defined system specifications at the behavioral and architectural level, the separated hardware and software design flows, the troublesome verification of the integrated hardware/software system, and the long design iteration loops. Hence, new system design methodologies are required for co-designing and verifying hardware/software systems to overcome the problems of traditional design methods. As shown in figure 2.1, hardware/software co-design methods offer a well-defined design flow, emphasizing system specification at the behavioral and architectural level, and establishing synergy between the hardware and software design flow. Hardware/software co-design eliminates the long re-design iteration loops of traditional design methods. (Obviously, there are short, local iteration loops between any two subsequent stages in the design flow, both in traditional design methods as well as in hardware/software co-design methods. For clarity, these short iteration loops are not shown in figure 2.1.)

Many hardware/software co-design projects are currently under development at universities and industrial research laboratories. The industry is putting these co-design methods into practice very rapidly, and EDA (Electronic Design Automation) companies are working on tools to support hardware/software co-design.

2.2 Classifying Hardware/Software Co-Design

At the moment, there is a large number of ongoing research projects on hardware/software co-design in universities and industry. Although all projects focus on designing hardware/software systems, they have different goals and use different approaches to reach these goals. We can roughly classify hardware/software co-design methods into co-design of Application-Specific Instruction Processors (ASIPs), co-design of software-oriented systems, and co-design of heterogeneous hardware/software systems.

2.2.1 Co-Design of ASIPs

An Application-Specific Instruction Processor (ASIP) is a dedicated processor in which the hardware architecture and the instruction set are optimized for executing algorithms in a specific application domain [G^+96]. An ASIP combines the concepts of an ASIC and a programmable processor. An ASIP is software programmable and hence more flexible than an ASIC, and at the same time an ASIP provides higher performance for executing software than a standard processor.

The hardware architecture and the instruction set of an ASIP suit the requirements of a specific application domain, such as high-speed video and audio processing. Examples of ASIPs are Philips' programmable Video Signal Processors VSP1 and VSP2, which are targeted towards high-speed video signal processing algorithms [V^+95b].

ASIPs are closely related to Digital Signal Processors (DSPs). The hardware architecture and the instruction set of DSPs is targeted towards executing digital signal processing algorithms, in which multiplication of data samples with coefficients and the accumulation of products are basic operations. However, the hardware architectures and instruction sets in ASIPs are more application-specific than in DSPs.

The definition of an ASIP's hardware architecture and instruction-set architecture (ISA) requires both hardware and software considerations. The ISA should support efficient software implementations of algorithms in the specific application domain, as well as efficient usage of the features in the ASIP's hardware architecture. The hardware architecture of an ASIP typically contains a number of parallel microcode-controlled datapaths, memory buffers, and communication switches. In parallel with developing the ASIP's hardware architecture, software tools have to be developed, like a processor simulator and a specific compiler targeted towards the ASIP's hardware architecture. The compiler should exploit the features of the hardware architecture, like pipelining, caches, and efficient use of the parallel hardware modules.

2.2.2 Co-Design of Software-Oriented Systems

This flavor of co-design focuses on system design in which has been decided a priori that the system is to be implemented in software. Co-design now deals with selecting an appropriate hardware platform on which the software can be executed efficiently. Selecting the hardware platform implies choosing the processor on which the software is executed, the bus architecture, and the memory architecture. If performance constraints cannot be met, the system can be accelerated by porting some parts of the software into dedicated hardware. Section 2.6 elaborates on co-design of software-oriented systems.

2.2.3 Co-Design of Heterogeneous Hardware/Software Systems

Co-design of heterogeneous hardware/software systems deals with system design starting from an implementation-independent specification. The goal of co-design is to find an optimal hardware/software architecture that implements the system specification and meets the constraints on real-time behavior, performance, speed, area, memory, power consumption, flexibility, etcetera. In co-design, the implementation decisions for hardware, software and communication interfaces are closely related: changes in one will immediately affect the other two. The focus of co-design is on designing at higher levels of abstraction while increasing design automation at lower levels through synthesis tools.

Examples of co-design projects focusing on heterogeneous systems design are POLIS at Berkeley University [C$^+$94b], Ptolemy at Berkeley University [KL93, BHLM94], work performed at Carnegie Mellon University [TAS93, AT96], Chinook at the University of Washington [COB95a, BCO96], CASTLE at GMD (Germany) [TSV94], work performed at the University of Illinois at Urbana-Champaign [Gup95, GM96], Tosca at the University of Milan (Italy) [BFS96a, BFS96b], COSMOS at INPG (France) [IAJ94, JO95], CoWare at IMEC (Belgium) [RVBM96, M$^+$96, VRBM96], and work performed at the Eindhoven University of Technology [vdPVS95, VvdPS96, vdPV97].

The sections 2.3, 2.4 and 2.5 elaborate on co-design of heterogeneous hardware/software systems. The focus in this thesis is primarily on co-design of heterogeneous systems. In the remainder of this thesis, the term hardware/software co-design is used only to indicate co-design of heterogeneous hardware/software systems.

2.3 Hardware/Software Co-Design

Hardware/software co-design methods define the subsequent design steps to proceed from a conceptual description of the desired system behavior to the actual hardware/software implementation of the system [BSV95, CW96, Mic96, LSVH96, P$^+$96a, GVNG94, GV95, Wol94]. The subsequent design steps in the hardware/software co-design process differ among the numerous co-design projects in universities and industry. These differences are mainly due to different application domains and different target hardware architectures. Control-oriented systems require different approaches than data-oriented systems. Examples of co-design application domains are control systems, communication and telecommunication systems, signal and image processing systems, multimedia systems, automotive systems, domestic systems, and aerospace systems.

In most hardware/software co-design methods, typically three stages can be distinguished: specification of the functional system behavior, defining the hardware/software system architecture, and synthesis of hardware and software. During the co-design process, system design proceeds from behavioral representations of the functional specification, to structural representations of the hardware/software architecture, and to physical representations of the hardware and software components. Usually, the co-design flows are fairly complex and they have many short iteration loops, but they avoid the long re-design iterations of the traditional design methods.

A general method for hardware/software co-design consists of six steps that roughly define the co-design flow from system conceptualization to system implementation, as shown in figure 2.2. The design flow in figure 2.2 suggests a strict, phased sequence of successive design steps. In practice this is seldom the case: usually a spiral approach is practiced, implying concurrent work at several stages (particularly system requirements capture, system specification, and architecture exploration) and many local iteration loops [vdPV97].

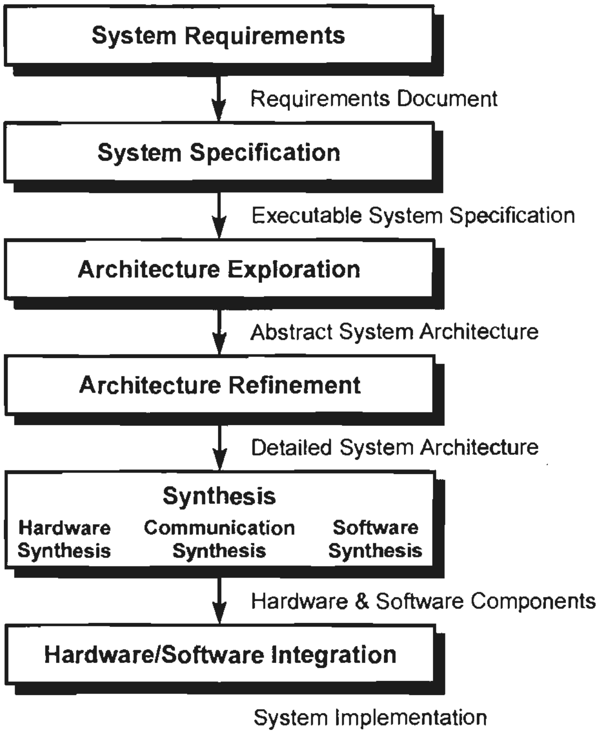


Figure 2.2 Hardware/software co-design flow

A concise description of the subsequent steps in the hardware/software co-design flow is presented next. A more detailed description will be provided in the sections 2.3.1 through 2.3.6.

1. The first step is to gather the **system requirements**. Usually, the system requirements are described informally using natural language. The resulting requirements document states the requirements on the system behavior and the environment in which the system should operate.

2. During **system specification**, the functional behavior of the system is captured into a conceptual, formal model. Usually, a hierarchical model of the system is created in which the system's functional behavior is decomposed into a number of parallel, communicating processes. The system specification is described using a formal specification language. The system specification is validated and verified using simulation and formal verification techniques. In most literature on co-design is stated that the system specification should not contain an implicit or explicit architectural model. The system specification should model the functional system behavior, independent of the implementation architecture. However, the organization of the system specification often already implies an implicit architectural model, because the same organization is used in the system implementation. A priori knowledge on the physical system architecture, like distributed processing nodes using a specific communication architecture (e.g. a shared bus), or dictated hardware and software

components, should already be reflected in the organization (i.e. communication mechanisms and decomposition) of the system specification [vdPV97]. The transition from behavior specification to hardware/software architecture is facilitated if the organization of the specification corresponds directly to the organization of the system architecture: this allows one-to-one mapping of objects in the specification to hardware or software components.

3. In the **architecture exploration** phase, numerous alternative hardware/software architectures are explored to find a system architecture that satisfies constraints like performance and costs. Exploration of the design space is achieved by defining a number of system architectures. A system architecture is a set of hardware and software components, each component implementing a part of the functional specification. The physical constraints like power consumption, silicon area, memory size, and performance of the hardware and software components are specified. The quality of the alternative system architectures is estimated by considering these constraints. The optimal system architecture is selected and this selected architecture is detailed in the subsequent design steps. Architecture exploration is one of the key issues in the current research on hardware/software co-design.

4. During **architecture refinement**, the system architecture is refined by adding more details on the implementation of hardware components, software components, and communication interfaces. The refined architecture describes the system's processors, memories, and buses. Co-simulation is used to verify whether the refined description is equivalent to the initial system specification. The result of architecture refinement is a system-level description, providing high-level, behavioral descriptions of the hardware components, the software components, and their communication interfaces.

5. **Synthesis** consists of three parallel activities: **hardware synthesis**, **software synthesis**, and **communication synthesis**. The task of synthesis is to convert an abstract, behavioral description into a description containing detailed implementation information.

   Hardware synthesis of custom components like ASICs consists of high-level synthesis and logic synthesis. High-level synthesis converts a behavioral-level hardware description into a structure of components at the register-transfer level (RTL) like ALUs and registers. Logic synthesis converts the RTL components into gate-level descriptions. Software synthesis determines the scheduling of software processes and inclusion of a real-time kernel or an operating system. In general, software synthesis converts a behavioral description into a traditional software program compilable by traditional compilers. Communication synthesis generates the communication interfaces between hardware and software components.

   Hardware components, software components and their communication interfaces can be co-simulated at various levels of abstraction. When synthesis and co-simulation is completed, the physical hardware design is done. The gate-level descriptions of custom hardware components are converted into switch-level descriptions and layout data for FPGA or custom ASICs is generated using design tools for placement and routing.

   The automatic synthesis of hardware, software and communication interfaces is a key issue in the current research on hardware/software co-design.

6. During **system integration**, all the hardware and software components are integrated and the final system implementation or a prototype implementation is obtained. Testing is

used to check whether the integrated system is a correct implementation of the system specification.

The co-design flow is based upon the principle of stepwise refinement. Initially, a high-level behavioral model of the system is created. In each subsequent design step, more implementation details are added and a more refined description of the system is obtained. In each design phase, the designer focuses on a different abstraction level. During system specification, the designer concentrates on the completeness and correctness of the functional system behavior. When defining the hardware/software architecture, the designer focuses on system performance and communication protocols between hardware and software components. During synthesis, the designer proceeds from behavioral-level descriptions (behavioral-level VHDL, C code) to lower level descriptions (register-transfer level and gate-level VHDL, assembly code and object code). During physical hardware design, the designer concentrates on detailed timing and electrical characteristics.

In the following sections, the six steps in hardware/software co-design are described in detail.

2.3.1 System Requirements

System requirements are requirements on the system's functional behavior, requirements on the environment in which the system has to operate, and non-functional system requirements. The system requirements are gathered and stated in the requirements document.

Functional system requirements define the functional behavior of the system. Often the behavior is described as sequences of input stimuli and the subsequent responses of the system. Requirements on the system environment define the objects in the system environment that interact with the system. Non-functional requirements are general requirements or implementation requirements, like performance, reliability or a prescribed system architecture.

The process of defining the system requirements is still a very amorphous and ad hoc process in most co-design methods. An exception is the SHE method [vdPV97], which particularly emphasizes requirements capture and system specification.

2.3.2 System Specification

During system specification, the functional behavior of the system is defined. A conceptual model is created to capture the system's functional behavior. The conceptual model is a hierarchical model, using functional decomposition to handle complexity. Besides hierarchy, the conceptual model concentrates on essential characteristics of embedded, real-time systems like concurrency, communication, synchronization, states and state transitions.

The system specification forms the basis for the subsequent design phases. A correct, complete and consistent specification is therefore a necessity. Unfortunately, system specification does not receive as much attention as subsequent implementation stages in the design flow. Consequently, many functional, conceptual errors are not detected until late in the design process or even during integration. However, design errors are far more difficult and more expensive to correct in the late stages of system design. System specification therefore should be one of the basic ingredients of hardware/software co-design. The goal of the system specification is to unambiguously define the system's functional behavior, resulting in a formal description of the system that can be validated and verified.

System specification consists of three tasks: selecting an appropriate modeling technique and description language, specifying the system using the modeling technique and description language, and finally validating and verifying the model. Usually the modeling and description, and the validation and verification tasks must be iterated several times before a complete, correct and consistent functional system specification is obtained.

There are many modeling techniques to capture functional behavior. Most commonly used in current co-design projects are:

- Finite-State Machine (FSM) model
  A system can be modeled as a hierarchical set of concurrent, communicating FSMs. In the basic model, the communication between FSMs is asynchronous communication, and the concurrent FSMs proceed synchronously in lock step. Various extensions to this basic FSM model have been proposed. The FSM model is very useful for modeling control-oriented systems that do not require complex data operations.

- Communicating Sequential Processes (CSP) model
  A system can be modeled as a hierarchical set of concurrent, communicating sequential processes. The communication between processes can be synchronous or asynchronous communication, and usually the processes proceed asynchronously. The CSP model is very useful for describing the behavior of control-oriented systems that incorporate much concurrency and also perform complex operations on data.

- Data-Flow Graph (DFG) model
  A data-flow graph is useful for describing systems with complex data operations, for instance digital signal processing systems. A data-flow graph decomposes the system's functionality into data transformation entities and data-flows between them. Data-flow graphs model the flow of data through a system, the variables in which data is stored, conditions, operations on data, and data dependencies. While FSM models and CSP models concentrate on coarse-grain partitioning of the specification at the task or process level, the DFG model focuses on fine-grain partitioning at the operation level. Obviously, DFG models are less suited for modeling control-oriented systems.

There is a variety of other models, such as Petri nets, control-data-flow graphs, object-oriented models, and queuing models. No modeling technique is ideal for all classes of systems. Selecting an appropriate model is most important to understand and define the system functionality during system specification. The best model is the one that most closely matches the characteristics of the system. CSP models are used in most co-design methods for heterogeneous system design, focusing on characteristics like hierarchy, concurrency, communication, synchronization, states and state transitions.

System specifications are described in a language. The language should be able to express the required system model. For instance, a FSM-based language is more appropriate to describe a control-oriented system than a data-flow language. Hence, there should be a one-to-one correspondence between the model characteristics and the expressive power of the language constructs. A close correspondence between the characteristics of model and language eases description and prevents errors. Examples of FSM-based languages are StateCharts [HPSS87, H^+90a] and synchronous languages such as Esterel [BB91, BS91b]; examples of data-flow based languages are Silage [H^+90b] and DFL [W^+94]; examples of process-based languages are SDL and behavioral-level VHDL.

Describing the system specification in a formal language should result in an executable system specification. Simulation of the executable specification allows the designer to validate and verify the system specification. Appropriate formal models and languages are very useful for capturing functional, behavioral system requirements. Non-functional requirements, such as constraints on performance, safety, reliability, and costs, are more difficult to express. These constraints provide essential information in the exploration phase when defining the hardware/software architecture.

A general problem in hardware/software co-design is modeling the system at a level that is high enough to capture and analyze the system-level characteristics, and low enough to allow efficient synthesis of the system. Several approaches have been used in hardware/software co-design methods:

- **Single-language approach**
  In single-language approaches, the system specification is described in one specification language. System-level specification languages like SDL and StateCharts allow to model a system at a high level of abstraction, in terms of communicating processes or communicating FSMs. Although the origin of these specification languages is in software engineering, they can be used to specify systems that will eventually be implemented as a heterogeneous system containing both hardware and software.

  However, high-level specification languages like SDL are not suited for detailed hardware and software design. After partitioning of the system, the software parts of the system-level specification have to be translated into software languages like C, while the hardware parts have to be translated into hardware languages like VHDL. Also the communication interfaces between hardware and software must be translated into a software or hardware language. An example of a co-design method using this approach is COSMOS [IAJ94, IJ95, JO95].

  A related approach is to use a hardware description language such as VHDL or Verilog for system specification. VHDL and Verilog allow specification at various levels of abstraction, ranging from the behavioral level using communicating processes, to detailed gate-level structural descriptions. The main advantage of this approach is that after hardware/software partitioning, the VHDL specification of hardware components can be used directly as input for hardware synthesis tools. The VHDL specification of software components however still requires translation into a programming language like C. Furthermore, the specification of complex software architectures in VHDL is cumbersome.

- **Mixed-language approach**
  In mixed-language approaches, the system specification is captured using a number of diverse languages. The underlying idea in this approach is that a heterogeneous system con-

sists of various parts, and each part can be described best in its own specification language. For instance, hardware is described best using a hardware description language like VHDL, software is described best using a software programming language like C, control-oriented algorithms are described best using a FSM-based language like StateCharts, and digital signal processing algorithms are described best using a data-flow language like Silage. Most co-design projects use a mixed-language approach, using a variety of different languages. A disadvantage of mixed-language approaches is that there often is already an implicit hardware/software partitioning and repartitioning is difficult to achieve. A more severe problem is that there is no common development environment because each language has its own development tools. The semantic difference between the languages and the interfacing between various design environments are basic problems. However, most co-design projects that use a mixed-language approach, like CoWare [RVBM96, VRBM96, M$^+$96] and Ptolemy [KL93, BHLM94], offer effective solutions to deal with these problems.

2.3.3 Architecture Exploration

During exploration of the design space, numerous alternative hardware/software architectures are explored to find a system architecture that satisfies constraints on performance, silicon area, memory size, power, costs, etcetera. During exploration, the system specification is partitioned into hardware parts, software parts, and communication interfaces.

Exploration usually starts with performance analysis of the system specification, resolving a first outline for hardware/software partitioning. Performance analysis mainly deals with profiling the system specification, analyzing execution rates of processes and uncovering communication bottlenecks [Ben96]. During partitioning, the system specification is divided into parts that will be implemented in hardware or software. An important issue is the granularity of partitioning, indicating the smallest indivisible functional object used in partitioning such as processes, subroutines, or blocks of statements. Higher granularity implies fewer parts with less communication interfaces, faster simulation and faster performance estimation. However, higher granularity also implies fewer possible partitions. In most co-design projects, manual or interactive partitioning is proposed, where the designer manually partitions the system. An experienced system designer is still needed to guide hardware/software partitioning. The co-design environment provides tools to support interactive partitioning and to compute metrics for evaluating the quality of the partitioning. An example is the PARTIF tool [IOJ94] for interactive partitioning in the COSMOS co-design environment.

After hardware/software partitioning, allocation is performed, which means selecting a proper set of software and hardware components to implement the hardware and software parts. Partitioning and allocation are usually iterated several times, because detailed metrics on performance and costs of the proposed partitioning can only be computed after allocation. The designer usually has hundreds of components to choose from. At one extreme, there are very fast but expensive custom hardware components like ASICs. At the other extreme, there are cheaper but slower general-purpose programmable microprocessors. Between these two extremes lie innumerable components that vary in costs, performance, power, size, flexibility, etcetera. During allocation, variables and data structures in the system specification are assigned to memory components, processes are assigned to hardware modules or software processes, and communication channels are assigned to hardware or software communication interfaces.

Partitioning and allocation assume that the objects in the system specification can be mapped one-to-one on hardware or software components. However, this one-to-one mapping is often impossible. The organization of the system specification is primarily intended for readability. Using the same organization structure in the hardware/software implementation may not lead to the optimal design. Therefore, transformations are required to reorganize the specification to achieve one-to-one mapping of objects to hardware or software components. Common examples of specification transformations are function inlining (i.e. replacing a function call with an instance of the function's body), merging processes, flattening hierarchy, splitting processes, grouping statements into procedures, and merging variables into arrays. Transformations reorganize the specification structure, but they should not modify the behavior stated in the specification. Hence, transformations should be behavior-preserving [VvdPS96].

In most co-design methods, usually some restrictions are imposed on the target architecture. In most cases, the target architecture consists of some application-specific hardware like an ASIC or FPGA, connected to the system bus of a general-purpose microprocessor. The microprocessor executes a small operating system and the application software. Communication between the application-specific hardware and the application software takes place on the system bus using device drivers from the operating system. Many co-design methods support multi-processor architectures, resulting in a distributed system. Distributed systems often offer the best implementation for embedded systems. For instance, time-critical tasks allocated to different processors may ensure that all hard deadlines are met, and often the usage of several small processors may be cheaper than using one complex processor.

During exploration, estimation is required to evaluate design metrics such as costs, performance, communication rates, power consumption, silicon area, testability, reliability, program size, data size, and execution time for a large number of system architectures. Usually multiple estimated metrics are combined to obtain a single cost value that defines the quality of a partitioning. It is required to weigh each metric by its relative importance to the overall design. The single cost value should give way to compare various partitionings and to select one that best satisfies the constraints.

In the exploration stage, an abstract description of the hardware/software architecture is created. Many implementation details of the hardware, software and communication components still have to be determined in the subsequent design stages. The design metric values derived from this rough hardware/software architecture are therefore estimates. More accurate design metric values can be derived from a more detailed hardware/software architecture, but this requires far too much time. Accuracy of the metrics and time required to compute the metrics are competing factors. Improving accuracy requires a more complete implementation, while reducing computation time requires a less detailed implementation. Estimating accurate metrics for hardware size, software size, and performance is not easy, because the mapping from a behavioral description into hardware or software is not straightforward. The complexity is introduced by optimization at different levels of abstraction, such as:

- Compilers use various optimizing algorithms. It is very difficult to predict the results of code reduction and performance enhancement.

- Architectural features of microprocessors like caching, pipelining, and multiple instruction execution make it difficult to predict the performance of software execution.
- Hardware synthesis uses optimization techniques like control logic optimization and state optimization. These optimizing algorithms make it difficult to predict the performance and size of hardware.

In general, the rough estimates derived from an abstract hardware/software architectural description, are sufficient to determine the quality of hardware/software partitioning and allocation in the exploration stage.

It can be concluded that exploration in most co-design methods consists of partitioning, allocation, transformation, and estimation. These four tasks are usually performed in various orders, passing through many iteration loops before a satisfied system-level hardware/software architecture is obtained.

2.3.4 Architecture Refinement

Design space exploration provides an abstract system architecture, modeled as a set of interconnected hardware and software components. During architecture refinement, the system architecture is refined by adding implementation details to the descriptions of the hardware components, the software components, and the communication interfaces. Examples are:

- A bus can implement a single communication channel or a group of communication channels in the system specification. During architecture refinement, details are added on the bus width and bus rate.
- During architecture refinement, the exact communication protocol for transferring data over the buses is defined, like a handshake protocol or a protocol using fixed time slots. Also addressing details and decomposing data for serial transmission must be determined.
- Often two components with fixed protocols have to communicate, like communication between a software process running on a microprocessor and an application-specific hardware component. Insertion of hardware and software is required to implement the hardware/software communication interface.
- When concurrently executing processes access the same resource (e.g. a bus or memory), mutual exclusive access must be ensured. During architecture refinement, arbitration processes are inserted to provide that only one process at a time is granted to access the shared resource in case of simultaneous requests. Arbitration schemes can use fixed priorities, in which priorities are assigned to each process statically, or dynamic priorities, in which priorities are assigned dynamically at run-time based on the access pattern of the processes.

The result of architecture refinement is a detailed system-level description of the system architecture that contains implementation details, but that is still largely functional. The system architecture is described as a set of behavioral models for hardware components, software components, and interface components. These behavioral models can serve as inputs for hardware, software and communication synthesis tools. The system architecture can be verified by co-simulation of the behavioral models.

2.3.5 Synthesis

Synthesis is the task of transforming a high-level specification or description into a detailed implementation description. Hardware synthesis and software synthesis can be performed on several abstraction levels. Hardware synthesis combines high-level synthesis and logic synthesis. High-level synthesis transforms a functional description into a structure of RTL components, such as registers, multiplexers, and ALUs. Logic synthesis transforms the RTL components into combinational and sequential hardware. Software synthesis is the task of converting a complex description into a traditional software program, compilable by traditional compilers. Scheduling of concurrent software processes is also a task of software synthesis. Any system compromising software and hardware components will need hardware/software interfaces to accommodate communication between the various components. Communication synthesis is the task of generating hardware and software to implement these communication interfaces.

2.3.5.1 Hardware Synthesis

Hardware components can be standard, off-the-shelf components or dedicated, custom components. Hardware synthesis is required for custom components like ASICs or FPGAs. Hardware synthesis combines high-level synthesis and logic synthesis [Mic94].

High-level synthesis or behavioral synthesis converts a hardware component's behavioral description into a structure of RTL components like ALUs and registers. The RTL description usually consists of a controller and a datapath. The datapath executes arithmetical and logical operations on data. The controller implements a finite-state machine which controls register transfers and operation modes in the datapath and generates signals for communication with the external world.

High-level synthesis consists of several tasks. Usually, the behavioral specification is compiled first into an intermediate representation, exposing control and data dependencies. Next, allocation selects hardware modules such as memory modules, functional modules, and bus modules from an RTL component database. Allocation determines the proper number and type of hardware resources needed to satisfy constraints on costs, performance and power. Scheduling assigns the operations in the behavioral description to clock cycles, taking into account data dependencies and control steps. Finally, during binding, variables are assigned to registers and memories, operations are assigned to functional modules, and data transfers are assigned to buses. There are many tools and design environments available for high-level synthesis.

After high-level synthesis, sequential and logic synthesis transform the RTL components into combinational and sequential hardware. A controller, implementing a FSM, is transformed into a hardware structure consisting of a state register and a combinational circuit that generates the next state and the outputs. This involves tasks like state minimization, state encoding, and logic minimization. A variety of sequential and logic synthesis techniques is available.

After hardware synthesis, the physical hardware design is performed to generate manufacturing data. The technology-independent gate-level netlists are converted into layout data for gate arrays, FPGAs, or custom ASICs. Tools are used for placement and routing, timing analysis and power analysis.

2.3.5.2 Software Synthesis

Architecture refinement offers a high-level description of concurrent, communicating software processes. When the concurrent processes are executed on a single processor, the processes must be scheduled for sequential execution. Scheduling should ensure that the processes are executed without deadlock and starvation and that timing constraints are satisfied. Furthermore, scheduling should minimize the amount of busy-waiting. Scheduling of software processes cannot be performed by compilers for traditional languages like C. The task of software synthesis is to convert the high-level description of concurrent software processes into a traditional software program, compilable by traditional compilers. Software synthesis determines the execution order of software processes, satisfying resource and performance constraints. The run-time scheduling is carried out by a real-time kernel or a small operating system. Hence, software synthesis also implies including a real-time kernel or an operating system.

Many compilers use standard optimization techniques that are well-suited for all processors. Most compilers do not attempt to optimize code for a particular processor. Since developing a new compiler for each processor or custom datapath would be very costly, much research is done on retargetable compilers that can generate optimized code for different instruction sets [G$^+$96].

2.3.5.3 Communication Synthesis

Communication synthesis is the task of converting high-level descriptions of communication interfaces between hardware and software components into detailed descriptions. Communication synthesis typically consists of generating glue logic to enable communication between hardware components, and generating device drivers to enable communication between software and hardware components.

In the current co-design methods, two approaches towards communication synthesis can be identified. Some co-design methods try to generalize communication mechanisms on a high level, without a clear bias towards target implementation. This approach avoids an early limitation of the design space. An example is communication synthesis in the COSMOS co-design method [OIJ93]. The second approach is that some co-design methods prefer a limited design space with a fixed hardware/software target architecture that matches a certain application domain. The communication protocols are limited to standard processor protocols or unbuffered point-to-point communication, which facilitates communication synthesis. Examples of co-design methods that employ automated synthesis of hardware/software communication interfaces are CoWare [LV94, LVM96], Chinook [COB92, COB95b], and work performed at the University of California at Irvine [NG94b, NG94a, NG95].

2.3.6 Integration

When synthesis is completed, all hardware components (ASICs, FPGAs, processors, ...) and software components (memories containing program code) are integrated. The goal of hardware/-software integration is to integrate all hardware and software components, such that the final system implementation or a prototype implementation is obtained. In the traditional design methods, hardware/software integration is very troublesome due to incorrect hardware/software interfaces. The required testing and debugging activities are extremely time and resource intensive. Correcting these errors usually results in long design iteration loops, as shown in figure 2.1. Co-design methods relax the problems of hardware/software integration, because the hardware/software interaction has already been verified earlier in the design process by co-simulation. However, also co-design methods still require testing and debugging of the integrated hardware/software system.

2.4 Validation and Verification

Validation and verification are very important issues in hardware/software co-design. Validation answers the question whether the right system is being built. Verification answers the question whether the system is being built right. Validation usually implies checking whether the system specification is a correct description of the desired system as stated in the system requirements. When validation is completed, the system specification should be a correct and complete description of the required system behavior.

In each design phase in the co-design process, the system description is transformed into a new, more refined system description. Verification is required to check the correctness of each design phase. Verification usually implies checking whether the refined system description corresponds to the initial system description in each design step.

The verification techniques can be classified into formal verification, simulation, and testing.

2.4.1 Formal Verification

Formal verification techniques can be applied to formal system descriptions. Formal languages provide precise semantics that allow mathematical proving. Formal verification is the only verification technique that can guarantee the correctness of a model. Formal verification techniques can be classified into two categories:

- Formal verification techniques can be used to prove certain properties of a model [Mel94]. The verified properties are usually safety properties and liveness properties. Safety properties state that the system will never perform some erroneous behavior, like deadlocking or emitting undesired outputs. Liveness properties state that the system will perform some behavior eventually or infinitely often, such as eventually emitting an expected response to an input.

  Formal verification of properties is very powerful, because proofs provide 100% accuracy and coverage. However, only a limited number of properties can be proven, and state space explosion causes that formal verification can only be applied to system models of moderate complexity.

- Formal verification techniques can be used to prove the equivalence between two models [Mar95]. For instance, formal verification can be used to prove the equivalence of two system models that are both described using process algebra. However, state space explosion causes that only systems of moderate complexity and size can be verified. Equivalence proving is also powerful to prove the equivalence between RTL descriptions and gate-level descriptions of hardware components. In this case, formal verification allows to check the results of hardware synthesis tools.

Examples of formal verification tools are theorem provers, model checkers and logic checkers. Theorem provers [BKM95, GM93, Mel94, Mar95] assist the designer in carrying out a formal proof of a property or of equivalence, either by checking the correctness of the proof or by automatically performing some steps in the proof. Model checkers [BCMD90] assist the designer by proving a safety or liveness property that is described as a formula in temporal logic. Logic checkers [P$^+$96b] are used to check the equivalence between RTL models, gate-level models or switch-level models.

It can be concluded that formal verification is very powerful, but practical use is limited due to state space explosion. Formal verification is not widely applied in industry yet, also because designers are usually not familiar with creating formal models of a system, with describing its properties formally, and with applying formal verification techniques.

2.4.2 Simulation

Simulation means execution of a system model and comparing the simulation results with the expected results. Contrasting to formal verification, proving correctness with simulation is very difficult, because exhaustive simulation is usually unfeasible and simulation coverage is difficult to measure. Nevertheless, simulation is currently the most common verification technique. Simulation can be applied to verify the system specification, but also to verify the more detailed descriptions in subsequent stages of co-design process.

- The system specification is described in a language, resulting in an executable specification. The executable specification can be simulated to validate the specification and to verify the completeness and correctness of the specification. Simulation of the system specification also provides performance measures.

- The system architecture is described in terms of interconnected hardware and software components. After architecture refinement, the system architecture is a behavioral-level description of hardware components, software components, and communication components. Simulation allows verification of the behavioral-level descriptions. Simulation of the system architecture results in more detailed performance measures.

- During synthesis, the behavioral-level descriptions of hardware, software, and communication components are gradually refined into more detailed descriptions. Hardware descriptions typically proceed from behavioral-level descriptions to RTL descriptions, gate-level descriptions, and switch-level descriptions. Software descriptions typically proceed from concurrent processes in C to assembly code and object code. Co-simulation of hardware and software components at various levels of abstraction is one of the most powerful benefits of hardware/software co-design.

- The physical, switch-level description of hardware components can be simulated to verify detailed electrical and timing properties.

Co-simulation of hardware and software components is one of the key benefits of the current hardware/software co-design methods. Each hardware and software component is verified separately by means of simulation or formal verification, to verify its functional correctness. However, the correctness of the complete system after integration of hardware and software components cannot be concluded, even if all individual components have been verified to work properly. Verification of the communication between the hardware and software components is required. Co-simulation of hardware and software components is very powerful, because it allows verification of the hardware/software interaction early in the design process. In the traditional design methods, system integration problems were not detected until integration of software components and the physical hardware components. It has been reported that typically 30–40% of the design effort in traditional design methods is spent on hardware/software integration, testing and debugging. Co-simulation allows very early integration of hardware and software, before the physical hardware components are manufactured. The goal of co-simulation is to move the point of integration and test closer to the design phase, where both hardware and software designers can see the others work and make changes before the real hardware is built. This has a major impact on the design cycle, eliminating long design iteration cycles.

Hardware/software co-simulation has two competing goals: simulation speed and resolution of simulation results. Simulation speed is usually orders of magnitude slower than the real-time execution of the system. At high levels of abstraction, typically the behavioral level, high simulation speed can be obtained, but the resolution of time and data is rather low. At lower levels of abstraction, data and time resolution is higher, but simulation speed is much slower. Various approaches towards co-simulation have been developed [CKL96, Row94, ME95]:

- **Hardware simulator**

  Co-simulation can be achieved by using a single simulation environment in which all components are simulated. Typically a hardware simulator is used, like a VHDL-simulator or a Verilog-simulator. These hardware simulators are EDA tools that use detailed software models to simulate the function and timing of electronic circuits.

  VHDL and Verilog simulators are typically discrete-event simulators, which means that every event occurs at some discrete point in time that is tagged with a time stamp. There is a totally ordered relationship between any two events: the two events occur either simultaneously, or one event precedes the other one. During simulation, the simulator has to sort the events by evaluating their time stamps, so that events with the earliest time stamps are processed first. This sorting can be computationally expensive, which makes event-driven simulators rather slow.

  An alternative model for totally ordered, discrete events is the discrete-time model. In this model, events are related to a clock. Any two events occur either simultaneously at the same clock tick, or they occur at different clock ticks so one event precedes the other event. Simulators based on the discrete-time model are called cycle-based simulators. They are simpler than event-driven simulators, because sorting of events is less expensive. Processing all events at a given clock tick forms a cycle. Within a cycle, the order in which events are processed is determined by data precedences, which define microsteps.

Co-simulation is achieved by creating discrete-event or discrete-time models of both the custom hardware components and the processor(s). The software components are modeled as memories containing object code. This requires that the software has already been compiled into a binary format and loaded into the simulated memories. Although co-simulation yields very accurate results, executing the software at this binary level requires the simulator to do many calculations during each processor cycle. Hence, this co-simulation approach is very slow, resulting in simulation times measured in days, weeks or months. Another disadvantage is that detailed functional models of microprocessors are usually not available. Most processor vendors do not provide such detailed models, because they reveal a great deal about the internal processor architecture.

Instead of modeling the detailed internal hardware architecture of a processor, a bus model of the processor can be created. A bus model is a discrete-event model that simulates only the bus interface of the processor. The software is executed using a high-level instruction interpreter that provides information about the number of clock cycles required for a sequence of instructions between a pair of I/O operations on the bus. This approach is useful for verifying low-level interactions such as communication on the processor bus or memory access. However, it is not easy to create accurate bus models and it is difficult to accurately simulate the hardware/software interaction.

- **Instruction-set simulator**
  Faster co-simulation can be achieved by using an instruction-set simulator linked to a hardware simulator. An instruction-set simulator for a particular processor is based on a model of the processor's instruction-set architecture that does not contain all the details of a full-function, discrete-event processor model. Instruction-set simulators allow faster simulation of software running on a microprocessor. The custom hardware components are still simulated using a hardware simulator. The hardware and software simulators are linked so that the instruction-set simulator runs in lock step with the hardware simulator. This co-simulation approach achieves only a modest gain in speed over co-simulation using a single hardware simulator.

- **Virtual prototyping**
  New co-simulation environments provide solutions to link hardware and software development tools and enable fast execution of software application code on simulated hardware. These co-simulation environments allow virtual prototyping, solving basic problems in co-simulation like speeding up software simulation and synchronizing hardware and software simulations. Examples of virtual prototyping tools are the Power PC Virtual System from IBM, the Eagle-i and Eagle-v tools from Eagle Design, and the Seamless CVE tool from Mentor Graphics [MCC96]. These virtual prototyping tools allow co-simulation of the processor, memory, and the application software at various levels of abstraction. Co-simulation may be distributed over a heterogeneous network of workstations and PCs.

Despite the progress in hardware/software co-simulation environments, co-simulation speed is often not sufficient for large scale simulations and consequently it takes too long to run. Especially for data-oriented systems with hardware accelerators, the co-simulation times are unacceptable. Co-simulation can be speeded up by using emulation techniques [HO96, MCC96].

- Software is usually executed on a standard, off-the-shelf processor. Instead of simulating software execution, the software can be executed directly in real-time on the physical target processor. This approach is called a **hardware model**.

  The custom hardware components are simulated using a hardware simulator. The hardware model should be running in lock step with the hardware simulator. Hence, this approach is still very slow. Furthermore, the software has to be loaded into memory to be executed by the target processor. This provides no support for software development and debugging.

  Software can also be executed on the target processor using an **in-circuit emulator (ICE)**. The ICE replaces the physical target processor, and allows access to the internal registers of the target processor. This approach supports software development and debugging.

- Prototypes of custom hardware components can be obtained using **hardware emulation** techniques. Hardware emulation implies the use of programmable components such as FPGAs to implement the custom hardware components. Hardware emulation systems consist of a fixed architecture of programmable-hardware devices, i.e. FPGAs and their interconnection scheme, and a software compiler required to map the hardware description onto the emulation architecture. The advantage of FPGA emulators is speed. However, FPGA emulators typically run at about one-tenth of the real time and therefore cannot be used for debugging of real-time behavior. Furthermore, FPGA emulators are expensive and take some time to be programmed. Debugging the emulation system is usually performed using a logic analyzer, probes and a stimulus generator board which is incorporated into the emulation system.

Hardware and software emulation techniques can be combined, for instance using an FPGA emulator for custom hardware and an ICE for the target processor that executes the software. These emulation techniques allow rapid prototyping. Building a prototype is very common in embedded systems design. Emulation by FPGAs can improve performance up to four or five orders over logic simulation [HO96]. Emulation is also commonly used for designing high performance microprocessors [P+96b]. A logic emulation prototype of a microprocessor can execute pseudorandom verification vectors and software application programs (e.g. booting Unix) up to six orders of magnitude faster than conventional software logic simulators [KSFM95].

Table 2.1 shows an overview of the performance of various commercial tools for co-simulation and emulation. Performance is measured as the number of clock cycles that are simulated or emulated per second.

Tool	**Performance** (clock cycles/second)
Event-driven simulator (Verilog)	5
Cycle-based simulator (SpeedSim)	200
Simulator Accelerator (Zycad)	2,000
Virtual prototyping (Eagle)	5,000 – 50,000
Virtual prototyping (IBM PVS)	100,000
Hardware emulation (Quickturn)	200,000
Real Hardware	1,000,000 – 100,000,000

Table 2.1 *Performance of co-simulation and emulation tools [MCC96]*

It can be concluded that there are various options available for co-simulation and emulation. The options differ in: performance, which is the speed at which the simulated or emulated system runs; costs for tools (simulators) and equipment (hardware emulators, ICE); time required to set up the simulation or emulation environment; and debugging support, which deals mainly with the visibility into hardware and software components. The ideal solution offers high performance, requires low costs and short set-up time, and provides excellent debugging support.

Traditional co-simulation methods using hardware simulators and instruction-set simulators suffer from low performance, but they require low costs and short set-up times. Virtual prototyping environments offer higher performance. The debugging support in design environments for co-simulation and virtual prototyping is excellent, offering high visibility into hardware and software components.

Rapid prototyping using hardware emulation and ICE offers high performance, but requires high costs and long set-up times. The debugging support of ICE is excellent for software, but the debugging of the surrounding hardware is limited. The debugging support in hardware emulation is usually very limited. Another disadvantage of rapid prototyping is that the prototype does not use the same technology as the final system implementation. Hence, the final system implementation may still contain design errors, like timing errors, that were not present in the prototype. The inherent problem of rapid prototyping however is that the designer winds up debugging the prototype instead of the final system implementation. Debugging the prototype can be as troublesome as debugging the final system implementation, which has been shown to be a major bottleneck in the design process.

2.4.3 Testing

Verification of the prototype implementation or the final implementation of a hardware/software system is done by testing. Testing is performed by offering test stimuli to the system, and observing and evaluating the responses of the system. An incremental approach to testing is required: each hardware and software component is tested first in isolation; when the component passes the tests, it can be integrated into the system. Testing the final system is still required to check the correctness of the interactions between all hardware and software components. When testing reveals the presence of an error in the hardware/software system, debugging is required to locate the error and to find out the exact cause that effected the error.

Hardware/software integration, testing and debugging, and the subsequent re-design cycles to correct the errors, form the main bottleneck in traditional hardware/software design methods. Typically 30–40% of design time and costs in embedded system design [MCC96], and 50–70% of design time and costs in distributed, real-time systems design [TY95], are spent on these activities. When an error is detected during testing of the hardware/software system, it is often not clear whether the error is caused by hardware or by software components. Usually the software designers blame the hardware, while the hardware designers blame the software. Because the integration problems are discovered very late in the design process, it is usually too expensive to redo the hardware design. Integration problems are therefore generally fixed in software. However, in systems with very complex software architectures like communication switches, it is more cost effective to re-design the hardware than modifying the software once it is stable. Nevertheless, often systems are shipped to the customer without the full functionality or meeting all performance requirements.

Hardware/software co-design methods improve this situation, because they allow to verify hardware/software interaction much earlier in the design process using co-simulation. Design errors can be detected and repaired before the actual system or prototype system is built. Hence, fewer design errors will be detected during hardware/software integration. However, co-simulation and formal verification techniques cannot guarantee the complete absence of design errors. Consequently, some design errors will not be revealed until hardware/software integration. Testing and debugging therefore is a necessity during hardware/software integration.

Testing and debugging of hardware/software systems is a very troublesome process. Testing a system though its external interfaces is often not sufficient to test the interactions between components inside the system: many aspects of the internal operation of the system are not externally visible. Furthermore, it is very difficult or even impossible to achieve visibility into an individual hardware or software component. Visibility is achieved by using auxiliary instrumentation, such as oscilloscopes, logic analyzers and measurement equipment. These instruments provide some visibility into hardware and software components. In-circuit emulators and embedded software monitors improve software visibility, but this has a major impact on the hardware/software system. Unfortunately, the current hardware/software co-design methods provide no support to improve the testing and debugging of hardware/software systems.

2.5 Co-Design Methods

In this section two co-design methods for heterogeneous hardware/software systems are outlined: the COSMOS co-design environment developed at TIMA-INPG (Grenoble, France), and the CoWare co-design environment developed at IMEC (Leuven, Belgium). Both co-design methods emphasize communication synthesis.

2.5.1 COSMOS

COSMOS is a co-design environment for the specification and synthesis of hardware/software systems [IAJ94, IJ95, VIJK94, Ism96, V$^+$95a]. The COSMOS design flow consists of system specification, system partitioning, communication synthesis, and architecture generation.

2.5.1.1 System Specification

The system specification is described first in SDL. Simulation is used to validate the SDL specification. Next, the SDL specification is automatically translated into an equivalent specification in SOLAR [JO95]. SOLAR is an intermediate design representation for heterogeneous hardware/software systems, modeling system-level concepts such as hierarchy, inter-process communication, synchronization and concurrency. SOLAR provides an intermediate design representation that can be translated into VHDL and C.

Most SDL concepts can be translated directly into SOLAR, because both languages are based on the extended FSM model. However, SDL and SOLAR use different communication concepts. Each SDL process is translated into an equivalent process in SOLAR that consists of a Channel Unit (CU) and a Design Unit (DU). The separation between CU and DU allows to describe the functional behavior (DU) separately from the communication behavior (CU) of a process. The DUs execute concurrently and asynchronously. The behavior of each DU is described using Harel's StateCharts [HPSS87, H$^+$90a].

Communication and synchronization between the DUs is performed over channels. A channel is regarded as a shared resource between DUs. A CU is an abstract representation of a channel. A CU contains a controller, a collection of methods, and a number of ports. Whenever a DU performs a communication operation, it accesses a method in the CU by means of a remote procedure call (RPC). The remote procedure in the CU unpacks parameters, calls the requested method in the CU, and sends a reply back to the DU. The actual communication protocol is described in the methods, which can be any kind of communication such as message passing, using shared data, or a complex layered protocol. Because the contents of the methods are invisible for the DUs, the communication protocol is completely transparent. The CU controller handles conflicts when multiple DUs access the channel simultaneously.

SOLAR allows a strict separation between functional behavior (DUs) and communication behavior (CUs). In the subsequent design steps, the DUs and CUs can be refined separately. A disadvantage of the SOLAR language is that it has no formal semantics. SOLAR's execution model is defined in an informal way similar to many programming languages: certain elements of the language are simply defined by the operations they perform. The absence of formal semantics prevents the use of formal verification techniques, like verification of properties or equivalence checking between two SOLAR descriptions.

2.5.1.2 System Partitioning

COSMOS uses an interactive approach to system partitioning, based on the PARTIF tool (PARTItioning of extended Finite-state machines) [IOJ94]. PARTIF uses as inputs the SOLAR system specification, a library of communication protocols, and user-imposed constraints like the maximum number of parts and the maximum number of states in any part. The output of PARTIF is a graph, where the edges represent channel accesses and the nodes represent hardware DUs, software DUs, or CUs. PARTIF allows interactive partitioning through a number of transformation primitives, like reordering the hierarchy, merging processes, and splitting processes. The designer controls what transformations are performed and in what order. PARTIF outputs information like interconnections between parts, shared variables between parts, silicon area for hardware modules, operations, local variables, number of states, and execution time (number of clock cycles). The designer can use this information to evaluate the quality of a partitioning and to compare different partitioning alternatives.

2.5.1.3 Communication Synthesis

Communication synthesis in COSMOS transforms the CUs into hardware/software communication mechanisms. The behavior of the CUs is distributed among the DUs and communication controllers. Communication synthesis consists of channel binding and channel mapping, as shown in figure 2.3.

- Channel binding implies protocol synthesis, in which a particular communication protocol is selected for each CU. The communication protocols are selected from a library of communication protocols, considering the communication type (serial or parallel, synchronous or asynchronous), the required performance, and the implementation (hardware or software) of the DUs that communicate via the CU.

- Channel mapping implies interface synthesis, in which the CU is distributed among the DUs and a communication controller. The communication controller is selected from a library, based on criteria like data transfer rates and memory buffering capacity. The communication controller may be a software component described in C or a hardware component described in VHDL.

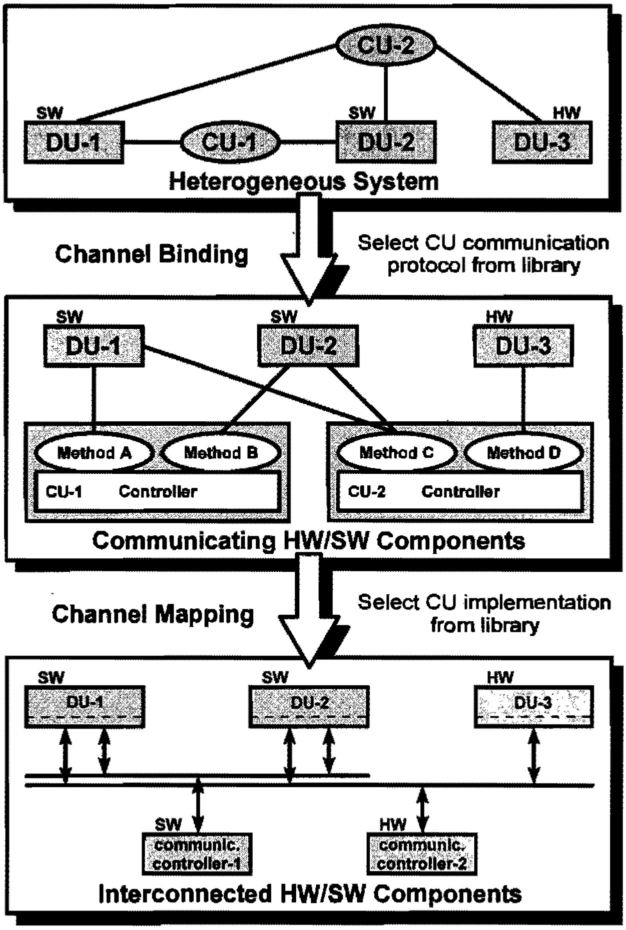


Figure 2.3 Communication synthesis in COSMOS

2.5.1.4 Architecture Generation

Architecture generation consists of virtual prototyping and architecture mapping.

- The virtual prototype is an abstract architecture represented in VHDL for the hardware elements, C for the software elements, and VHDL or C for the communication controllers. The virtual prototype is verified by co-simulation.
- During architecture mapping, the abstract architecture is mapped onto a physical architecture containing hardware subsystems, software subsystems, and communication subsystems. This is achieved using standard compilers to transform C into assembler code and synthesis tools to translate VHDL into ASICs.

2.5.2 CoWare

CoWare is a co-design environment for the design of heterogeneous hardware/software systems [Bol96, RVBM96, VRBM96, M$^+$96, LV94, LVM96]. CoWare concentrates on designing DSP systems, which typically consist of one or more data-flow paths with different data rates and execution rates for digital signal processing, slow control loops for mode selection and parameter setting in the data-flow paths, and a reactive control system for handling events from the environment like a user interface.

2.5.2.1 System Specification

CoWare uses a mixed-language approach: the system specification is a heterogeneous description of communicating processes. The DSP paths consist of data-flow components, performing operations on data streams, which can be specified best in an applicative data-flow language like DFL or SILAGE. Other DSP paths can be described best in VHDL as a datapath controlled by a FSM. The slow synchronization loops and control loops are specified best using a FSM-based language like StateCharts. These control loops are good candidates for software implementation and can therefore be described in C.

The benefit of CoWare's mixed-language approach is that various specification paradigms, specification languages, design tools, and simulation environments can be coupled. However, the system specification already incorporates an implicit system architecture and a hardware/software partitioning. Furthermore, analysis of the specification and formal verification is more difficult in a mixed-language specification than in a single-language specification.

A basic principle in CoWare is the strict separation between functional behavior and communication. System components are modeled by means of processes. Communication between processes takes place through a behavioral interface, which consists of ports. The ports of two communicating processes are connected by a channel. The communication semantics are based on the concept of the remote procedure call (RPC): one process triggers the execution of a thread in another process. The channel carries both data and control information for/from the remote thread.

A *process* describes the behavior of a system component. A primitive process is described in C, DFL, or VHDL. It consists of a context and a number of threads. The context contains code that is common to all threads in the process, such as variable declarations and shared functions. A hierarchical process models a set of processes and is described in CoWare's internal language.

A *thread* is a single flow of control within a process. A process can contain multiple threads. A slave thread is associated to a slave port, and is executed when the slave port is activated. A time-loop thread is not associated to any port, and is executed in an infinite time-loop.

A *port* is an object through which processes communicate. A primitive port consists of a protocol and a datatype parameter. A hierarchical port is used to describe protocol conversions and data formatting.

A *protocol* defines the communication semantics of a port. A primitive protocol indicates the way of data transport: it indicates the data direction and whether the protocol acts as master (activating a RPC) or slave (servicing a RPC). A hierarchical protocol defines a specific protocol implementation. It uses a number of terminals and a timing diagram that indicates how the logic values on the terminals evolve over time during data transport.

A *channel* connects a master port to a slave port for point-to-point communication. A primitive channel provides unbuffered communication. It has no behavior, and is just a medium for data transport. A hierarchical channel refines a primitive channel by specifying behavior. At the conceptual level, a hierarchical channel can be used to model a communication channel, e.g. a communication channel with bandwidth limitations or noise sources interacting with it. At the implementation level, a hierarchical channel is used to specify communication buffers, such as FIFOs and stacks.

The CoWare environment supports three communication mechanisms:

- *Intra-process communication* is inter-thread communication within a process via shared variables in the context of the process.
- *Inter-process communication with a primitive protocol* is communication between a master thread and a slave thread based on RPC semantics.
- *Inter-process communication with a hierarchical protocol* is communication between two processes defined by the terminals and timing diagram of the protocol.

The use of primitive ports, primitive protocols, primitive channels and RPC as basic communication mechanism provides that the designer can concentrate on the system's functional behavior. As the design process progresses, the primitive ports, protocols, and channels are refined by making them hierarchical and by replacing the RPC communication.

Transforming ports, protocols, and channels from primitive to hierarchical descriptions, can be done without modifying the functional behavior descriptions of the processes. Traditionally, the description of a process contains both functional and communication behavior in an interleaved way. Re-using such process often implies that the communication part has to be changed. The

RPC communication mechanism in CoWare avoids this, and libraries with communication behaviors can be constructed. In this sense, CoWare supports design for re-use and re-use of designs.

The system specification is validated by simulation on a Unix workstation. Unix communication primitives are used to implement communication between various simulators (VHDL, multithreaded C library).

2.5.2.2 Partitioning and Refinement

The system specification is modeled as a collection of processes described in C, VHDL or DFL. During partitioning, the processes are allocated to hardware or software components. A process is never split over several components, because the process is the finest grain of partitioning. However, a number of processes that is mapped on the same component can be merged into a single process. Alternative merges may require that for some processes a description is available in more than one specification language. Hardware/software trade-offs are based on the required degree of flexibility and performance constraints. After partitioning and binding, a partition is obtained such that there is a one-to-one binding of merged processes to processors. The designer can refine the communication mechanisms between the processors by expanding primitive channels into hierarchical channels.

2.5.2.3 Synthesis

Each process is synthesized, which implies a compilation and an encapsulation step. During compilation, hardware compilers and software compilers are used to generate hardware and software components. Encapsulation is required to encapsulate the generated components in such a way that their interfaces are consistent with the original specification.

Software processes, described in C, are assigned to a programmable processor core. Generally, the interface of the processor core is fixed and does not correspond to the required interface of the C process that has to be executed on it. Therefore, a number of software I/O device drivers is added. These I/O device drivers link the original C process to the software interface of the processor. Additionally, hardware interfaces are inserted to link the hardware interface of the processor to the hardware interface of external hardware components. The addition of software I/O device drivers and hardware interfaces is performed automatically. Figure 2.4 depicts the automated insertion of I/O device drivers and hardware interfaces.

2.5.2.4 Verification

CoWare uses co-simulation to verify the functional behavior of the system at several levels of abstraction. Unfortunately, the time required for co-simulation becomes excessive at the lower abstraction levels. The simulation times are less excessive at higher abstraction levels, but these levels provide not enough detail to verify real-time behavior.

CoWare's automated approach to communication synthesis offers many advantages. However, refinement of the communication interfaces during synthesis may alter the timing behavior of the system. This can introduce errors like deadlock and starvation of processes, which were not

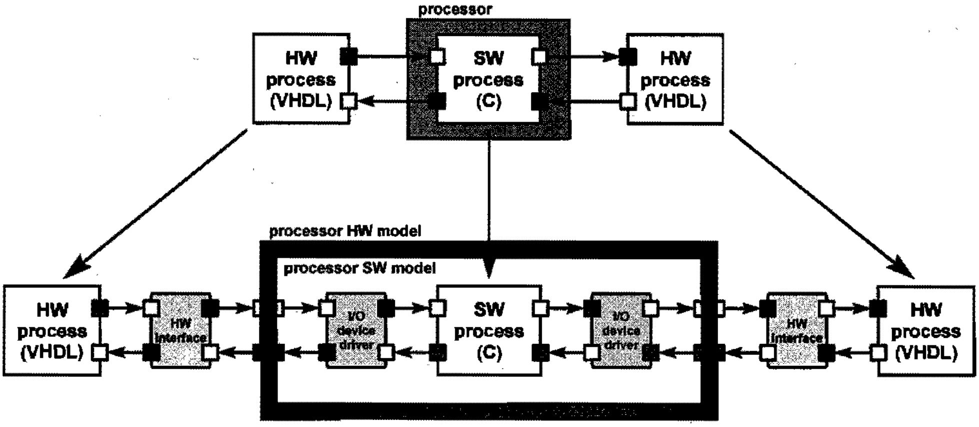


Figure 2.4 Interface synthesis in CoWare

present in the system description prior to communication synthesis. Co-simulation is required to detect these errors. However, co-simulation the system after communication synthesis is time-consuming.

2.6 Analysis & Design Methods

In software engineering, analysis & design methods have been developed to structure the software development process. In the analysis phase, the system requirements and the system specification are captured; in the design phase, the software implementation is constructed. The first generation of analysis & design methods appeared in the middle 1980's, by means of structured analysis and design (SAD) methods such as Ward & Mellor [WM86] and Hatley & Pirbhai [HP87]. These SAD methods are extensions to Yourdon's data-flow analysis [You89]. In the early 1990's, the second generation of analysis & design methods appeared by means of object-oriented analysis and design (OOAD) methods, such as Object Modeling Technique (OMT) [R$^+$91], Object-Oriented Software Engineering (OOSE) [J$^+$92], and Software/Hardware Engineering (SHE) [vdPV97].

Although the SAD and OOAD methods are intended for software development, they have also been proposed for developing hardware/software systems. SAD and OOAD methods provide suitable analysis techniques to capture the system requirements and system specification. However, transforming the system specification into a hardware/software architecture is hardly supported. Currently, SAD and OOAD methods focus more on selecting a hardware platform on which the software is executed (see also section 2.2.2) than on genuine co-design of heterogeneous hardware/software systems.

The SAD and OOAD methods have evolved in a top-down direction through the system design process. They emphasize requirements and specification capture, but they are still very weak at the subsequent design stages like architecture selection and synthesis. On the other hand, hardware/software co-design methods have evolved in the course of time in a bottom-up way. Initially, co-design methods focused on integrating design environments and tools for co-simulation and automated synthesis of hardware and software. As time progressed, the focus shifted from the lower abstraction levels (synthesis) to higher abstraction levels (specification, architecture selection). At the moment, co-design methods cover the entire system design process, from system specification to system implementation.

2.6.1 Structured Analysis & Design

Structured analysis & design (SAD) methods are very useful for specifying the functional behavior of complex systems, focusing on data, control, and functions. There is a variety of SAD methods, which are all closely related and adopt the same philosophy, such as Ward & Mellor [WM86], Hatley & Pirbhai [HP87], Edwards [Edw93], Calvez [Cal93], and Gomaa [Gom93].

In SAD methods, the system specification is a hierarchical model, capturing the system's functional behavior and its interactions with the external environment. Each hierarchical level consists of a number of processes ('transformations') that communicate by exchanging events and data. A transformation can be decomposed at the lower levels. Data transformations specify functions and operations on data; control transformations model finite-state machines and are mainly used to coordinate the data transformations. The system specification is graphically depicted in figures ('bubbles and arrows') and state transition diagrams, complemented by tables and natural language. Although SAD methods provide a natural and structured approach for system specification, they have some well-known shortcomings:

- SAD methods are informal. The language constructs and the graphical symbols used to specify a system, have no well-defined semantics and sometimes even no well-defined syntax. The system specification is not executable. This implies that simulation and formal verification techniques cannot be applied.

- Manual reviewing is the only way to validate and verify the system specification. Manual reviewing should be done using fundamental mode operation. This implies that for each process only one input at a time is allowed to change, and that no new inputs are allowed until all internal computations are completed and all outputs are generated. It is assumed that the internal computations are performed infinitely fast and take no time. Hence, all the parallel processes in the behavior model are executed synchronously.

  Fundamental mode implies that no inputs may occur simultaneously. This is not a severe restriction, because when the time scale is chosen small enough, two inputs will never occur at exact the same point of time. However, in digital systems all actions are related to a clock signal and inputs are evaluated on the clock edges. When two inputs occur in the interval between two clock edges, the inputs must be considered as simultaneous. In this case, the fundamental mode is no longer valid.

- Due to the absence of verification techniques, it is very difficult to keep the system specification consistent when making changes and applying transformations. There is no support the guarantee the equivalence of the models before and after transformation.

- In the design stage, the system specification must be transformed into a hardware/software architecture. This step is hardly supported in SAD methods.

2.6.2 Object-Oriented Analysis & Design

Object-oriented analysis & design (OOAD) methods combine the structured, systematic analysis approach from SAD with object-oriented analysis and specification techniques. There is a large variety of OOAD methods, of which Object Modeling Technique (OMT) [R+91] and Object-Oriented Software Engineering (OOSE) [J+92] are among the most popular. Most OOAD methods are intended for object-oriented software development, and offer no support for hardware design. Furthermore, OOAD methods have several shortcomings, like using inconsistent system views and using inappropriate modeling concepts and modeling primitives [vdPV97].

A recent development is the SHE (Software/Hardware Engineering) method, which aims at designing reactive hardware/software systems [vdPVS95, VvdPS96, vdPV97]. SHE combines concepts from SAD and OOAD methods. Like SAD and OOAD methods, SHE focuses on requirements capture and system specification, although work on transforming a system specification into a hardware/software architecture is on its way. SHE defines a framework for design activities combined with a formal description language called POOSL (Parallel Object-Oriented Specification Language). Detailed information on the SHE method and the POOSL language is provided in chapter 7.

2.7 Discussion

This chapter described the state-of-the-art on hardware/software co-design. We examined the improvements brought by hardware/software co-design methods over the traditional design methods, focusing on hardware/software verification issues. This section discusses our findings.

Traditional design methods versus co-design
The basic problems of traditional design methods are the lack of a well-defined design flow, the lack of well-defined system specifications at the behavioral and architectural level, the separated hardware and software design flows, the troublesome verification of the integrated hardware/software system, and the long design iteration loops. Hardware/software co-design methods offer improvements to deal with all of these problems.

The importance of experienced system designers
Traditional hardware/software design methods rely heavily upon experienced system designers for defining the hardware/software architecture of the system. Designers make implementation choices based upon their experience and knowledge.

Hardware/software co-design methods support the designer when defining the hardware/software architecture. The increasing system complexity and the expanding design space cause that designing hardware/software systems is becoming extremely difficult even for experienced and talented system designers. Tools for performance analysis and design space exploration assist the designer to select a suitable hardware/software architecture that satisfies all constraints. Performance analysis tools offer information to the designer like execution rates of processes and average usage of channels. Design space exploration tools generate values for metrics such as silicon area and software program size. However, an experienced designer is still required to evaluate the figures offered by performance analysis and design space exploration tools. The hardware/software partitioning in co-design of complex systems is performed by an experienced designer either manually or using an interactive approach. Hence, experienced system designers are required even in modern hardware/software co-design methods.

Hardware/software co-design flow

The design flows in hardware/software co-design methods differ due to different application domains and different target architectures. Usually, the co-design flows are fairly complex and incorporate many short iteration loops, but avoiding the long re-design iterations of the traditional design methods. Hardware/software co-design flows generally consist of six steps: system requirements capture, system specification, architecture exploration, architecture refinement, synthesis, and hardware/software integration. The focus is on designing at higher levels of abstraction while increasing design automation at lower levels through synthesis tools.

Hardware/software integration

Hardware/software integration, the corresponding testing and debugging activities, and the subsequent re-design iteration loops form the major bottleneck in traditional design methods. These problems account for 30–40% of the design time and costs in embedded systems [MCC96], and even 50–70% in distributed, real-time systems [TY95]. Typically 50% of ASIC prototypes cannot be integrated correctly due to errors in communication interfaces with other hardware and software components [Bou90, Sch93a]. Because the integration problems are discovered very late in the design process, it is usually too expensive to re-design the hardware. Integration problems are therefore generally solved in software. However, in systems with very complex software architectures like communication switches, it is more cost effective to re-design the hardware than modifying the software once it is stable. Nevertheless, often systems are shipped to the customer without the full functionality or meeting all performance requirements.

Hardware/software co-design methods allow to verify hardware/software interaction much earlier in the design process using co-simulation. Design errors can be detected and repaired before the actual system or prototype system is built. Hence, hardware/software integration in co-design is less troublesome. Another improvement brought by co-design is communication synthesis for automated generation of hardware/software communication interfaces. In particular the COSMOS and CoWare co-design methods emphasize communication synthesis, offering libraries and tools to achieve that the functional behavior of communication interfaces is correct-by-construction. However, communication synthesis affects the timing behavior of the system, which can introduce timing errors. Hence, verification of the timing behavior after communication synthesis is required.

Verification

Hardware/software co-design methods emphasize verification for checking correctness in every step of the co-design flow. The verification techniques can be classified into formal verification, (co-)simulation, emulation and testing.

Formal verification

Formal verification techniques for property proving and equivalence checking, allow 100% accuracy and coverage. However, formal verification techniques are restricted to models of moderate complexity, mainly due to state space explosion. Formal verification is not widely applied in industry yet, also because designers are usually not familiar with creating formal models of a system, with describing the system properties formally, and with applying formal verification techniques.

Co-simulation

All hardware/software co-design methods use simulation for verification in each phase of the co-design flow. Co-simulation is one of the key benefits of hardware/software co-design. Co-simulation allows verification of the hardware/software interaction early in the design process. Hence, integration problems can be detected and solved before the hardware is manufactured. This eliminates the long re-design iteration cycles in traditional design methods.

Co-simulation is achieved using hardware simulators, instruction-set simulators and virtual prototyping tools. The debugging support in co-simulation environments is excellent, offering high visibility into hardware and software components.

A major limitation of co-simulation is simulation speed, which is usually orders of magnitude slower than the real-time execution of the system. At high levels of abstraction, typically the behavioral level, high simulation speed can be obtained, but time and data resolution is rather low. At lower levels of abstraction, data and time resolution is higher, but simulation speed is much slower. Especially for data-oriented systems with hardware accelerators, the co-simulation times are unacceptable.

Emulation

Co-simulation can be speeded up by building a prototype implementation of the system, using emulation techniques like hardware models, in-circuit emulation, and hardware emulation. Testing a prototype offers higher performance than co-simulation, but requires higher costs and long set-up times. Furthermore, the debugging support is limited. In-circuit emulators provide excellent debugging support for software, but the debugging of the surrounding hardware is limited. The debugging support in hardware emulation is usually very limited.

Another disadvantage of emulation is that the prototype is built using a different technology (e.g. FPGAs) than the final system implementation. Hence, the final system implementation may still contain design errors, like timing errors, that are not present in the prototype. The inherent problem of prototyping is that the designer winds up testing and debugging the prototype instead of the final system implementation.

Testing and debugging

The prototype implementation or the final implementation of a hardware/software system is verified by means of testing. System testing is required to check the correctness of the interactions between the hardware and software components. When testing reveals the presence of errors in the hardware/software system, debugging is required to locate the errors and to find out the exact causes that effected the errors. Although formal verification and co-simulation will reveal many interaction problems early in the design flow, they cannot guarantee the complete absence of design errors during hardware/software integration. Consequently, testing is required in co-design to verify the hardware/software implementation.

Testing and debugging an integrated hardware/software system is a very troublesome process. When an error is detected, it is often not clear whether the error is caused by hardware or by software. Usually the software designers blame the hardware, while the hardware designers blame the software. Testing the system through its external interfaces is often not sufficient to test the interactions between components inside the system: many aspects of the internal operation of the system are not externally visible. Furthermore, it is very difficult or even impossible to achieve visibility into an individual hardware or software component. Visibility is achieved by using auxiliary instrumentation, like oscilloscopes, logic analyzers and measurement equipment. These instruments provide some visibility into hardware and software components. In-circuit emulators and embedded software monitors improve software visibility, but this has a major impact on the hardware/software system. Unfortunately, the current hardware/software co-design methods provide no support to improve the testing and debugging of hardware/software systems. Design-for-test and design-for-debug techniques for system-level testing and debugging of hardware/software systems are not considered in the current hardware/software co-design methods. The goal of this thesis is to fill this gap by introducing design-for-test and design-for-debug techniques in hardware/software co-design.

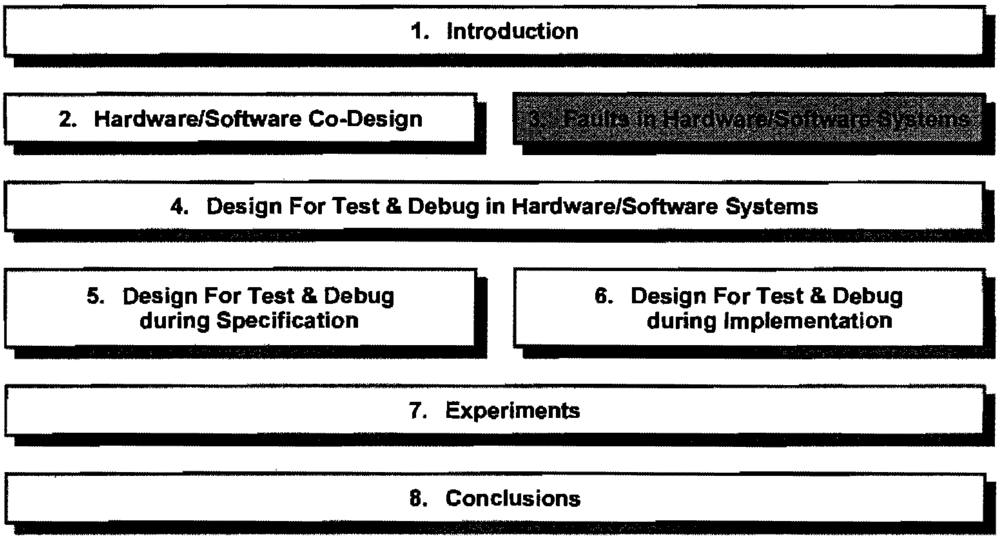
2.8 Summary

In this chapter we described the state-of-the-art on hardware/software co-design. We defined a co-design flow consisting of six steps: system requirements capture, system specification, architecture exploration, architecture refinement, synthesis, and hardware/software integration. Co-design methods strongly emphasize verification, using formal verification, (co-)simulation, and emulation. These verification techniques are used to check correctness in every design step, providing that design errors are uncovered early. However, we showed that testing the prototype implementation or the final hardware/software implementation of a system is still necessary to reveal design errors.

We argued that testing and debugging of hardware/software systems is very troublesome, which is mainly due to the limited visibility into the internal operation of a system. At present, auxiliary instruments such as oscilloscopes, logic analyzers and measurement equipment are used, but they do not provide adequate solutions. Unfortunately, the current co-design methods provide no support to improve the testing and debugging of hardware/software systems and they do not address design-for-test and design-for-debug. The main objective of this thesis therefore is to fill this gap by introducing design for test & debug in hardware/software co-design.

Chapter 3

Faults in Hardware/Software Systems



1. Introduction

2. Hardware/Software Co-Design

3. Faults in Hardware/Software Systems

4. Design For Test & Debug in Hardware/Software Systems

5. Design For Test & Debug during Specification

6. Design For Test & Debug during Implementation

7. Experiments

8. Conclusions

This chapter elaborates on faults in hardware/software systems. The goal of this chapter is to examine and classify faults in hardware/software systems that are typically encountered during hardware/software integration testing and system testing. System-level testing and debugging approaches should be capable to detect and diagnose these faults. This chapter also characterizes the dependability and the architecture of hardware/software systems, where the focus is on hardware/software interfaces.

3.1 Introduction

During hardware/software integration, **integration testing** is required to verify the correct cooperation of the hardware and software components that build up the system. We recommend an incremental, building-block approach to integration and integration testing, in which first the individual components are tested separately. When a component passes its test, it can be integrated into a larger aggregate of components. This aggregate is tested next, and subsequently integrated into an even larger aggregate. This procedure is repeated until all components and all aggregates have been tested and integrated. Finally, the complete hardware/software system is obtained.

Integration testing primarily focuses on testing the interfaces between the various hardware and software components in the system. The goal of integration testing is to verify whether a component or an aggregate of components correctly interacts with the other components or aggregates in the system.

When integration and integration testing is completed, the entire hardware/software system is available. Finally, **system testing** is required to verify the correctness of the system as a whole. The goal of system testing is to verify whether the system implementation conforms to the system specification. Integration testing concentrates on testing local communication interfaces in the system, while system testing concentrates on testing the overall system behavior constituted of all hardware and software components. When integration testing or system testing reveals the presence of an error in the system, debugging is required next to locate the error and to diagnose the exact cause that effected the error.

In order to thoroughly understand integration testing and system testing, a conscientious notion of system architecture, communication interfaces, and faults in hardware/software systems is required. Therefore, this chapter first elaborates on the architecture of hardware/software systems, focusing on communication interfaces. Next, dependability of hardware/software systems is examined. Finally, faults in hardware/software systems are explored in depth.

3.2 Hardware/Software System Architecture

The system architecture of a hardware/software system reflects the organizational structure of the system in terms of interconnected hardware and software components. During the system design process, the behavioral specification is transformed into a hardware/software system architecture. The processes and communication channels in the specification are mapped onto hardware and software components and communication mechanisms.

The system architecture of embedded systems typically consists of application software and application-specific hardware components. Execution of the application software requires a hardware nucleus and system software. Figure 3.1 shows a generic view on the system architecture of embedded hardware/software systems.

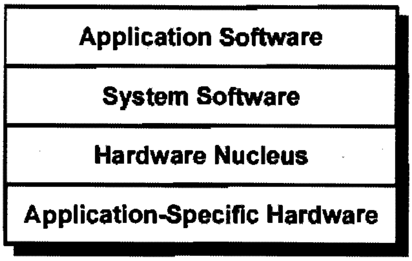


Figure 3.1 *HW/SW system architecture*

The hardware nucleus, shown in figure 3.2, consists of a processor (microprocessor, microcontroller, DSP, or ASIP) that executes the software, and memory that contains the binary software code. The processor fetches program code from memory, decodes the program instructions, and executes the instructions. The system software typically consists of an operating system, that constitutes the interface between the application software, the hardware nucleus, and the application-specific hardware components like ASICs and FPGAs.

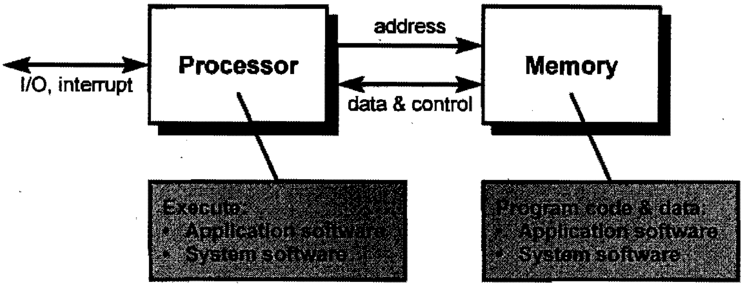


Figure 3.2 *Hardware nucleus*

Figure 3.2 shows a simple hardware nucleus consisting of a single processor and a single memory. In more complex hardware nuclei, there can be multiple processors and multiple memories for execution and storage of system software and application software.

3.2.1 Application Software

The software architecture defines how the software is decomposed into components, how these components are interconnected and how they communicate and interact with each other. Software architecture can be described from different viewpoints, like conceptual architecture, module interconnection architecture, execution architecture, and code architecture [SNH95].

The application software in embedded systems typically consists of a collection of cooperating software processes. The system software provides a scheduler to schedule the application software processes for execution on the processor at run-time. Process scheduling can be dynamic or static. In dynamic scheduling, there is no fixed execution order of processes: the scheduler determines which process is executed next by evaluating the states and priorities of processes. Static scheduling implies a fixed execution order of processes.

An alternative for cooperating processes is to code the application software into a single sequential program. A sequential program implies an a priori, fixed execution order of the processes: it has a single thread of control and requires no run-time scheduling. However, state space explosion due to parallel composition of processes, lack of modularity, and difficulties in meeting requirements on response time and performance, often make a sequential software implementation inadequate. The reactive, real-time behavior of embedded systems requires that the software reacts on events in the system environment within a certain amount of time. It is very difficult to capture this reactive, real-time behavior in a sequential program. Hence, the application software typically consists of cooperating software processes, scheduled at run-time by the system software.

3.2.2 System Software

The system software provides two primary functions to the application software: it constitutes a high-level interface to the hardware and handles parallelism in the application software.

The interface between the application software and the system software is often referred to as the Application Program Interface (API). The API defines the calling conventions for communication between the application software and the system software, and the services that the system software provides to the application software.

The system software incorporates I/O device drivers that provide an abstraction from the low-level I/O operations required for accessing I/O devices. The application software interacts with the hardware by calling these device drivers. The device drivers take care of the sequence of low-level reads, writes, and interrupts by using special I/O instructions or memory-mapped I/O. The interfaces between the processor and hardware devices can range from simple bus interfaces for protocol conversion to complex DMA controllers performing block transfers [SB92]. A device driver typically establishes a physical path to transfer basic information such as data and addresses between the processor and the hardware device, and controls the sequence of information transfers. I/O can be performed using polling or interrupt-driven techniques. Polling implies that the processor periodically checks whether the next I/O operation can be performed. The overhead of polling (busy-waiting) is avoided by interrupt-driven I/O.

Parallelism in application software consists of concurrent software processes that are executed on multiple processors (physical concurrency), or on a single processor (logical concurrency). Physical concurrency implies a multi-processor implementation, where each software process is executed on a separate processor. Logical concurrency is provided by the system software, which schedules the application software processes for sequential, interleaved execution on a single processor. The system software provides mechanisms for process scheduling, using preemptive or non-preemptive scheduling based on static or dynamic priorities, and mechanisms for communication and synchronization between processes.

The architectures of large operating systems, typically multi-user operating systems like Unix, incorporate concepts of hierarchical layers and abstraction [SS94b, Sta92, Tan95]. An operating system should separate policies from mechanisms. Mechanisms provide the basic primitives to perform a certain task, such as process scheduling. Policies refer to how the mechanisms are used. For instance, process scheduling policies like first-come first-served (FCFS) or priority-based scheduling, are implemented by using the scheduling mechanisms.

The current trend in operating systems is to evolve from monolithic kernel (macro-kernel) architectures to micro-kernel (or nano-kernel, pico-kernel, ...) operating systems, in which the kernel only provides the basic mechanisms. Policies are implemented in the layers that are built around the kernel. This architecture contributes to more flexible, modular, and efficient operating systems. It is interesting to notice that the same trend can be identified in microprocessor architectures, moving from CISC to RISC architectures. The instruction sets of RISC architectures provide basic, simple instructions only. Complex operations, as found in CISC architectures, are realized by a sequence of RISC instructions.

In traditional operating systems, each process has a private address space and a single thread of control. In multi-user operating systems, like Unix, the amount of process state information is large, which makes creating, maintaining, and context switching between processes expensive. 'Lightweight processes' or 'threads' have been proposed to improve performance in situations where creating, maintaining, and switching between processes occurs frequently [Tan95]. Each lightweight process has its own program counter, register file and stack, while all lightweight processes share the same address space and other state information. In multi-user operating systems, lightweight processes give a significant efficiency improvement. Some multi-user operating systems provide kernel support for both processes and threads (e.g. Mach [T$^+$87]), while other multi-user operating systems offer kernel support for processes only and use libraries to implement multiple threads per process (e.g. Unix).

In real-time operating systems, the amount of process state information is usually much smaller [BS91a, LHC93, CR95]: processes share the same address space and are not enclosed in separate, protected, virtual address spaces. Hence, real-time operating systems are already lightweight. Constraints on memory size, speed, and costs in embedded systems often make the overhead for a large operating system unacceptable. Often, ad hoc system software is written to handle I/O and resource scheduling in embedded systems. Real-time operating systems are typically small-kernel operating systems, providing only the basic services of process management, interprocess communication, and I/O drivers. In embedded systems, synchronization between processes is often provided implicitly by the scheduler in case of non-preemptive scheduling, and interprocess communication is efficiently supported by shared data.

Distributed, real-time systems comprise a multi-processor architecture in which each processing node is often specialized to perform a certain function. Instead of a single, multi-processor operating system, usually a set of heterogeneous, mono-processor operating systems is used so that each processing node has its own local operating system.

3.2.3 Hardware Nucleus

The hardware nucleus, as shown in figure 3.2, provides the processor on which the software is executed. The hardware nucleus is also referred to as the software interpreter [LA90, Eck93] or the hardware platform. The hardware nucleus provides the physical interface between hardware and software: the software is stored as binary code in the nucleus' memory, and executed by the nucleus' processor.

The hardware/software interface between the processor's hardware architecture and the software instructions executed on the processor is often referred to as the instruction-set architecture (ISA) [PH94]. The ISA is an abstract description of the processor architecture and defines the software model of the processor, including the processor's instruction set, addressing modes, data formats, the main functional components in the processor architecture (e.g. data ALUs, address ALUs, registers, stacks, caches, timers, interrupt control, and DMA control), the memory map, and I/O ports. The ISA provides a functional view on the external processor architecture at the instruction level. The internal processor architecture describes the physical, structural architecture of the processor. The internal processor architecture can be described at various levels of abstraction, such as the machine-cycle level, the clock-cycle level, and the discrete-event level [Anc86].

The external processor buses can be monitored for testing and debugging purposes. However, it is difficult to resolve the processor's internal state by monitoring the external buses. Architectural features such as pipelining (parallel fetch, decode, and execution of instructions), superscalar architectures (parallel execution of instructions, using techniques like out-of-order execution, branch prediction and speculative execution), caches (both for program instructions and data storage), and DMA make it very difficult to determine the processor's internal state.

3.2.4 Hardware Component Architecture

The hardware nucleus components and the application-specific hardware components can be described at various levels of abstraction. (The same abstraction levels are used in hardware synthesis, as described in section 2.3.5.1.)

- At the register-transfer level (RTL), the hardware components are described in terms of registers, combinational circuitry, low-level buses, and control circuits that implement finite state machines. The RTL models describe the internal architecture and control logic within a hardware component, independent of hardware implementation technology.
- At the logic level, the hardware components are described in terms of Boolean logic functions and simple memory elements such as flipflops. The logic-level model describes the functional behavior, and does not describe the exact implementation in logic gates or the hardware implementation technology.
- At the gate level, the function, timing, and structure of the hardware components are described in terms of the structural interconnection of Boolean logic blocks like NAND, NOR, NOT, AND, OR, and XOR gates. The gate-level model describes the actual implementation structure in terms of interconnected elements from a specific logic family library.
- At the switch level, the hardware components are described as networks of interconnected transistors. The transistors are modeled as simple voltage-controlled on-off switches.
- At the circuit level, the operation of the hardware components is described in terms of the voltage-current behaviors of resistors, capacitors, inductors, semi-conductor circuit elements and their interconnections.

3.2.5 Communication Interfaces

The communication between the application software, the system software, the hardware nucleus, and the application-specific hardware takes place at various communication interfaces. We identify nine classes of communication interfaces in the hardware/software system architecture, as shown in figure 3.3.

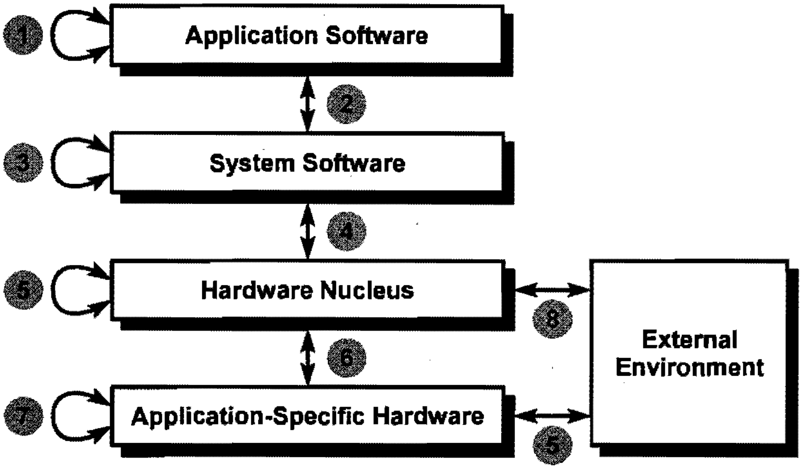


Figure 3.3 HW/SW communication interfaces

1. **Application software** communication interface
   The application software typically consists of a collection of cooperating processes. The application software communication interface represents intraprocess communication within an application software process. A process in the application software is a sequential process that consists of a number of methods or functions. Examples of intraprocess communication are accessing local process data or function calls within a process.

2. **Application Software - System Software** communication interface
   Interprocess communication between the application software processes is performed using communication mechanisms provided by the system software. Examples of interprocess communication are remote procedure calls, communication by shared data buffers, software interrupts, or message passing between processes.

   The application software processes are executed sequentially on a processor. The system software controls the process scheduling, using preemptive or non-preemptive scheduling policies, and static or dynamic scheduling schemes.

   Furthermore, the system software provides an abstract interface to the underlying hardware nucleus and the application-specific hardware. The system software provides I/O device drivers for accessing hardware features like timers and D/A converters, and for communication with application-specific hardware components.

   The application software - system software communication interface is usually called the Application Program Interface (API).

3. **System Software** communication interface
   The architecture of the system software consists of processes or modules, providing various functions like scheduling of application software processes and I/O device drivers. The system software communication interface represents the internal communication between the software components that build up the system software.

4. **System Software - Hardware Nucleus** communication interface
   The hardware nucleus operates as software interpreter: it provides one or more processors that execute both system software and application software, and memory in which the software code is stored.

   The system software - hardware nucleus communication interface also provides access to data structures in memory, handling of hardware interrupts, and access to the processor's external communication buses like the processor-memory bus and I/O buses.

5. **Hardware Nucleus** communication interface
   Internal communication in the hardware nucleus includes low-level hardware communication between the processor and peripherals like memories and controllers. The external processor buses (address, data, and control) are involved in this communication, such as reading/writing memories and receiving interrupt signals. Often the processor core, memories and peripherals such as timers, A/D-D/A converters and serial I/O ports are integrated on a single chip.

6. **Hardware Nucleus - Application-Specific Hardware** communication interface
   Buses and wires form the communication infrastructure for the communication between the hardware components in the hardware nucleus and the application-specific hardware components. Communication protocols like handshaking or fixed-time access are used to communicate over these buses and wires.

7. **Application-Specific Hardware** communication interface
   Likewise, the application-specific hardware components are interconnected by buses and wires over which communication takes place.

8. **Hardware Nucleus - External Environment** communication interface
   Objects in the external environment of the system communicate with the system by means of sensors and actuators. The hardware nucleus - external environment interface is typically concerned with communication between objects in the environment and processes in the application software.

9. **Application-Specific Hardware - External Environment** communication interface
   Additionally, objects in the external environment of the system communicate with the application-specific hardware components.

The communication interfaces 5, 6 and 7 deal with communication between hardware components. Buses and wires constitute the infrastructure between the hardware components. The electrical signals that are transported over these buses and wires constitute the lowest level of communication protocols. Low-level hardware communication is based on basic read/write operations on buses, such as accessing memory and I/O ports. Processor-memory buses are high-speed, synchronous buses, maximizing processor-memory bandwidth, using a fixed communication protocol relative to the clock. I/O buses connect the processor to various I/O devices. Backplane buses allow processors, memories, and I/O devices to coexist on a single bus, balancing the demands of processor-memory communication with I/O device-memory communication. Bus access is controlled using a master-slave configuration or using an arbitration scheme such as daisy-chain arbitration, centralized arbitration, distributed arbitration by self-selection, or distributed arbitration by collision detection [PH94].

The hardware/software system architecture in figure 3.3 shows a layered software architecture of application software and system software. This layered software architecture provides a logical, conceptual view on the software. However, in the physical view on software, the software is embedded in the hardware nucleus as binary code stored in memories. In this physical view, there is no notion of a logical software architecture. A similar notion holds for the application-specific hardware. In the physical view, the hardware architecture consists of interconnected hardware modules. The hardware modules themselves are conceptually equivalent to the application software processes: each hardware module performs a certain application function. Control logic and glue logic support parallel execution, synchronization and communication between the hardware modules. The control logic and glue logic between the hardware modules is conceptually equivalent to the system software. However, there are some differences in the concepts of concurrency and communication between software and hardware.

- Concurrency in hardware is physical concurrency, which means that concurrent events in hardware occur simultaneously and that concurrent hardware modules operate truly parallel in time. Concurrency in software can be either physical concurrency or logical concurrency. Physical software concurrency implies that software processes are executed on separate processors. Hence, physical software concurrency is achieved by physical concurrency in hardware. Logical software concurrency implies that the software processes are scheduled for sequential, interleaved execution on a single processor.

- Communication between hardware modules is performed over fixed, static interconnections like buses and wires. Although hardware communication switches provide more flexible connections between hardware modules, the connections are basically static. Software provides more dynamic communication mechanisms. Furthermore, software processes can be started, suspended, resumed and stopped, and even created and killed at run-time.

The communication mechanisms in hardware and software can be categorized into communication via message passing or shared data, synchronous or asynchronous communication, unidirectional or bidirectional communication, and point-to-point or broadcast communication.

Message passing/shared data

Communication between two processes implies a sending process, which outputs messages on a communication channel, and a receiving process, which inputs messages from the communication channel. Message passing indicates that a message can be received at most once by the receiving process. Message passing can be buffered or unbuffered.

- Buffered message passing implies that there is a buffer in the communication channel to store messages. The sending process writes messages into the buffer and the receiving process reads messages from the buffer. Although the buffer size may be theoretically unbounded, in practice a buffer has a finite size. The sending process may be blocked if the buffer is full, while the receiving process may be blocked if the buffer is empty. In the case of non-blocking communication, the messages from the sending process are lost when the buffer is full. A buffer is usually a FIFO (First-In First-Out) buffer, but there are various other strategies for reading and writing a buffer, for instance using priorities or LIFO (Last-In First-Out).
- Unbuffered message passing implies that there is no buffer in the communication channel. The sending process outputs a message on the channel, which is input by the receiving process. In rendezvous communication, the sending process is blocked if the receiver is not ready to accept data, and the receiving process is blocked if it is waiting for the sender to send data. Non-blocking, unbuffered message passing implies asynchronous communication, where the sending process outputs a message regardless if the receiving process is ready or not.

Communication by shared data implies communication using a shared memory, which is written by the sending process and read by the receiving process. The receiving process can read the same data multiple times from a shared memory (non-destructive read), while message passing implies that data can be read at most once (destructive read).

Synchronous/asynchronous

Synchronous communication implies that the sending and the receiving process are synchronized during communication, hence the send and receive operations occur at the same time. In asynchronous communication, the sending and receiving process are not synchronized. Note that the concept of synchronous/asynchronous communication is orthogonal to the concept of synchronous/asynchronous execution of concurrent processes.

Synchronous communication implies rendezvous communication. The sending and receiving processes have to wait until both processes are ready to interact. During the communication interaction, the processes are locked together. Rendezvous communication generally implies message passing between asynchronous processes at synchronization points.

Communication using buffers always implies asynchronous communication. Unbuffered communication is either synchronous communication, i.e. communication using rendezvous where both the sending and receiving process can be blocked, or asynchronous communication, i.e. communication where the sending process is never blocked and messages are lost if the receiving process is not ready.

In hardware, synchronous communication is implemented by handshaking protocols, where request, ready, and acknowledge signals are exchanged between the sending and the receiving process. Hardware communication on buses and wires using fixed time slots is asynchronous communication. In software, synchronous communication requires explicit synchronization to implement the rendezvous. Communication via buffers or shared memory in hardware or software implies asynchronous communication.

Unidirectional/bidirectional

Unidirectional communication implies that data is transferred in one way, from the sending process to the receiving process. Bidirectional communication implies that processes perform both send and receive operations.

Point-to-point/broadcast

Point-to-point communication implies communication between one sending process and one receiving process, which is one-to-one communication. Broadcast communication implies one sending process and multiple receiving processes, which is one-to-many communication. In addition, many-to-many communication or many-to-one communication is possible.

3.2.6 Taxonomies

The system architecture of a hardware/software system can be represented in a structural model that reflects the physical organization of the system implementation. Besides the structural view on a hardware/software system, also other views on the system are relevant. Various taxonomies have been proposed to define the various aspects of hardware/software systems, such as Ecker's design cube [EH92], Gajski's Y-chart [GK83], and Madisetti's taxonomy [Mad95]. However, these taxonomies do not thoroughly address hardware/software co-design aspects.

The RASSP program (Rapid-prototyping of Application Specific Signal Processors) [RASb], initiated by the U.S. Department of Defense, aims at developing new design and prototyping methods to deal with the increasing complexity, time-to-market pressures, and life-cycle costs of digital signal processing systems. Within RASSP, a multi-axis taxonomy has been developed to define terminology and model characteristics for hardware/software systems [RASa]. The RASSP taxonomy, shown in figure 3.4, identifies four orthogonal axes: the temporal, data, functional, and structural axis. A fifth axis has been added to describe the relation between hardware and software: this axis represents the level of software programmability, indicating the granularity of software instructions.

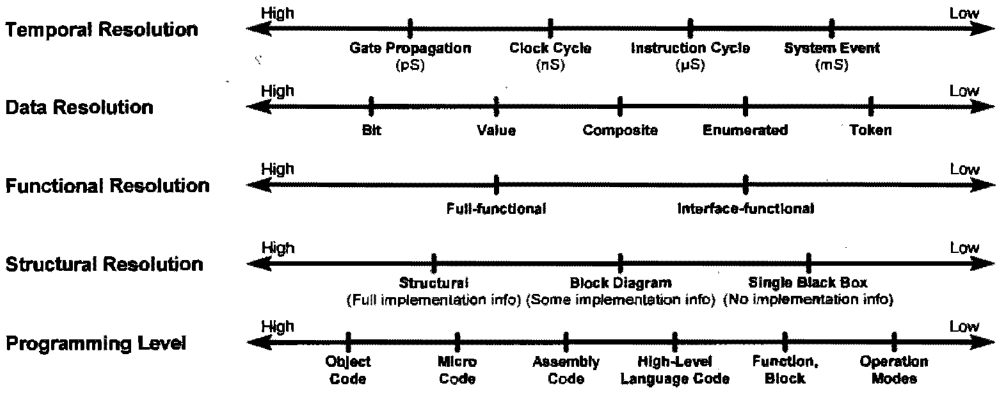


Figure 3.4 RASSP taxonomy

In the RASSP taxonomy, each axis addresses a system aspect at various levels of abstraction. Abstraction depends on the resolution of details: low resolution of details corresponds with a high level of abstraction, and high resolution of details corresponds with a low level of abstraction. Resolution is analogous to precision, as distinguished from accuracy.

3.3 Dependability

Dependability is defined as the trustworthiness of a hardware/software system such that reliance can justifiably be placed on the service that the system delivers [AL86, LA90, Lap92b, Lap92a]. The service delivered by the system is defined as the system behavior perceived by the user(s), where a user is another system (human or physical) which interacts with the system. Table 3.1 shows a taxonomy for dependability characteristics, introducing the concepts of dependability attributes, impairments and means.

attributes	reliability
	availability
	safety
	security
impairments	faults
	errors
	failures
means	fault avoidance
	fault removal
	fault tolerance

Table 3.1 Dependability characteristics (based on [Lap92b])

3.3.1 Dependability Attributes

Dependability is a generic term that implies reliability, availability, safety and security. Reliability refers to the continuity of service: reliability of a system is the probability that the system will behave conform to its specification throughout a period of time. Reliability can be expressed in terms of the Mean Time Between Failures (MTBF) and the Mean Time To Repair (MTTR). Availability refers to the readiness for usage. Availability can be expressed as MTBF/(MTBF+MTTR). Safety refers to the non-occurrence of catastrophic failures. Security refers to the prevention of unauthorized access to the system and/or unauthorized handling of information.

3.3.2 Dependability Impairments

An impairment is an anomaly in a hardware or software component that causes a component to deviate from its intended function. An impairment can be the cause or the effect of the system becoming undependable. Impairments can arise during all stages of the system life cycle, such as specification, design, development, manufacturing, assembly, installation, operational use, and maintenance.

As shown in table 3.1, impairments can be classified into failures, errors, and faults. The notions of fault, error and failure are used to indicate cause-effect relations and the evolution of an impairment in time. A fault is the root cause of the impairment, like a physical defect in hardware. A fault causes an error, affecting the service delivered by the system. A failure occurs when the user notices that the service delivered by the system no longer complies with the system specification. The delivered service may be functionally incorrect, or the timing of the service delivery may be incorrect. Hence, an error is the manifestation of a fault in the system, while a failure is the effect of an error on the delivered service.

A fault may be dormant or active. A dormant fault is not causing an error. For instance, a fault in a redundant or inactive hardware circuit, or a fault in a software module that is not being executed, will not cause an error. A dormant fault becomes an active fault when the component that contains the dormant fault, is executed. If the active fault causes a deviation of the delivered service, then an error occurs. For instance, a physical fault in a hardware circuit may cause that the function or timing of the circuit is incorrect, and hence the fault induces an error.

An actual example of a dormant software fault is the 'year 2000 problem'. In many software programs, dates are stored using only two decimal digits to represent the year. For instance, the year 1987 is stored as 87, and consequently the year 2000 is stored as 00. Software that uses this format will produce correct results for the years 1900 to 1999, but incorrect results after 1999. The software cannot differentiate 2000 from 1900, which causes that sorting and computations produce incorrect results. For instance, a person born on June 6, 1969 will be 30 years old on January 1, 2000, but the software will conclude that his age is −70.

An error may be latent or detected. A latent error is an error that has not been recognized (yet). A latent error may disappear before it is detected. An error often propagates, creating new errors. Hence, an error in a component may originate from an active fault in the same component, or the error may be due to propagation of an error from another component. Furthermore, errors can be independent errors or related errors. Independent errors are due to different faults, while related errors are due to a common fault. For instance, a fault in the power supply or the clock signal may cause multiple, related errors. Related errors usually cause common-mode failures.

The time between the fault occurrence and the first appearance of an error is called the fault latency. Multiple errors can originate from the same fault, and all errors may propagate through the system. The error latency is defined as the time between the first appearance of the error and the moment that the error is detected (i.e. causes a failure).

Examples of faults, errors and failures are:

- A software programmer can make a mistake when writing software code. This results in a (dormant) fault in the software, such as a faulty instruction or faulty data. When the software code is executed, the fault becomes active and produces an error. If the user notices that the software behaves erroneously, a failure occurs.
- An electromagnetic field may cause a hardware fault when electromagnetic interference affects the electrical charges on wires. The fault causes an error when the hardware circuit

produces incorrect results. Subsequently, a failure may occur when these erroneous results are used by other hardware circuits.

The electromagnetic field may affect the values on a memory input during a write operation. Subsequently, faulty information is stored in the memory which results in a latent error. The latent error is not detected until the erroneous memory locations are read.

- An operator that enters an inappropriate command on a computer keyboard, induces a fault. The faulty command will cause the computer to perform an undesired operation, resulting in an error. A failure occurs if the operator notices that the computer has performed an undesired operation.

Basically, any fault can be viewed as a permanent design fault. Electromagnetic interference can be considered as a design fault, because the designer did not provide adequate shielding. A fault caused by an operator typing an inappropriate command can also be considered as a design fault, because the designer should have provided fault tolerance against inappropriate user inputs. However, a designer is not capable to foresee all situations in the system's operational life that can cause faults.

The definitions of faults, errors, and failure are based on precise cause-effect relations. However, in practice the cause-effect relations can be very complex, which makes it difficult to determine whether an impairment is a fault, an error, or a failure.

Various viewpoints can be used to classify faults, such as fault origin and fault persistence.

3.3.2.1 Fault Origin

Fault origin indicates the root cause of the fault. The fault origin can reside either inside or outside the system boundary. Furthermore, the fault origin can be described by the phenomenological cause of the fault.

- Faults can be caused either within the system (internal faults) or outside the system boundary (external faults). External faults result from interference or from interaction of the system with its physical environment. Examples of external hardware faults are electromagnetic perturbations, radiation, extreme temperatures, or vibration. External software faults are mistakes made by the operator, which enters inappropriate commands or faulty data. Internal hardware faults result from physicochemical disorders such as threshold changes, short circuits and open circuits. Internal software faults are design faults.

- The phenomenological cause can be either a physical fault or a human-made fault. A physical fault is due to physical phenomena, and can be either an internal or an external fault. Human-made faults are due to human imperfections, like design faults or faults during operation and maintenance activities.

3.3.2.2 Fault Persistence

Faults can be either permanent faults or temporary faults, where a temporary fault is either a transient fault or an intermittent fault.

- Permanent faults are irreversible. Permanent hardware faults can occur due to improper manufacturing, damage, or wear out. A hardware component containing a permanent fault can be restored only by replacement or repair.

  Obviously, software cannot wear out or be physically damaged. Permanent software faults are design faults. Also design faults in hardware are permanent faults. Design faults can be removed only by redesign.

- A transient fault is a temporary, external fault originating from disturbances in the physical environment. Transient hardware faults are caused by disturbances such as power supply fluctuations, particle radiation, and interference by electromagnetic noise. These disturbances typically have a short duration. Afterwards, the affected hardware circuit usually returns to a normal operating state. Although a transient hardware fault usually causes no permanent damage, it may bring the system into an erroneous state.

- An intermittent fault is a temporary, internal fault resulting from the presence of rarely occurring combinations of conditions. An intermittent fault usually occurs periodically. Intermittent hardware faults are often due to marginal or unstable hardware. Examples are pattern sensitive faults in semiconductor memories and parametric faults that are sensitive to voltage or temperature fluctuations. Parametric faults can cause races, hazards, and changes in delay properties. Intermittent hardware faults are in fact permanent design faults, that are activated rarely, like circuits with minimal timing margins.

Although software faults are design faults and therefore permanent faults, they can appear as temporary faults. Temporary software faults typically occur during exceptional conditions and are often caused by non-determinism in the software. An example is preemptive scheduling of software processes. The number of different interleaved executions of software processes is usually too large for exhaustive verification or testing. During an exceptional condition, the processes typically will be interleaved in a completely different order than during normal system operation. Now software faults can be revealed that were dormant during normal system operation. Usually, temporary faults in hardware or in software cannot be reproduced in subsequent system runs, because the exact circumstances under which the fault occurred, cannot be reproduced. Therefore, the debugging of temporary faults is extremely difficult.

The term 'bug' is commonly used to indicate a software fault. In [Gra86], an original description of permanent and temporary software faults is presented on the analogy of quantum mechanics. Permanent software bugs are called 'Bohrbugs' and temporary software bugs are called 'Heisenbugs'. Gray writes: "Bohrbugs, like the Bohr atom, are solid, easily detected by standard techniques, and hence boring. But Heisenbugs may elude a bugcatcher for years of execution. Indeed, the bugcatcher may perturb the situation just enough to make the Heisenbug disappear. This is analogous to the Heisenberg uncertainty principle in physics."

3.3.3 Dependability Means

Dependability means are methods and techniques that provide dependability. Dependability means are concerned with providing the ability for the system to deliver its service conform the specification. Dependability means can be categorized by fault avoidance, fault removal, and fault tolerance. Fault avoidance is concerned with preventing the occurrence or introduction of faults; fault removal is concerned with reducing the number of faults present in the system; fault tolerance is concerned with providing correct operation of the system in spite of faults.

3.3.4 Fault Avoidance

Fault avoidance is concerned with selecting appropriate design methodologies and implementation technologies that reduce the probability of faults being introduced in the system. For instance, hardware/software co-design methods improve the design process for complex heterogeneous hardware/software systems. Systems designed using co-design methods will contain less design faults than systems designed using traditional design methods. The use of reliable implementation technologies, such as reliable hardware technology and re-use of hardware and software components, will reduce the probability of implementation faults.

The ultimate form of fault avoidance is correctness-by-construction during the design and implementation of hardware/software systems. Unfortunately, correctness-by-construction is still an utopia. Fault avoidance alone is therefore insufficient for designing and implementing faultless hardware/software systems.

3.3.5 Fault Removal

Fault removal is composed of verification, diagnosis and correction. Verification implies checking the correctness of the system, and is aimed at revealing faults. As indicated in section 2.4, verification techniques can be categorized into formal verification, (co-)simulation, and testing. When verification reveals the presence of an error, diagnosis (debugging) is required to find the exact cause ('the fault') that effected the error. Subsequently, correction is required to correct the fault.

3.3.6 Fault Tolerance

Fault-tolerant systems are capable of performing self-recovery. Fault tolerance aims at preventing that faults lead to system failures, which is achieved by incorporating redundancy in the system. When an error is detected in a system component, a redundant component will take over the functions of the faulty component. Fault-tolerant systems are typically applied for fail-safe, critical applications, such as spacecrafts and telephone exchanges.

Fault tolerance is constituted of four phases [Joh89]: detecting the occurrence of an error, determining how the error affected the system, recovering the system, and repairing the fault that caused the error.

The definition of a fault implies that a fault cannot be detected directly. A fault is manifested as an error in the system. Hence, the usual starting point for fault tolerance is the detection of an erroneous state. There are various techniques for error detection [Ben94a]:

- replication checks, using hardware redundancy or time redundancy;

- timing checks, using watchdog timers that expire if an expected response does not occur within a certain period of time;
- reversal checks, that compute the expected input from the actual output and compare the expected input with the actual input;
- coding checks, using error detecting/correcting codes such as parity bits, Hamming codes, cyclic redundancy checks (CRC), and checksums;
- reasonableness checks, using run-time checks such as software assertions;
- structural checks, performing checks on data structures such as double-linked lists;
- diagnostic checks, performing diagnostic tests by applying test stimuli and evaluating the responses.

Fault-tolerant systems incorporate static redundancy or dynamic redundancy. Static redundancy (or masking redundancy) is achieved by duplicate components. When an error is detected in a component, the redundant component takes over the operation of the faulty component. Dynamic redundancy is achieved by distributing the functions of a faulty component among the other components in the system. An example is a distributed system in which each processing node carries out some specific functions. When a particular node fails, its functions are relocated to the other processing nodes. Incorporating redundancy in the system will increase the size and complexity of the system, which can lead to a decrease in reliability. Therefore, incorporating redundancy should be performed with great care.

Fault tolerance provides error detection and subsequent error recovery to return the system into an error-free state. However, it is still required to analyze the fault that effected the error. The relation between faults and errors can be very complicated. Usually, various faults can lead to the same error. For instance, a parity error on a memory read may indicate a faulty memory chip, a faulty memory bus, a faulty memory power supply, a faulty parity check, etcetera. Permanent faults have to be repaired by means of replacing a hardware component or modifying a software component; temporary faults usually require no repair actions if they occur very seldom.

Since the 1950's, fault tolerance has been primarily concerned with physical faults in hardware. Fault-tolerance strategies however are not effective for dealing with design faults. The more complex the system, the more design faults are likely to be introduced (and to remain) in the system, both in hardware and in software. Design faults are unanticipated faults, with unanticipated effects on the system.

Hardware faults can affect the contents of memories, which may induce changes in software code. Hence, hardware faults may cause software errors.

Software faults are design faults and software redundancy cannot be achieved by replacing a software module with an exact copy. Examples of fault tolerance techniques for dealing with software faults are recovery blocks and N-version programming.

Design fault tolerance is usually achieved through two approaches: modular decomposition and design diversity. Modular decomposition implies factoring the system's functions into a set of components in such a way that the failure of one component will be confined. A failure in one component will not prevent the execution of the total system function, although it may cause the system to switch into a degraded mode (graceful degradation). Design diversity implies that multiple, redundant components are designed and implemented independently. Examples of fault-tolerant systems based on modular decomposition and design diversity, are:

- The NASA space shuttle employs fly-by-wire, which implies that any command from the astronauts to control the vehicle, is processed and verified by computers first, and next issued to the proper actuators [NAS88]. The avionics system of the space shuttle consists of over 300 electronic black boxes, offering dual or triple redundancy for every function. The black boxes are connected by serial data buses to a set of five general-purpose computers (IBM AP-101). These computers employ redundancy both in hardware and in software. Four of the five identical general-purpose computers are redundant. The computers run two completely independent coded versions of the flight control software.

- The Boeing 777 airplane employs a fly-by-wire flight control system [Yeh96]. The flight control system uses triple redundancy for all hardware resources: the computing system, the airplane electrical power systems, the hydraulic systems, and the communication paths.

- The Airbus A320, A330 and A340 airplanes also employ fly-by-wire flight control systems [BT93, Fav94]. The flight control systems consist of several redundant computers, running different software versions. The redundant computers are all functionally equivalent, but different hardware architectures were used to implement them. For instance, the A320 uses two types of computers, one based on the 68010 microprocessor and one based on the 80186 microprocessor. Both computer systems were designed and implemented by two different companies. Additionally, the fly-by-wire system is complemented by a mechanical back-up system.

In [LA90, Lee94], a layered model for hardware/software systems is proposed. Similar to the OSI model, each layer provides services to its upper layer. In figure 3.5, the *(N)*-layer receives service requests from the *(N+1)*-layer. The *(N)*-layer interprets the service requests and generates service requests to the *(N-1)*-layer. The *(N-1)*-layer will provide responses back to the *(N)*-layer, which in turn will respond back to the *(N+1)*-layer.

The layered system model is extended with fault tolerance by means of exceptions and exception handlers. If the *(N)*-layer receives an illegal service request from the *(N+1)*-layer, then the *(N)*-layer returns an interface exception to the *(N+1)*-layer. An interface exception is often caused by a design fault in the *(N+1)*-layer.

The *(N)*-layer generates an internal exception when its internal error detection mechanisms reveal the presence of an internal error. An internal exception activates the *(N)*-layer exception handler which provides fault tolerance services that may correct the error. The *(N)*-layer returns to normal service if the error is corrected. However, if the error cannot be corrected, a failure exception is generated to the *(N+1)*-layer. The failure exception indicates to the *(N+1)*-layer that the *(N)*-layer cannot provide correct services any more. The exception handler in the *(N+1)*-layer is called to deal with the failure exception. The differentiation between interface exceptions and failure exceptions allows the *(N+1)*-layer to discriminate a design fault in the *(N+1)*-layer from a fault in the *(N)*-layer.

The *(N)*-layer exception handler can receive a failure exception from the *(N-1)*-layer exception handler, for instance due to an unrecoverable error in the *(N-1)*-layer. The *(N)*-layer exception handler may be able to recover the error. However, if the *(N)*-layer cannot recover the error, then a failure exception is generated to the *(N+1)*-layer exception handler, etcetera.

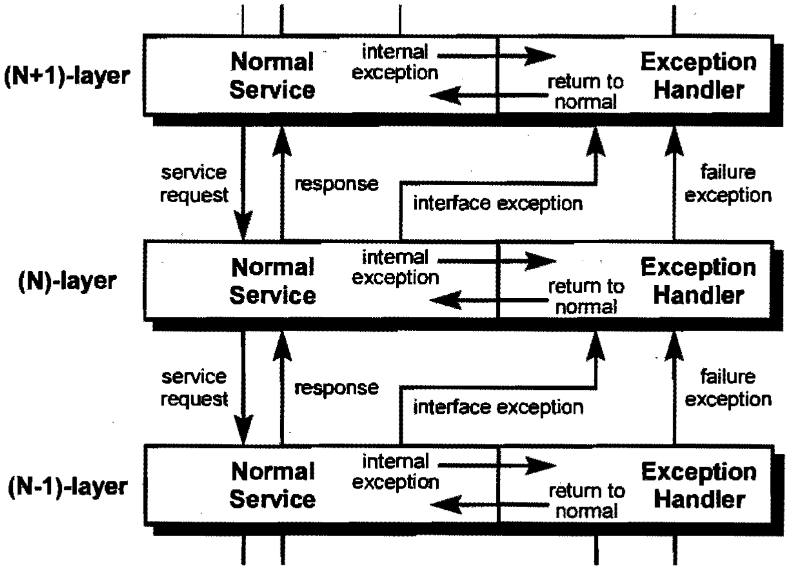


***Figure 3.5** Layered fault-tolerant system*

3.4 Case Studies on Faults in Hardware/Software Systems

In literature various studies are documented on faults in hardware/software systems. We examined case studies on fault-tolerant computer systems, multi-user mainframe computer systems, and distributed real-time systems.

3.4.1 Fault-Tolerant Computer Systems

In [Gra90, GR93] an analysis on failure statistics is presented for Tandem Computers systems between 1985 and 1990. Tandem systems are fault-tolerant computer systems, typically consisting of multiple processors and discs, a few hundred terminals (e.g. automated-teller machines) and communication lines. The study showed a continuing increase of hardware reliability caused by improvements in hardware technology and fault tolerance. The growth in system complexity was mainly in software. The increasing software complexity, together with the reliability improvement of hardware, caused that in 1990 most failures were due to software faults.

3.4.2 Multi-User Operating System

In [SC91] an analysis on failure statistics is presented for the IBM MVS Operating System between 1985 and 1989. The study describes software faults, the conditions in which the faults caused errors, and the subsequent failures.

Most errors where either concurrency-related errors or overlay errors. Examples of concurrency-related errors are deadlocks, unexpected sequences of events, undefined states, and synchronization errors. Overlay errors are related to management of pointers and memory allocation.

Most errors were due to boundary conditions, bug fixes, error recovery and timing. Examples of boundary conditions are unusual user input, high system workload with buffer and memory overflows, and changes in timing and performance characteristics due to the installation of new hardware. Frequently, new errors were introduced when fixing previous errors. Either the fix itself was incorrect, or the fix uncovered errors that were already present. Error recovery code and exception handlers are very difficult to test and debug, and therefore often contain latent errors. Timing errors are unanticipated sequences of events, like an interrupt at an inopportune moment. It is unfeasible to test all possible interleavings of events before the software is released. Errors in untested interleavings may be revealed after months or years of operational use. Although timing errors occur infrequently, they usually cause failures that have a high impact on the system.

A follow-up study in [SC92] reports an analysis on failures in the MVS operating system and in two large database management systems running on an IBM mainframe. Although the three software systems showed different error type distributions, most errors in all three software systems were undefined state errors. Undefined state errors arise when the system goes into an unspecified state due to unanticipated events. The system either misinterprets the events and moves into an erroneous state, or the system has no code to handle the events and ignores them.

3.4.3 Distributed, Real-Time Software

In [PS93] an analysis is presented on software faults in a large distributed, real-time system of AT&T. About 50% of the software faults were interfacing faults, i.e. faults in the communication interfaces between various components. Examples of interfacing faults are:

- faults in protocols for interprocess communication;
- incorrect exception handling or recovery from exceptions;
- race conditions due to incorrect coordination of shared data;
- violation of performance constraints such as resource access time and response time;
- incorrect resource allocation and deallocation;
- incorrect design and use of dynamic data structures;
- unexpected interactions between parts of the system;
- unexpected dependencies between parts of the system.

3.4.4 The Ariane 5 Failure

The first flight of the European space rocket Ariane 5 on June 4, 1996 ended in a failure [ESA96]. About 35 seconds after lift off, the launcher veered off its flight path and exploded at an altitude of 3,700 meter. The Ariane 5 carried four scientific satellites.

The root cause of the failure was in the Inertial Reference System (SRI) which measures the attitude of the Ariane 5 and its movements in space. The SRI is composed of a computer that calculates angles and velocities based on information from an inertial platform with laser gyros and accelerometers. The data from the SRI are transmitted over a data bus to the On-Board Computer (OBC) which executes the flight control program. The OBC controls the nozzles of the solid boosters and the main engine (see figure 3.6). The Ariane 5 contains two SRIs operating in parallel to provide fault tolerance. Both SRIs have identical hardware and software. One SRI is the active system and the other SRI is a stand-by back-up system. If the OBC detects a failure in the active SRI, it immediately switches to the stand-by SRI.

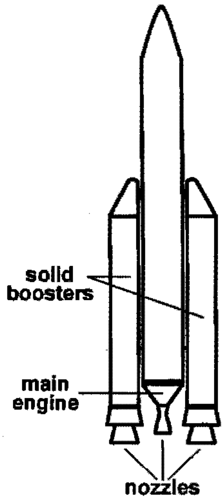


Figure 3.6 Ariane 5

The Ariane 5 performed a nominal lift off and flight until *H*0+36 seconds. (The point of time *H*0 indicates initiation of the launch sequence and ignition of the main engine and the solid boosters; lift off occurs a few seconds later.) At *H*0+36.7 seconds, the computer in the back-up SRI became inoperative and about 0.05 seconds later, the computer in the active SRI failed for the same reason. Loss of the mission was now inevitable, because both SRIs were inoperative and could no longer provide correct guidance and attitude information.

Upon its failure, the active SRI transmitted diagnostic information to the OBC. However, the OBC misinterpreted this diagnostic information as flight data. Due to this misinterpretation, the OBC concluded that a large attitude deviation had occurred. The OBC tried to correct this attitude deviation by commanding the booster nozzles and somewhat later the main engine nozzle to full deflection. These extreme nozzle deflections caused the launcher to veer abruptly to an angle of more than 20 degrees. Subsequently at about *H*0+39 seconds, high aerodynamic forces caused disintegration by rupture of the links between the solid boosters and the core state. The Ariane 5 initiated its self-destruction mechanism and exploded.

The failure of the SRI computers was due to a software exception that occurred during the data conversion from a 64-bit floating point to a 16-bit signed integer value. The floating point value was too large to be represented by a 16-bit signed integer, which resulted in an Operand Error. The Operand Error triggered execution of the exception handler. The exception handler transmitted diagnostic information via the data bus to the OBC, stored the failure context in an EEPROM, and shut down the SRI processor. The design of the exception handler was based on the assumption that exceptions are due to hardware failures. When the OBC receives a failure indication from the active SRI, it immediately switches to the back-up SRI. Obviously, this approach does not provide fault tolerance against design errors such as software errors, which cause that both SRIs fail simultaneously in the same way.

The large floating point value that caused the Operand Error, was the result of an internal alignment function that calculates the horizontal bias (BH) related to the horizontal velocity. In the Ariane 5, this function computes meaningful results only before lift off, and serves no purpose after lift off. Nevertheless, the function was operative after lift off. The reason for this is that the SRI software in the Ariane 5 was re-used from the Ariane 4. The Ariane 4 requires that the alignment function is operative for about 40 seconds after lift off.

When designing the SRI software, all operations that could give raise to an exception were analyzed. The analysis indicated that seven variables could lead to an Operand Error during conversion from floating point to integer representations. Four variables required protection and extra software code was added. The other three variables, including the variable BH, were either physically limited or there was a large safety margin. Therefore, it was decided to leaf these three variables unprotected. An extra reason for not protecting these variables was the requirement that the maximum workload of the SRI computer should not proceed 80%. However, this analysis was based solely on trajectory data of Ariane 4 and was not repeated for trajectory data of Ariane 5. Unfortunately, the early parts of the trajectories of Ariane 4 and Ariane 5 differ considerably. The horizontal velocity of Ariane 5 is five times larger than for Ariane 4, which caused the Operand Error during conversion of the variable BH. Also during simulation and system testing, no trajectory data of Ariane 5 were used to verify the correct operation of the SRI software. System testing focused only on verifying the interfaces, while not verifying the operation of the system as a whole.

3.5 Software Faults

3.5.1 Programming Languages

Currently, there is a large variety of software programming languages, e.g. C/C++, Ada, Java, Smalltalk, Pascal, Cobol, etcetera. The application domain usually imposes which programming language or family of programming languages to use. High-level and object-oriented programming languages allow rapid development of complex software. On the other hand, low-level programming languages usually provide more efficient software code regarding program size and execution time. For instance, in [P^{+}96a] is reported that assembly language is still widely applied in embedded systems for programming microcontrollers (75% assembly language, 25% C) and DSPs (90% assembly language, 10% C). Although selecting an appropriate programming language is very important, there are additional requirements for obtaining high-quality software, such as appropriate software development tools (e.g. compilers and debuggers), extensive software testing procedures, and good skills of programmers [Spu94].

The high-level programming language C and its object-oriented successor C++ are currently widely used. The software systems described in section 3.4.2 (IBM operating system and application software) and section 3.4.3 (AT&T real-time, distributed software) were implemented in the C/C++ language. C/C++ is the native language in computer systems running the Unix operating system. (In fact the Unix operating system itself is written in the C language.) The C/C++ language is also generally approved for writing 'technical' software, such as software in embedded, distributed or real-time systems. C/C++ language is the primary programming language in most hardware/software co-design projects, both in industry and in universities [MCC96].

There are many tools available for software development in C/C++, such as compilers, libraries and debuggers. Writing programs in C/C++ usually yields efficient software code regarded to program size and execution time. Nevertheless, the C/C++ language incorporates some well-known pitfalls. For instance, the programmer is responsible for allocating and de-allocating dynamic data structures in memory, for pointer management, and for providing mutual exclusive access to shared resources. Consequently, C/C++ is not the most suitable language for obtaining reliable software code. However, there are many techniques and tools for preventing and detecting errors in C/C++ programs [Spu94].

Another prevalent programming language is Ada [Ada95], particularly in safety-critical application domains. Ada was originally developed for the U.S. Department of Defense, targeted towards reliable, real-time software. Ada is currently being used in most U.S. defense systems, but also in other application domains like aerospace systems (e.g. Ariane 5), telecommunication systems, medical systems, power plants, and railroad control systems. Ada supports concurrent tasks. Furthermore, Ada provides powerful error detection mechanisms, both at compile-time and at run-time, and exception handling mechanisms.

3.5.2 Taxonomy of Software Faults

General taxonomies of software faults are provided in [Bei90, ANS94]. The majority of software faults in these taxonomies is related to faults in software components, i.e. faults that can be detected by testing a software component in isolation. Interfacing faults are due to incorrect communication interactions between multiple components. Interfacing faults can be detected during integration testing, where larger and larger aggregates of components are integrated and tested. Proper integration testing will reveal most interfacing faults. However, some interfacing faults are due to very complex interactions between components in application software, system software, and hardware. These complex interfacing faults are system-level faults, because they can be detected only during system testing, where the system is tested as a whole. Examples of interfacing faults and system-level faults are:

- A component tries to communicate with the wrong or a non-existing component;
- Incorrect parameters are passed during communication;
- Incorrect interrupt handling, for instance using an incorrect interrupt handler, or incorrect masking/unmasking of interrupts;
- Incorrect communication protocols for communication with device drivers, such as incorrect initialization, incorrect commands, or incorrect timing of commands;
- Incorrect communication of the application software with the operating system;

- Faults in interprocess communication, such as faults in locking and unlocking of processes or resources, and faults in setting or resetting of semaphores;
- Incorrect priorities for processes;
- Incorrect management of shared resources;
- Incorrect reentrance of software code;
- Violation of response times;
- Incorrect handling of exceptions;
- Faults due to high system load;
- Faults due to unanticipated sequences of events.

3.5.3 Faults in Memory Access

Many software faults in C/C++ programs, both in application software and system software, are related to corrupted memory and pointers [SC91, SC92, Dac93, Mag93, PS93, Spu94]. These software faults typically appear as temporary faults at run-time and often cannot be reproduced during debugging. Consequently, these faults are often not detected during testing, and they are typically revealed under exceptional conditions in the field.

The heap is the memory area where programs can allocate and free blocks of memory at run-time for storage of dynamic data. In the C/C++ language, the functions 'malloc' and 'free' provide access to the heap. Initially, the heap memory is unallocated and uninitialized. When a program requires memory from the heap, it has to allocate a memory block of the appropriate size using the 'malloc' function. Upon allocation, the program owns the memory block and may start using it. Obviously, a program is not allowed to read, write or free unallocated memory blocks. The contents of a newly allocated memory block is uninitialized, which implies that the memory block contains random data. Therefore, reading from an uninitialized memory block usually indicates a software fault. The program initializes the memory block by writing appropriate data in it. Subsequently, the program may read and write the memory block. If the memory block is no longer needed by the program, the memory block must be freed using the 'free' function. The memory block is now returned to the heap's pool of unallocated memory. Figure 3.7 depicts the sequences to allocate, write, read and free a block from the heap memory.

Software faults related to corrupted dynamic data occur frequently, because the programmer himself is responsible for implementing correct sequences to allocate, write, read, and free memory blocks. Common faults related to accessing dynamic data are reading from uninitialized memory, reading and writing beyond the bounds of a memory block (e.g. reading or writing beyond the bounds of an array, a stack, or a buffer), reading from or writing to freed memory, and freeing unallocated or non-heap memory.

Another class of software faults related to memory access are memory leaks. A memory leak is a memory block that cannot be accessed or freed anymore. When a memory block is allocated, a

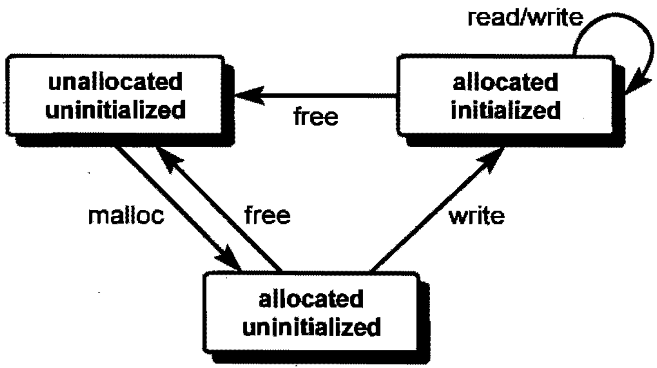


Figure 3.7 Accessing heap memory blocks

pointer is returned that refers to the first location in the memory block. A block can be freed by the 'free' function, which requires as input parameter a pointer to the first location in the memory block. A memory leak occurs when there are no more pointers that refer to the memory block, for instance when the last pointer referring to the memory block is cleared or changed to a location outside the memory block. A potential memory leak occurs if there are pointers referring to the memory block, but no pointers that refer to the first location in the memory block.

Memory leaks may cause fragmentation of the heap memory, which in turn may reduce performance. If the program section containing the memory allocation and the memory leak is executed repeatedly, the program may eventually run out of memory.

Fortunately, there are powerful tools for detecting and debugging faults like memory corruption and memory leaks. The Purify tool [Pur94] instruments C/C++ programs to detect run-time memory corruption faults and memory leaks. Purify inserts instructions into the object code at link-time. At run-time, these instructions monitor the program execution and check every read, write, allocation, and free of heap memory and stack memory. Purify maintains a table containing two bits for each byte of heap/stack memory used by the program. The first bit records whether the corresponding byte may be accessed, i.e. whether it has been allocated; the second bit records whether the byte has been initialized (see figure 3.7).

Because of the close relation between pointers and dynamic memory access, corrupted pointers are often due to software faults such as memory corruption and memory leaks. Common faults are corrupted or dangling pointers (referring to random locations) and null pointers (uninitialized pointers, referring to address zero). Purify detects reading or writing through null pointers and reading or writing the first page of memory. Besides reporting memory corruption errors and memory leaks, Purify also provides watchpoints that can be configured to monitor memory accesses (allocate, write, read or free) to a particular memory location.

Despite the benefits of Purify, the tester himself is responsible for executing the right paths through the software. Purify uncovers faults only in software paths that are actually executed, and hence it does not detect faults in parts of the software that are not executed. Exhaustive software testing, in which all control-flow and data-flow paths through the software are executed, is unfeasible even for very small programs. Consequently, exhaustive testing for all memory corruptions and memory leaks using Purify is unfeasible. Furthermore, there are some faults that Purify cannot reveal, such as a wild pointer, which is a faulty pointer that coincidentally refers to an allocated memory location.

3.5.4 Concurrency-Related Faults

Another significant class of software faults is related to concurrency [SC91, SC92, Sta92, PS93]. Concurrency-related faults are faults in the interactions between concurrent software processes. Concurrency-related software faults typically manifest themselves as temporary faults.

Concurrent software processes may be dependent or independent processes [Sta92]:

- Independent software processes are processes that do not communicate with each other and that are unaware of each other. Processes in different parts of the application software may be performing completely unrelated tasks, and hence they are independent processes. The operating system however has to schedule all independent processes and provide mutual exclusive access to shared resources like I/O devices. Although the processes are functionally independent, the timing of a process is affected by other processes. A process may have to wait until its request to access a shared resource is granted. Process scheduling implies interleaved execution of the processes on a single processor. Hence, the processor is a shared resource on which the processes are executed in turn.
- Indirectly dependent processes are processes that communicate indirectly with each other using shared memory. These processes are indirectly aware of each other: they do not know each other by name but they share access to the same data. As with independent processes, the timing of a process is affected by other processes due to scheduling. The processes are also functionally dependent, because they operate on the same data.
- Directly dependent processes are processes that communicate directly with each other using some form of message passing. Obviously, these processes are directly aware of each other and they are both functionally and time dependent.

We can conclude from the previous that four factors affect the interactions between concurrent software processes:

- Communication and synchronization protocols provide direct interaction between concurrent software processes. Synchronization protocols are used to transfer timing or control information between processes; communication protocols are used to transfer data and possibly timing or control information between processes.
- Shared data allows indirect interaction between concurrent software processes that access the shared data. Mutual exclusive access to shared data is required, which implies that only one process at a time may access the shared data. The timing of a process can be affected, because the process may have to wait until its request for accessing the shared data is granted.

- Process scheduling is required for interleaved execution of the processes on a single processor. The timing of a process is affected, because the process must wait on its turn for execution. Hence, the timing of a process depends on the scheduling of all other processes.

- Shared resources like I/O devices require mutual exclusive access. The timing of a process can be affected, because the process may have to wait until access to the shared resource is granted. Hence, the timing of a process depends on all other processes that also request mutual exclusive access to the shared resource.

Examples of concurrency-related faults occurring in the interactions between concurrent software processes are:

- **Faulty communication and synchronization protocols**
  Faults in the specification, design or implementation of communication and synchronization protocols may cause incorrect transfer of data or timing/control information between processes. An example is a faulty handshaking protocol, where the sequence of request, ready, and acknowledge signals is implemented incorrectly.

- **Faulty mutual exclusive access to shared data or shared resources**
  Incorrect mutual exclusive access may imply that multiple processes can access the shared resource simultaneously. Furthermore, requests from various processes should be scheduled in a fair way: each access request to the shared resource from each process should eventually be granted. If this condition is not met, some processes may never gain access and will be starved. The amount of waiting time for a process should be bounded, particularly for real-time processes that have to satisfy timing constraints.

- **Faulty process scheduling**
  Process scheduling implies suspending and resuming process execution and context switching. Processes should be scheduled in a fair way to avoid starvation. Furthermore, sufficient processing time should be assigned to each process to avoid violation of performance and timing constraints.

- **Deadlock**
  Deadlock implies circular waiting among multiple processes in such a way that no process can make any progress. For instance, if process A is waiting for a message from process B, while process B is waiting for a message from process A, neither process A nor process B can proceed. Deadlocks often occur due to incorrect scheduling of shared resources. For instance, if process A is granted access to shared resource P and subsequently requests access to shared resource Q, while process B is granted access to shared resource Q and subsequently requests access to shared resource P, neither process A nor process B can proceed.

- **Race conditions**
  Race conditions are faults that depend on the relative timing between processes. For instance, in figure 3.8 is shown how process *A* reads data, transforms the data, and writes data back into a shared memory. Between the read and write access of process *A*, process *B* writes data into the shared memory. Hence, process *A* overwrites the data written by process *B*.

This race condition can be avoided by changing the relative timing of the processes. The fault in figure 3.8 occurs because process *A* accesses the shared memory before process *B*. The fault may not occur when the processes are scheduled in a different way, i.e. when process *B* accesses the shared memory before process *A* as shown in figure 3.9. Different process scheduling may be due to different timing of external events or non-determinism in the scheduling algorithm.

In preemptive multitasking systems, a process can be preempted while it is updating a shared data structure, and the preempting process may access and modify the data. When the preempted process resumes its execution, it will find a modified, inconsistent data structure. A common error in assembly language is that a register or memory location is written, for instance by an interrupt service routine, without saving and restoring its original contents.

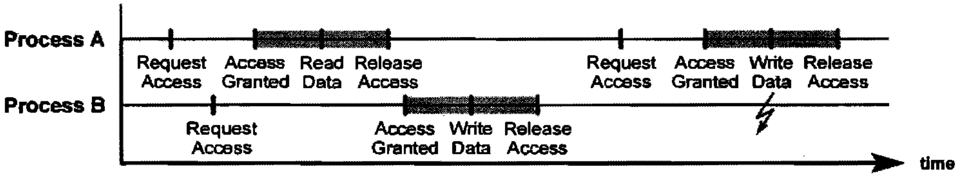


Figure 3.8 Race condition

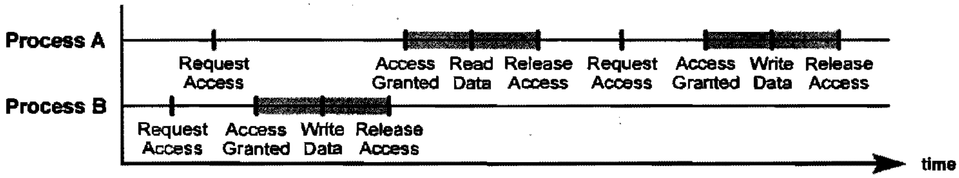


Figure 3.9 Avoiding race condition by different scheduling

- **Interrupt handling**
  The arrival of an interrupt causes that an interrupt handler is started, which interferes in the execution of other processes. Consequently, interrupts affects the timing of processes. Figure 3.10 shows an example of a concurrency-related software fault due to an interrupt. In the first situation, the interrupt interferes in the process scheduling but does not introduce a fault. In the second situation, the interrupt occurs slightly earlier and causes process *B* to miss its deadline.

It is extremely difficult to verify the absence of concurrency-related software faults by means of simulation or testing. The difficulty resides in the large number of possible sequences of events and the large number of possible interleavings of the different processes. For instance, a subtle fault in the software part providing mutual exclusive access to shared data, may remain dormant in most sequences of events or in most interleavings of processes. The fault may cause an error only in some exceptional sequences of events or interleaving of processes. It is usually impossible to simulate or test exhaustively all possible sequences of events or interleavings of processes.

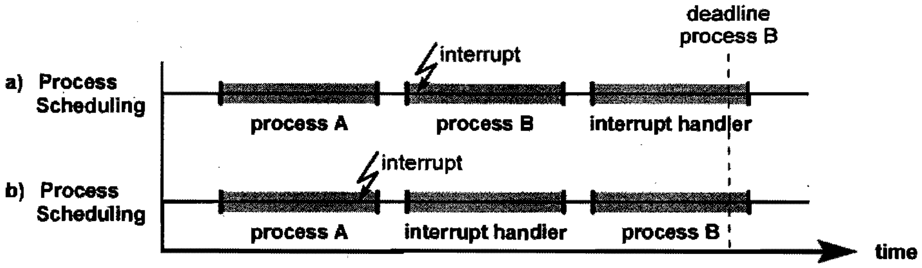


Figure 3.10 Violating timing constraints due to interrupt

Furthermore, it may be very difficult to provoke a particular interleaving, because one cannot control directly the order in which processes are scheduled. Process scheduling is often performed non-deterministically, which causes that the interleaving of processes may differ every time the software is executed. Consequently, it is generally impossible to reproduce an error, and therefore concurrency-related software faults often manifest themselves as temporary faults.

Real-time, reactive systems respond to events occurring in the system environment. The timing of events affects the behavior of the system. Some dormant, concurrency-related software faults may only be triggered in some exceptional conditions with extraordinary timing of events.

Concurrency-related and timing-dependent faults may remain dormant forever, because some sequences of events or interleavings of processes can never occur, even under exceptional conditions. However, when the hardware is changed, for instance when upgrading the system to a processor offering higher performance, or when porting application software to another operating system and hardware platform, the impossible sequences or interleavings may become possible. Consequently, real-time software that behaved correctly under all circumstances may suddenly fail.

3.6 Hardware Faults

Hardware faults are physical hardware defects that can be introduced during manufacturing or field operation. The manufacturing of hardware such as ICs or PCBs, is a complicated technological process. Although high yields can be obtained, they are never 100% and hence some hardware components will contain manufacturing defects. Common manufacturing defects on PCBs are interconnection defects such as opens and bridging faults. Typical manufacturing defects on VLSI ICs are interconnection defects such as opens and bridges and transistor defects such as short and open transistors.

Production testing is required to detect the physical hardware defects in ICs and PCBs that are introduced in the manufacturing process. Production testing can be considered as a filter, where the defect components are filtered out [Ben94b]. Production testing typically is structural testing, which means that the test stimuli focus on testing the physical structure instead of the functional behavior.

During field operation, hardware faults can be introduced due to wear out because hardware components have a finite life time. Hardware faults may also be due to physical effects, such as physical damage, deformations, high voltages, high temperatures, vibrations, humidity, radiation, or electromagnetic interference. Wear out of hardware always causes permanent hardware faults, while physical effects may cause either permanent or temporary faults.

We defined hardware faults as physical hardware defects, occurring during manufacturing or field operation. Hence, the phenomenological causes of hardware faults are physical faults, and not human-made design faults (see section 3.3.2.1 on fault origin).

Obviously, design faults during the specification or design of hardware also result in incorrect hardware. Complex VLSI hardware circuits are typically specified and designed using hardware description languages such as VHDL and Verilog. Describing hardware in a hardware description language is to a certain extent comparable to writing software code in a software programming language. Consequently, programming faults in hardware descriptions are likely to occur. As for software described in the previous section, many faults in hardware components can be detected by simulating or testing a hardware component in isolation. However, the increasing complexity of hardware components makes exhaustive simulation and testing of hardware components impossible. A notorious example of a hardware design fault that was only uncovered in the field, is the fault in the floating-point division algorithm of the Intel Pentium microprocessor [SB94, Pra95].

Physical faults and design faults in hardware are often correlated, particularly in the case of parametric hardware faults. Parametric faults are dormant faults that only become active during fluctuations in parameters such as temperature, voltage, current, humidity, etcetera. Parametric faults are due to marginal hardware design with small tolerance margins for parameter fluctuations: a slight variation in a parameter value causes a small alteration in hardware timing that is still sufficient to cause an error.

Interfacing faults are an important class of hardware faults. In [Bou90, Sch93a] is reported that although 90% of ASIC prototypes pass component testing, 50% fail during integration testing due to interfacing faults with other hardware and/or software components. Interfacing faults can be detected when simulating or testing larger aggregates of hardware and software components. Complex interfacing faults, which we will call system-level faults, can only be uncovered when simulating or testing the system as a whole, incorporating all hardware and software components. Interfacing faults and system-level faults are due to design faults in hardware and/or software. Faulty communication and synchronization protocols, faulty mutual exclusion, deadlocks, and race conditions are design faults that can occur in software as well as in hardware.

3.7 Fault Models

The ultimate goal of testing is to uncover all faults, both physical hardware faults and hardware/software design faults, with minimal effort. Testing implies offering test stimuli to the system and observing and evaluating the responses of the system. However, exhaustive testing of hardware/software systems is unfeasible, because of the extremely large number of test cases.

For instance, exhaustive testing of an ASIC (combinational logic) with 100 inputs requires 2^{100} test cases and would take 4×10^{14} years using a 100 MHz tester! Hence, in practice only a limited number of test cases can be applied. It is very important to select a set of test cases that will uncover as many faults as possible. In general, there are two approaches for deriving test cases:

- Test cases can be derived from a description of the hardware/software system, at any level of abstraction. However, this approach implies an implicit fault model and exhaustive testing: the test cases should detect all possible faults.
- Test cases can be derived using a specific, explicit fault model. Test cases are generated to detect the faults defined in the fault model. This approach produces a limited set of test cases. Furthermore, it is possible to measure the fault coverage. Using an explicit fault model provides that there is a finite number of faults that can be enumerated. The fault coverage is defined as the ratio of faults that can be uncovered using the generated test cases.

Explicit fault models are commonly used for deriving test cases, both in hardware and in software. Creating an appropriate explicit fault model is impeded by the large number and the complexity of physical hardware faults and hardware/software design faults. Therefore, explicit fault models often describe faults at a higher level of abstraction. Many faults on lower abstraction levels cause the same effect, and may therefore be modeled by the same fault at a higher level of abstraction. However, this approach implies a trade-off between accuracy and ease of modeling [B+92a].

3.7.1 Hardware Fault Models

Typical manufacturing defects in VLSI MOS technology are spots of extra or missing conducting or semiconducting material, spots of extra or missing insulating material, and parasitic devices [BA82, MA80, Mal87]. These defects typically result in faulty transistors (shorts and opens) and faulty interconnections (shorts, opens, bridges). There is no single fault model that can possibly cover all physical hardware defects. Instead, logical fault models are used to represent the effect of physical faults on the behavior of the modeled hardware circuit [ABF90]. Many different physical faults may be modeled by the same logical fault. Furthermore, logical faults may be applicable to many technologies like MOS or bipolar technologies. Finally, logical fault models may be used to model physical faults whose effects are not completely understood yet. The logical hardware fault models can be classified into structural fault models and functional fault models.

3.7.1.1 Structural Fault Models

Structural fault models are related to structural hardware models, i.e. models that describe the hardware structure in terms of hardware components and their interconnections. A simple structural model describes a hardware circuit as a network of logic gates and their interconnections. In general, a structural fault model assumes that the components are fault-free and that faults only reside in the interconnections between the components. Typical structural faults are shorts (stuck-at one), opens (stuck-at zero) and bridging faults.

The classical structural fault model for digital hardware circuits is the single stuck-at fault model. The stuck-at fault model is based on the assumption that physical faults in gates and interconnections can be modeled as stuck-at-zero faults or stuck-at-one faults on the input and output lines of logic gates. Although it has been demonstrated that the stuck-at fault model cannot accurately model faults in MOS technologies [BA82, Mal87, MA80], the stuck-at fault model is still widely used. Practical experiences show that test vectors derived using the stuck-at fault model, can detect most physical faults, achieving quality levels of about 200 ppm. (A quality level of 200 ppm indicates that 200 parts per million, i.e. 0.02%, are faulty.) The main advantage of the single stuck-at fault model is that the number of stuck-at faults in a circuit is small when compared to other fault models. Consequently, the single stuck-at fault model is the only structural fault model that allows computationally efficient test case generation.

The multiple stuck-at fault model is an extension of the single stuck-at fault model, in which multiple stuck-at faults are modeled simultaneously. However, usually the number of multiple faults is too large for practical use.

The quality level during production testing of CMOS circuits can be improved further up to approximately 10 ppm by using I_{ddq} testing. I_{ddq} is the IEEE symbol for the quiescent power supply current in MOS circuits. In CMOS circuits, the transistors are arranged in such a way that the I_{ddq} current is very small, typically in the order of micro or nano-ampères. Most defects cause an elevation in I_{ddq}, and hence defects can be detected by measuring I_{ddq}.

3.7.1.2 Functional Fault Models

Functional faults are related to functional hardware models, i.e. models that describe the functional behavior rather than the hardware structure. Functional hardware models are independent of the hardware implementation technology. Examples of functional faults are changes in Karnaugh diagrams or truth tables, and design faults in RTL descriptions. Functional fault models can model both physical hardware faults and design faults.

Functional fault models aim at reducing the complexity of test case generation by modeling faults at higher levels of abstraction. Functional faults can represent the effect of multiple physical faults. Inevitably, this is only possible at the expense of accuracy.

A functional fault model for microprocessors has been proposed in [TA80]. A functional model of a microprocessor is created by capturing the register architecture and the instruction set into a graph. Every user-accessible register is represented by a node in the graph. Two additional nodes, IN and OUT, denote the connections between the microprocessor and the external world. Directed edges between nodes indicate that a particular instruction transfers information from one node to the other node. This functional fault model incorporates faults in register decoding, instruction decoding, instruction sequencing, data storage, data transfer, and data manipulation.

The challenge in functional fault models is to develop accurate models that represent realistic physical faults. Often, heuristic or ad hoc methods are used for functional testing, in which there is no well-defined fault model: functional testing simply attempts to exercise the fault-free behavior of each system function. The major problem of heuristic methods is that the quality and fault coverage is unknown. Experience shows that functional testing of hardware typically detects 50 to 70 percent of the physical faults that are revealed by structural testing [ABF90].

The system specification is usually organized in a hierarchical manner. At the highest level, the system is composed of several components. Each component is described into more detail at the lower levels. A hierarchical, functional fault model may be established corresponding to the hierarchical system specification. For instance, faults in the interconnections between components can be modeled at each level of abstraction. Each component is decomposed into subcomponents at lower levels, and hence faults in the interconnections between these subcomponents model internal faults in the component.

An explicit, hierarchical, functional fault model is proposed in [CCP93b, CCMP94]. A system is modeled as a set of parallel processes that communicate over channels. The processes are described in process algebra. The fault model assumes that the processes themselves are fault free, and that faults reside only in the communication channels. A fault in a channel is modeled by introducing a new process that models the effect of the fault. For instance, in figure 3.11 a system is modeled consisting of the processes *A* and *B* that communicate over a channel. A fault in the channel is modeled by introducing an extra process *Fault*.

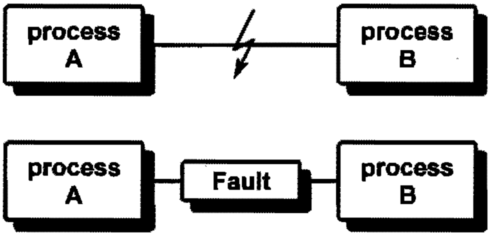


***Figure 3.11** Functional fault model ([CCP93b])*

The process *Fault* can model various faults, for instance:

- The process *Fault* may model faults in a single channel (shown in figure 3.11), such as loss of a message on the channel or a spurious message. In more complex cases, the process *Fault* can model faults in multiple channels, such as crosstalk between two channels.
- The process *Fault* may model a permanent fault, a transient fault, or an intermittent fault.
- The process *Fault* may model memoryless faults, whose behavior does not dependent on the past sequence of messages on the channel, or triggered faults that only occur after a given sequence of messages.

Test pattern generation is performed by checking weak bisimulation equivalence between the original, fault-free description and the description including the *Fault* process.[1] Although this approach provides a general functional fault model, there are severe restrictions that limit practical use. Checking bisimulation equivalence between process algebra descriptions is only possible for rather small systems, due to state space explosion caused by the parallel composition of processes. Furthermore, the fault universe is very large, because the process *Fault* can model any fault effect. It is unfeasible to model and bisimulate all possible functional faults.

[1] We will show in chapter 5 (page 136) that the approach in [CCP93b] is incorrect. Inserting a *Fault* process in general causes that the two descriptions are not weak bisimulation equivalent, regardless of the behavior of the *Fault* process.

3.7.2 Software Fault Models

There is a very large number of different types of software faults, which makes it very hard to define explicit fault models for software. Therefore, implicit fault models are used which typically assume that faults reside in control-flow or data-flow paths. Path testing is a structural testing technique, based on exercising the control-flow and data-flow paths in the software code. Unfortunately, the number of paths increases rapidly with increasing size of the software code. This causes that path testing is only applicable for testing small software components or aggregates of components. Nonetheless, path testing techniques can reveal approximately 65% of all software faults during component testing [Bei90].

The fault assumption in path testing is that faults reside in control-flow paths and data-flow paths. During software testing, a fault will cause that the traversed path differs from the intended path. Control-flow paths describe the control structure of the software, typically in terms of conditional or unconditional branches, loops, if-then-else constructs, etcetera. Data-flow paths describe how data objects in the software are defined and used in predicates and computations.

There are various path testing techniques based on control-flow and data-flow paths [Bei90]:

- Statement testing (P_1) is a form of testing control-flow paths in which every statement in the software code is executed at least once. Statement testing is the weakest form of path testing.
- Branch testing (P_2) is another form of testing control-flow paths in which every branch alternative in the software code is executed at least once. Branch testing is a stronger form of path testing. In fact, branch testing includes statement testing.
- In complete-path testing (P_∞), all possible control-flow and data-flow paths through the software code are executed. Complete-path testing, which implies 100% coverage of all control-flow and data-flow paths, is generally impossible to achieve.

The notation P_1, P_2, ..., P_∞ indicates that there is an infinite number of path-testing strategies stronger than branch testing and weaker than complete-path testing. A 100% coverage of branch testing or statement testing is a minimal requirement in software component testing. Additional paths should cover extreme cases for loops and nested loops, such as executing a loop zero times, once, twice, one less than the maximum number of times, and the maximum number of times.

The gap between complete-path testing and branch testing can be filled by testing data-flow paths. Data-flow testing is based on selecting control-flow paths through the software code in which sequences of operations on data objects are performed. Operations on data objects can be classified into:

- Define (d), create, or initialize a data object;
- Kill (k), undefine, or release a data object;
- Use (u) a data object either in a computation (c) or a predicate (p).

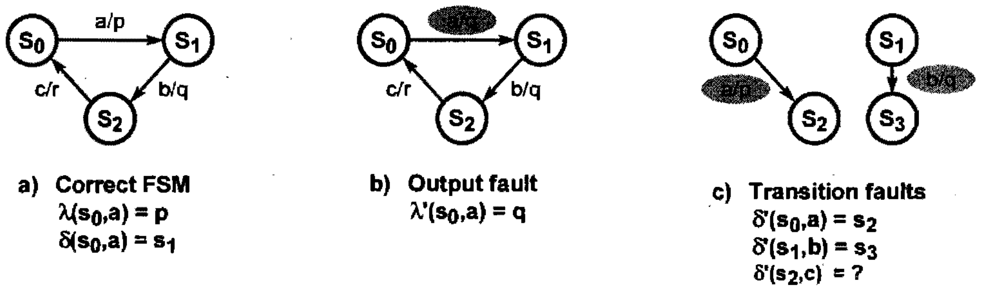
There are various data-flow path testing strategies. They differ in the sequences of operations that are performed on data objects on the traversed data-flow paths. Common data-flow path testing strategies are testing all-*du* paths, all-*u* paths, all-*p*/some-*c* paths, all-*p* paths, all-*c*/some-*p* paths, all-*c* paths, and all-*d* paths [Bei90]. Formal proofs of the relations between these data-flow testing strategies and control-flow testing strategies are provided in [RW85, FW88].

3.7.3 FSM-Based Fault Models

Fault models based on Finite-State Machine (FSM) descriptions have been extensively used in conformance testing of communication protocols [Hol91, Sar93]. The FSM model is an excellent technique for specifying communication protocols. A FSM is a 6-tuple $\langle S, I, O, s_0, \delta, \lambda \rangle$, where:

- S is the finite set of states;
- I is the finite set of input symbols;
- O is the finite set of output symbols;
- $s_0 \in S$ is the initial state;
- δ is the state transition function, $\delta : S \times I \rightarrow S$;
- λ is the output function, $\lambda : S \times I \rightarrow O$.

Fault models in FSM descriptions typically model output faults and state transition faults, as shown in figure 3.12. An output fault indicates that on a state transition, a faulty output is produced that differs from the expected output as specified in the output function. A state transition fault indicates that either a state transition is missing, or that a state transition transfers to a faulty new state that differs from the expected new state as specified in the transition function. The faulty new state can be an element of the state set S, or the state can be an additional state that is not an element of the state set S.



a) **Correct FSM**
$\lambda(s_0,a) = p$
$\delta(s_0,a) = s_1$

b) **Output fault**
$\lambda'(s_0,a) = q$

c) **Transition faults**
$\delta'(s_0,a) = s_2$
$\delta'(s_1,b) = s_3$
$\delta'(s_2,c) = ?$

Figure 3.12 FSM output faults and state transition faults

A number of methods has been proposed in literature for generating test suites from a FSM description to detect output faults and transition faults. The test methods are based on black-box testing: input sequences are applied to the FSM, and the output sequences are observed and evaluated. Common methods for test suite generation from FSM descriptions are the unique-input-output (UIO) method [SD88], the transition tour method [NT81], the distinguishing sequence method [Gön70], the UIOv method [VCI90], the W-method [Cho78] and the Wp-method [F$^+$91]. These methods are computationally intensive, and are restricted to FSMs with moderate numbers of states and state transitions. Furthermore, these methods detect output faults and transition faults, assuming that there are no state transitions to additional states that are not elements of the state set S. The length of the test suite and the costs for test suite generation generally increase exponentially with the number of additional states.

3.8 Discussion

In this chapter we explored in depth the architecture, communication interfaces, dependability aspects, and faults in hardware/software systems. The exploration of these topics yields a thorough understanding of hardware/software systems, which is a prerequisite when considering integration testing and system testing. We summarize and discuss our findings in this section.

Hardware/software architecture

We introduced a general model for hardware/software architecture, consisting of application software, system software, a hardware nucleus, and application-specific hardware. The system software is composed of an operating system, that constitutes the interface between the application software and the hardware. The system software provides a high-level interface for the application software to the hardware. The application software typically consists of concurrent, cooperating software processes, scheduled at run-time by the system software. The hardware nucleus provides one or more processors for executing the application software and the system software, and memory for storing the binary software code. The application-specific hardware comprises hardware components such as ASICs and FPGAs.

Communication interfaces

The communication between the application software, the system software, the hardware nucleus, and the application-specific hardware takes place at various communication interfaces. We identified nine communication interfaces in hardware/software systems. Four interfaces are concerned with internal communications inside the application software, the system software, the hardware nucleus, and the application-specific hardware. Two interfaces deal with communication between the system and the external environment. The remaining three interfaces are concerned with communication between the application software and the system software (API), communication between the system software and the hardware nucleus, and communication between the hardware nucleus and the application-specific hardware. We argued that integration testing and system testing should primarily focus on verifying these communication interfaces.

Dependability of hardware/software systems

We provided an overview of dependability aspects in hardware/software systems. Dependability is a generic term implying reliability, availability, safety and security. Dependability in hardware/software systems is achieved by fault avoidance, fault removal, and fault tolerance.

- Fault avoidance is concerned with preventing the occurrence or introduction of faults. Fault avoidance is accomplished by selecting appropriate design methodologies and reliable implementation technologies.

- Fault removal is concerned with reducing the number of faults present in the system. Fault removal is accomplished by verification activities such as verification, (co-)simulation, and testing, which all aim at checking the correctness of the system and uncovering faults. When verification reveals the presence of an error, debugging is required to diagnose the fault that effected the error.

- Fault tolerance is concerned with providing correct operation of the system in spite of faults. Traditional fault-tolerance strategies typically incorporate redundancy in hardware or in time. These traditional fault-tolerance strategies are effective for dealing with physical hardware faults, but not for hardware/software design faults, as became evident once again by the Ariane 5 failure. Fault tolerance against design faults should be provided by modular decomposition and design diversity.

Fault, error, and failure

The notion of fault, error and failure is used to indicate cause-effect relations and to indicate how an impairment evolves in time. A fault is the root cause of an impairment in a system. A fault may affect the system behavior, resulting in an error. A failure occurs when the user notices that the system behavior no longer complies with the system specification. The system behavior may be functionally incorrect or the timing may be impaired. A fault is either dormant or active, and an error is either latent or detected.

The fault origin can reside either inside the system (internal fault) or outside the system (external fault). The phenomenological cause of a fault is either a physical phenomenon (physical fault) or human imperfection (human-made fault). Furthermore, a fault is either a permanent fault or a temporary fault.

Permanent faults are irreversible, such as permanent hardware faults or hardware/software design faults. A temporary hardware fault is either a transient fault, which is an external fault originating from disturbances in the physical environment, or an intermittent fault, which is an internal fault resulting from the presence of rarely occurring combinations of conditions. Although software faults are design faults and therefore permanent faults, they can be experienced as temporary faults ('Heisenbugs').

Faults in hardware/software systems

We argued that most design faults in hardware and software can be detected by testing hardware and software components in isolation. The remaining design faults are interfacing faults, that are due to incorrect communication interactions between multiple components. Interfacing faults can be detected during integration testing, where larger and larger aggregates of components are integrated and tested. Proper integration testing will reveal most interfacing faults. However, some interfacing faults are due to very complex interactions between components in application software, system software, and hardware. These complex interfacing faults are called system-level faults, because they can be detected only during system testing, where the system is tested as a whole.

Software interfacing faults

We showed that many software interfacing faults are related to concurrency and memory access. These software faults typically appear as temporary faults at run-time and often cannot be reproduced during debugging. Consequently, these faults are often not detected during testing, and they are typically revealed under exceptional conditions in the field.

Software faults related to memory access are faults in dynamic memory management, pointer management, and memory leaks. There are powerful tools for detecting and debugging these kinds of faults at run-time by instrumenting the software code. However, these tools reveal faults only in software paths that are actually executed. In general, exhaustive software testing in which all possible paths are traversed, is impossible to achieve.

Concurrency-related software faults are faults in the interactions between concurrent software processes. We demonstrated that concurrent software processes may be independent processes (processes that do not communicate with each other), indirectly dependent processes (processes that communicate indirectly with each other using shared memory), or directly dependent processes (processes that communicate directly with each other using some form of message passing). The interactions between concurrent software processes is affected by communication and synchronization protocols, shared data, process scheduling, and shared hardware resources. Typical concurrency-related software faults are faulty communication and synchronization protocols, faulty mutual exclusive access to shared data or shared resources, faulty process scheduling, deadlocks, race conditions, and faulty interrupt handling.

We argued that it is extremely difficult to show the absence of concurrency-related software faults by means of testing. Exhaustive testing is unfeasible due to the large number of possible sequences of events and interleavings of processes. Furthermore, it may be very difficult to provoke a particular interleaving, because one cannot control directly the non-determinism in process scheduling. The timing and sequences of events affects the behavior of the system. Some software faults may only be triggered in some exceptional conditions with extraordinary timing of events. Furthermore, timing dependencies and non-determinism make it usually impossible to reproduce an error during debugging.

Hardware interfacing faults

Hardware faults are physical hardware defects that can be introduced during manufacturing or field operation. An important class of hardware design faults are interfacing faults. In [Bou90, Sch93a] is reported that although 90% of ASIC prototypes pass component testing, 50% fail during integration testing due to interfacing faults with other hardware and/or software components. Examples of hardware interfacing faults are faulty communication and synchronization protocols, faulty mutual exclusion, race conditions, and deadlocks.

Fault models

The ultimate goal of testing is to uncover all faults, both physical hardware faults and hardware/software design faults, with minimal effort. Testing implies offering test stimuli to the system and observing and evaluating the responses of the system. However, exhaustive testing of hardware/software systems is generally unfeasible, because of the extremely large number of test cases. Test cases can be derived using specific, explicit fault models. Fault models are widely used in hardware testing (stuck-at fault model and functional fault models), software testing (faults in control-flow paths and data-flow paths), and conformance testing of communication protocols (output faults and state transition faults in FSM models). Fault models are very effective for deriving test cases in component testing. However, we showed that the current fault models fall short in integration testing and system testing. There are no effective, logical fault models yet for modeling interfacing faults and system-level faults. The large number of interfacing faults and system-level faults prohibits computationally efficient test case generation.

Integration testing and system testing
The most important conclusion of this chapter is that integration testing and system testing should primarily focus on testing the interfaces between hardware and software components in the system. Integration testing aims at detecting local interfacing faults between a limited number of hardware and software components. System testing aims at revealing system-level faults, which are faults that are due to complex interactions between many hardware and software components. Integration testing and system testing are both essential parts for checking that the system implementation behaves conform the system specification. The Ariane 5 failure clearly demonstrated that both integration testing and system testing are required, and that selecting appropriate test cases is a prerequisite.

The goal of this thesis is to develop design-for-test and design-for-debug techniques that support and improve integration testing, system testing and debugging. In the previous chapter, we already showed the necessity for design-for-test and design-for-debug. In this chapter, we showed that system-level testing and debugging is concerned primarily with interfacing faults. Hence, design-for-test and design-for-debug should provide means to deal with interfacing faults.

3.9 Summary

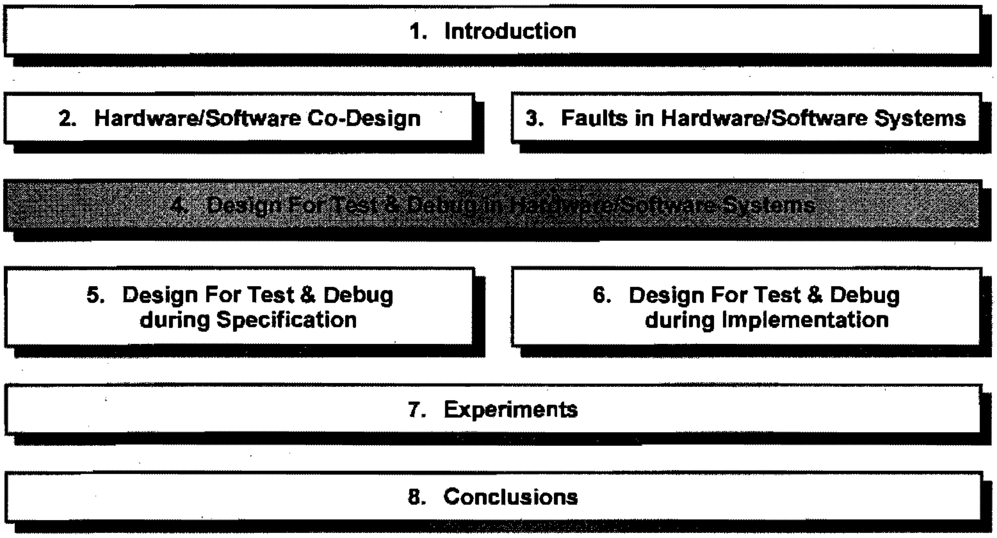
In this chapter we introduced a generic architectural model for hardware/software systems, consisting of application software, system software, hardware nucleus, application-specific hardware, and communication interfaces. We argued that integration testing and system testing should primarily focus on verifying the communication interfaces in our architectural model. However, we also argued that exhaustive testing for interfacing faults and system-level faults is unfeasible due to the large number of possible sequences of events, the interleaved execution of processes, timing dependencies and non-determinism.

We classified faults considering fault origin and fault persistence, and we introduced the notions of fault, error and failure to reason about cause-effect relations. We showed that there are no effective, logical fault models yet for modeling interfacing faults and system-level faults. We argued that many interfacing faults and system-level faults in hardware and software are related to concurrency. We classified these faults into faulty communication and synchronization protocols, faulty mutual exclusive access to shared data or shared resources, faulty process scheduling, deadlocks, race conditions, and faulty interrupt handling. These faults typically appear as temporary faults at run-time, and they often cannot be reproduced during debugging.

The main conclusion of this chapter is that our method towards design for test & debug should primarily aim at detecting faults in communication interfaces.

Chapter 4

Design For Test & Debug in Hardware/Software Systems


1. Introduction
2. Hardware/Software Co-Design
3. Faults in Hardware/Software Systems
4. Design For Test & Debug in Hardware/Software Systems
5. Design For Test & Debug during Specification
6. Design For Test & Debug during Implementation
7. Experiments
8. Conclusions

In this chapter we present our approach to design for test & debug in hardware/software systems. The term 'design for test & debug' indicates that we address both design-for-test and design-for-debug simultaneously. We discuss the basic principles of our design for test & debug approach and we show how this approach affects the design of hardware/software systems. We elaborate on the concept of Point of Control and Observation (PCO), which forms the key element of our approach. We demonstrate the benefits of PCOs for improving system-level testing and debugging.

4.1 Introduction

In chapter 2 we reviewed the state-of-the-art on hardware/software co-design. We concluded that hardware/software co-design methods offer considerable improvements over traditional design methods. However, we also concluded that hardware/software integration testing, system testing, and debugging are still very troublesome, which is mainly due to the limited visibility into the internal operation of the system. The current hardware/software co-design methods do not consider or support testing and debugging of hardware/software systems. We argued that design-for-test and design-for-debug techniques for system-level testing and debugging should be integral elements of hardware/software co-design.

In chapter 3 we examined faults in hardware/software systems. We concluded that the primary aim of integration testing and system testing is to detect interfacing faults. Interfacing faults are due to incorrect communication interactions between multiple hardware and/or software components. These interfacing faults can hardly be detected by testing hardware and software components in isolation. Therefore, they should be revealed during integration testing, when larger and larger aggregates of components are integrated and tested. Some interfacing faults are due to very complex interactions between components in the application software, the system software, the hardware nucleus, and the application-specific hardware. These complex, system-level interfacing faults should be detected during system testing, when the system is tested as a whole.

We showed in chapter 3 that interfacing faults in software are generally related to concurrency and memory access. We also showed that a significant amount of interfacing faults are hardware design faults. Typical interfacing faults are faulty communication and synchronization protocols, faulty mutual exclusive access to shared data or shared resources, faulty process scheduling, deadlocks, race conditions, and faulty interrupt handling.

We illustrated that hardware/software integration testing, system testing, and debugging are very troublesome. Exhaustive testing is unfeasible due to the large number of possible sequences of events and the unpredictable timing of events in the system environment. Hardware/software systems often incorporate non-deterministic behavior due to the interleaved execution of software processes. Timing dependencies and non-determinism make it usually impossible to reproduce an error during debugging. Testing and debugging a system through its external interfaces does usually not provide sufficient control and observation of the internal system operation.

In this chapter we present our approach to design for test & debug (i.e. design-for-test and design-for-debug) in hardware/software systems to deal with the problems of integration testing, system testing and debugging. We first introduce the basic principles on which our approach is based. Next, we describe our approach in detail. Our initial ideas on design for test & debug were published in [VSSvR94, Vra94, VWvW96], while more recent work is published in [VSS96, VSS97].

4.2 Basic Principles

In our opinion, integration testing, system testing and debugging of hardware/software systems can be improved by considering the following three basic principles.

Basic Principle 1

Design for test & debug is required to improve integration testing, system testing and debugging of hardware/software systems.

Integration testing, system testing and debugging are impeded by the limited visibility into the internal system operation. The external system interfaces provide insufficient observation and control of the hardware and software components inside the system and their interactions. The visibility can be improved by using test & debug equipment to collect additional information about the internal system operation. Test & debug equipment typically consists of a logic analyzer, an oscilloscope and measurement equipment that are connected by probes to hardware buses and wires in the system. Although test & debug equipment can provide very detailed and accurate measurement, there are severe restrictions. Using test & debug equipment requires the probing of hardware buses and wires. However, it is generally unfeasible to connect probes to the internal circuitry of an IC. Unfortunately, the current trend in hardware/software systems is to integrate more and more functions on a single chip: yesterday's systems are today's chips. The increasing complexity of ICs and the inability to probe the internal IC circuitry seriously impede the use of traditional test & debug equipment.

Probes can be connected to off-chip buses and wires. However, monitoring the external buses and wires will not easily resolve an IC's internal operation and state. For instance, modern microprocessors incorporate architectural features such as pipelining (parallel fetch, decode and execution of instructions), superscalar architectures (parallel execution of instructions, using techniques like out-of-order execution, branch prediction and speculative execution), caches (both for program instructions and data storage) and DMA. These architectural features make it very difficult to determine the microprocessor's internal state and operation.

In addition, probes connected to buses and wires will collect huge amounts of low-level data. Storing, processing and analyzing the data requires large memories and powerful data processing systems. It is very difficult to obtain high-level information about the activities of the operating system and application software processes by monitoring low-level data on hardware buses and wires. Nevertheless, modern test & debug equipment provides very sophisticated test & debug tools, in which hardware equipment (probes, logic analyzer, oscilloscope) is linked to software tools running on a workstation. The software tools offer user interfaces to control the hardware equipment, storage of measured data, data analysis, data visualization, and symbolic debugging.

Visibility into microprocessor operation can be improved by using an in-circuit emulator. The target processor is removed from the system and the in-circuit emulator is plugged into the same socket. The in-circuit emulator acts identical to the target processor, but the processor's registers can be observed and controlled. Although an in-circuit emulator considerably improves visibility into a microprocessor, the obtained data is still of a low level. Furthermore, the target processor has to be replaced by the in-circuit emulator, and observation and control is achieved by stealing processor cycles. This causes interference in the real-time behavior of the system.

Software is typically developed on a host system, e.g. a workstation, that offers powerful software development tools, such as compilers, assemblers, testing tools, debuggers and simulators. Testing and debugging on a host system is very useful for software components and aggregates of software components. However, the real-time software behavior and the interaction with the hardware components cannot be verified until the software is integrated into the target system. Unfortunately, the target system provides no software development tools, and visibility into the software operation can only be achieved using test & debug equipment such as an in-circuit emulator. Hence, testing and debugging software in a target system is very difficult.

The bottom line is that traditional test & debug equipment and traditional test & debug approaches are inadequate for hardware/software integration testing, system testing and debugging. The only way to achieve adequate visibility into the system's internals is by designing-in test and debug features into the hardware and/or software. Hence, design for test & debug (i.e. design-for-test and design-for-debug) is required to improve integration testing, system testing, and debugging of hardware/software systems.

Basic Principle 2

Design for test & debug should provide visibility into communication interfaces and into state information of software processes and hardware components.

The focus of integration testing and system testing is on verifying the communication interactions between hardware and software components in the system. System debugging is mainly concerned with diagnosis of interfacing faults. Design for test & debug should therefore concentrate on providing visibility of the communication interfaces in the system. Furthermore, testing and debugging is considerably improved when visibility is provided into the state information of software processes and hardware components.

Figure 4.1 illustrates the basic concept of design for test & debug on the process level. A process generally consists of operations and data. The operations process input events, transform the data, and generate output events. This general view on a process is applicable to both hardware and software.

- In hardware, a process is implemented as a hardware module that typically consists of combinational logic which performs operations on data stored in elements such as flipflops, registers, or memory.

- In software, a process can be an object in an object-oriented programming language. The object consists of methods that perform operations on variables and data structures. In traditional programming languages, a process with a single thread of control is implemented as a set of functions that perform operations on local and global variables and data structures.

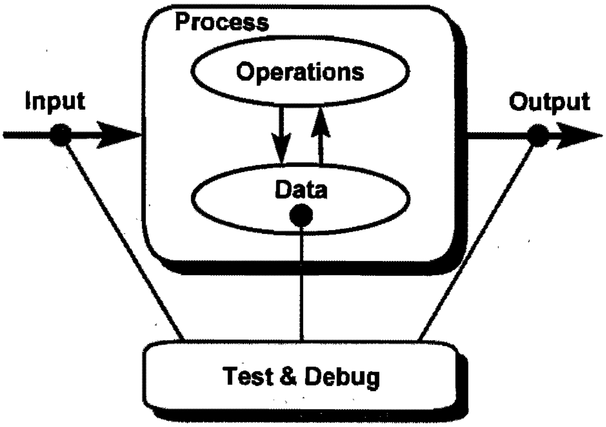


Figure 4.1 *Process-level design for test & debug*

Design for test & debug on the process level as shown in figure 4.1 implies that the communication interfaces and the state information of the process are visible to an external tester/debugger. This is achieved by probing the inputs and outputs of the process (the communication interfaces) and by probing the data that contains state information. Visibility into the communication interfaces and the state information of processes provides essential information about the internal system operation during testing and debugging.

In section 3.2 we stated that the system specification consists of processes that are implemented in software (as processes in the application software) or in hardware (as modules in the application-specific hardware). Visibility is required into the operational state of processes, i.e. whether a process is running/active or stopped/inactive.

Executing concurrent processes in the application software on a single processor requires process scheduling. The operational state of a process can be ready, running, or blocked, as shown in figure 4.2.

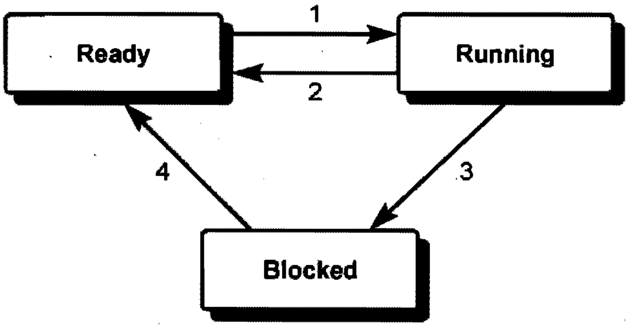


Figure 4.2 *Operational states of a software process*

A process in the ready state is waiting to be executed on the processor. When the process is executed, it goes into the running state (1). After some time, the process is suspended and goes back into the ready state (2). For instance, in preemptive process scheduling a process is suspended when its allotted amount of processor time has been spent, or when another process with higher priority is waiting to be executed on the processor. A running process is blocked (3) when it is waiting for an external event, like waiting for access to a shared resource or waiting for an event from another process or from the system environment. A blocked process is taken away from the processor by the operating system and put into the blocked state. When the operating system detects the occurrence of the event that the blocked process is waiting for, the process state becomes ready (4). The scheduling of processes is performed by the dispatcher in the operating system. Hence, visibility into the operational state of software processes can be achieved by probing the dispatcher.

A similar notion holds for processes that are implemented as modules in the application-specific hardware. A hardware module may be active or inactive. The operation of hardware modules is usually controlled by some control logic. Visibility into the operational state of hardware processes can be achieved by probing this control logic. An example is a bus that is shared among several hardware modules under control of a bus arbiter. The bus arbiter selects the active hardware module that may write data on the bus.

Basic Principle 3

Design for test & debug should be an elementary part of hardware/software co-design, and should be considered in all steps of the design flow: system specification, architecture exploration, architecture refinement, and synthesis.

Design-for-test (DFT) and design-for-debug (DFD) are not new concepts. DFT and DFD have been practiced for decades and in the course of time many techniques for DFT and DFD have been developed for both hardware and software. The traditional techniques for DFT and DFD are typically applied during hardware synthesis and software synthesis, as shown in figure 4.3. These traditional DFT and DFD techniques are very useful for component testing and debugging. However, they are of limited use for integration testing, system testing and system debugging.

- Hardware DFT techniques mainly concentrate on detecting physical hardware faults during production testing and field testing. Hardware DFT techniques primarily aim at structural testing and therefore do not address functional design verification, i.e. detecting hardware design faults. Hardware DFT techniques are well developed, both at the IC level (e.g. scan paths, BIST), at the PCB level (boundary scan) and at the system level (backplane test buses).

  The hardware DFT facilities can be used to a certain extent for design verification, like scan-based debugging in which an IC's scan paths and boundary scan architecture are used for debugging hardware design faults. Scan paths provide that flipflops in an IC can be connected in series, resulting in a serial shift register. Scan-based debugging therefore provides serial access to an IC's state information. However, scan-based debugging yields low-level information, only a single serial line is available for shifting out data, and during shifting the

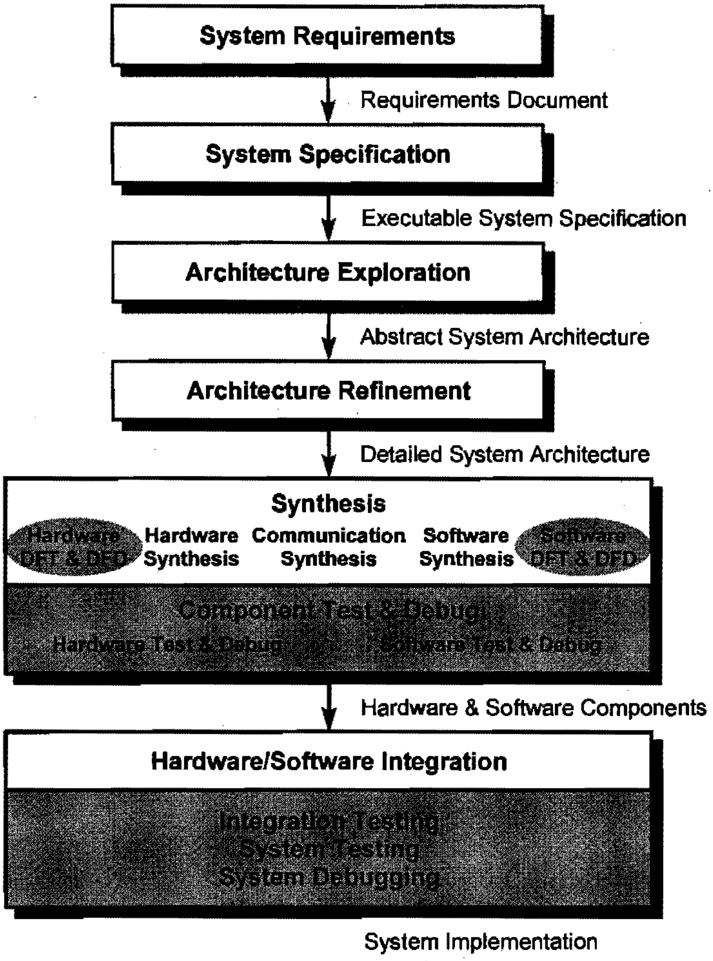


Figure 4.3 Current practice on design for test & debug

IC's state is affected. Hence, scan-based debugging is not really suited for on-line testing and debugging of real-time behavior.

Modern microprocessors incorporate hardware DFD techniques, like hardware breakpoints and debug modes. These hardware DFD techniques are primarily intended for debugging the software that is executed on the processor. However, this does not provide visibility into the surrounding, dedicated hardware components.

- Software DFT and DFD techniques for testing and debugging in the target system typically imply software instrumentation. There is a large variety of techniques for software instrumentation, ranging from very ad hoc solutions, like inserting additional debug statements (e.g. additional 'printf' statements in C code), to structured solutions like software monitors. Software instrumentation is very useful for software testing and debugging in the target system, but does not provide any visibility into dedicated hardware components.

> Furthermore, software instrumentation is typically performed by inserting additional code, which implies modifying the software. The interference caused by software instrumentation may change the software behavior and particularly the timing behavior, which is intolerable for real-time systems.

We gave only a brief description of hardware and software DFT and DFD techniques. A comprehensive overview of hardware DFT and DFD techniques as well as software instrumentation techniques and tools will be provided in chapter 6.

The traditional DFT and DFD techniques are typically considered during the synthesis phase in the design flow, as shown in figure 4.3. In our opinion, DFT and DFD should be considered much earlier in the design flow, starting already from the system specification. During system specification, the functional behavior of the system is captured in a formal description, describing the system as a set of concurrent, communicating processes. The essential information for system-level testing and debugging, i.e. the communication interfaces and process state information, is stated explicitly in the system specification. Our design for test & debug approach is based on extending the system specification with additional functional behavior to achieve visibility into communication interfaces and process state information. Subsequently, these test & debug functions are taken into account in all the successive stages of the design process. Our approach provides that the DFT and DFD facilities can be used effectively for integration testing, system testing and system debugging.

Our approach to design for test & debug provides access and hence visibility into the system internals by means of additional test & debug functions. The primary purpose of the test & debug functions is to collect essential information on the system behavior, i.e. observing or monitoring the internal system operation. In addition, the test & debug functions may offer capabilities to control the internal operation of the system for testing and debugging purposes. For instance, messages can be inserted on communication interfaces, or process state information can be modified. Hence, the test & debug functions provide both observation and control of the system behavior.

4.3 Design For Test & Debug

Based on the three basic principles described in the previous section, we propose an approach for system-level design for test & debug. In the following sections we will describe how our design for test & debug approach is applied in the various stages of the design flow. Figure 4.4 provides an overview of our design for test & debug approach in hardware/software co-design.

4.3.1 System Specification

In the system specification, we insert additional functional behavior to achieve visibility into communication interfaces between processes and visibility into process state information. The system specification provides a system-level view, describing the functional system behavior in terms of concurrent, communicating processes. The system specification explicitly describes interprocess communication and the process behaviors. Hence, the system specification is most suitable for determining where to insert additional test & debug functions for achieving visibility into communication interfaces and process state information.

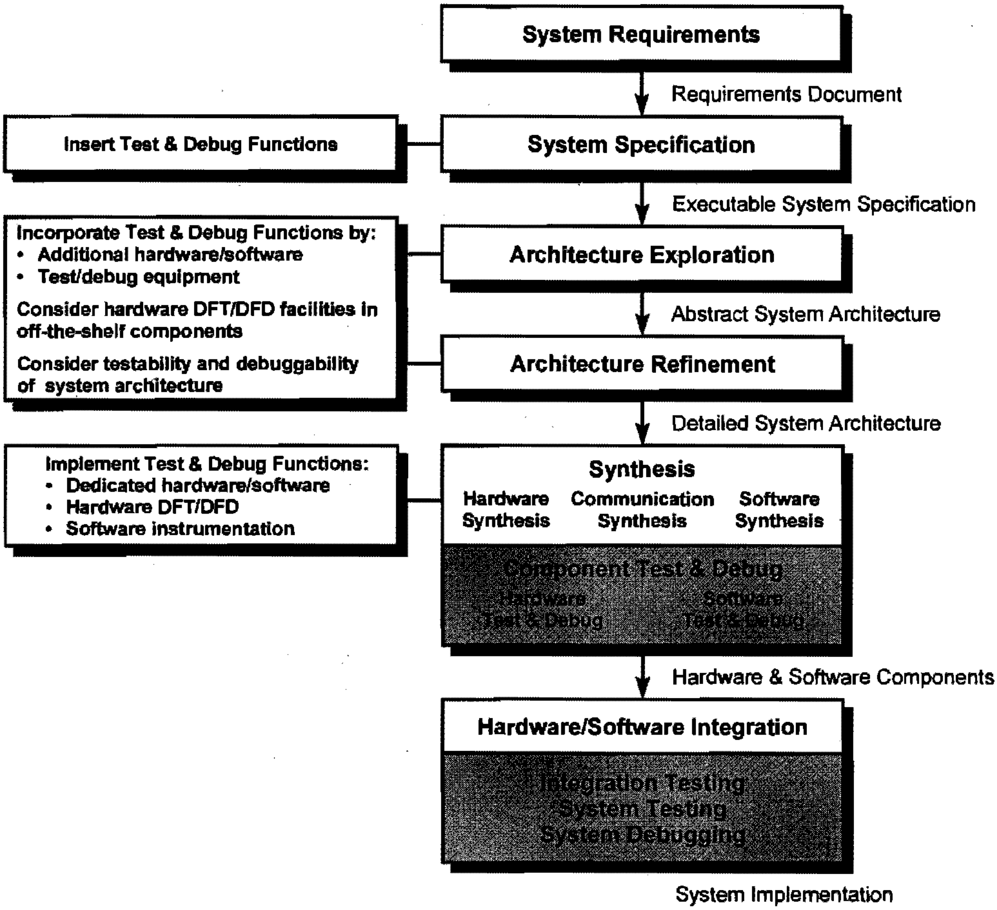

System Requirements
Requirements Document
Insert Test & Debug Functions
System Specification
Executable System Specification
Incorporate Test & Debug Functions by:
• Additional hardware/software
• Test/debug equipment
Consider hardware DFT/DFD facilities in off-the-shelf components
Consider testability and debuggability of system architecture
Architecture Exploration
Abstract System Architecture
Architecture Refinement
Detailed System Architecture
Implement Test & Debug Functions:
• Dedicated hardware/software
• Hardware DFT/DFD
• Software instrumentation
Synthesis
Hardware Synthesis
Communication Synthesis
Software Synthesis
Component Test & Debug
Hardware Test & Debug
Software Test & Debug
Hardware & Software Components
Hardware/Software Integration
Integration Testing
System Testing
System Debugging
System Implementation

Figure 4.4 Design for test & debug in hardware/software co-design

The notion of test & debug function is an abstract concept, indicating some functional behavior which provides that interprocess communication or process state information can be observed and controlled in the system environment. In figure 4.1, we showed the concept of test & debug functions on the process level to access the communication interfaces and state information of a process. In section 4.4 we will make the concept of test & debug function more concrete by introducing Points of Control and Observation (PCOs).

4.3.2 Architecture Exploration

The test & debug functions that are inserted in the system specification, are incorporated next into the system architecture during architecture exploration. The test & debug functions can be realized in three ways:

Realizing test & debug functions with test & debug equipment
Test & debug functions can be realized by using test & debug equipment, like a logic analyzer, an oscilloscope, an in-circuit emulator or measurement equipment. The advantage of this approach is that there is no need for additional hardware and software in the system to realize the test & debug functions.

During architecture exploration, the hardware/software system architecture is defined: the processes and communication channels in the system specification are mapped onto hardware and software components and communication mechanisms. Because the test & debug functions are part of the system specification, they are also included in the system architecture. Instead of adding hardware/software in the system architecture to implement the test & debug functions, the test & debug functions point out those places were probes should be connected, or whether a processor should be replaced by an in-circuit emulator. Hence, the test & debug functions indicate where to probe for what information. Visibility into communication interfaces or process state information is now achieved by connecting probes to a hardware bus, or by using the capabilities of the in-circuit emulator.

We already indicated that the use of test & debug equipment is severely hindered by the huge amounts of low-level data that have to be analyzed, by the limited accessibility into the internal circuitry of complex ICs, and by the probe effect. Furthermore, a logic analyzer and an oscilloscope provide only observability and no controllability of the internal system behavior. Hence, often test & debug equipment cannot be used effectively for realizing our test & debug functions.

Realizing test & debug functions with dedicated hardware/software
The test & debug functions can be realized by implementing them directly into hardware and/or software. In this approach, the test & debug functions are treated just like the other functions in the system specification. During architecture exploration, the complete system specification (including the test & debug functions) is mapped onto hardware and software components and communication mechanisms. Obviously, this approach introduces a certain amount of overhead costs, because additional hardware and/or software is required to realize the test & debug functions. In general, the preferable approach is to realize the test & debug functions with test & debug equipment whenever possible, and to implement the test & debug functions in hardware/software if additional accessibility is required which cannot be achieved by using test & debug equipment.

In the architecture exploration phase, various alternative hardware/software architectures are explored. As indicated in section 2.3.3, architecture exploration consists of partitioning, allocation, transformation and estimation. Because the test & debug functions are part of the system specification, they are automatically taken into account during architecture exploration. Estimating the quality of a system architecture, i.e. evaluating criteria like costs, performance, silicon area and memory size, therefore also considers the effects of additional test & debug functions.

Realizing test & debug functions with traditional DFT and DFD techniques
The test & debug functions may also be realized by using the traditional DFT and DFD techniques. For instance, the test & debug functions may be realized by using hardware DFT facilities, like scan paths, or by using hardware DFD facilities, like hardware breakpoints, or by using software instrumentation, like a software monitor.

Hardware DFT facilities are typically incorporated in ICs and PCBs to test for physical hardware faults during production testing and field testing. In some cases, it may be possible to use these hardware DFT facilities for realizing our test & debug functions. This is very attractive, because hardware DFT can now be used to test for physical hardware faults as well as to test for hardware/software design faults. Likewise, modern microprocessors offer built-in debug facilities, to support debugging of software that is executed on the microprocessor. These DFD facilities can be used to realize our test & debug functions.

A limitation of using traditional DFT and DFD techniques is that usually not all of the specified test & debug functions can be realized. For instance, scan-based debugging is a technique in which the scan paths inside an IC are used to observe and control the IC's internal state. Although scan-based debugging provides observability and controllability, data has to be shifted out over a single serial line and the internal state is affected during shifting. Hence, scan-based debugging cannot be used for on-line debugging of real-time behavior. DFT and DFD techniques for testing and debugging of software in a target system typically imply the use of a software monitor. However, an off-the-shelf software monitor may not provide the required degree of visibility, and usually offers only observation capabilities while lacking control capabilities.

An intermediate solution is to use the traditional DFT and DFD facilities for realizing our test & debug functions, and to extend them with some dedicated hardware and/or software. For instance, extra hardware may be added on-chip to improve scan-based debugging. In fact, this approach is proposed in [vRBMV97]. In chapter 6 we will elaborate further on how our test & debug functions can be realized by using the traditional DFT and DFD techniques in hardware and software.

Using the traditional DFT and DFD techniques implies that during architecture exploration the testability and debuggability of off-the-shelf hardware components should be considered. For instance, during architecture exploration a particular target processor is selected to execute the application software and the system software. Some processors incorporate DFD facilities, like hardware breakpoints and advanced debug modes, while other processors lack these DFD facilities. It may be preferable to select a processor that incorporates DFD facilities. The traditional DFT and DFD techniques can be used to implement test & debug functions in application-specific hardware components. This implies that in our design for test & debug approach, hardware DFT and DFD is already considered during architecture exploration. This in contrast to the traditional approaches were hardware DFT and DFD is not considered until the synthesis stage.

4.3.3 Architecture Refinement & Synthesis

During architecture exploration we make decisions on how to realize the test & debug functions. Additional hardware/software is required when we decide to implement the test & debug functions using dedicated hardware/software or using traditional hardware/software DFT and DFD techniques. During architecture refinement and synthesis, the detailed implementation descriptions for the hardware and software components are generated, which include the hardware and software for the test & debug functions.

4.3.4 Dealing with the Side Effects

In traditional DFT and DFD methods, additional hardware and software for realizing test & debug functions is typically introduced during synthesis, as shown in figure 4.3. The main problem of traditional DFT and DFD methods resides in the side effects caused by this additional hardware and software:

- test & debug functions require overhead costs due to extra hardware/software;
- test & debug functions affect the system performance;
- test & debug functions interfere with the dynamic behavior of the system.

A typical software DFT technique is to add a software monitor in the target system, which collects information about the internal system operation at run-time. Adding a software monitor requires additional program memory in the target system to store the monitor's program code. The software monitor causes performance degradation, because the processor has to execute the application software, the system software as well as the monitor software. The software monitor interferes with the system behavior, because it disturbs the real-time behavior and the relative timing between processes. This interference, known as the 'probe effect' or the 'instrumentation uncertainty principle', can either introduce new faults or prevent some faults from occurring.

In our design for test & debug approach, we try to avoid the side effects of additional hardware and/or software in advance. We consider test & debug functions already in the system specification. In the subsequent design steps, the test & debug functions are treated just like the normal system functions. The effects of the test & debug functions can therefore be analyzed in advance, and measures can be taken if these effects are intolerable. For instance, analysis during architecture exploration may indicate that a software monitor requires a certain amount of additional processor time. A possible solution is to select a processor which offers high enough performance, such that there is sufficient processing time left for executing the software monitor. An alternative solution is to replace the software monitor partly or completely by a hardware monitor.

We use various techniques in our design for test & debug approach to deal with side effects. First, the additional test & debug functions are confined to minimize the amount of additional hardware/software and performance degradation. Furthermore, the probe effect is hidden behind logical time, which implies that the order in which events occur inside the system is not affected by the additional test & debug functions. Hence, although the timing of events may be affected, their relative ordering remains the same. Furthermore, the probe effect can be avoided completely by leaving in the test & debug functions, or by realizing truly non-intrusive test & debug functions by means of dedicated hardware or test & debug equipment. We will elaborate further on these topics in chapter 5 and 6.

4.3.5 Testability and System Architecture

Our design for test & debug approach is based on improving observability and controllability by inserting test & debug functions. Hence, we improve testability by inserting additional functional behavior. An alternative approach for improving testability is to consider the inherent testability of the system architecture. The characteristics of the system architecture, such as the hardware/software partitioning and the selected hardware components, affect how good a system can be tested. During architecture exploration, a number of alternative system architectures is evaluated, considering criteria such as costs and performance. Testability can also be used as a criterion for evaluating the quality of system architectures.

In [AHLTR96, THR96] a testability-oriented approach is proposed for hardware/software partitioning. The system specification is decomposed first into a number of (implementation-independent) components. Each component can be implemented in hardware or in software. A number of alternative hardware/software partitionings is obtained by selecting different combinations of component implementations. The testability of each component, both for its hardware implementation and for its software implementation, is estimated by considering the number of required test vectors to test the component. The overall system testability is obtained by considering the testability of all the individual components in the system. The overall system testability figure provides hardly any meaningful information. However, this approach is useful to identify those components in the system that have a major impact on the system testability, and to evaluate the effects on testability when implementing these components in hardware or software. Hence, the inherent system testability can be improved by selecting an appropriate hardware/software partitioning.

In [Sch93b] it is argued that time-triggered architectures are preferable to event-triggered architectures when considering testability of distributed, real-time software. In event-triggered systems, all actions in the system (computations, communication protocols, and interactions with the environment) are initiated by the observation of an event, such as receiving a message or an interrupt from another process or from the external environment. On the other hand, in time-triggered systems all actions are initiated exclusively at predefined points in time, governed by a global clock. A time-triggered architecture implies that the computations, the communication protocols, as well as the interactions with the external system environment are time-triggered actions. Time-triggered computations imply that each computation is started and ended at predefined points in time. Time-triggered communication denotes that each communication protocol is started on a predefined point in time and ended before a predefined end time. Time-triggered interaction with the external environment implies that the system periodically polls the inputs at predefined points in time to detect whether an event has occurred since the last poll. If multiple events have occurred, the events are processed in a fixed order. Although a time-triggered architecture puts stringent restrictions on the system architecture, it yields a completely predetermined, deterministic system behavior. These properties significantly enhance system testability. Hence, a time-triggered system architecture inherently provides better testability and debuggability than an event-triggered system architecture.

4.4 Test & Debug Functions

A basic element of our design for test & debug approach is the insertion of test & debug functions in the system specification. The goal of the test & debug functions is to create access and visibility into communication interfaces and process state information. Our basic test & debug function is the Point of Control and Observation (PCO). Figure 4.5 shows a PCO that is inserted in the communication channel between *process A* and *process B*.

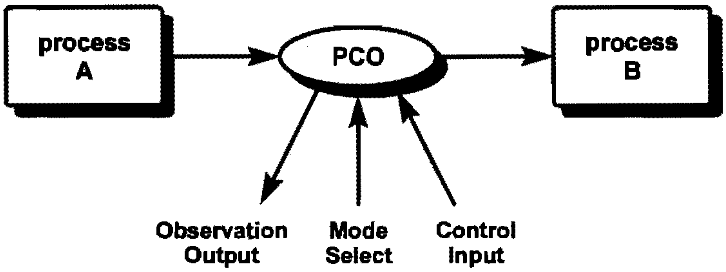


Figure 4.5 *Controlling and observing a communication interface*

A PCO has three modes of operation, as shown in figure 4.6: transparent mode, observation mode, and test mode. The PCO operation mode is selected by the PCO's *Mode Select* input.

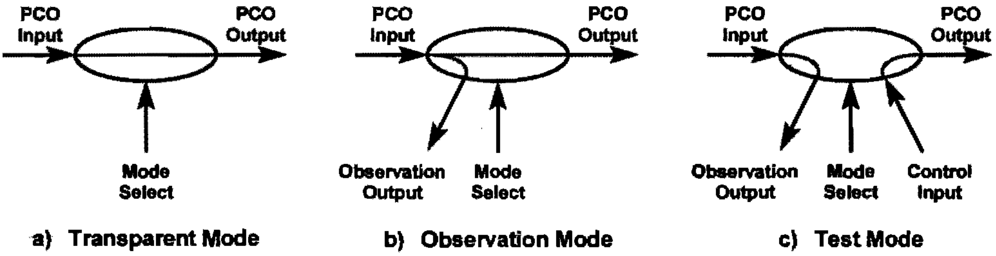


Figure 4.6 *PCO operation modes*

1. When the PCO operates in transparent mode, the *PCO Input* is connected directly to the *PCO Output*. The *Observation Output* and *Control Input* are not used.

   The transparent mode is the default mode during normal system operation. The PCO is completely transparent and performs no observation or control.

2. When the PCO operates in observation mode, the *PCO Input* is connected to both the *PCO Output* and the *Observation Output*. The *Control Input* is not used.

   The observation mode is used to monitor the normal system behavior. The messages that pass through the PCO can be observed on the *Observation Output*.

3. When the PCO operates in test mode, the *PCO Input* is connected to the *Observation Output* and the *Control Input* is connected to the *PCO Output*. Hence, there is no connection between the *PCO Input* and the *PCO Output*.

   The test mode is used during testing and debugging activities. The messages that are received on the *PCO Input* can be monitored, and simultaneously messages can be inserted on the *PCO Output*. The observation and control activities can be performed independently.

A PCO can be inserted in the communication interface between two processes, as shown in figure 4.5. When the PCO operates in transparent mode, the messages from *process A* are transferred to *process B* without any interference of the PCO. In observation mode, the messages are transferred transparently also, but the messages can be monitored as well on the *Observation Output*. In test mode, the messages sent by *process A* can be monitored on the *Observation Output*, and test messages can be sent to *process B* through the *Control Input*. Hence, in test mode *process A* and *process B* can be tested independently and in parallel.

A complete PCO as shown in figure 4.6, has three modes of operation. A partial PCO has only two modes of operation, and can be either a Point of Observation (PO) or a Point of Control (PC). A PO can operate in transparent mode or in observation mode (as shown in figure 4.7), while a PC can operate in transparent mode or in test mode (as shown in figure 4.8).

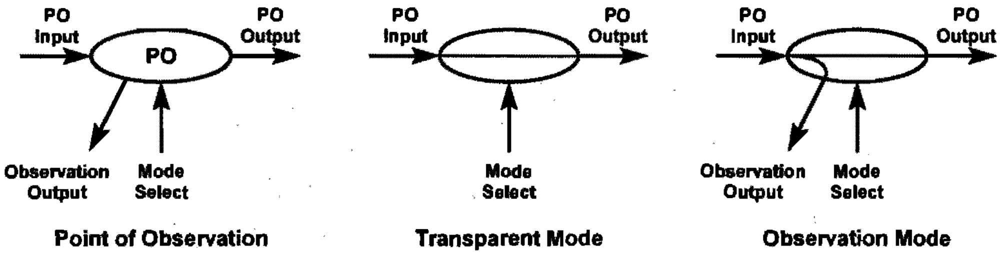


Figure 4.7 Point of Observation (PO)

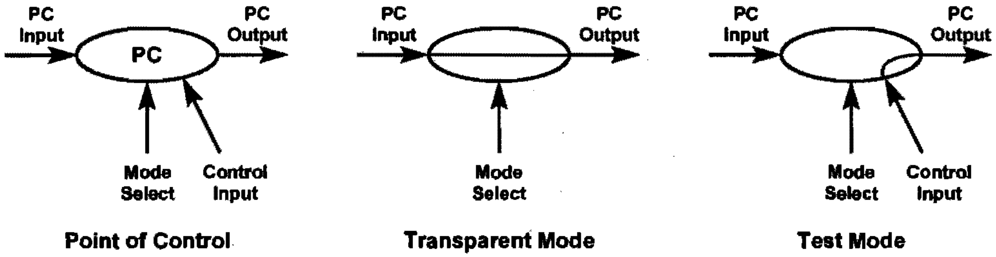


Figure 4.8 Point of Control (PC)

A PCO can be inserted in a process to access the process state information, as shown in figure 4.9. The process is represented as an object, containing methods and data. In traditional object-oriented programming languages like C++, an object is activated when an input message occurs. The input message identifies a particular method, and the selected method is executed which typically implies reading and/or writing the object's encapsulated data and generating output messages. In the POOSL language [Voe95a, Voe95b, vdPV97], a process object is an autonomous entity that is active continuously. An input message is processed by a particular method, identified by the current state of the process object. The method will typically read and/or write the encapsulated data and generate output messages.

Inserting a PCO to access state information is achieved by adding a separate method that implements the PCO behavior. In figure 4.9 the *method PCO* is inserted, and in addition the *Mode Select*, *Observation Output*, and *Control Input* are appended. In transparent mode, the PCO performs no operation. In observation mode, the process state information, i.e. some or all data, can be monitored on the *Observation Output*. In test mode, the process state information can be modified through the *Control Input*. Messages on the *Control Input* are provided by an external tester/debugger, and hence also the timing of these messages is controlled by the external tester/debugger. On the other hand, the *Observation Output* is controlled by the process itself, and hence the process is responsible for the number and the timing of messages sent on the *Observation Output*. In general, only changes in the process state information need to be monitored. Hence, whenever a method modifies the process data, it can invoke the *method PCO* to output the modified state information. Another solution is to extend a method with a Point of Observation. In figure 4.9 the *method B* is equipped with a PO. Whenever the *method B* modifies the process data, it uses the PO to output the modified state information on the *Observation Output*.

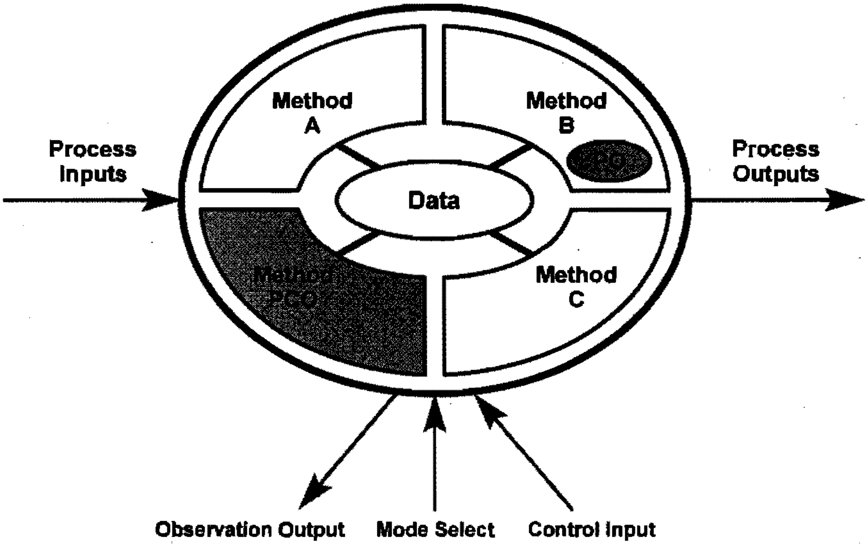


Figure 4.9 Controlling and observing process state information

4.5 Accessing Test & Debug Functions

A PCO provides that a communication interface or process state information can be observed and controlled. However, it is required that the *Mode Select*, *Observation Output*, and *Control Input* of each PCO are connected to a tester/debugger in the external system environment. Hence, communication channels between an external tester/debugger and the PCOs are required for accessing the PCOs. The external tester/debugger controls the operation of each PCO: the PCO operation mode is controlled through the PCO *Mode Select* input, test data is inserted through the PCO *Control Input*, and test results are collected through the PCO *Observation Output*. The communication channels between the external tester/debugger and the PCOs can be provided by using individual PCO channels or a shared PCO channel, as depicted in figure 4.10.

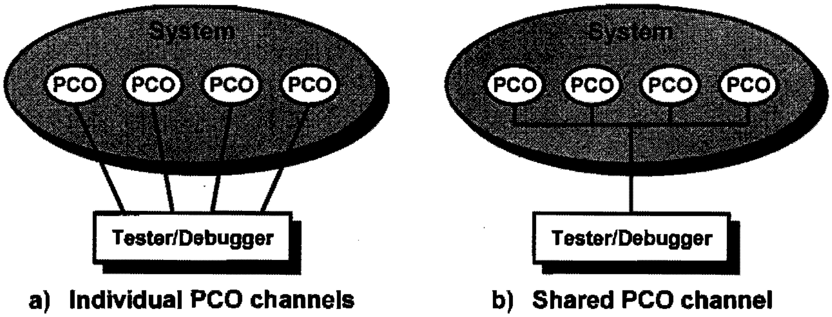


Figure 4.10 PCO communication channels

- Each PCO can be connected directly to the external tester/debugger through a private, dedicated communication channel, as shown in figure 4.10a. This approach is simple, and causes no interference with the normal system behavior. The disadvantage however is that the number of additional communication channels increases linearly with the number of PCOs in the system. Hence, the number of additional communication channels may become very large, and the external tester/debugger should provide many connectors, one for each communication channel.
- The overhead of additional communication channels can be reduced by multiplexing a channel. For instance, a single communication channel can be shared by all PCOs, as shown in figure 4.10b. Channel arbitration is required to secure mutual exclusive access for each PCO to the channel, for instance using time-multiplexing or a bus-arbiter in the external tester/debugger. In addition, the tester/debugger must be able to address each individual PCO on the channel. An advantage of this approach is that a single *Mode Select* signal can be used to switch all the PCOs simultaneously into the same operation mode.

Instead of using dedicated communication channels for PCOs, the system's communication channels can be used to transport the PCO messages. This requires carefully multiplexing the communication channels for transporting both PCO messages and normal system messages. In this case, there is no direct connection between a PCO and the external tester/debugger. Although this approach avoids dedicated PCO communication channels, it is less transparent and less flexible. Furthermore, this approach is suitable for PCOs that access process state information, but less suitable for PCOs that access communication interfaces.

4.6 Using Test & Debug Functions

During integration testing and system testing, the system behavior is exercised by offering test stimuli to the primary system inputs and by observing the system responses on the primary system outputs. Additional control and observation is provided by the PCOs, which offer secondary system inputs (i.e. the *Mode Select* and *Control Input* inputs) and secondary outputs (i.e. the *Observation Output* outputs).

Testing is typically performed in four steps: bring the system into a particular state, offer test stimuli to the system, observe the system's responses and the system's new state, and evaluate the observed responses and system state. The test & debug functions (PCOs) can be used in the following ways during testing:

1. Prior to offering test stimuli to the system, the system should be in a particular, predefined state. This particular state is usually enforced in two steps: first, the system is reset, forcing the system into its initial state; next, a sequence of stimuli is applied to the system to transfer the system from the reset state into the required state.

   This initialization procedure can be shortened by switching the PCOs into test mode. The state information of the processes can now be modified directly using the PCO's *Control Input*. Hence, the system can be enforced directly into the required state using the PCOs.

2. After initialization, the PCOs are switched into observation mode and test stimuli are applied to the system. The PCOs are used to observe the internal system responses and the new system state using the *Observation Output*.

Evaluation is typically performed by comparing the observed system responses with the expected system responses, and by comparing the observed new state with the expected new state of the system. An error is detected if comparison yields a mismatch between the observed and the expected behavior. When an error is detected, debugging is required next to uncover the exact fault that induced the error.

Debugging can be facilitated to a large extent by using PCOs. The PCOs in observation mode provide detailed information during testing about the internal system behavior. If testing uncovers an error, the faulty system part that caused the error can therefore be identified rather easily. Hence, the fault can already be localized to some extent at the beginning of the debugging process. Debugging is usually performed by testing the local system behavior in which the fault resides. The PCOs in test mode can provide direct access to control and observe the local system behavior, which provides excellent debugging support.

Figure 4.11 shows an example of debugging using PCOs. Testing indicated that a fault resides in system part *C*. Debugging is performed by testing part *C* in isolation. By switching PCO_1, PCO_2 and PCO_3 in test mode, part *C* can be disconnected from its surrounding system parts. Test stimuli are applied to part *C* through its primary input and the *Control Input* of PCO_1. The responses of part *C* are observed through its primary output and the *Observation Output* of PCO_2 and PCO_3. In addition, the internal behavior of part *C* can be monitored through the *Observation Output* of PCO_4 and PCO_5, which both operate in observation mode.

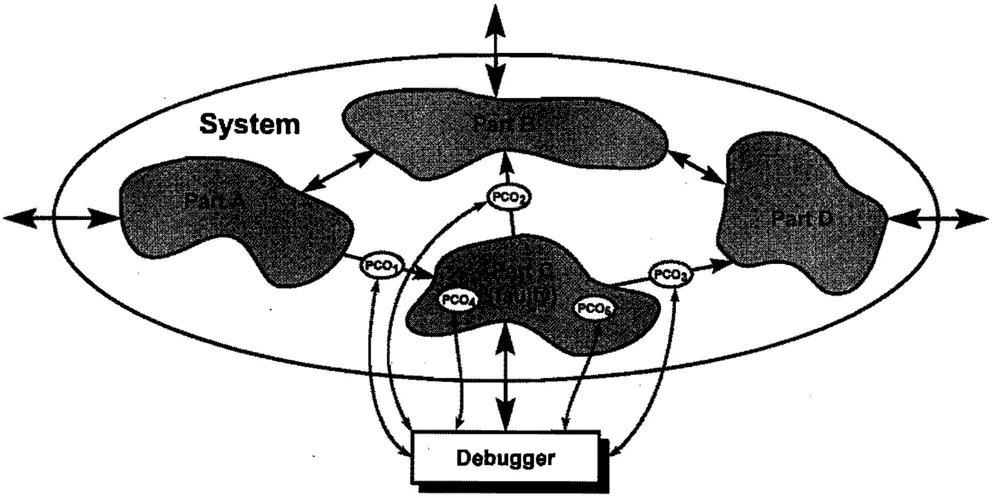


Figure 4.11 System debugging using PCOs

4.7 Test & Debug Functions and System Architecture

In section 3.2 we elaborated on hardware/software system architecture, and we identified nine classes of communication interfaces. When applying our design for test & debug approach by inserting PCOs in the system for accessing communication interfaces and process state information, we obtain the modified system architecture shown in figure 4.12. The communication interfaces 8 and 9 model communication between the system and the external system environment. There are no PCOs required in these communication interfaces, because these interfaces can be accessed directly in the external environment. PCOs are inserted in the communication interfaces 1 through 7.

1. Communication interface 1 represents intraprocess communication, i.e. communication within the application software processes. PCOs in communication interface 1 therefore allow access to the internal communication of application software processes.

   In the previous sections, we stated that we typically insert PCOs during system specification to access the interprocess communication interfaces, which does not include intraprocess communication. Nevertheless, PCOs can also be inserted during system specification to access intraprocess communication, which offers improved control and observation of the internal operation of application software processes. The PCOs in interprocess communication interfaces offer observation and control of the external behavior of application processes, while PCOs in intraprocess communication allow observation and control of the internal process behavior. Hence, PCOs in intraprocess communication interfaces allow more detailed control and observation, which may be very useful during system debugging when detailed information about some local behavior in the system is required. However, the insertion of PCOs in intraprocess communication introduces a large overhead, because many additional PCOs are required.

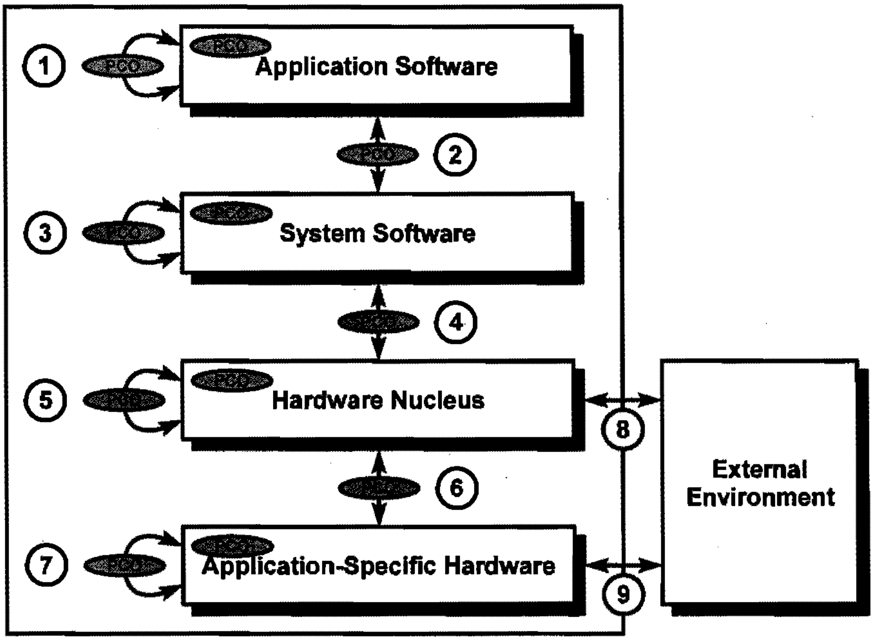


Figure 4.12 PCOs in hardware/software system architecture

2. Communication interface 2 represents interprocess communication, i.e. communication between the application software processes. PCOs in communication interface 2 therefore allow access to the communication interfaces between application software processes. These PCOs are typically inserted during system specification, when PCOs are introduced to access communication interfaces.

   In addition, communication interface 2 represents the scheduling of application software processes for execution on the processor(s) provided in the hardware nucleus. We discussed in section 4.2 that information on process scheduling and the processes' operational state (figure 4.2) is essential during testing an debugging. This information can typically be provided by inserting PCOs in the dispatcher of the system software. These PCOs are typically introduced during architecture exploration and architecture refinement. Obviously, these PCOs cannot be explicitly modeled in the system specification, because the system specification does not include implementation details about the system architecture and the system software.

3. Communication interface 3 represents internal communication in the system software, i.e. communication between the software components that build up the system software such as the dispatcher and I/O drivers. PCOs in communication interface 3 are typically introduced during architecture exploration and architecture refinement. These PCOs provide information about the internal operation of the system software.

   The system software offers mechanisms to provide mutual exclusive access of shared resources, like semaphores or monitors. PCOs in the system software may provide detailed visibility into these mechanisms, for instance observation and control of a semaphore.

4. Communication interface 4 represents low-level communication between the hardware nucleus and the software, typically at the abstraction level of the ISA (Instruction-Set Architecture). In fact, communication interface 4 is a low-level representation of the communication interfaces 1, 2 and 3. The application software and the system software are executed on the processor(s) in the hardware nucleus, and also all the software communication mechanisms defined in the interfaces 1, 2 and 3 are performed by the processor.

   PCOs in communication interface 4 provide access to the internal operation of the processor(s), like access to registers and functional units as defined in the ISA. These PCOs are typically defined by the DFT and DFD capabilities of the processor. For instance, modern microprocessors allow debug modes in which information about the processor's internal operation is output on some dedicated processor pins. More detailed information about the processor's internal state can be obtained by using scan paths.

   A microprocessor is typically an off-the-shelf component that a designer uses to implement a hardware/software system. The feasibility of PCOs inside the microprocessor is therefore defined by the built-in DFT and DFD features of the microprocessor. Although the designer has no direct control on designing-in PCOs in an off-the-shelf microprocessor, he can select a microprocessor that offers advanced DFT and DFD features instead of a microprocessor that lacks these features.

   Detailed observation and control of a processor's internal operation can be obtained by using an in-circuit emulator. In this case, the PCOs are defined by the capabilities offered by the in-circuit emulator.

5. Communication interface 5 represents internal communication inside the hardware nucleus, which is typically processor-memory communication. PCOs in communication interface 5 therefore allow access to the processor-memory communication. PCOs at this level are typically implemented by test equipment like a logic analyzer, that is connected by probes to the processor-memory bus.

   The current technological trend is to implement embedded processor cores together with memories and peripherals on a single chip. This prevents the use of test & debug equipment, because probing the internal circuitry is general unfeasible. In this case, realizing on-chip PCOs requires building-in hardware test & debug facilities.

6. Communication interface 6 represents communication between the hardware components in the hardware nucleus and application-specific hardware components, constituted by hardware buses and wires. PCOs in these communication interfaces offer access to low-level communication. Likewise the PCOs in communication interface 5, realizing PCOs requires test equipment or built-in hardware DFT and DFD facilities.

7. Communication interface 7 represent the communication between the application-specific hardware components. Communication interface 7 is conceptually equivalent to the communication interfaces 1, 2 and 3. Processes in the system specification are either implemented in software (processes in the application software) or in hardware (modules in the application-specific hardware). Hence, PCOs in intraprocess and interprocess communication in the system specification are either implemented in communication interface 1 and 2 for software, or in communication interface 7 in case of hardware. As in communication

interface 5 and 6, PCOs can be realized by test equipment or built-in hardware DFT and DFD facilities.

Besides PCOs in the communication interfaces, figure 4.12 also indicates that PCOs are included inside the application software, the system software, the hardware nucleus, and the application-specific hardware. These PCOs are used to access process state information. The system specification models the functional system behavior as a set of concurrent, communicating processes. These processes are implemented as application software processes or as hardware modules. Hence, the PCOs inside the application software provide access to the state information of process implemented as application software processes, while the PCOs inside the application-specific hardware provide access to the state information of processes implemented as hardware modules.

The PCOs inside the system software are closely related to the PCOs in communication interface 2 and 3. These PCOs provide access to the internal operation of the system software. In a similar way, the PCOs inside the hardware nucleus are closely related to the PCOs in communication interface 4 and 5. These PCOs provide access to the internal operation of the hardware nucleus.

4.8 System-Level Test Cases

Our design for test & debug approach is primarily concerned with adding test & debug functions. A second concern is selecting appropriate test cases for integration testing and system testing. These system-level test cases should exercise the system behavior, by offering test stimuli at the system's primary inputs and observing the system's responses at the system's primary outputs, and by using the control and observation capabilities of the test & debug functions (PCOs) as explained in the previous sections.

The selection of appropriate test cases, together with the insertion of test & debug functions in the system, should be considered in each phase of the system design flow. This process is graphically depicted in the V-model of figure 4.13.

In the system specification, the functional system behavior is captured into a formal model of concurrent, communicating processes. The test cases used during system testing can be derived from the system specification. The test cases should concentrate on exercising the system behavior, regarding the system as a set of concurrent, communicating processes.

During architecture exploration and architecture refinement, the hardware/software architecture of the system is selected and refined. The test cases used during hardware/software integration testing are derived from the system architecture. The test cases should concentrate on testing the communication interfaces between the various hardware and software components that build up the system.

During hardware and software synthesis, the individual hardware and software components are developed. The test cases used during component testing can be derived from the descriptions of the individual hardware and software components. Most tools for test case generation, like automatic test pattern generator (ATPG) tools for hardware and test case generators for software data-flow and control-flow testing, are aimed at component testing.

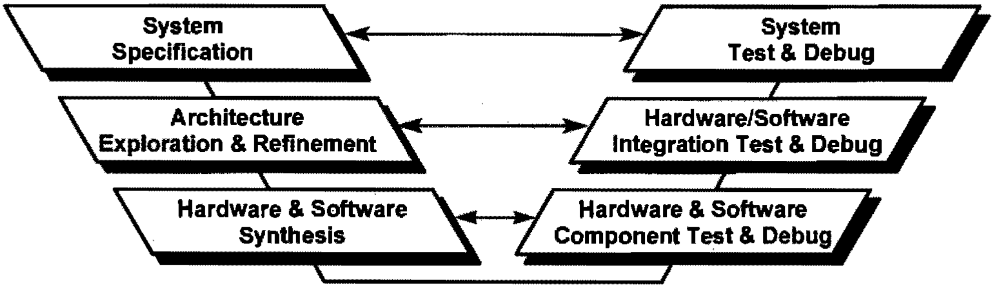

System Specification
System Test & Debug
Architecture Exploration & Refinement
Hardware/Software Integration Test & Debug
Hardware & Software Synthesis
Hardware & Software Component Test & Debug

Figure 4.13 V-model for hardware/software co-design

In modern analysis & design methods, like structured analysis & design (SAD) methods and object-oriented analysis & design (OOAD) methods, the system specification is typically constructed in the analysis stage by considering the events that occur in the system environment and the subsequent behavior of the system in response to these events.

- In Ward & Mellor's structured analysis & design method [WM86], an environment-based modeling approach is proposed for system analysis. This approach is based on identifying first the events in the environment to which the system must respond. Next, the system's response to each event in the environment is specified.

  In [Vra94] we used the event-response list to derive event-traces. An event-trace describes a sequence of events, starting with an event in the external environment that is input to the system, the subsequent internal events, and the response events that are the output of the system. An event-trace defines the causal relations between the various events.

  In a case study, we modeled an elevator control system using Ward & Mellor's structured analysis method. The elevator control system is used to control the operation of four elevators in a 40-floor building. We derived event-traces from the system specification, based upon the event-response list. The event-traces were used as system-level cases, and we applied them to both validation and verification of the system specification, as well as to testing of a software implementation of the elevator control system. We will elaborate on this case study in chapter 7.

- In Rumbaugh's OMT method for object-oriented analysis & design [R$^+$91], first an object model is constructed during system analysis, which represents the static structure in terms of object classes and their relations. Next, a dynamic model is constructed, which represents the sequences of operations that occur in response to external stimuli. The dynamic model consists of a collection of state diagrams that interact with each other via events. A scenario is defined as a sequence of events that occurs during one particular execution of a system. A scenario may include all events in the system, or only a subset of events of interest. A scenario can be regarded as the historical record of executing a system or a thought experiment of executing a system. The sequence of events and the objects exchanging events are illustrated in an event-trace diagram.

- In Jacobson's OOSE method for object-oriented analysis & design [J$^+$92], a similar concept to scenarios is used, called use cases. A use case is defined as a behaviorally related sequence of transitions that a user performs in a dialogue with the system. The complete system behavior is described by the set of all use cases. Interaction diagrams are created for each use case, showing the interactions between communicating objects.

- In Van der Putten & Voeten's SHE method for object-oriented analysis & design [vdPV97], a related approach to the OMT scenarios and the OOSE use cases is proposed. A scenario in SHE defines and describes the responses elicited by communication with one or more objects outside the (sub)system under analysis. A scenario in SHE also describes the internal communication between the processes that model the system behavior. This in contrast to scenarios in OMT and use cases in OOSE, which consider the system as a black box and only describe the communication between the system and its external environment. A scenario is graphically represented in a Message Flow Diagram, in which the stimuli and responses are modeled of the processes that participate in a specific scenario. In addition, a scenario is described in a narrative form. A scenario narrative is a text that describes the causal relations between events in the environment and responses of objects in the system, as well as causal relations between events in the system and responses of objects in the environment.

  Scenarios are one of the key elements in the SHE method. The system specification is created by playing scenarios. Scenarios are used to identify objects and their communication flows, and to reason about behavior, the ordering of events, and the reactions of collaborating processes.

  We also modeled our elevator control system using the SHE method, and created a formal system specification in the POOSL language. We used the scenarios for validation and verification of the system specification. We will elaborate also on this case study in chapter 7.

The concepts of event-traces, use cases and scenarios all define the interactions between concurrent, communicating objects in the system and in the system environment. They are most suited to be used as system-level test cases, both for validation and verification of the system specification as well as for testing of the system implementation.

System-level test cases can be derived from the system specification. An executable system specification implies that the system specification can be validated and verified by means of simulation. During simulation, typically scenarios are simulated. Evaluation of the simulation results is usually performed by the designer, who manually compares the simulated behavior with the expected behavior. During system testing, the same scenarios can be used as test cases. Evaluation of the test results can be automated, by automatically comparing the executed behavior of the system implementation with the simulated behavior of the system specification.

As indicated, we used event-traces/scenarios in a case study as system-level test cases. These event-traces/scenarios describe the sequence of external events applied to the primary system inputs, the events observed by the PCOs in observation mode inside the system, and the events observed on the primary system outputs. Furthermore, we used PCOs in test mode to quickly bring the system into a predefined state before applying a particular test case.

4.9 Discussion

In this chapter we introduced our approach to system-level design for test & debug in hardware/software systems. We discuss and summarize the major results in this section.

In the previous chapters we argued that the current hardware/software co-design methods do not consider or support testing and debugging of hardware/software systems. We demonstrated that integration testing, system testing and system debugging should aim at detecting interfacing faults and system-level faults. Furthermore, we explained that testing and debugging of hardware/software systems is very troublesome due to limited visibility into the internal system operation, due to the large number of possible sequences of events and the unpredictable timing of external events, and due to non-deterministic behavior caused by the interleaved execution of software processes.

We developed an approach to design for test & debug of hardware/software systems to deal with the previous problems. Our approach is based upon the following basic principles:

1. Design for test & debug is required to improve integration testing, system testing and debugging of hardware/software systems.

2. Design for test & debug should provide visibility into communication interfaces and into state information of software processes and hardware components.

3. Design for test & debug should be an elementary part of hardware/software co-design, and should be considered in all steps of the design flow: system specification, architecture exploration, architecture refinement, and synthesis.

Based upon these three basic principles, we derived an approach for system-level design for test & debug. The key element is that we insert test & debug functions in the system specification. These test & debug functions are taken into account in all the successive stages of the design process. This approach offers the following advantages:

- The system specification provides a system-level view on the functional system behavior, consisting of concurrent, communicating processes. The system specification is therefore most suitable to identify the communication interfaces and process state information that require insertion of test & debug functions to improve accessibility.

- Test & debug functions are part of the system specification, and hence they are automatically considered during architecture exploration. The consequences of test & debug functions on the system architecture can therefore be predicted in advance and appropriate measures can be taken to avoid intolerable side effects, like performance degradation and the probe effect.

During architecture exploration our test & debug functions are incorporated in the system architecture. The preferable approach is to realize the test & debug functions with test & debug equipment (e.g. logic analyzer, ICE) whenever possible, and to implement the test & debug functions in hardware and/or software if additional accessibility is required which cannot be achieved by using test & debug equipment.

Implementing the test & debug functions can be performed either by dedicated hardware/software or by using the current DFT and DFD techniques in hardware and software. Using dedicated hardware/software implies overhead costs, but this approach is transparent and allows to implement test & debug functions exactly as they are specified. Using current DFT and DFD techniques may reduce the overhead costs. This requires that already during architecture exploration requirements are stated on the testability and debuggability of off-the-shelf hardware components, the application-specific hardware components, and software components. In traditional design methods these decisions are not made until the synthesis stage.

Our basic test & debug function is the Point of Control and Observation (PCO). A PCO can operate in transparent mode during normal system operation, in observation mode during testing and debugging to monitor the internal system behavior, and in test mode during testing and debugging to provide direct control and observation of the internal system behavior. Related to the concept of PCO are the Point of Observation (PO) and the Point of Control (PC), which have only two modes of operation.

We demonstrated how PCOs can be used to access communication interfaces and process state information. In addition, we discussed how PCOs can be accessed from an external tester/debugger, using dedicated PCO channels (either individual PCO channels or a shared PCO channel) or sharing the normal system channels.

We typically insert PCOs in the system specification to access interprocess communication interfaces and process state information. In addition, we insert PCOs during architecture exploration in the system software and the application-specific hardware. PCOs in the system software provide visibility into the scheduling of processes and into the internal operation of the system software. PCOs in the application-specific hardware provide visibility into control logic that controls the operation of hardware modules.

We argued that system-level test cases should be derived from the system specification. We showed how these test cases can be used for both validation and verification of the system specification, as well as for testing of the system implementation.

In this chapter, we provided a rough outline of the basic elements and concepts in our design for test & debug approach. Obviously, many issues need further exploration. In the remainder of this thesis, we will address two basic subjects: the specification of PCOs and the implementation of PCOs. In chapter 5, we will elaborate on the specification of PCOs and on the interference caused by inserting PCOs. In chapter 6, we will discuss the implementation of PCOs in hardware and/or software and how the current DFT and DFD techniques in hardware and software can be utilized. Finally, in chapter 7 we will present experiments in which we demonstrate how our design for test & debug approach is applied during analysis and design of an elevator control system.

4.10 Summary

Our method towards design for test & debug in hardware/software systems is based upon three basic principles:

- Design for test & debug is required to improve integration testing, system testing and debugging of hardware/software systems.
- Design for test & debug should provide visibility into communication interfaces and into state information of software processes and hardware components.
- Design for test & debug should be an elementary part of hardware/software co-design, and should be considered in all steps of the design flow: system specification, architecture exploration, architecture refinement, and synthesis.

The key element of our method is to insert Points of Control and Observation (PCOs) in the system specification to access communication interfaces and process state information. The system specification provides a system-level view on the functional behavior, and is therefore most suitable to identify communication interfaces and process state information.

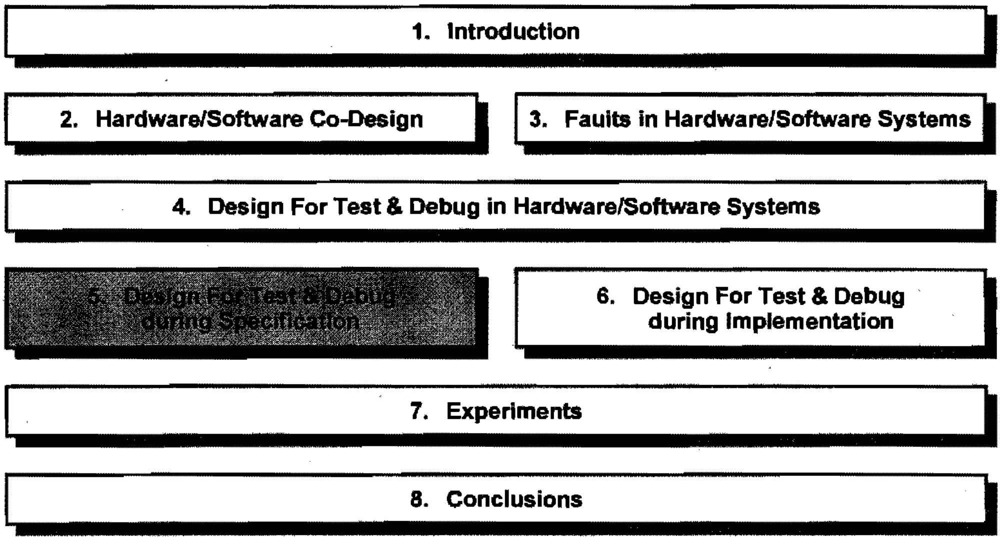
We next incorporate PCOs into the system architecture. Our approach provides that the effects of PCOs on the system architecture can be predicted in advance and appropriate measures can be taken to avoid intolerable side effects, such as performance degradation and the probe effect. Additional PCOs can be inserted during architecture exploration in the system software and in the application-specific hardware. These PCOs can provide visibility into the interleaved execution of processes, the system software and control hardware.

In some cases, PCOs can be realized by test & debug equipment such as a logic analyzer or an in-circuit emulator. However, test & debug equipment often cannot offer the required controllability and observability and hence PCOs have to be implemented in hardware and/or software. Implementing PCOs can be performed either by dedicated hardware/software or by using the current DFT and DFD techniques in hardware and software. Using dedicated hardware/software implies overhead costs, but this approach is transparent and allows to implement PCOs exactly as they are specified. Using current DFT and DFD techniques may reduce the overhead costs. This requires that already during architecture exploration requirements are stated on the testability and debuggability of off-the-shelf hardware components, the application-specific hardware components, and software components. In traditional design methods these decisions are not made until the synthesis stage.

Finally, we showed that system-level test cases should be derived from the system specification. These test cases can be used for both validation and verification of the system specification, as well as for testing of the system implementation.

Chapter 5

Design For Test & Debug during Specification


1. Introduction
2. Hardware/Software Co-Design
3. Faults in Hardware/Software Systems
4. Design For Test & Debug in Hardware/Software Systems
5. Design For Test & Debug during Specification
6. Design For Test & Debug during Implementation
7. Experiments
8. Conclusions

In this chapter we elaborate on design for test & debug during system specification. We concentrate on two key questions: where should PCOs be inserted in the system specification, and what are the effects of PCO insertion on the system behavior.

5.1 Introduction

Our design for test & debug approach is based on the insertion of Points of Control and Observation (PCOs) in the system specification. An important question is where PCOs should be inserted in the system. On the one hand, there should be a sufficient number of PCOs to provide an adequate level of observability and controllability during testing and debugging. On the other hand, the more PCOs the system contains, the more overhead is introduced because each PCO in the system specification has to be realized by test equipment or additional hardware/software in the system implementation. The number of PCOs also influences the complexity of the external tester/debugger: in observation mode the data on the observation output of each PCO has to be analyzed, and in test mode also test data has to be provided to the control input of each PCO. Hence, there should be a balanced number of PCOs, which provides maximum observability and controllability with minimum overhead costs.

In this chapter we will explore various methods to identify those places in the system specification where insertion of a PCO is desirable or required. We will first examine related approaches from literature, namely OSI protocol testing, hardware testability analysis, and system-level testability analysis. Next, we propose scenario-based testability analysis to guide PCO insertion.

A second, important question is how PCO insertion affects the system behavior. Inserting PCOs in the system specification implies modification of the system specification. The insertion of PCOs may cause that the system behavior becomes incorrect, i.e. the modified system behavior does not comply with the original system behavior. In this chapter, we will discuss the effects of PCO insertion. Furthermore, we will give a formal proof based on CCS process algebra that PCOs can be inserted while preserving the externally observable behavior.

5.2 OSI Protocol Testing

Design for test & debug in hardware/software systems is still a largely unexplored field. Moreover, the current research initiatives on system-level design for test & debug mainly concentrate on the implementation of test & debug facilities in hardware and/or software. Very few research initiatives deal with design for test & debug during system specification, where the focus is on the implementation-independent, functional system behavior. One of the rare exceptions is the field of telecommunication systems and computer networks.

Much work has been performed on the specification, verification, and testing of communication protocols. The ISO standards provide a framework, the OSI (Open Systems Interconnection) Reference Model, in which an overall architecture for communication protocols in seven layers is defined [ISO94b, HS88, JA90]. Besides protocol specifications, also topics related to testing have been standardized within the OSI framework, in particular protocol conformance testing and test management.

In the OSI terminology, the term PCO is used to indicate any point where a tester can observe or control the system behavior. A PCO usually corresponds to an external system input/output. This definition does not correspond to our notion of a PCO as an internal test & debug function that is inserted to improve observation and control inside the system. Therefore, we initially referred to our test & debug functions as Test & Debug Points (TDPs). However, in more recent OSI standards the term PCO is used also to indicate observation and control facilities inside a system, which matches exactly our definition of TDPs. To comply with the OSI terminology, we replaced the term TDP by the term PCO.

5.2.1 OSI Protocol Conformance Testing

Conformance testing generally indicates testing a system implementation to check whether the externally observable behavior conforms to the system specification. Conformance testing typically means black-box testing: observation and control is performed only at the external inputs and outputs of the system and implementation details about the internal system architecture are not considered or are even unknown.

In the OSI framework, conformance testing indicates verifying by means of testing whether the implementation of an OSI protocol stack conforms to the protocol specifications as stated in the OSI standards. The objective of the OSI standards is to provide an overall architecture for communication systems, so that systems which conform to the OSI standards are able to communicate which each other. OSI conformance testing is a necessary but not sufficient condition to guarantee the correct interworking of system implementations.

A framework for OSI conformance testing has been formalized in ISO Standard 9646 [ISO94c, Kni93]. In this standard, abstract test methods are described for testing an implementation under test (IUT), which can be a single protocol layer, multiple protocol layers, or embedded protocol layer(s) in a protocol stack. Conformance testing of a protocol stack is normally performed in a bottom-up manner: each protocol layer is tested after its underlying protocol layers in the stack have passed their conformance test.

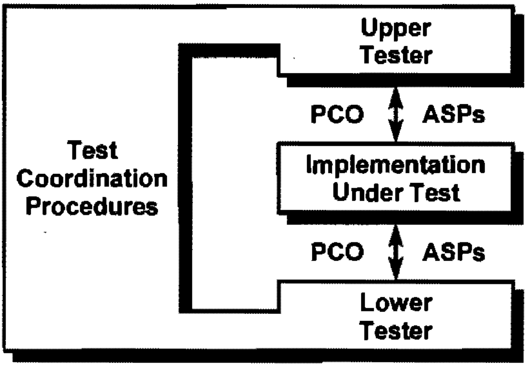


Figure 5.1 Conceptual test method

The basic architecture for the abstract test methods is the conceptual test method, shown in figure 5.1. Testing is performed using a Lower Tester (LT), an Upper Tester (UT), and Test Coordination Procedures (TCPs) to coordinate the operation of the LT and the UT. According to the OSI terminology, the PCOs correspond to the Service Access Points (SAPs) of the IUT and the information exchanged between the IUT and the testers corresponds to the Abstract Service Primitives (ASPs).

The SAPs of a protocol layer are excellent places for observation and control during conformance testing. However, the SAPs in a protocol stack usually cannot be accessed directly by a tester. The abstract test methods in ISO 9646 therefore specify PCOs, which correspond to SAPs that can be accessed directly by a tester. A PCO is called local if it resides within the system under test (SUT), and external if it resides outside of the SUT.

ISO 9646 describes two types of abstract test methods: the local test method and the external test method. In the local test method, both the UT, the LT, the TCPs and the IUT are part of the SUT. The local test method closely resembles the conceptual test method as shown in figure 5.1. The local test method is not applicable for remote testing.

The external test method is suitable for remote testing. As shown in figure 5.2, the external test method can be classified into the distributed, the coordinated, and the remote test method. Each external test method can be used for testing a single protocol layer, multiple protocol layers, or embedded protocol layer(s). TCPs are required to coordinate the tests. The implementation of the TCPs differs in each of the test methods.

- In the distributed test method (figure 5.2a), the LT is external from the SUT. The LT is communicating with the SUT via the service provider. The UT is embedded in the SUT, and a PCO is available between the UT and the IUT. The TCPs are exchanged between the LT and the UT through the IUT and the service provider.
- The coordinated test method (figure 5.2b) is an enhanced version of the distributed test method. There is no PCO between the UT and the IUT. A Test Management Protocol (TMP) is defined for the coordination of the UT and LT.
- In the remote test method (figure 5.2c), there is no explicit UT. The SUT is considered as a black box. Although the remote test method is convenient to use, its control and observation capabilities are limited.

The local test method is preferable, because it allows direct control and observation of the IUT and easy implementation of the TCPs. However, the local test method cannot be used for remote testing. In the external test methods, the remote test method requires no test facilities in the SUT, however it offers inadequate observation and control. The distributed and the coordinated test method require implementation of an UT in the SUT, and they are deficient for guaranteeing synchronization between the UT and the LT. Practical experiences with the abstract test methods resulted in low test coverages, high testing costs and complex test equipment.

An improved test method has been proposed by Zeng et al. [ZR86, ZDH88, ZCS89], introducing the 'ferry-clip concept'. This test method is based on the local test method, but modifications have been made to perform remote testing of the SUT using an external test system. The SUT is equipped with a 'clip', providing access to the lower and the upper interface of the IUT. Test data is transferred between the test system and the clip using a transparent connection ('a ferryboat').

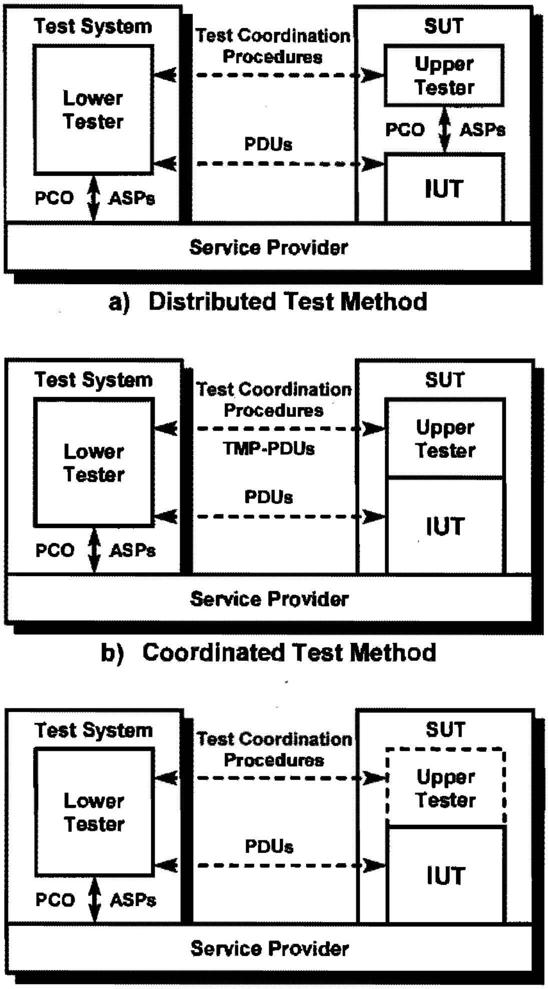


Figure 5.2 External test methods

The ferry-clip concept has been improved further by Witteman & Van Wuijtswinkel at KPN Research. They applied the enhanced ferry-clip concept in an experimental ATM broadband ISDN system, as shown in figure 5.3 [VWvW96, WvW94]. PCOs are inserted to access the external interfaces of the IUT (layers under test). Furthermore, the PCOs allow to disconnect the IUT from its upper and lower layers, so that it can be tested in isolation. The PCOs are managed in the SUT by the Passive Ferry Control, which is connected to the Active Ferry Control in the test system. The test manager in the test system controls the connection between the SUT and the test system, while the test suite manages the actual test cases. The PCOs in the enhanced ferry-clip concept are functionally identical to the PCOs we defined in section 4.4.

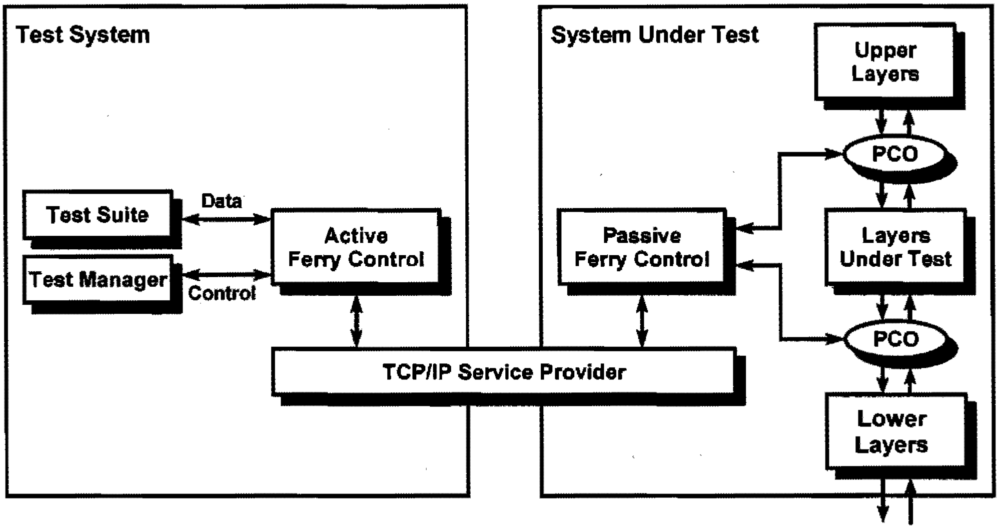


Figure 5.3 Enhanced ferry-clip concept

5.2.2 OSI Test Management

The OSI standards define the exchange of management information between open systems, considering accounting management, configuration management, fault management, performance management and security management [ISO89, ISO92]. Fault management deals with the detection, isolation and correction of faults. One of the functions of fault management is diagnostic testing, which has been standardized in the Test Management Function (TMF) [ISO94a, McR95]. The Confidence and Diagnostic Test Categories [ISO96] specify general test interfaces using the TMF framework. Each test category specifies test characteristics and test management information for a specific test, like connection testing, loopback testing and resource boundary testing.

The Resource Boundary Test Category [ISO96, vWW95] is depicted in figure 5.4. The test conductor in the managing system controls the testing by sending test requests to the test performer in the managed system. The test requests indicate which resources are involved in the test. The tests are carried out by the test performer in the managed system. The Resource Boundary Test Category allows testing of individual resources in the system, using the PCOs at the resource boundaries for observation and control of signals. In the OSI management perspective, resources and PCOs are modelled as objects. Resources are represented by Managed Objects; a resource that is being tested is represented by a Managed Object Referring to Test (MORT); and a PCO is represented by an Associated Object (AO).

The Resource Boundary Test Category can be related to interoperability testing and conformance testing. Each resource represents a protocol layer in a protocol stack, while the PCOs represent the corresponding SAPs. In this configuration, the Resource Boundary Test Category allows to test each protocol layer in isolation.

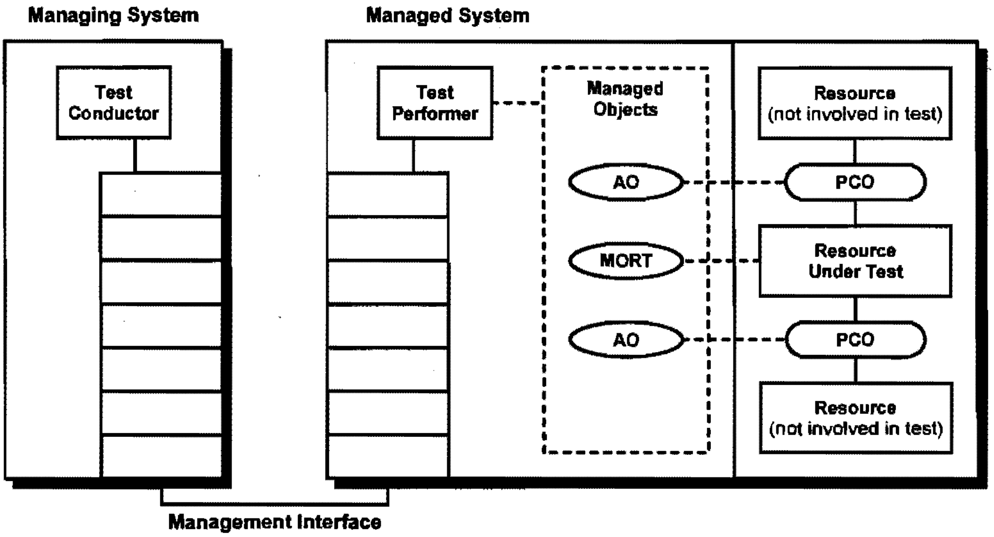


Figure 5.4 Resource Boundary Test Category

5.2.3 Discussion

In this section, we concentrated on one particular application domain: communication systems. The evolution in the OSI standards for remote testing of communication systems, both for conformance testing and diagnostic testing, clearly demonstrates the necessity of design-for-test during system specification. The concept of PCO in the OSI standards corresponds directly to our concept of PCO in the behavioral system specification. The main conclusion is that the communication interfaces between layers in the protocol stack of a communication system are appropriate places for inserting PCOs.

5.3 VLSI Testability Analysis

System testability is related to controllability and observability of the internal system behavior. A general guideline for PCO insertion is therefore to insert PCOs at those points which are most difficult to observe and/or control. These points can be identified by performing testability analysis, i.e. an analysis for evaluating testability characteristics of a system such as observability and controllability. Various approaches to testability analysis have been proposed in literature. In this section we will review testability analysis techniques for digital VLSI circuits.

In the course of time, testability analysis for hardware circuits has evolved from the gate level to the behavioral level. On the gate level, a circuit is described in a structural model as a network of logic gates. On the behavioral level, a circuit is usually described in a mixed structural/functional model: the circuit structure is modeled as a network of modules, while the function of each module is described in a behavioral model. In the following sections we will briefly discuss gate-level and behavioral-level testability analysis.

5.3.1 Gate-Level Testability Analysis

Testability analysis on gate-level descriptions of digital circuits was first introduced in the early 1970s. The basic idea is that testability can be analyzed by measuring controllability and observability of individual lines in a circuit. Testability analysis is performed by considering the circuit structure, i.e. the network topology, and the information propagation through logic gates. The purpose of testability analysis is to provide guidelines for modifying a circuit. A modified circuit should have improved testability, which in turn should lead to a reduction of test generation costs. This approach however requires efficient testability analysis techniques, because the costs for testability analysis may not cancel out the savings in ATPG costs.

Gate-level testability analysis became generally approved with the introduction of SCOAP (Scandia Controllability/Observability Analysis Program) [Gol79, GT80] and related tools. These tools perform testability analysis by computing observability and controllability measures for each line in a circuit. The controllability of a line represents the relative difficulty for justifying the logic value on a line to 0 or 1 from the primary inputs. The observability of a line represents the relative difficulty for propagating the logic value on the line to the primary outputs. In general, large controllability and observability values indicate that a line is difficult to test. Testability can be improved by modifying the circuit, such as adding an observation point in a line with a high observability value or adding control circuitry in a line with a high controllability value (e.g. [CB85]).

Several algorithms have been proposed for calculating the controllability and observability measures. The SCOAP algorithm computes the controllability value for each line by calculating the number of line assignments in the circuit that is required to justify a 0 or 1 from the primary inputs to the line. In a similar way, the observability value for each line is derived by calculating the number of line assignments in the circuit that is required to propagate the logic value on the line to the primary outputs. Several enhancements to the SCOAP algorithm have been proposed, such as [JCDZ86] which considers the influence of reconvergent fanouts. In [Dus78], information theory is used for computing the controllability and observability measures for each line by means of conditional entropies. In [SG76, Gra79], the observability and controllability measures are computed by considering the input-output mapping of logic components.

Although testability analysis on gate-level descriptions is widely used, it has several weaknesses. Statistics indicated that there is only a moderate correlation between SCOAP testability analysis and the ease of fault detection [AM82, MU84]. Furthermore, computing controllability and observability measures may not always identify testability problems [Sav83]. Gate-level testability analysis provides a large amount of controllability and observability values. It is up to the designer to interpret these relative values and to decide whether testability improvement is required or not. Moreover, gate-level testability analysis is performed late in the hardware design flow when redesign may be too costly.

A common DFT technique for sequential circuits is scan design, which provides that the flipflops in a circuit can be chained into a serial shift register (scan chain) during test mode. Scan design allows direct control and observation of the flipflops in the scan chain, and hence it allows to test the combinational and sequential parts of the circuit separately. In full-scan design, all flipflops are included in the scan chain. However, full-scan design may entail silicon area overhead and performance degradation. An alternative is partial-scan design, in which only a subset of the flipflops is included in the scan chain. The methods for selecting a subset of flipflops to be included in the partial-scan chain, can roughly be divided into three categories: methods based on gate-level testability analysis that use observability and controllability measures for selecting partial-scan flipflops; methods based on test generation that exploit the information provided by sequential ATPGs for selecting partial-scan flipflops; and methods based on structural analysis which select flipflops to break feedback cycles in the circuit.

Experiments on sequential circuits showed that test generation complexity grows exponentially with the length of feedback cycles in a sequential circuit, while this complexity grows only linearly with sequential depth [CA90]. A feedback cycle is constituted of a flipflop whose output is fed back into the flipflop's input; sequential depth is the largest number of flipflops on a path between primary inputs and outputs. Similar views are stated in [HS89, GMG90], where feedback loops and reconvergent fanouts are identified as major testability bottlenecks for sequential ATPG. However, more recent experiments somewhat weaken the importance of feedback cycles to sequential ATPG complexity [MEMMR95]. Several algorithms for selecting partial-scan flipflops to break feedback cycles have been proposed [CA90, LR90, PA92, AM94, CBA94].

5.3.2 Behavioral-Level Testability Analysis

Testability analysis on behavioral hardware descriptions offers several advantages over gate-level testability analysis. Behavioral-level testability analysis can be performed earlier in the design flow before synthesis, and it is able to cope with the increasing complexity of VLSI circuits. Testability analysis on the behavioral level considers both the circuit structure, i.e. the topology of interconnected modules, and the information propagation through modules. Numerous approaches have been proposed for behavioral-level testability analysis.

- A simplified approach is proposed in [SGP83], considering only the network topology while ignoring information propagation through modules.

- In TDES (Testable Design Expert System) [AB85], a circuit is modeled as a network of modules. Testability analysis is performed by identifying paths for information propagation through a module (the module identity mode: I-mode), and paths for propagating information from one module to another module (I-path).

- A precise approach is described in [MH88, MH91, Mur94], where an algebraic theory of propagation is introduced that is used for hierarchical test generation and design-for-test. The theory of propagation provides a formal description of the propagation characteristics of a module, together with algebraic operations to calculate the propagation characteristics of a network of modules.

- In [JK93], the controllability and observability measures of SCOAP are extended to the functional level. A circuit is modeled as a graph, where nodes represent modules and edges represent interconnections between modules. Each module is described by its inputs, its outputs, and a boolean function which models the dependencies between inputs and outputs.

- In [TA89], information theory is proposed for testability analysis on data-flow graphs. The nodes in the graph correspond to functional modules and the edges represent signal paths. Entropy and mutual information are used to quantify the amount of information that is transferred from a primary input to a node, and from a node to a primary output.

- In [VAA92, VVA93], testability analysis is performed during high-level synthesis. A behavioral VHDL description is analyzed first by ATKET (Automatic Test Knowledge Extraction Tool), which computes test modes (i.e. initialization mode, propagation mode, hold mode, and status mode) for each module. Based on the evaluation of these test modes, the behavioral description may be modified. Next, the behavioral description is input into a high-level synthesis tool, performing scheduling and allocation of hardware resources. The synthesized design is analyzed by ATKET again. Testability bottlenecks may be removed by performing re-scheduling or re-allocation.

- In [CM89, CWS91, CS92, CS93, CKS94], testability analysis is performed by BETA (Behavioral-level Testability Analyzer). BETA first performs path analysis on the control-flow graph (CFG) of a circuit. Next, paths are examined for controlling and observing variables, and each variable is classified as completely or partially controllable, and completely or partially observable. This information is used to select test points or partial-scan flipflops. Alternatively, test statement insertion in the CFG is proposed to improve testability by making variables completely controllable or observable in test mode.

- A common structure for hardware circuits is a datapath controlled by a control section. In [AM89], an approach for symbolic test generation is proposed. The datapath is described as an interconnection of functional modules and represented as a graph. Each node in the graph has a behavioral path model, which describes a propagation mode, a justification mode, and a status mode. The control section is defined as a finite-state machine, described by a state table. The test generation algorithm determines a symbolic test path for each target module in the datapath. The symbolic test path describes constraints for the primary inputs, and the sequence in the control section to activate the test path.

- In [LP93], testability analysis is proposed for microprocessor circuits which may be generally modeled as a datapath and a control section. The circuit behavior is represented in a structural data-flow graph (SDG). For each assembly instruction, a SDG represents the modules in the datapath that are involved in the instruction and the data-flow between them. Paths are computed to justify the inputs of a module and to propagate the module outputs to a primary output of the circuit. The test generation process is separated into path analysis and value analysis. During path analysis, an assembly instruction sequence is generated to apply the test vector to the module under test and to allow the fault effect to be observed at a primary output. During value analysis, the values at all modules are calculated. Testability analysis is used to measure the difficulty of generating an assembly instruction sequence and value assignments.

- In [CLP92a, CLP92b, Chi93, C$^+$94b], a circuit is hierarchically described in VHDL as a datapath and a control section. The circuit description is transformed into an Execution Flow Graph (EFG), which formalizes the data flow and the interaction between the datapath and the control section. For each module in the datapath, an EFG is generated, and controllability and observability sequence ranges are calculated. The controllability sequence range is an estimation of the minimum and maximum number of instruction cycles required to set a give value. The observability sequence range is an estimation of the minimum and maximum number of instruction cycles required to observe any value. By combining the sequence ranges of all modules, an overall testability sequence range is obtained, which is used for selecting flipflops in a partial-scan approach.

5.3.3 VLSI Testability Analysis and PCO Insertion

We conclude from the previous that testability analysis of VLSI circuits is widely applied to guide design-for-test. Testability analysis is performed by analyzing both the circuit structure, i.e. the topology of interconnected modules, and the information propagation through modules. In gate-level testability analysis, a module corresponds to a basic logic gate, while in behavioral-level testability analysis a module corresponds to a behavioral description of combinational or sequential logic.

An interesting question is whether the methods for VLSI testability analysis can be applied to guide PCO insertion in a system specification. A system specification typically consists of a model of concurrent, communicating processes. Testability analysis may be performed by considering both the structure of the specification, i.e. the topology of interconnected, communicating processes, and the information propagation through each process.

5.3.3.1 Analysis of Information Propagation

The effort for analyzing the structure of a system specification is to some extent comparable to the effort for analyzing the structure of a VLSI circuit. However, analyzing the information propagation through a process is usually much more complex than analyzing the information propagation through a logic gate or a relatively simple combinational or sequential hardware module. The behavioral specification of a process typically incorporates a non-trivial control flow and data flow. Analyzing the information propagation in the behavioral specification of a process is comparable to path analysis in software. We indicated in section 3.7.2 that it is generally impossible to analyze all control-flow and data-flow paths through a software module. Hence, in general testability analysis of the system specification cannot be performed efficiently when considering the information propagation through processes.

In chapter 7 we will discuss experiments on an elevator control system as a case study. We specified the elevator control system using the POOSL language. The POOSL specification consists of nine concurrently operating, communicating processes and all nine processes are of similar complexity. Figure 5.5 shows the control flow of one of the processes named *Individual Elevator Control*. Each node in the control-flow graph corresponds to a statement in the POOSL specification; for the sake of clarity we annotated the statements in the POOSL specification with line numbers. The POOSL specification of the process consists of a number of methods, indicated by the grey boxes, that call each other.

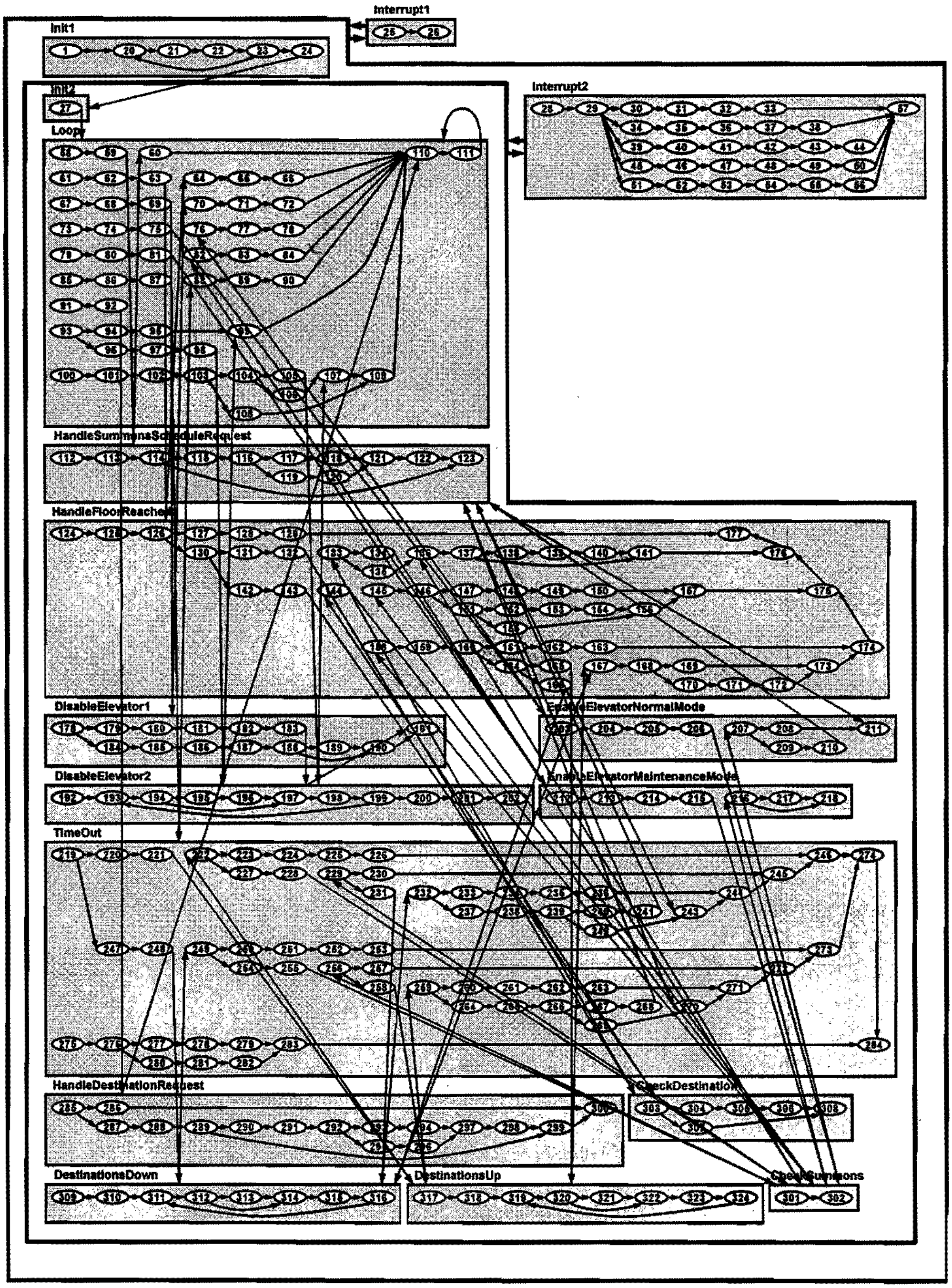


Figure 5.5 Control-flow graph of POOSL process 'Individual Elevator Control'

In figure 5.5, there are only 62 basic control-flow paths between the initial node 1 and node 111 in the method *loop*. However, these 62 basic control-flow paths do not include any loop statements, tail recursion and interrupts, because this would lead to an exponential increase in the number of control-flow paths as explained below.

- Counting in loops causes an exponential increase in the number of control-flow paths. For instance, the method *disableElevator2* contains the following loop:

  ```
  (192) a:=1;
  (193) while (a<=40)
  (194) do if ( ... )
  (195)     then ...
  (196)          ...
  (197)     fi;
  (198)     a:=a+1;
  (199) od;
  ```

  There are 2^{40} control-flow paths through this piece of code due to the if-statement in the loop.

- A POOSL process typically exhibits infinite, non-terminating behavior, which is achieved by tail recursion. The process in figure 5.5 contains tail recursion in the method *loop*: node 111, the last statement of the method *loop*, recursively calls the method *loop*. The 62 basic control-flow paths between node 1 and node 111 are simple paths, because they do not incorporate the tail recursion.

- A POOSL process can be interrupted, which is specified by an interrupt or an abort statement. An interrupt or abort statement is used to specify that a process should be willing to accept some message at any instant. The process in figure 5.5 has two levels of interrupts. *Interrupt2* may interrupt all methods except for the method *init1*, while *interrupt1* may interrupt all methods. (Note that *interrupt1* may interrupt *interrupt2*.)

  Considering interrupts causes an enormous increase in the number of control-flow paths. For instance, the statement $(ch?n_1; ch?n_2; ch?n_3)$ *interrupt* $ch?n$ is equivalent to the statement $(ch?n_1; m_2)$ *or* $(ch?n; m_1)$, where m_1, m_2 and m_3 are methods defined as:
  $m_1 = (ch?n_1; m_2)\ or\ (ch?n; m_1)$
  $m_2 = (ch?n_2; m_3)\ or\ (ch?n; m_2)$
  $m_3 = (ch?n_3)\ or\ (ch?n; m_3)$
  Obviously, it is impossible to give an upper bound for the number of control-flow paths through this statement.

It can be concluded that it is unfeasible to give an upper bound for the number of control-flow paths when considering loop statements, tail recursion and interrupts. The example in figure 5.5 clearly demonstrates that analysis of information propagation through a process by means of exhaustive path analysis cannot be performed efficiently. Hence, testability analysis of a system specification cannot be based on an analysis of information propagation.

5.3.3.2 Analysis of Specification Structure

A simplified approach to testability analysis of a system specification is to consider only the structure of the specification, while ignoring information propagation through processes. The structure of a specification can be represented in a directed graph (digraph), where each node represents a process and each directed edge represents communication between a sending process and a receiving process.

A directed graph $G = (V, E)$ consists of a set of vertices V and a set of directed edges E. A path in a directed graph is a sequence of vertices $v_1, \ldots, v_n$ such that $v_1 \rightarrow v_2, \ldots, v_{n-1} \rightarrow v_n$ are edges. The length of the path is the number of edges on the path. A path is simple if all vertices on the path, except for possibly the first and last, are distinct. A simple cycle is a simple path of length at least one that begins and ends at the same vertex. The number of vertices in graph G is $n = |V|$, and the number of edges is $e = |E|$.

Testability analysis can be performed by analyzing the structure of the graph. There are various well-known algorithms for analyzing directed graphs:

- The single-source shortest paths algorithm computes the shortest paths in a digraph from a source vertex to each vertex in V. The notion of shortest path can be extended to the path with minimum cost when each edge has an associated label representing the cost. The cost of a path is the sum of the costs of the edges on the path. An efficient algorithm of complexity $O(e \log n)$ has been developed by Dijkstra [Dij59]. A related algorithm of complexity $O(en \log n)$ has been developed by Floyd [Flo62] to determine the shortest paths between any pair of vertices in the graph.
- The eccentricity of a vertex v in a digraph is defined as the maximum length of the shortest path from any vertex w to v. The center of G is the vertex of minimum eccentricity, which is the vertex that is closest to the vertex most distant from it. The eccentricity of vertices and the center of the graph can be determined efficiently by Floyd's algorithm.
- A strongly connected component of a digraph is a maximal set of vertices $S \subseteq V$ in which there is a path between any two vertices in S. The strongly connected components in a digraph can be determined efficiently by depth-first search traversal of the graph. An efficient algorithm has been developed by Tarjan [Tar72], computing the strongly connected components of a graph in a single depth-first search traversal.

  Identifying the strongly connected components in a digraph corresponds to identifying feedback loops in the graph. The problem of identifying strongly connected components is of linear-time complexity. However, the problem of finding a minimal feedback vertex set (MFVS) to break all feedback cycles is NP-complete [AHU74]. Heuristic and exact algorithms for identification of the MFVS have been proposed in [SW75, LSW88], and in research on selecting partial-scan flipflops to break feedback cycles in sequential circuits [CA90, LR90, PA92, AM94, CBA94].

Testability analysis can be performed by analyzing the observability and controllability of each node in the digraph. The controllability of a node can be determined by computing the eccentricity of the node with respect to the system inputs, and the observability of a node can be determined by computing the eccentricity of the node with respect to the system outputs. Analysis of the graph structure may provide useful information. However, the behavioral view on the system is not captured and consequently the results of testability analysis may be inaccurate. For instance, figure 5.6 shows the digraph representation of the specification of the elevator control system. The nodes $P_1, \ldots, P_9$ correspond to the nine POOSL process objects; the nodes $I_1, \ldots, I_9$ correspond to the system inputs; and the nodes $O_1, \ldots, O_9$ correspond to the system outputs. Figure 5.7 shows the length of the shortest path between each process node and the system inputs and outputs, and the eccentricity of each process node with respect to the system inputs (E_I) and the system outputs (E_O). It follows from figure 5.7 that the nodes P_3, P_5, P_7 and P_9 have the largest eccentricity, and hence they are most difficult to control and observe from the system inputs and outputs. For instance, process P_3 is difficult to access because of paths like $I_5 \rightarrow P_3$ and $P_3 \rightarrow O_5$ of length 5. However, P_3 and P_5 are behaviorally independent processes, and hence process P_3 is not correlated to I_5 and O_5. This example clearly demonstrates that a pure structural analysis of the specification, which ignores the behavioral view, is inadequate for testability analysis. The structural analysis can be improved by considering only those paths that can actually be traversed. This would eliminate the paths $I_5 \rightarrow P_3$ and $P_3 \rightarrow O_5$. We will elaborate further on this improved approach in section 5.5.

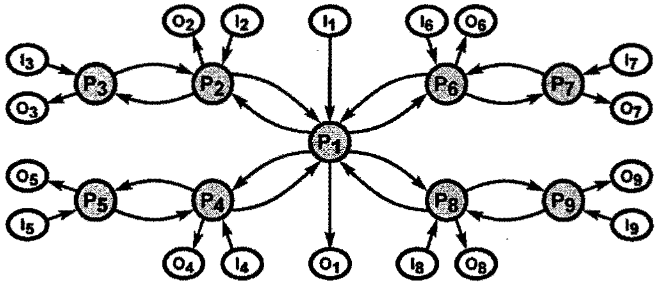


Figure 5.6 Digraph of the specification structure of the Elevator Control System

	I_1, O_1	I_2, O_2	I_3, O_3	I_4, O_4	I_5, O_5	I_6, O_6	I_7, O_7	I_8, O_8	I_9, O_9	E_I, E_O
P_1	1	2	3	2	3	2	3	2	3	3
P_2	2	1	2	3	4	3	4	3	4	4
P_3	3	2	1	4	5	4	5	4	5	5
P_4	2	3	4	1	2	3	4	3	4	4
P_5	3	4	5	2	1	4	5	4	5	5
P_6	2	3	4	3	4	1	2	3	4	4
P_7	3	4	5	4	5	2	1	4	5	5
P_8	2	3	4	3	4	3	4	1	2	4
P_9	3	4	5	4	5	4	5	2	1	5

Figure 5.7 Shortest paths and eccentricity

Another aspect of structural analysis is to identify feedback loops in a specification. Feedback loops are typically key contributors to the complexity of a system specification. PCOs may be inserted to break feedback cycles. This approach is analogous to selecting partial-scan flipflops for breaking feedback cycles. For instance, a minimum feedback vertex set (MFVS) for the digraph in figure 5.6 is $\{P_2, P_4, P_6, P_8\}$. Removing the nodes in the MFVS eliminates all cycles in the digraph. However, also in this case structural analysis alone is insufficient for guiding PCO insertion because the behavioral view on the system is ignored.

5.3.3.3 Discussion

In this section, we argued that techniques for VLSI testability analysis cannot be applied efficiently to guide PCO insertion in a specification. The analysis of information propagation through processes in a system specification implies analysis of all control-flow and data-flow paths in a process, which is generally impossible to achieve. Analysis of the specification structure while ignoring the information propagation through processes, yields inaccurate results because the behavioral view on the system is not taken into account. The only reasonable approach is to analyze the specification structure and to identify a limited set of paths through the specification. This implies that for each process a limited set of paths is analyzed. This approach is elaborated further in section 5.5.

Another topic that impedes testability analysis is that the system specification consists of a number of concurrent processes. These processes operate in parallel, and hence the control flow in the system specification is in fact composed of the control flows of all the concurrent processes in the system specification. The control flows of the individual processes are closely related, because the processes communicate with each other. Hence, testability analysis of the system specification should in fact analyze concurrent, interacting control-flow paths.

5.4 System-Level Testability Analysis

System-level testability analysis has been proposed by Sheppard & Simpson (ARINC, USA) in their work on integrated diagnostics, and by Robach et al. (LSR-IMAG, France) in their work on testability-oriented hardware/software partitioning.

5.4.1 Testability Analysis for Integrated Diagnostics

Sheppard & Simpson developed an approach towards integrated diagnostics for US military systems [SS91b, SS91a, SS92b, SS92a, SS93b, SS93a, SS94a]. Until the 1980s, the design of military systems concentrated on meeting performance constraints, while field maintenance and diagnostics were hardly considered. Consequently, systems in the field showed retest-ok rates over 40%, no-fault-found rates over 50%, and false-alarm rates exceeding valid alarm rates. To deal with these problems, a shift took place from design-for-performance to design-for-field operation. Initially, this resulted in ad hoc design modifications that did not lead to significant improvements in the field. A more structured approach has been introduced by Sheppard & Simpson. Although their primary application domain is US military systems, their approach is generally applicable.

The Sheppard & Simpson approach is based on the assumption that a failure during field operation is due to a physical hardware fault. The fault can be diagnosed by applying a particular set of test

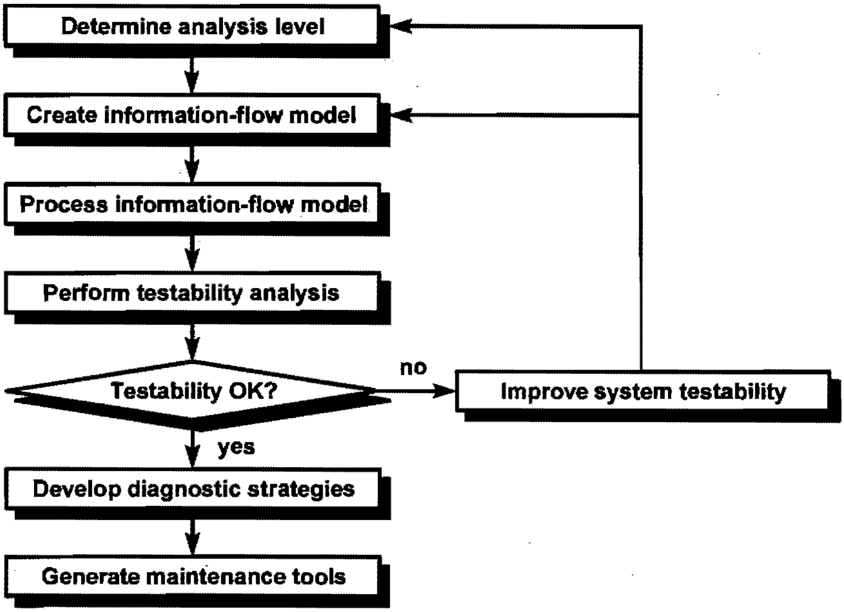

Determine analysis level
Create information-flow model
Process information-flow model
Perform testability analysis
Testability OK?
no
Improve system testability
yes
Develop diagnostic strategies
Generate maintenance tools

Figure 5.8 Sheppard & Simpson approach to integrated diagnostics

cases to the system, possibly using the system's built-in test facilities. The Sheppard & Simpson approach deals with two key issues: testability analysis, which means analyzing how well a fault can be diagnosed by applying a set of tests, and developing efficient diagnostic strategies. The approach is based on creating a mathematical model of a system. Figure 5.8 shows the subsequent stages of this approach.

First the analysis level is determined. This level defines the smallest isolatable part during diagnosis, such as an IC, a PCB, or a collection of PCBs. Next an information-flow model of the system is created. The information-flow model is a directed graph, where each node represents a test or a fault-isolation conclusion, and each edge represents a (first-order) dependency between a test and a fault-isolation conclusion. Tests are regarded as information sources, and for each test the consequences of a pass or fail are evaluated. A fault-isolation conclusion corresponds to an element that can be isolated, like a PCB or an individual component. The purpose of the information-flow model is to reason about combinations of tests for diagnosing faults. When the information-model is completed, it is processed which implies calculating higher order dependencies between tests and fault-conclusions in the information-flow model.

The information-flow model is used to identify several characteristics of system testability, such as conclusion ambiguity and test redundancy. Conclusion ambiguity indicates that a set of tests cannot distinguish among two or more fault-conclusions. Test redundancy indicates that two or more tests provide identical information. Testability analysis is used to evaluate system testability. If the system testability is inadequate, an iteration step is performed to improve testability. Testability can be improved by specifying additional test cases and/or by removing redundant test cases. If testability analysis is performed before the system is actually implemented, redesign can be performed. However, redesign usually requires extra costs and may affect the system performance. Furthermore, redesign may solve the current testability problems but at the same time it may introduce new testability problems. A simple form of redesign is to repack the system in such a way that all the testability problems are grouped into a single isolatable unit.

When an adequate level of system testability is reached, diagnostic strategies are developed. A diagnostic strategy consists of a particular set of tests and the order in which these tests are applied in order to locate a fault. Optimization is required to minimize the number of tests needed for isolating a fault. The problem of constructing an optimal diagnostic strategy is known to be NP-complete, and therefore exhaustive search is unfeasible. Sheppard & Simpson proposed a number of heuristics of which an entropy-directed search offers the best results. Finally, the diagnostics strategies are incorporated in maintenance tools, like technical manuals and automated test equipment.

The Sheppard & Simpson approach concentrates on providing diagnostic strategies for dealing with hardware faults. The system architecture, the built-in test facilities, and the test cases are all modeled to some extent in the information-flow model. Hence, the information-flow model and testability analysis are based on the physical implementation of the system. Our design for test & debug approach is completely different. We concentrate on hardware/software design faults, in particular on interfacing faults, and we propose testability analysis on the implementation-independent system specification. The main conclusion is that both approaches are complementary. Our approach focuses on design for test & debug during system specification and architecture design, while the Sheppard & Simpson approach concentrates on deriving diagnostic strategies from the physical implementation. However, this also implies that the techniques for testability analysis proposed by Sheppard & Simpson are not applicable for guiding PCO insertion.

5.4.2 Testability Analysis for Hardware/Software Partitioning

Robach et al. propose system-level testability analysis to guide hardware/software partitioning. Initially, Robach et al. used Information Transfer Graphs (ITG) [TR95b, TR95a, AHLTR96, AHTR97] to model hardware circuits. An ITG is a directed graph that is closely related to the data-flow model. Each node corresponds to a function of the circuit and each edge models the possibility to transfer information from one node to another one. The SATAN (System's Automatic Testability ANalysis) tool [RMM84] is used for testability analysis and automatic generation of test cases from the ITG. This process is performed in three steps. First, the information flows in the ITG are identified. An information flow is a path in the ITG from inputs to output. Next, a set of information flows is selected to be used as test cases. Finally, a testability measure is computed. Each node in the ITG corresponds to a function or module. A weight is assigned to each node to represent the effort for testing the node. System testability is measured as a function of the weights of the nodes in all information flows in the test set. In later work, the ITG is replaced by the control-flow graph (CFG) [THR96, AHTR97].

Each node in an ITG or a CFG represents some functional behavior that can be implemented in hardware or in software. There are two weights assigned to each node, which represent the testing effort, i.e. the number of test cases, for respectively the hardware implementation and the software implementation of the node [AHLTR96, HR96a, HR96b, THR96, AHTR97]. The functional behavior of a node is specified in behavioral VHDL, and this behavioral VHDL specification is regarded as the software implementation of the node. Robach et al. propose mutation analysis to derive a test set for the software implementation. Mutation analysis is based on creating a set of faulty versions or mutants of the software implementation. Each mutant differs from the correct software implementation by a single, unique and syntactically correct fault. Fault classes can be modeled by a set of mutation operators. A mutant is created by applying a mutation operator to the original software implementation. Robach et al. defined a set of simple mutation operators for behavioral VHDL descriptions, such as changing an arithmetic, logic, relational, or unary operator, or replacing a constant or a variable by another constant or variable. The test set for the hardware implementation of the node is derived in exactly the same way, however additional test cases are defined to detect physical hardware faults.

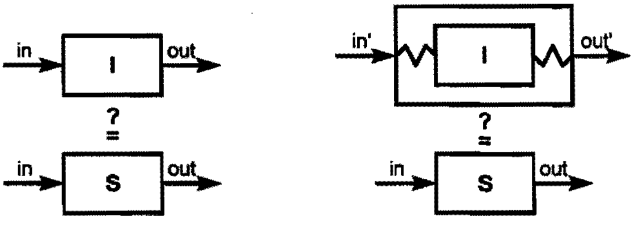
Various hardware/software partitionings can be obtained by considering the hardware or software implementation for each node in the ITG or CFG of the system. The testability of each hardware/software partitioning is computed by considering the paths through the ITG or CFG and by the weights of the individual nodes.

Robach et al. propose an interesting approach to address system testability, however several weaknesses can be identified. They assume that a behavioral specification of a hardware module is comparable to a software program. This is a valid argument to some extent, however the reversed argument is definitely not valid. The control flow and data flow of a software program or a system specification is typically much more complex than the behavioral specification of a hardware module. We clearly demonstrated this in section 5.3.3.1 where we discussed the complexity of a POOSL process. In section 5.3, we also concluded that path analysis and information propagation cannot be applied efficiently for testability analysis of behavioral system specifications. However, the approach of Robach et al. is based on exhaustive path analysis. Hence, the work of Robach is applicable to hardware systems, but it does not truly address testability of hardware/software systems.

Robach et al. propose mutation analysis for obtaining a test set, focusing on simple design faults, such as incorrect arithmetic, logic, relational or unary operators and incorrect variables or constants. It is assumed that more complex design faults can be detected by the same test set. However, in chapter 3 we argued that testing of hardware/software systems should concentrate on interfacing faults. Faults like deadlocks and race conditions may not be identified easily by a test set that is obtained from mutation analysis. Moreover, mutation analysis as proposed by Robach et al. implies that testing a hardware implementation always requires more effort than testing a software implementation.

5.4.3 Formal Analysis of System Testability

In [DASC93] a formal analysis is presented on conformance testability of embedded components in a system. Conformance testing of a component in isolation implies checking whether the externally observable behavior of the component's implementation I is conforming to the component's specification S. The relation $I\ conf\ S$ indicates that implementation I conforms to specification S when tested in isolation, as indicated in figure 5.9a.



a) *I conf S ?* b) *I confE S ?*

Figure 5.9 *Conformance testing of a component*

In [DASC93] is analyzed how conformance testability of a component may degrade when the component is embedded in a system. The embedded component cannot be accessed directly from the system environment due to its surrounding components in the system. Consequently, testing the embedded component implementation I for conformance with respect to its specification S requires that test stimuli and test responses are passed through the surrounding components. The relation $I\ conf_E\ S$ denotes that the embedded implementation I conforms to specification S, as indicated in figure 5.9b.

Testing an embedded component in a system for conformance may lead to four possible conclusions:

1. $(I\ conf\ S)$ and $(I\ conf_E\ S)$
2. $\neg(I\ conf\ S)$ and $\neg(I\ conf_E\ S)$
3. $(I\ conf\ S)$ and $\neg(I\ conf_E\ S)$
4. $\neg(I\ conf\ S)$ and $(I\ conf_E\ S)$

The conclusions 1 and 2 indicate correct situations: if the component is found to be conforming or non-conforming to its specification when tested in isolation, then the same result is obtained when testing the embedded component.

Conclusion 3 indicates that a conforming implementation may become non-conforming when embedded in the system. Conformance testing aims at verifying the behavior of the implementation with respect to the specified behavior. However, conformance testing does not verify the implementation for non-specified behavior. (Conformance testing does not include robustness testing.) When embedded in the system, non-specified inputs or input sequences may be offered to the component, which imposes non-specified behavior to the component that may lead to the conclusion of non-conformance.

Conclusion 4 indicates that a non-conforming implementation may become conforming when embedded in the system. It may be impossible to offer (specified) inputs or input sequences to the embedded component which would impose non-conforming behavior to the component. Hence, due to testability degradation, the non-conforming behavior may be non-detectable.

In [DASC93], a formal approach using refusal graphs is proposed for finding the limits of testability degradation when testing embedded components. This approach is based on Brinksma's formal definition of conformance testing [BSS87, Bri88].

5.4.4 Discussion

In this section, we discussed three approaches to system-level testability analysis. The approach of Sheppard & Simpson concentrates on diagnostic strategies for hardware systems. We showed that this approach is complementary to our design for test & debug approach. Consequently, their testability analysis method is not applicable to guide PCO insertion in a system specification.

We argued that the approach of Robach et al. does not truly address hardware/software systems. Their testability analysis method is applicable to hardware systems, but it is not suited for guiding PCO insertion in a system specification.

Finally, we provided a theoretic discussion on testability degradation of embedded components during conformance testing. Although this discussion did not provide any guidelines for PCO insertion, it clearly demonstrated how the testability of individual components may degrade when embedded in a system. A plausible approach is to insert PCOs at the boundaries of those embedded components that suffer from testability degradation. This approach would require an extension of the formal framework in [DASC93]. However, we did not study this approach any further.

5.5 Scenario-Based PCO Insertion

In section 4.8 we elaborated on the role of scenarios during system specification. We showed that scenarios are key elements for creating the system specification. Moreover, we argued that scenarios are well suited to be used as system-level test cases, both during validation and verification of the system specification, as well as during testing of the system implementation.

In this section we will show that scenario-based testability analysis of the system specification is suited for guiding PCO insertion. The system behavior is defined by a set of scenarios. Each scenario defines some specific system behavior. A scenario is defined by events that occur in the system environment to which the system must respond, the subsequent behavior of the processes in the system and their interactions, and the responses of the system. In general, only a subset of all the processes and communication channels in the system is involved in a particular scenario.

The goal of scenario-based testability analysis is to identify a subset of processes and communication channels that are involved in a particular scenario and that provide essential information on the system behavior. The essential information should be sufficient for an external observer to explain the system behavior.

When executing a particular scenario, the involved processes will interact and exchange messages. Whenever a process receives a message, the process will typically evaluate both its current state and the input message, and subsequently the process will generate output messages and modify its state. Usually, only part of the process state is relevant for evaluating the input message and for deciding what the subsequent behavior of the process will be, and only part of the process state is modified. Hence, the essential information is that part of the process state which is evaluated and modified during the scenario. Also the input and/or output messages of the process (i.e. communication channels) can be identified as essential information that is required to explain the behavior of the process.

The second element of scenario-based testability analysis is to analyze the external visibility of the relevant processes and communication channels. If the essential state information of a relevant process cannot be easily determined by observing just the external system behavior, then inserting a PCO to access the process state information is desirable. Likewise, if the messages on a relevant communication channel cannot be easily determined by observing just the external system behavior, then inserting a PCO to access the communication channel is desirable.

In summary, scenario-based PCO insertion addresses two basic questions for each scenario:

- What is the essential information in the system for the particular scenario, which allows an external observer to explain the system behavior?
- How well is this essential information visible to an external observer? If the information is not reasonably visible to the external observer, then PCO insertion is desirable.

A scenario can be regarded as a path through the system specification in which multiple processes are involved. We showed in section 5.3.3.1 that an exhaustive path analysis through a single process is unfeasible due to the large number of control-flow and data-flow paths. Furthermore, we showed that the system specification consists of a number of concurrent, communicating processes and hence the control flow through the system specification is composed of the control flows of all the concurrent processes. A scenario can be regarded as a single path through the system specification, which is composed of subpaths through the processes that are involved in the scenario. Hence, for each process only a limited number of paths is considered for one particular scenario.

We showed in section 5.3.3.1 that the number of control-flow paths through a process may become very large when considering loops, tail recursion and interrupts. We limit the number of paths during scenario-based testability analysis in the following way:

- We do not perform exhaustive path analysis through a loop. Usually it suffices to consider loop invariants, pre- and postconditions, and/or boundary conditions to extract the essential information in the loop. For instance, the example in section 5.3.3.1 contained the following loop:

```
a:=1;
while (a<=40)
do if (destinations get(a))
   then db!indicateDestination(a,false);
        destinations put(a,false)
   fi;
   a:=a+1;
od;
```

The postcondition of this loop is: $\forall a : 1 \leq a \leq 40 : destinations[a]{=}false$. The essential information in this loop is provided by the array *destinations*. However, the essential information is directly visible in the system environment, because changes in *destinations* are communicated on the channel *db* which is connected to objects in the system environment. (In fact, the statement *db!indicateDestination(a,false)* models that the light of a destination request button in the elevator cage is turned off.)

- A scenario indicates how many times tail-recursive calls need to be evaluated. For instance, the method *loop* in figure 5.5 contains tail recursion. A particular scenario is that the system operator first enables an elevator and next disables the elevator. This scenario will traverse the path from node 73 to 111 (enabling the elevator), a tail-recursive call to node 67, and the path from node 67 to 111 (disabling the elevator). Hence, the scenario indicates which paths are traversed in the two subsequent executions of the method *loop*.
- In a similar way, the scenario indicates when interrupts should be considered. For instance, a particular scenario defines the system behavior when a passenger presses a summons button. In figure 5.5, pressing a summons button will cause an interrupt (node 25). The interrupt behavior will subsequently modify a variable (node 26). In the method *loop*, this variable is only considered in the path from node 58 to node 111, and not in all other paths through the method *loop*. Hence, *interrupt1* in figure 5.5 needs only to be considered in this particular scenario that traverses the path between node 58 and 111, and may be ignored in all other scenarios.

A remaining question is the completeness of scenario-based testability analysis. As indicated, scenario-based testability analysis traverses only a limited number of paths through the system specification for each individual scenario. However, analysis of the individual scenarios results in traversal of all relevant paths. For instance, we stated in section 5.3.3.1 that there are 62 basic control-flow paths through the process in figure 5.5. When scenario-based testability analysis of all the individual scenarios is completed, all 62 basic paths will have been traversed. Furthermore, additional paths will be traversed that contain relevant loops, tail recursion and interrupts.

It should be noted that scenario-based analysis of individual scenarios is sufficient for PCO insertion. However, this approach does not provide an exhaustive analysis. Consequently, it is insufficient for test case generation, because the execution of multiple, simultaneous scenarios in the system should be considered during testing. Scenario-based PCO insertion provides that PCOs are inserted to access the essential information for each individual scenario. Hence, even if there are multiple, simultaneous scenarios being executed in the system, the PCOs provide that the essential information of each individual scenario is visible to an external observer.

5.5.1 Example of Scenario-Based PCO Insertion

In the case study on the elevator control system, one particular scenario defines the system behavior when a passenger in an elevator cage presses a button to indicate that he wants to be transported to a particular floor. Pressing the particular button is modeled in the system specification as a *destinationRequest(Floor)* message that is input to the system. The parameter *Floor* indicates the floornumber where the passenger wants to be transported to. The subsequent system behavior is as follows:

- If the elevator is halted and if the elevator is not positioned at the floor of the destination request, then the elevator will be scheduled to service the destination request. The elevator doors will be closed, and subsequently the elevator will start moving.

- If the elevator is halted or stopped and if the elevator is positioned at the same floor as specified in *Floor*, then the system will generate no responses.

- If the elevator is disabled, then the system will not generate responses.

- If the elevator is moving or if the elevator is stopped at a floor differing from *Floor*, then the destination request is stored and no explicit scheduling is required.

It is easy to see that the essential information in this scenario is the elevator state (which can be halted, stopped, moving, or disabled) and the current floor at which the elevator is positioned. In the external environment, the current floor of the elevator is directly visible, because each elevator is equipped with arrival lights that indicate the current position of the elevator. However, the elevator state is not directly visible in the system environment. For instance, an external observer who observes an elevator that is waiting on a particular floor, is unable to determine whether the elevator is halted or stopped. Hence, it is desirable to insert a PCO in the system specification so that the elevator state can be observed and/or controlled. The specification of the elevator control system will be discussed in depth in chapter 7. We will show in chapter 7 that the elevator state is contained in a process named *Individual Elevator Control*. This process will indeed be equipped with a PCO to access the elevator state. Further examples of scenario-based PCO insertion will be provided in chapter 7.

5.6 Effects of PCO Insertion

5.6.1 Formal Analysis of PCO Insertion

In this section, we will examine the effects of PCO insertion on the system specification. The insertion of PCOs implies that the system specification is modified. However, the modified system specification should still be correct with respect to the original system specification: the insertion of PCOs in the system specification should not induce faulty behavior. In order to examine the effects of PCO insertion, we consider the system specification to be a collection of concurrently operating, communicating processes. We use the CCS process algebra [Mil80, Mil89, Koo91] to describe such a system specification for the following reasons:

- Process algebra provides a formal, mathematical theory on communication and concurrency. The behavior of a system is typically expressed in process algebra as a composition of concurrent, communicating processes. This corresponds directly to our view on the system specification.

- Process algebra provides a formal, mathematical theory on equivalence relations between systems. Hence, process algebra allows to formally express whether two systems are equivalent or not and under what conditions this equivalence holds. The effects of PCO insertion can be studied by examining whether the system behavior including PCOs is equivalent to the system behavior without PCOs.

- There is variety of process algebras, such as ACP, CCS, CIRCAL and LOTOS, which have slightly different semantics and characteristics. In this thesis we will use Milner's CCS process algebra. This choice is mainly motivated by our focus on the POOSL language for system specification. POOSL is largely based on CCS.

The previous arguments clearly show that CCS process algebra is a valid choice for analyzing the effects of PCO insertion. However, there are some restrictions. In CCS process algebra, the communication between processes is based solely on synchronous pair-wise message passing. Furthermore, in CCS process algebra there is no notion of time. Only the ordering of actions performed by a process and the ordering of interprocess communication actions are considered. Hence, analyzing the effects of PCO insertion will focus on whether PCO insertion causes a change in the ordering of actions in the system. The absence of time is not a severe restriction, because in the system specification the primary focus is on a high-level, abstract description of the functional system behavior which concentrates on topics like concurrency, communication, synchronization and ordering of events. The notion of time becomes more important during architecture exploration and the subsequent stages of the design flow.

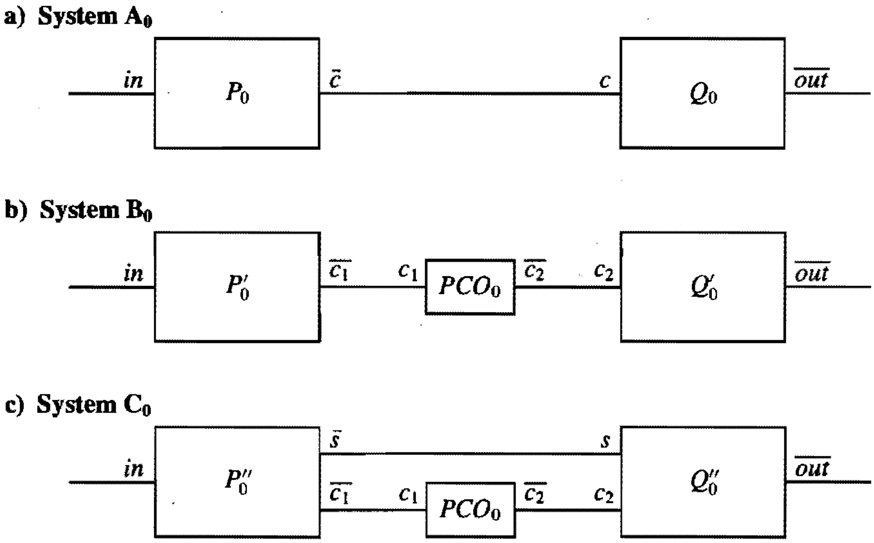


Figure 5.10 *Inserting a PCO in a communication channel*

Figure 5.10a shows an example system described in the CCS process algebra. The system A_0 is composed of two concurrent, communicating processes (*agents* in CCS terminology) P_0 and Q_0. The two agents communicate when agent P_0 performs output action $\bar{c}$ and agent Q_0 performs input action c simultaneously. The behavior of the agents P_0 and Q_0 is defined as:

$$P_0 \stackrel{\text{def}}{=} in.P_1 \qquad Q_0 \stackrel{\text{def}}{=} c.Q_1$$
$$P_1 \stackrel{\text{def}}{=} \bar{c}.P_0 \qquad Q_1 \stackrel{\text{def}}{=} \overline{out}.Q_0$$

The behavior of system A_0 is defined as: $A_0 \overset{\text{def}}{=} (P_0 | Q_0)\backslash c$. This indicates that system A_0 is constituted by the parallel composition of the agents P_0 and Q_0, and restriction $\backslash\{c\}$ (or shortly $\backslash c$). Restriction $\backslash c$ implies that the agents may perform the actions $\bar{c}$ and c only to communicate with each other. The monolithic behavior of the system A_0 is:

$$\begin{aligned} A_0 &= in.A_1 \\ A_1 &= \tau.A_2 \\ A_2 &= in.A_3 + \overline{out}.A_0 \\ A_3 &= \overline{out}.A_1 \end{aligned}$$

The behavior of A_0 can be minimized under observation equivalence to A_0':

$$\begin{aligned} A_0' &= in.A_1' \\ A_1' &= in.A_2' + \overline{out}.A_0' \\ A_2' &= \overline{out}.A_1' \end{aligned}$$

Next, we insert a *PCO* to observe and control the communication between the agents P_0 and Q_0. Inserting a *PCO* however should not change the externally observable system behavior. An observer in the system environment that interacts with the system, should not be able to distinguish the original system without a *PCO* from the system with a *PCO* operating in transparent mode. In CCS process algebra, this constraint can be formalized as: the behavior of the original system A_0 should be observational equivalent to the behavior of the system B_0, where B_0 is the modified system in which a *PCO* has been inserted that operates in transparent mode.

In [Mil89], two agents are said to be observational equivalent if they can both perform the same externally observable actions. This corresponds to the notion of bisimulation, i.e. both agents should be able to simulate each other's behavior. The formal definition in [Mil89] is:

A binary relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ over agents is a (weak) bisimulation if $(P, Q) \in \mathcal{S}$ implies, for all $\alpha \in Act$,

(i) Whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \overset{\hat{\alpha}}{\Rightarrow} Q'$ and $(P', Q') \in \mathcal{S}$.

(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then, for some P', $P \overset{\hat{\alpha}}{\Rightarrow} P'$ and $(P', Q') \in \mathcal{S}$.

In the course of time, many equivalence relations have been defined in process algebras, such as observation equivalence [Mil89], trace equivalence [Mil89], failure equivalence [Hoa85], and testing equivalence [NH84]. We will use observation equivalence here, because it closely corresponds to the intuitive notion that two systems are equivalent if and only if an external observer that is interacting with the systems, cannot distinguish between them. Hence, two systems are observational equivalent if their externally observable behavior is equivalent. Observation equivalence incorporates all distinctions which reasonably can be made by an external observer [Abr87]. Observation equivalence is quite a strong equivalence relation: if two systems are observational equivalent, then they are also equivalent under many other equivalence relations such as trace equivalence, failure equivalence and testing equivalence. However, observation equivalence is sometimes considered as too strong, because it goes beyond those observations that really can be made by an external observer. For instance, the agents $A = \tau.a.b + \tau.a.c$, $B = a.(\tau.b + \tau.c)$ and $C = a.b + a.c$ are not observational equivalent.

On the other hand, for our application area it is preferable to use a strong equivalence relation such as observation equivalence. The stronger the equivalence relation, the better we can restrict the effects of inserting a *PCO*. Another appealing property is that observation equivalence can be proven by means of bisimulations.

In figure 5.10b, we insert a *PCO* in the communication channel between the agents P'_0 and Q'_0. The agents P'_0, PCO_0 and Q'_0 are defined as:

$$\begin{array}{lll} P'_0 \overset{\text{def}}{=} in.P'_1 & PCO_0 \overset{\text{def}}{=} c_1.PCO_1 & Q'_0 \overset{\text{def}}{=} c_2.Q'_1 \\ P'_1 \overset{\text{def}}{=} \overline{c_1}.P'_0 & PCO_1 \overset{\text{def}}{=} \overline{c_2}.PCO_0 & Q'_1 \overset{\text{def}}{=} \overline{out}.Q'_0 \end{array}$$

The monolithic behavior of the composed system $B_0 \overset{\text{def}}{=} (P'_0|PCO_0|Q'_0)\backslash\{c_1, c_2\}$ in figure 5.10b is given by:

$$\begin{aligned} B_0 &= in.B_1 \\ B_1 &= \tau.B_2 \\ B_2 &= \tau.B_3 + in.B_4 \\ B_3 &= in.B_5 + \overline{out}.B_0 \\ B_4 &= \tau.B_5 \\ B_5 &= \tau.B_6 + \overline{out}.B_1 \\ B_6 &= in.B_7 + \overline{out}.B_2 \\ B_7 &= \overline{out}.B_4 \end{aligned}$$

The behavior of B_0 can be minimized under observation equivalence to B'_0:

$$\begin{aligned} B'_0 &= in.B'_1 \\ B'_1 &= in.B'_2 + \overline{out}.B'_0 \\ B'_2 &= \overline{out}.B'_1 + in.B'_3 \\ B'_3 &= \overline{out}.B'_2 \end{aligned}$$

It can easily be shown by bisimulations that the agents A_0 and B_0 are not observational equivalent. Hence, the insertion of the *PCO* in figure 5.10b causes a change in the system behavior. In fact, the *PCO* is acting as a buffer in the communication channel. As a consequence, agent B_0 in figure 5.10b can perform the sequence *in.in.in*, which cannot be performed by agent A_0 in figure 5.10a.

In order to insert a *PCO* while preserving the externally observable behavior, we adjust the behavior of the agents P_0 and Q_0 as follows (see figure 5.10c):

$$\begin{array}{lll} P''_0 \overset{\text{def}}{=} in.P''_1 & PCO_0 \overset{\text{def}}{=} c_1.PCO_1 & Q''_0 \overset{\text{def}}{=} s.Q''_1 \\ P''_1 \overset{\text{def}}{=} \overline{s}.P''_2 & PCO_1 \overset{\text{def}}{=} \overline{c_2}.PCO_0 & Q''_1 \overset{\text{def}}{=} c_2.Q''_2 \\ P''_2 \overset{\text{def}}{=} \overline{c_1}.P''_0 & & Q''_2 \overset{\text{def}}{=} \overline{out}.Q''_0 \end{array}$$

The monolithic behavior of the composed system $C_0 \overset{\text{def}}{=} (P''_0|PCO_0|Q''_0)\backslash\{s, c_1, c_2\}$ in figure 5.10c is:

$$
\begin{aligned}
C_0 &= in.C_1 \\
C_1 &= \tau.C_2 \\
C_2 &= \tau.C_3 \\
C_3 &= \tau.C_4 + in.C_5 \\
C_4 &= in.C_6 + \overline{out}.C_0 \\
C_5 &= \tau.C_6 \\
C_6 &= \overline{out}.C_1
\end{aligned}
$$

The behavior of C_0 can be minimized under observation equivalence to C_0':

$$
\begin{aligned}
C_0' &= in.C_1' \\
C_1' &= in.C_2' + \overline{out}.C_0' \\
C_2' &= \overline{out}.C_1'
\end{aligned}
$$

It can easily be seen that the systems A_0 in figure 5.10a and C_0 in figure 5.10c are observational equivalent. Hence, we inserted a *PCO* in the communication channel while preserving the externally observable system behavior. However, this required adjusting the behavior of the agents P_0 and Q_0 to avoid the *PCO* acting as a buffer in the communication channel. In fact, we replaced the communication behavior in system A_0 (passing message c) by the communication behavior in system C_0 (passing the messages s, c_1 and c_2). Intuitively, the message s can be considered as a synchronization signal between the agents P_0'' and Q_0'', and the messages c_1 and c_2 can be considered as the original message c that passes through the *PCO*.

In section 3.7.1.2 we discussed the functional fault model as proposed in [CCP93b, CCP93a, C$^+$94a]. In this work, the CCS process algebra is used to model a system as a set of concurrent processes. A fault in a communication channel is modeled by introducing a new process *Fault* that models the fault effect. For instance, inserting a fault in communication channel c in system A_0 (figure 5.10a) results in system B_0 (figure 5.10b), where the agent PCO_0 is replaced by the agent *Fault*. However, inserting an agent *Fault* suffers from the same problem as inserting an agent PCO_0. The system A_0 is not observational equivalent to system B_0, regardless of the behavior of the agent *Fault*. In later work [CCMP94], this problem is overcome by switching from CCS to CIRCAL process algebra [Mil85, Mil91]. CIRCAL allows multi-way rendez-vous communication and simultaneous actions. The agent PCO_0 in figure 5.10b would be defined in CIRCAL as $PCO_0 = \{c_1, \overline{c_2}\}\ .\ PCO_0$, which specifies that the actions c_1 and $\overline{c_2}$ are performed simultaneously. However, this solution rather misuses the semantical features offered by CIRCAL. Furthermore, the behavior of such agent PCO_0 cannot be implemented directly in hardware or in software.

5.6.2 Transformation Functions for PCO Insertion

In general, adjusting the behavior of agents for *PCO* insertion is not a trivial task. In appendix A, we formally define the transformation functions $\mathcal{F}$ and $\mathcal{T}$ for agents that send messages on a communication channel containing a *PCO*, and the transformation functions $\mathcal{G}$ and $\mathcal{Z}$ for agents that receive messages from a communication channel containing a *PCO*. In the example in figure 5.10c we obtained the agents P_0'' and Q_0'' by: $P_0'' = \mathcal{F}(\mathcal{T}(P_0))$ and $Q_0'' = \mathcal{G}(\mathcal{Z}(Q_0))$.

The formal definitions in appendix A define the transformation functions as parameterized functions (see figure 5.11):

- The function $\mathcal{T}_{\bar{c}}$ is applied to an agent that outputs the message $\bar{c}$ on a channel in which a PCO is inserted. The function application $\mathcal{T}_{\bar{c}}(P)$ returns an agent that is syntactically identical to agent P except that every occurrence of action $\bar{c}$ in P is replaced by $\bar{c}.\theta$ in $\mathcal{T}_{\bar{c}}(P)$.

  The function application $\mathcal{F}_{\bar{a}}(P)$ returns an agent that is syntactically identical to agent P except that every occurrence of θ in P is replaced by $\bar{a}$ in $\mathcal{F}_{\bar{a}}(P)$.

  Hence, $\mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(P))$ returns an agent that is syntactically identical to agent P except that every occurrence of action $\bar{c}$ in P is replaced by $\bar{c}.\bar{a}$ in $\mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(P))$, as illustrated in figure 5.11.

- The function $\mathcal{Z}_c$ is applied to an agent that inputs the message c from a channel in which a PCO is inserted. The function application $\mathcal{Z}_c(Q)$ returns an agent that is syntactically identical to agent Q except that every occurrence of action c in Q is replaced by $c.\zeta$ in $\mathcal{Z}_c(Q)$.

  The function application $\mathcal{G}_b(Q)$ returns an agent that is syntactically identical to agent Q except that every occurrence of ζ in Q is replaced by b in $\mathcal{G}_b(Q)$.

  Hence, $\mathcal{G}_b(\mathcal{Z}_c(Q))$ returns an agent that is syntactically identical to agent Q except that every occurrence of action c in Q is replaced by $c.b$ in $\mathcal{G}_b(\mathcal{Z}_c(Q))$, as illustrated in figure 5.11.

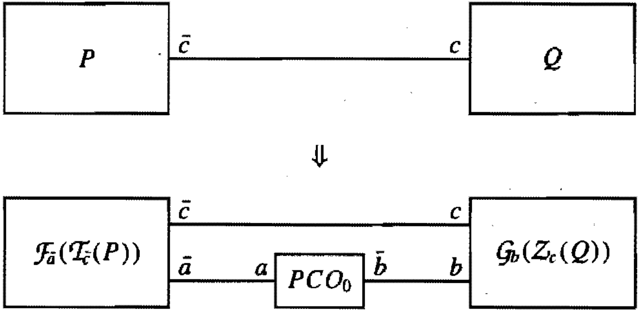

P
$\bar{c}$
c
Q
⇓
$\mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(P))$
$\bar{c}$
c
$\bar{a}$
a
PCO_0
$\bar{b}$
b
$\mathcal{G}_b(\mathcal{Z}_c(Q))$

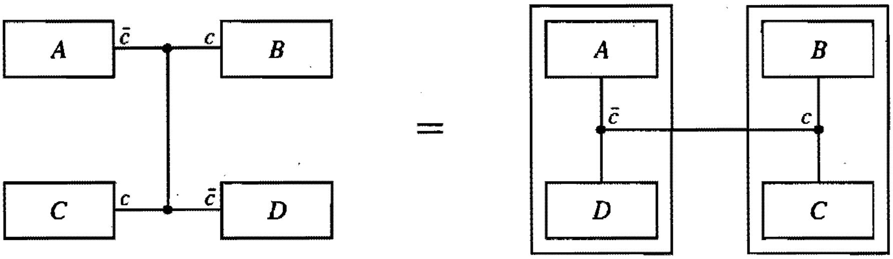
Figure 5.11 *Applying transformation functions for PCO insertion*

The transformation functions can be generally applied to all agents in the complete calculus of CCS and even to non-finite state agents. The behavior of these agents may be expressed using prefix, summation, composition, restriction and relabelling. Hence, the agents P and Q in figure 5.11 do not necessarily have to be monolithic; they may be complete systems consisting of several concurrent agents.

In appendix A, we also give a mathematical proof that the modified system, in which a PCO has been inserted and in which transformation functions have been applied to the relevant agents, is observational equivalent to the original system when the PCO is operating in transparent mode or in observation mode. This is a very powerful result, because this proof implies that PCO insertion using the transformation functions preserves the externally observable system behavior. Hence, there is no need for an a posteriori proof of equivalence by means of bisimulation, which is in general a very complex or even impossible task. The transformation functions provide correctness-preserving PCO insertion.

The agent PCO_0 in figure 5.10 and figure 5.11 represents a PCO operating in transparent mode. However, the agent may also represent the composed behavior of a PCO operating in observation mode and an external observer. For instance, the behavior of PCO_0 in figure 5.11 can be replaced by a system $(PCO'_0|E_0)\backslash obs$. The agent PCO'_0 represents the PCO in observation mode, whose behavior is defined as: $PCO'_0 \overset{\text{def}}{=} a.\overline{obs}.\bar{b}.PCO'_0$. The behavior of the external observer E_0 is defined as $E_0 \overset{\text{def}}{=} obs.E_0$. It can easily be shown that $PCO_0 \approx (PCO'_0|E_0)\backslash obs$. Hence, the agent PCO_0 may be considered as representing a PCO in transparent mode or in observation mode.

The transformation functions can be applied also to systems in which multiple agents communicate over a single channel. For instance, the system in figure 5.12a consists of four agents that all communicate over channel c. From the associativity of the parallel composition, it follows that $(A|B|C|D)\backslash c = ((A|D)\,|\,(B|C))\backslash c$, as depicted in figure 5.12b.



a) $(A|B|C|D)\backslash c$ b) $((A|D)\,|\,(B|C))\backslash c$

Figure 5.12 Associativity of parallel composition

We now apply the transformation functions defined in appendix A to insert a PCO in channel c. The resulting system including the PCO is shown in figure 5.13. As proven in section A.3, it follows that: $((A|D)\,|\,(B|C))\backslash c \approx (\mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(A|D))\,|\,PCO\,|\,\mathcal{G}_b(\mathcal{Z}_c(B|C)))\backslash\{a,b,c\}$.

The transformation functions may be distributed over the parallel composition, as defined in section A.1 (definition 2 and definition 4), which results in the following:
$\mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(A|D)) = \mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(A)\,|\,\mathcal{T}_{\bar{c}}(D)) = \mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(A))\,|\,\mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(D))$ and
$\mathcal{G}_b(\mathcal{Z}_c(B|C)) = \mathcal{G}_b(\mathcal{Z}_c(B)\,|\,\mathcal{Z}_c(C)) = \mathcal{G}_b(\mathcal{Z}_c(B))\,|\,\mathcal{G}_b(\mathcal{Z}_c(C))$.
Hence, the systems shown in figure 5.12 and figure 5.13 are observational equivalent.

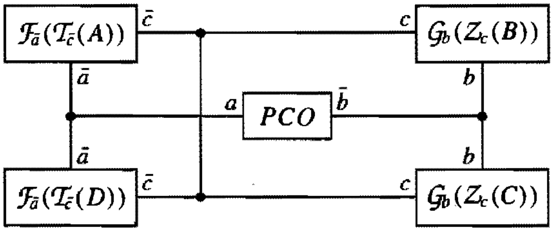

$\mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(A))$
$\bar{c}$
c
$\mathcal{G}_b(\mathcal{Z}_c(B))$
$\bar{a}$
b
a
PCO
$\bar{b}$
$\bar{a}$
b
$\mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(D))$
$\bar{c}$
c
$\mathcal{G}_b(\mathcal{Z}_c(C))$

Figure 5.13 Applying transformation functions for PCO insertion

5.6.3 Example of Correctness-Preserving PCO Insertion

In this section we provide an example of PCO insertion in a non-trivial system. We consider the system A_0 shown in figure 5.14a, which is composed of the agents P_0, Q_0 and R_0.

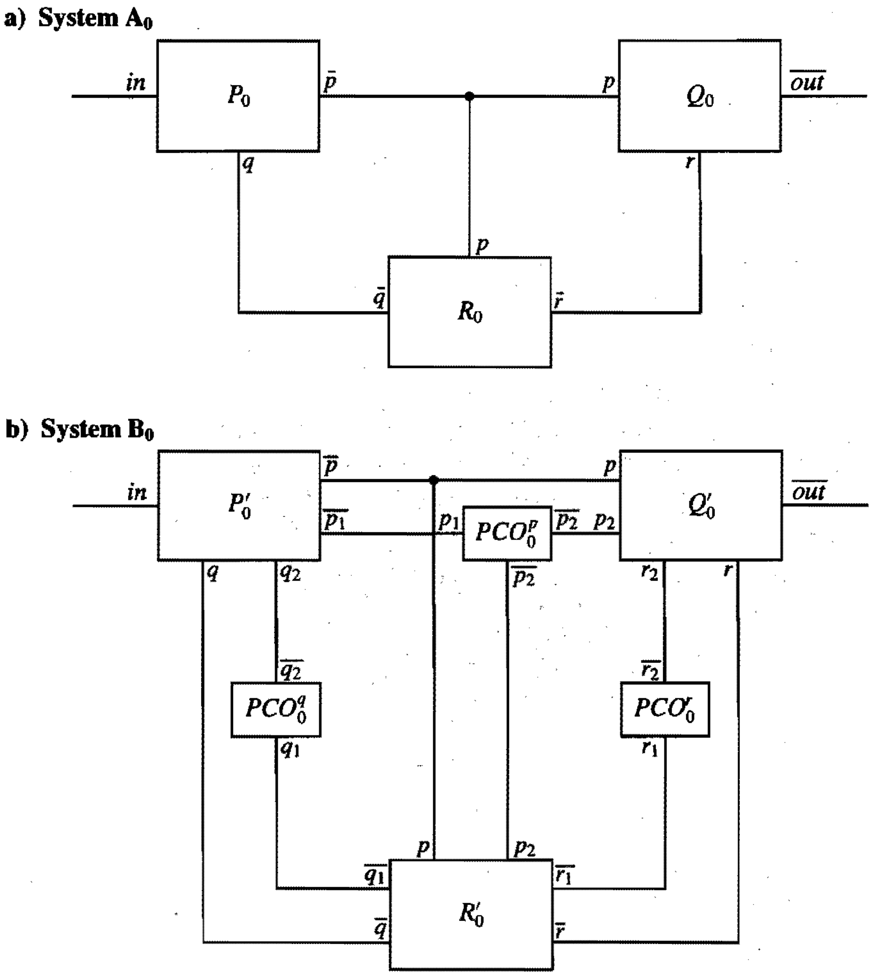


Figure 5.14 Example of PCO insertion

The agents P_0, Q_0 and R_0 are defined as:

$$\begin{array}{lll}
P_0 \overset{\text{def}}{=} in.P_1 & Q_0 \overset{\text{def}}{=} p.Q_1 + r.Q_0 & R_0 \overset{\text{def}}{=} p.R_1 \\
P_1 \overset{\text{def}}{=} \bar{p}.P_0 + \bar{p}.P_2 & Q_1 \overset{\text{def}}{=} \overline{out}.Q_0 & R_1 \overset{\text{def}}{=} \bar{q}.R_0 + \bar{r}.R_0 \\
P_2 \overset{\text{def}}{=} q.P_0 + in.P_1 & &
\end{array}$$

The monolithic behavior of the composed system $A_0 \overset{\text{def}}{=} (P_0|Q_0|R_0)\backslash\{p,q,r\}$ in figure 5.14a is:

$$\begin{aligned}
A_0 &= in.A_1\\
A_1 &= \tau.A_2 + \tau.A_3\\
A_2 &= in.A_4 + \overline{out}.A_0\\
A_3 &= \tau.A_0 + in.A_5\\
A_4 &= \tau.A_6 + \tau.A_7 + \overline{out}.A_1\\
A_5 &= \tau.A_6 + \tau.A_7 + \tau.A_1\\
A_6 &= in.A_8 + \overline{out}.A_3\\
A_7 &= \tau.A_2 + in.A_8 + \overline{out}.A_3\\
A_8 &= \overline{out}.A_5
\end{aligned}$$

We insert three PCOs in the communication channels p, q and r. We use the transformation functions to adjust the behaviors of the agents for PCO insertion, as discussed in the previous section.

- Inserting a PCO in channel p requires the transformations $\mathcal{F}_{\overline{p_1}}(\mathcal{T}_{\bar{p}}(P_0))$ and $\mathcal{G}_{p_2}(\mathcal{Z}_p(Q_0|R_0))$, where $\mathcal{G}_{p_2}(\mathcal{Z}_p(Q_0|R_0)) = (\mathcal{G}_{p_2}(\mathcal{Z}_p(Q_0))|\mathcal{G}_{p_2}(\mathcal{Z}_p(R_0)))$.
- Inserting a PCO in channel q requires the transformations $\mathcal{F}_{\overline{q_1}}(\mathcal{T}_{\bar{q}}(R_0))$ and $\mathcal{G}_{q_2}(\mathcal{Z}_q(P_0))$.
- Inserting a PCO in channel r requires the transformations $\mathcal{F}_{\overline{r_1}}(\mathcal{T}_{\bar{r}}(R_0))$ and $\mathcal{G}_{r_2}(\mathcal{Z}_r(Q_0))$.

The agents P_0', Q_0', R_0' in figure 5.14b are now obtained by:

$$P_0' = \mathcal{F}_{\overline{p_1}}(\mathcal{T}_{\bar{p}}(\mathcal{G}_{q_2}(\mathcal{Z}_q(P_0)))),$$

$$Q_0' = \mathcal{G}_{p_2}(\mathcal{Z}_p(\mathcal{G}_{r_2}(\mathcal{Z}_r(Q_0)))),$$

$$R_0' = \mathcal{G}_{p_2}(\mathcal{Z}_p(\mathcal{F}_{\overline{q_1}}(\mathcal{T}_{\bar{q}}(\mathcal{F}_{\overline{r_1}}(\mathcal{T}_{\bar{r}}(R_0)))))).$$

The resulting agents P_0', Q_0', R_0' and the agents PCO_0^p, PCO_0^q and PCO_0^r in figure 5.14b are given by:

$$\begin{array}{lll}
P_0' = in.P_1' & Q_0' = p.Q_2' + r.Q_1' & R_0' = p.R_1'\\
P_1' = \overline{p}.P_2' + \overline{p}.P_3' & Q_1' = r_2.Q_0' & R_1' = p_2.R_2'\\
P_2' = \overline{p_1}.P_0' & Q_2' = p_2.Q_3' & R_2' = \overline{q}.R_3' + \overline{r}.R_4'\\
P_3' = \overline{p_1}.P_4' & Q_3' = \overline{out}.Q_0' & R_3' = \overline{q_1}.R_0'\\
P_4' = q.P_5' + in.P_1' & & R_4' = \overline{r_1}.R_0'\\
P_5' = q_2.P_0' & &
\end{array}$$

$$\begin{array}{lll}
PCO_0^p \overset{\text{def}}{=} p_1.PCO_1^p & PCO_0^q \overset{\text{def}}{=} q_1.PCO_1^q & PCO_0^r \overset{\text{def}}{=} r_1.PCO_1^r\\
PCO_1^p \overset{\text{def}}{=} \overline{p_2}.PCO_0^p & PCO_1^q \overset{\text{def}}{=} \overline{q_2}.PCO_0^q & PCO_1^r \overset{\text{def}}{=} \overline{r_2}.PCO_0^r
\end{array}$$

The behavior of the composed system B_0 in figure 5.14b is defined as:
$B_0 \overset{\text{def}}{=} (P_0'|Q_0'|R_0'|PCO_0^p|PCO_0^q|PCO_0^r)\backslash\{p, p_1, p_2, q, q_1, q_2, r, r_1, r_2\}$.

As proven in appendix A, the transformation functions guarantee that system A_0 in figure 5.14a is observational equivalent to system B_0 in figure 5.14b: $A_0 \approx B_0$. Hence, there is no need for proving this equivalence afterwards by means of bisimulation, which is in general a very complex or even impossible task. This example clearly demonstrates the strength of our transformation functions for PCO insertion while preserving the externally observable system behavior.

5.7 Discussion

In this chapter we concentrated on two key questions: where should PCOs be inserted in the system specification, and what are the effects of PCO insertion in the system specification on the system behavior. We discuss and summarize our conclusions in this section.

5.7.1 Guidelines for PCO Insertion during System Specification

There should be a balanced number of PCOs in the system, which provides maximum observability and controllability of the internal system behavior at minimum costs. We studied three related fields in literature to derive guidelines for PCO insertion:

- Much work has been performed and standardized in the application domain of communication systems. The evolution in the OSI standards for remote testing clearly demonstrates the necessity of design-for-test during system specification. The communication interfaces between protocol layers in communication systems are appropriate places for inserting PCOs.

- Testability analysis of VLSI circuits is widely applied to guide hardware design-for-test. VLSI testability analysis techniques have evolved from the gate level to the behavioral level. These techniques analyze both the circuit structure, i.e. the topology of interconnected logic gates or modules, and the information propagation through these logic gates or modules. We argued that a similar approach can be applied to guide PCO insertion in the system specification, by analyzing both the topology of the specification structure and the information propagation through individual processes. However, we concluded that this approach cannot be applied efficiently. The analysis of information propagation through a process implies exhaustive analysis of all control-flow and data-flow paths, which is generally unfeasible. Furthermore, we argued that analysis of simply the specification structure while ignoring information propagation, is inaccurate because the behavioral view on the system specification is not taken into account.

- System-level testability analysis has been proposed by Sheppard & Simpson in their work on integrated diagnostics. However, we argued that their approach is complementary to our design for test & debug approach, and that their testability analysis is not applicable to guide PCO insertion. System-level testability analysis has also been proposed by Robach et al., however we showed that their work is directed at hardware systems instead of hardware/software systems. Finally, we outlined how formal analysis of conformance testability might be used to guide PCO insertion.

The overall conclusion is that the previous approaches are all relevant in their own application domain, however none of them is truly applicable to guide PCO insertion in the system specification of hardware/software systems. Instead, we propose scenario-based testability analysis to guide PCO insertion. Scenario-based PCO insertion addresses two key questions: what is the essential information in the system for a particular scenario, and how well is this essential information visible to an external observer in the system environment. We showed that scenario-based testability analysis traverses a relevant subset of paths through a system specification in the POOSL language, including loops, tail recursion and interrupts. Scenario-based testability analysis is suitable to identify the essential communication channels and process state information.

The discussion on PCO insertion in this chapter is entirely focused on the system specification. We did neither consider how to incorporate PCOs in the system architecture, nor how to implement PCOs in hardware and/or software. We will elaborate on these topics in chapter 6. Furthermore it should be noted that additional PCOs may be inserted during architecture exploration and the subsequent stages in the design flow. For instance, we discussed in chapter 4 that information on the scheduling of processes, both in hardware and in software, is essential for an external tester/debugger. The system specification usually does not specify the exact scheduling of processes, because this is often implementation-dependent. In order to observe and control the scheduling of processes, additional PCOs may be required. A typical example is to insert a software monitor in the operating system to observe the scheduling of software processes. Hence, PCO insertion in the system specification is necessary but not sufficient to obtain full controllability and observability of the internal operation of a hardware/software system.

5.7.2 Effects of PCO Insertion during System Specification

Inserting PCOs in the system specification implies modification of the system specification, and consequently PCO insertion may induce faulty system behavior. This problem is analogous to the probe effect or the Heisenberg principle in testing: inserting a probe in hardware or software for testing purposes may interfere in the behavior of the hardware or software in such a way that existing faults are masked or that new faults are introduced.

We provided a formal discussion on the effects of PCO insertion in the system specification using CCS process algebra. We demonstrated that inserting a PCO in a communication channel introduces a buffer in the channel. This causes that the behavior of the system including the PCO is generally not observational equivalent to the behavior of the system before PCO insertion. We showed that a PCO can be inserted in a communication channel while preserving observation equivalence. However, this requires adjusting the behaviors of the processes that communicate on the channel. We illustrated that adjusting the process behaviors is in generally not a trivial task. Furthermore, proving observational equivalence by means of bisimulation is a computationally-intensive task that is applicable only to systems of moderate size due to state space explosion. To overcome these problems, we formally defined a set of mathematically proven transformation functions. Inserting a PCO in a communication channel and applying the transformation functions to the involved processes, guarantees (mathematically proven) that the modified system including the PCO is observational equivalent to the original system without PCO. Hence, applying these transformation functions to processes provides correctness-preserving PCO insertion.

5.8 Summary

In this chapter we concentrated on two key questions: where should PCOs be inserted in the system specification, and what are the effects of PCO insertion on the system behavior.

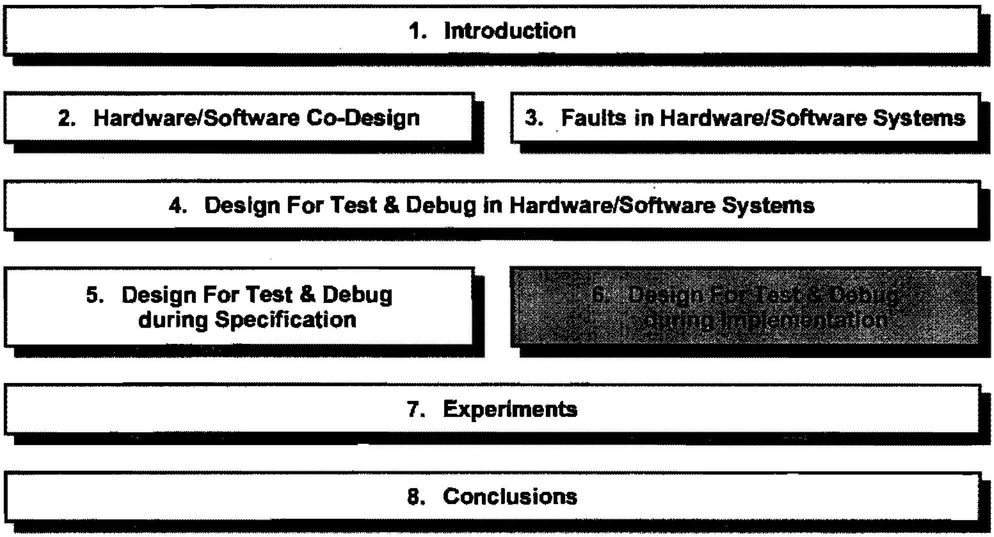
A balanced number of PCOs should be inserted in the system, providing maximum observability and controllability of the internal system behavior at minimum costs. We propose scenario-based testability analysis to guide PCO insertion. Scenarios are key elements in modern analysis and design methods for creating a system specification. A scenario describes some specific part of the system behavior. Our scenario-based approach to PCO insertion addresses two key questions: what is the essential information in the system for a particular scenario, and how well is this essential information visible to an external observer in the system environment. Scenario-based testability analysis is suitable to identify the essential communication channels and process state information in a system specification. We showed that scenario-based testability analysis traverses a relevant subset of paths through a system specification in the POOSL language, including loops, tail recursion and interrupts.

We discussed related approaches to PCO insertion in a system specification, considering PCOs in the OSI standards for communication systems, testability analysis in VLSI circuits, and system-level testability analysis techniques. We concluded that these approaches are all relevant in their own application domain, however none of them is truly applicable to guide PCO insertion in the system specification of hardware/software systems.

Inserting PCOs in a system specification implies modifying the system behavior. Unfortunately, the PCOs may interfere in the system behavior in such a way that the system behavior becomes incorrect. We provided a formal discussion on the interference of PCOs in the system behavior using CCS process algebra. We showed that the system behavior including PCOs is not necessarily observational equivalent to the system behavior before PCO insertion. However, we proved that a PCO can be inserted in a communication channel while preserving observation equivalence. We formally defined a set of transformation functions that should be applied to those processes that communicate over a channel in which a PCO is to be inserted. We gave a mathematical proof that the modified system behavior, obtained after applying the transformation functions and inserting PCOs, is observational equivalent to the original system behavior. Hence, we proved that we are able of inserting PCOs in a system specification in such a way that the interference of PCOs does not induce incorrect behavior.

Chapter 6

Design For Test & Debug during Implementation


1. Introduction
2. Hardware/Software Co-Design
3. Faults in Hardware/Software Systems
4. Design For Test & Debug in Hardware/Software Systems
5. Design For Test & Debug during Specification
6. Design For Test & Debug during Implementation
7. Experiments
8. Conclusions

In this chapter we elaborate on design for test & debug during system implementation. We provide an overview of the current design-for-test and design-for-debug techniques for both hardware and software. Next, we discuss how these techniques can be used to implement PCOs and to implement the infrastructure for accessing PCOs from the external environment.

6.1 Introduction

Our design for test & debug approach is based on the insertion of Points of Control and Observation (PCOs) in the system specification. In the subsequent phases of the design process, the PCOs are actually realized. We argued in chapter 4 that PCOs can be realized either by test & measurement equipment, or the PCOs can be incorporated into the hardware/software architecture of the system. In the latter case, the PCOs are implemented by dedicated hardware/software, by using the current hardware DFT and DFD facilities, or by software instrumentation. In this chapter, we provide an overview of the state-of-the-art on DFT and DFD techniques in both hardware and software. In addition, we discuss how these techniques can be used to implement PCOs and to implement the infrastructure for accessing the PCOs from an external tester/debugger.

6.2 Hardware DFT

Hardware DFT techniques primarily aim at detecting the physical faults in hardware components that arise during production or field operation. In the course of time, various DFT techniques have been developed for ICs, PCBs and hardware systems (i.e. a collection of PCBs).

Hardware DFT techniques can be employed to enhance external testing or to provide built-in self-test (BIST). For external testing, DFT provides built-in test facilities to improve observability and controllability of the internal system operation to an external tester. BIST is a DFT technique combining built-in test and self test, which implies that the system incorporates hardware and/or software to perform self testing. BIST can be performed off-line or on-line. Off-line BIST indicates that testing is performed by switching the system from normal operation mode into a dedicated test mode. On-line BIST indicates that testing is performed during normal system operation, and hence there is no dedicated test mode. On-line BIST can be either concurrent, in which testing is performed by redundancy and comparison techniques as with fault-tolerant techniques, or non-concurrent, in which testing is carried out while the system is in an idle state.

6.2.1 Hardware DFT on the IC Level

In the 1960's, IC testing was generally performed by means of exhaustive, functional testing using a black-box, functional view. This approach became unfeasible in the 1970's due to the increasing complexity of ICs. Consequently, structural testing techniques were developed in which test sequences are derived from the internal structure of the IC. In addition, DFT became a necessity to enable structural IC testing.

It is generally recognized (e.g. [Tex96]) that the cost of finding a faulty IC after assembly onto a PCB is an order of magnitude more than detecting the fault before assembly. When the fault is detected after integration in the system, the cost increases once more an order of magnitude. Finally, when a faulty IC is discovered in a system at a customer site, the cost is three orders of magnitude higher than the cost of discovering the fault before assembly. Hence, it is essential to test an IC thoroughly before it is assembled onto a PCB.

A typical ad hoc DFT technique on the IC level is to insert test points in the circuit that are accessible through IC pins (e.g. [CB85, GMG90, TM96]). A test point can be a control point or an observation point. Obviously, the major constraint using test points is the demand for additional IC pins. The number of additional IC pins for testing can be limited by multiplexing. Other ad hoc DFT techniques are improving circuits to ease their initialization or to disable internal oscillators and clocks during testing.

A promising, structured approach to DFT at the IC level is BIST, which implies that the IC is capable of self testing by means of built-in test features (e.g. [Fuj85, ABF90, AKS93a, AKS93b, Ben94b]). The basic architecture for IC-level BIST consists of a test pattern generator, a test response analyzer, and a test controller. The test controller controls the operation of the test pattern generator and the test response analyzer. A test pattern generator typically consists of a ROM with stored test patterns, a counter or a LFSRs (linear feedback shift register). A test response analyzer usually consists of a comparator with stored test responses or a LFSR or MISR (multiple-input signature register) used as signature analyzer. Advanced BIST facilities combine test pattern generation and test response analysis, such as STUMPS (self-test using MISR and parallel shift register sequence generator) and BILBOs (built-in logic block observer).

Testing sequential circuits is considerably more complex than testing combinational circuits. A common DFT technique for sequential circuits is to incorporate a (full or partial) scan path, which provides that the memory elements in the circuit can be switched into test mode to form a serial shift register. Scan paths allow separate testing of the memory elements and the combinational logic in the circuit. There are several forms of scan design which primarily differ in the design of the scan cells.

Philips developed an IC-level DFT approach called macro testability [B^+86, BDSvdS90, B^+92b, B^+93, MKW93, BBT95]. Macro testing is based on partitioning an IC into testable blocks (macros) and providing that every macro can be controlled and observed from the external IC pins. This approach allows the use of specific test strategies and specific test generation algorithms for each macro. Accessing a macro from the external IC pins is provided by means of test protocol expansion or by inserting test interface elements (TIEs). In test protocol expansion, the local macro test protocol described at the terminals of the macro, is transformed into a global test protocol described at the terminals of the IC. TIEs may be inserted at the interfaces of the macros to provide direct access, and they can be chained to form a serial shift register.

Test points, scan cells and TIEs provide observation and control inside a circuit and hence they can be regarded as low-level hardware implementations of PCOs. Although these facilities are intended for structural testing of physical defects, they can be re-used for functional testing and debugging. However, access to these test facilities is rather restricted: test points can only be accessed through multiplexed IC pins; scan cells and TIEs can only be accessed serially and their state is affected during shifting. Hence, these test facilities are of limited use for real-time observation and control.

6.2.2 Hardware DFT on the PCB Level

Initially, printed circuit boards (PCBs) were tested using in-circuit testing (ICT). ICT implies that the ICs and their interconnections on a PCB are tested by accessing the IC pins with probes or a

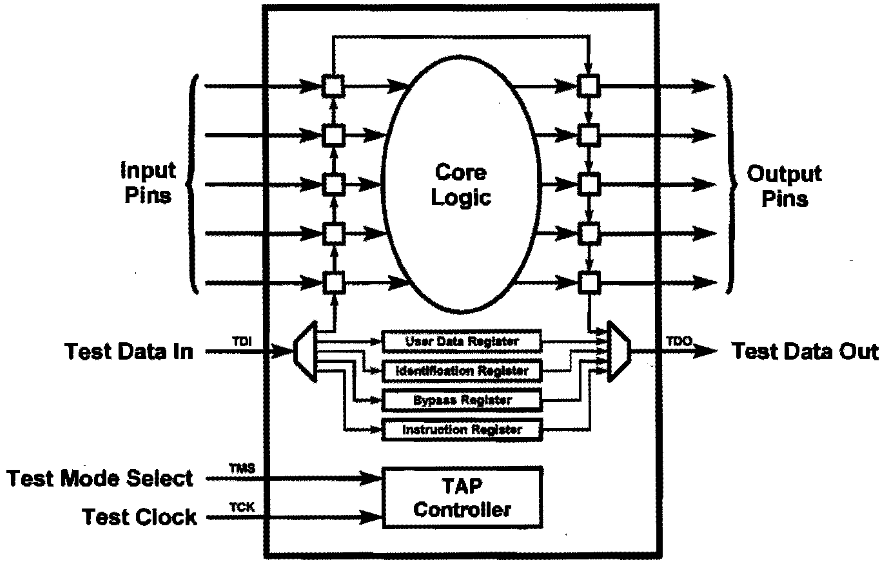


Figure 6.1 Boundary-scan architecture

bed-of-nails. However, ICT is effective only after the board has been removed from the system, and ICT became even impossible with the rise of surface-mount technology (SMT) in which components are mounted densely on both sides of the board. Traditional, hierarchical testing did not provide a satisfying solution to PCB testing. ICs mounted on a board are difficult to access from the board's edge connectors because access paths through other ICs are required. This is not only limiting control and observation capabilities, but also the generation of test cases is very difficult.

A solution at the PCB level has been provided by means of the IEEE 1149.1 boundary-scan standard [IEE90, MT90, Par92, BvdEdJ93, Tex96]. In the boundary-scan architecture, each IC incorporates a test access port (TAP), a test controller and test registers, as shown in figure 6.1.

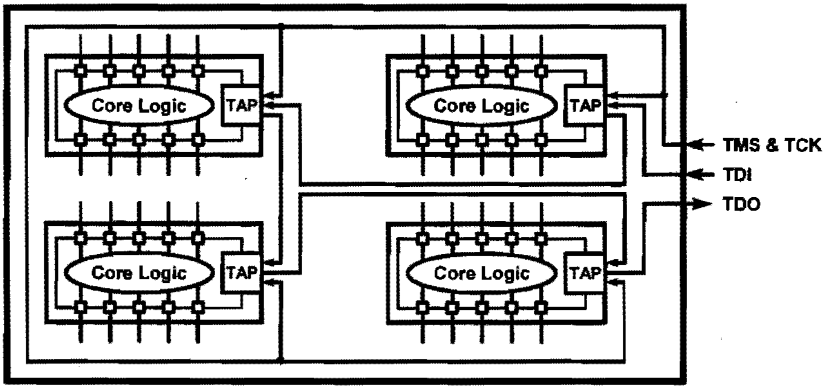


Figure 6.2 PCB with boundary-scan architecture

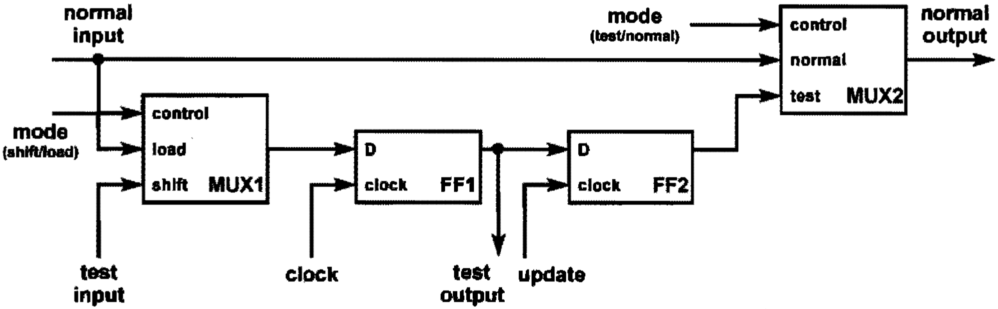


Figure 6.3 Boundary-scan cell

The TAP requires four IC pins: *test data input* (TDI), *test data output* (TDO), *test mode select* (TMS) and *test clock* (TCK). Boundary-scan cells are placed at all IC inputs and outputs. The boundary-scan cells can be chained to form a serial shift register (the boundary-scan register). Data can be shifted in and out of the boundary-scan register through the TAP.

The TDI and TDO pins of all ICs on a board are connected as shown in figure 6.2, providing a serial access path to the boundary-scan register of every IC from the board's edge connector. The boundary-scan architecture supports testing of both individual ICs and the interconnections between the ICs on the PCB.

A possible implementation of a boundary-scan cell is shown in figure 6.3. The control signals *clock*, *update*, *mode (shift/load)* and *mode (test/normal)* are provided by the TAP controller. The *test input* and *test output* are connected to adjacent boundary-scan cells, so that the chain of boundary-scan cells builds up the boundary-scan register. If the boundary-scan cell is placed in an IC input, the *normal input* is connected to the IC input pin and the *normal output* is connected to the core logic; if the boundary-scan cell is placed in an IC output, the *normal input* is connected to the core logic and the *normal output* is connected to the IC output pin. A boundary-scan cell can be considered as a hardware implementation of a PCO, providing a transparent mode, an observation mode and a test mode.

- In transparent mode, the multiplexer MUX2 is switched into normal mode by the *mode (test/normal)* signal, providing a direct connection between the *normal input* and the *normal output*.

- In observation mode, multiplexer MUX2 is switched into normal mode and multiplexer MUX1 is switched into load mode. Hence, the *normal input* signal is passed to the *normal output* and as well to the input of flipflop FF1. The *clock* signal is used to capture the data in flipflop FF1.

  Subsequently, MUX1 is switched into shift mode, providing a serial connection to the adjacent boundary-scan cells. The data in the boundary-scan register can now be shifted out under control of the *clock* signal. The data can be observed on the TDO pin. Note that shifting can be performed during normal system operation, because the transparent connection between *normal input* and *normal output* is not affected.

- In test mode, first a test stimulus is placed into the boundary-scan register. The multiplexer MUX1 is operating in shift mode to shift the test stimulus into flipflop FF1. Shifting in the test stimulus can be performed during normal system operation. When the test stimulus is in place, the signal *update* is generated to capture the test stimulus in FF2.

  Subsequently, multiplexer MUX1 is switched into load mode and multiplexer MUX2 is switched into test mode. Hence, the *normal input* is passed to FF1, while the test stimulus in FF2 is passed to the *normal output*. The *normal input* can be captured in FF1 by the *clock* signal, and next the boundary-scan register can be shifted out by switching MUX1 into shift mode.

The boundary-scan architecture is primarily intended for structural testing of interconnections on PCBs and for accessing individual ICs. Nevertheless, the boundary-scan architecture can be used also for functional testing and debugging purposes [Cro89, HYC89, FM90, Lef90, Dan92, Tex96]. We showed that a boundary-scan cell can be considered as a hardware implementation of a PCO. The major limitation however is that only serial access is provided, which restricts the observation and control of real-time system behavior to repeatedly sampling and shifting out the boundary-scan registers. In spite of this limitation, boundary scan is a powerful and cheap technique for system testing and debugging.

Effective boundary-scan testing requires that most ICs on a board incorporate the boundary-scan architecture. Fortunately, many ICs like microprocessors and DSPs currently incorporate the boundary-scan architecture. Furthermore, devices like buffers and latches are available that incorporate the boundary-scan architecture (e.g. Texas Instruments Scope octal devices [Cro89, HYC89, Lef90]).

At the moment, embedded cores and multi-chip modules (MCMs) are emerging. Consequently, PCBs containing multiple chips may be implemented in the near future on a single silicon chip that contains multiple cores/modules. As with PCBs, also multi-chip modules may be equipped with boundary-scan like facilities to test the interconnections between cores/modules and to access the individual cores/modules. For instance, the ARM7 microprocessor core incorporates a boundary-scan architecture that allows to access the core once it is embedded in an ASIC [ARM95b]. We will elaborate further on the use of the boundary-scan architecture for debugging purposes in section 6.3.

6.2.3 Hardware DFT on the System Level

Hardware systems typically have a hierarchical structure consisting of ICs, PCBs and racks of PCBs. The traditional, hierarchical testing techniques are insufficient on the PCB level and the system level. Test stimuli have to be transferred from the system level through many layers of circuitry down to the element under test, and vice versa the test results have to be transferred up again through many layers of circuitry.

A hierarchical solution to system-level testing is to equip all hierarchical levels with built-in test or BIST facilities [HK92, Mau93, HK94]. The system in figure 6.4 consists of several PCBs, where each PCB contains several ICs. Each level is equipped with a test manager which controls

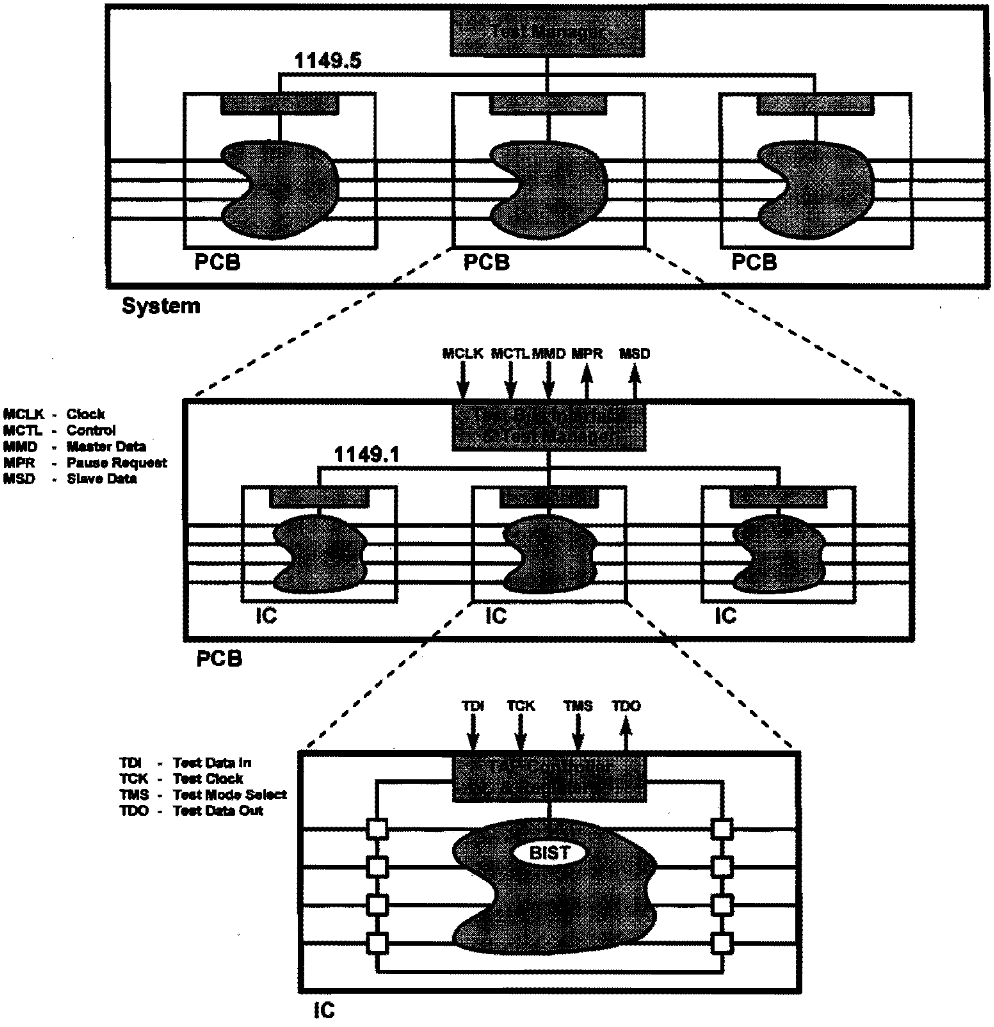


Figure 6.4 Hierarchical built-in test architecture

the built-in test or BIST facilities at the particular level. The ICs are equipped with the boundary-scan architecture and the TAP controller is used also to control the IC's BIST for self-testing of the chip's core logic. On the PCB level, the IEEE 1149.1 boundary-scan architecture provides an infrastructure for testing ICs and their interconnections. The test manager on a PCB is responsible for managing the testing of the board. The PCB test manager is not standardized in the IEEE 1149.1 standard, but several proposals have been described in literature (e.g. [Whe92, HK94]). A collection of boards can be tested using the module test and maintenance (MTM) bus, which has been standardized in the IEEE 1149.5 standard [IEE95]. The MTM bus provides a communication protocol for exchanging test and maintenance commands and serial data.

The hierarchical built-in test architecture of figure 6.4 provides that the test managers at each level can be accessed through standardized test interfaces. An important step is to make sure that the test infrastructure itself is correct. Therefore, the test buses, the test interfaces and the test managers themselves should be tested first.

The hierarchical built-in test architecture may adopt a centralized or a distributed system test strategy. In the distributed strategy, each level is equipped with BIST: test stimuli are generated and test responses are analyzed locally on each level. On the IC level, the IC-level test manager activates the IC's BIST. The BIST generates test stimuli and analyzes the test responses. Finally, the BIST transmits an OK or NOK message (and possibly diagnostic information) to the IC-level test manager. On the PCB level, the PCB-level test manager activates the IC-level test managers and performs a self-test of the IC interconnections on the board. Subsequently, the PCB-level test manager collects the test results of the IC-level test managers and evaluates the IC interconnection test. Finally, the PCB-level test manager transmits an OK or NOK message (and possibly diagnostic information) to the system-level test manager. In a similar way, the system-level test manager activates the PCB-level test managers and performs a self-test of the interconnections between the PCBs. Next, the system-level test manager collects the results of the PCB-level test managers and evaluates the PCB interconnection test. Finally, the system-level test manager transmits an OK or NOK message (and possibly diagnostic information) to the external system environment.

The centralized strategy is not based on BIST. The test stimuli are transmitted from the system-level test manager to the PCB level or the IC level, and vice versa the test responses are transmitted back to the system-level test manager. The system-level test manager is responsible for evaluating the test responses. The centralized strategy resembles the traditional, hierarchical testing approach. However, the test interfaces and the test managers provide that test stimuli and test responses can be transferred rather easily between the various levels. The system-level test manager in the centralized strategy is usually implemented in software.

Incorporating a hierarchical built-in test or BIST architecture in a system requires additional costs due to the extra hardware for the built-in test facilities at all levels. However, these costs pay off because of less expensive test equipment and improved testing and diagnosis during assembly, integration, and field operation. In general, the benefits of a hierarchical test architecture are small at the chip level, they are larger on the board level, and they are considerable at the system level.

The hierarchy of ICs on PCBs, PCBs in subsystems, and subsystems in a system can be extended when considering a number of systems connected in a communications network. As described in section 5.2.2, the ISO 10164 standard provides a test management interface which can be used for testing systems connected in a network.

We stated in the previous sections that test points, scan cells, TIEs and boundary-scan cells can be considered as hardware implementations of PCOs. The hierarchical test infrastructure as described in this section provides access to these PCOs.

6.3 Hardware DFD

Hardware DFD techniques aim at incorporating built-in hardware facilities to facilitate the debugging of hardware/software systems. When testing reveals an error in the system, debugging is required next to locate the fault in the system that caused the error. Hardware DFD techniques are useful for debugging physical hardware defects as well as for debugging design errors in hardware and/or software.

The hardware DFT techniques described in section 6.2 are primarily intended to facilitate structural testing for physical defects that arise during production, assembly, integration and field operation. Nevertheless, these hardware DFT facilities may be re-used for debugging purposes. We already discussed in section 6.2 that hardware DFT facilities may be considered as PCOs and hardware DFT facilities also provide a test infrastructure to access these PCOs. However, we also indicated that this approach is rather restricted when testing and debugging real-time behavior. Furthermore, these hardware DFT facilities concentrate on hardware components, while ignoring the observation and control of software.

At the moment, hardware DFD techniques are becoming more and more important. They provide improved observation and control of the internal system operation to facilitate debugging of hardware and/or software. Hardware DFD techniques are useful for silicon debugging and embedded system debugging. Silicon debugging indicates the debugging of physical defects and hardware design errors in ICs. Silicon debugging is typically required to debug the first silicon prototypes of an IC and is performed on an IC tester. (Hence the IC is debugged in isolation.) Embedded system debugging indicates the debugging of ICs and embedded software once they are integrated in a system. Faults may reside either in the embedded software or in the hardware (physical defects or design errors). Modern processors incorporate advanced DFD facilities for embedded system debugging. In this section we will review the state-of-the-art of hardware DFD techniques for silicon debugging and embedded system debugging.

6.3.1 Hardware DFD for Silicon Debugging

Due to the increasing complexity of VLSI chips, the internals of ICs are becoming less and less accessible. This problem is becoming even worse with the rise of embedded cores and multi-chip modules. Currently, test equipment is used for silicon debugging, such as IC testers and e-beam probers. IC testers provide access only to the IC's I/O pins, which is insufficient for silicon debugging. Improved access can be provided by e-beam probers, which allow to observe internal IC nodes by probing on metal wires. However, only the top metal layers can be probed. Lower metal layers can be probed only after removing the top metal layers by focused ion beam equipment. However, the use of focused ion beam equipment largely depends on the circuit layout and is time-consuming. Employing e-beam probers and focused ion beam equipment becomes nearly impossible with the rise of new IC process technologies that allow multiple metal layers and new IC packaging technologies. Hence, DFD to support silicon debugging is becoming a necessity.

An ad hoc DFD technique is to provide additional IC pins through which internal signals can be accessed directly. However, the cost of additional IC pins usually prohibits this approach. A compromise is to multiplex multiple signals on a single IC pin. For instance, Sun's UltraSPARC processor has 15 dedicated pins for monitoring, which allow to observe 75 internal signals [L$^+$95, Lev97].

A structured DFD approach is proposed in [PDW95], where DFD is considered during high-level synthesis of ASICs. Given a set of relevant variables, the CFG is adjusted to provide that these variables are controllable and observable during debugging on the chip's I/O pins. Pipelining techniques and I/O buffers provide that no dedicated IC pins are required.

A frequently used DFD technique is scan-based debugging, which is based on re-using the IC's internal scan paths and boundary-scan paths for debugging purposes [Cro89, HYC89, FM90, Lef90, Tex96]. Scan-based debugging has been promoted particularly by Texas Instruments, providing software tools and test equipment to support scan-based debugging. TI's ASSET (Advanced Support System for Emulation and Test) provides a software environment running on a host PC, from which the scan paths in a target system can be accessed through the IEEE 1149.1 TAP for testing, debugging and emulation [Cro89, HYC89, FM90]. Scan-based debugging is attractive because it provides detailed access to the internals of ICs and the interfaces of embedded cores. Moreover, scan-based debugging requires no additional costs because the scan paths, which are basically DFT facilities, are re-used. However, we showed in section 6.2 that scan-based debugging is restricted. The internal scan paths in the IC's core logic can only be accessed when the IC is halted. Furthermore, scan paths can be accessed only by serially shifting data in or out.

An improved scan-based debugging approach is proposed in [vRBMV97]. The IC runs at full speed, interacting at real-time with its environment, and is halted after a particular number of clock cycles. Next, the IC's state is examined and single step execution is provided by scan-based debugging. The IC is halted by a trigger mechanism, consisting of a trigger control block, shadow registers that contain trigger conditions, and comparators that detect matches between values in functional registers and the shadow registers.

Modern microprocessors and DSPs incorporate advanced DFD facilities that can be used for both silicon debugging and for debugging the software that is executed on the processor. A huge amount of papers is available in which various processors and their DFT and DFD facilities are described, such as the ARM7TDMI [ARM95b, ARM95a, ARM96a, ARM96b], DEC Alpha 21164 [BE94, BE97], Hewlett-Packard HP PA7100LC [JDA93], IBM/Motorola/Apple PowerPC 603 [HVTL94] and PowerPC 620 [Y$^+$95], Intel Pentium [Int93, NG96], Motorola 68HC16Z1 [L$^+$91], Motorola MC68060 [CPC94, KSFM95, Kum97], Sun Microsystems microSPARC [H$^+$94b, Kat94], Sun Microsystems SuperSPARC [PY93], Sun Microsystems SuperSPARC II [HB95, HA95], Sun Microsystems/Texas Instruments UltraSPARC I [L$^+$95, Lev97], and Texas Instruments TMS320C80 [HSBP95].

As an example, we will briefly outline the hardware DFD facilities in the SuperSPARC processors. The SuperSPARC [PY93] microprocessor includes DFT features as full scan, BIST, SRAM test mode for accessing on-chip caches, and IEEE 1149.1 boundary scan. These DFT features are intended for production testing. The processor also incorporates DFD features by means of 15 dedicated observability pins on which 75 internal nodes can be monitored. However, it was felt that these DFT and DFD facilities were insufficient for silicon debugging. Consequently, enhanced debug capabilities were included in the SuperSPARC II [HB95, HA95].

The DFD facilities in the SuperSPARC II consist of a controllable internal clock, full internal scan, IEEE 1149.1 boundary scan and embedded memory debug modes. All debug operations are controlled through the IEEE 1149.1 controller. The internal clock can be stopped through a dedicated pin or through the IEEE 1149.1 controller. When the clock is stopped, the chip enters a debug mode in which the clock operates under control of the IEEE 1149.1 controller. Afterwards, the clock can be restored.

The SuperSPARC II incorporates a single full-scan chain containing 7,939 flipflops. In debug mode, the data in the scan chain can be shifted out ('scandump' [H+94b, Kat94]) under control of the IEEE 1149.1 controller. Furthermore, the contents of the on-chip caches can be extracted ('cachedump'), which is also controlled by the IEEE 1149.1 controller. Other memories, such as the register files and the instruction queue, can be accessed indirectly. For each memory, a set of flipflops can be identified that controls the address and the read signals and observes the data output. These flipflops can be accessed via the scan chain.

6.3.2 Hardware DFD for Embedded System Debugging

Embedded systems typically contain both hardware components and embedded software. Once an error in an embedded system is detected, the aim of embedded system debugging is to locate the fault that caused the error. The fault may reside in the embedded software or in the hardware (physical defect or design error).

Modern microprocessors and DSPs include advanced DFD features to support embedded system debugging. Typically, these DFD features can be accessed through dedicated processor pins ('debug port'). In most processors, the debug port is provided by the IEEE 1149.1 TAP and additional, proprietary pins. The DFD facilities typically allow to enter and leave debug modes (e.g. by stopping and restoring the internal clock), to observe and control the internal scan chains (scan-based debugging), to observe the on-chip memories (e.g. cachedump) or to modify their contents, and to download software code into the on-chip program memory.

Most processors provide execution tracing, which means that the contents of caches, pipelines and processor status registers are output on the processor's I/O pins during execution. For instance, the Motorola MC68060 microprocessor [CPC94] provides a background debug mode, in which a 64-bit state vector is output on the external data and address bus during unused bus cycles in normal operation.

In addition, most processors provide both hardware and software breakpoints. A software breakpoint is a particular instruction (e.g. the instruction 'INT3' on the Intel Pentium) that is inserted in the program code. Whenever the instruction is executed, an exception handler is invoked. Hardware breakpoints are provided by dedicated debugging registers. A breakpoint occurs when the values in the debugging registers match some values on the address, data and/or control buses.

An example of DFD for embedded processor cores is the ARM7TDMI processor core [ARM95b, ARM95a, ARM96a, ARM96b]. The processor core incorporates the boundary-scan architecture and also uses a separate macrocell ('embeddedICE' or 'ICEbreaker') that provides hardware breakpoints. The IEEE 1149.1 TAP is used to access the boundary-scan register and to access the registers in the embeddedICE macrocell. The hardware breakpoints can be set up as breakpoints (on instruction fetches) or as watchpoints (on data load and store operations).

Hardware DFD facilities like scandumps, cachedumps and execution tracing provide observation and control for debugging and hence they can be considered as PCO implementations. Breakpoints can be used during debugging to stop the system when a particular part of software is executed (software breakpoints) or when particular values are detected on buses (hardware breakpoints). Breakpoints themselves do not provide PCO functionality, however the system state can be observed and controlled using other debug facilities after a breakpoint is detected.

6.4 Software Debugging

Typical software debugging techniques [Dac93, Mag93, Spu94] for debugging sequential software programs on a host computer are post-mortem debugging and symbolic debugging. Operating systems can produce a dump of the program state whenever an error is detected. For instance, the Unix operating system produces a coredump file on errors like segmentation faults and bus errors. Post-mortem debugging is performed by examining the coredump file using a symbolic debugging tool. Symbolic debugging tools may work directly with the source code, translating the low-level machine code and addresses into symbolic names. Interactive, symbolic debugging tools offer features like stack traces, breakpoints, watching variables and single-step execution. Obviously, these software debugging techniques are of limited use for debugging both hardware and software in a target system.

A frequently used software debugging technique is software instrumentation, which implies that additional code is inserted in the software to output debugging information or to perform run-time checks. There are various approaches to software instrumentation:

- The programmer may manually instrument the software by inserting output statements. These statements typically output variables at run-time or flags that a particular point in the code has been reached.

  Assertions are a special form of instrumentation instructions. An assertion is an expression which is evaluated at run-time. The expression should be true during normal conditions, but may become false in case of an error. Languages such as C and Eiffel provide direct support for assertions. Assertions can easily be inserted and removed from C code by preprocessing facilities such as conditional compilation.

- The compiler may add run-time checks to the software, such as checking for stack overflows and array bound violations.

- Debugging tools may instrument software at compile-time or link-time to detect errors at run-time. For instance, the Purify tool (see also section 3.5.3) inserts instructions at link-time to check allocation and accesses to heap memory and pointers at run-time.

6.5 Debugging of Distributed Real-Time Systems

Distributed real-time systems are hardware/software systems with real-time timing constraints and distributed processing nodes. In the course of time, various techniques have been developed to support the debugging of real-time behavior in distributed real-time systems [TY95].

A distributed real-time system is typically used to control physical processes in its environment, which requires real-time control and continuous system operation. The sequence of events and the exact timing of events in the system environment are usually difficult to predict. Stringent timing constraints require that the system responds to events within a predefined period of time. A distributed real-time system typically consists of communicating processes running on different processors. The traffic loads on the communication network may cause unpredictable communication delays, which in turn may lead to race conditions when using shared resources. Due to the unpredictable communication delays, race conditions, dynamic process scheduling, and unpredictable sequences and timing of events, the system behavior is non-deterministic. Consequently, the behavior of the system may be non-repeatable when re-running the system with the same input data. The processes in a distributed real-time system can be running on different processors with separate clocks, which makes it very difficult to determine the global system time and the global system state during testing and debugging.

The traditional debugging approach is cyclic debugging. For instance, a sequential software program is typically debugged on a host computer using interactive debugging tools. The program execution is repeatedly stopped to examine the program state, and next the program execution is continued or the program is re-executed to stop at an earlier point in the execution. However, cyclic debugging is inappropriate for distributed real-time systems [MH89, TY95] due to the non-repeatable system behavior, the lack of a global clock and the lack of interactive debugging tools in the distributed target system.

Debugging of distributed real-time systems is typically performed using event-based debuggers. The run-time behavior of the system is monitored and relevant events are recorded. The system execution is viewed as a sequence or several parallel sequences of events, which are stored as event histories. Event-based debugging [Bat89] is based on comparing the recorded event histories with the system specification. Various methods can be used to monitor event histories, ranging from additional statements in software code to hardware test and measurement equipment. One of the main problems with event-based debugging is the probe effect [MRW92]. Any attempt to gain information about the system execution will cause overhead costs and will interfere with the run-time behavior. The probe effect, which is analogous to the Heisenberg Uncertainty principle in quantum mechanics, may mask errors or introduce new errors. There are several solutions to deal with the probe effect in event-based debugging:

- The probe effect can be minimized by using fast monitoring operations in hardware and/or software that cause minimal intrusion in the system behavior. For instance, software code can be instrumented with additional output statements to provide run-time debugging information. Obviously, this approach implies reducing the probe effect but not eliminating it.
- The probe effect can be eliminated to a large extent by using test & measurement equipment

for monitoring. For instance, a logic analyzer can be used for monitoring data on a hardware bus. However, this requires that hardware probes are connected to the hardware bus. The probe effect cannot be eliminated completely due to the parasitic electrical properties of the hardware probes.

- The probe effect can be eliminated completely by designing in probes and leaving them permanently in the system. In fact, we apply this approach when incorporating PCOs in the specification and the subsequent implementation of hardware/software systems. The PCOs can be considered as permanent probes in the system.

  This approach avoids the probe effect in the system implementation. Monitoring can be performed using the built-in PCOs, and there is no need for auxiliary hardware, software and test & measurement equipment. However, the insertion of PCOs may induce a probe effect during system specification, because the insertion of PCOs implies modifying the system behavior, as indicated in section 5.6. Nevertheless, we showed in section 5.6 that we can insert PCOs in a system specification while preserving the externally observable behavior, which completely eliminates the probe effect.

  We argued in section 6.2 and section 6.3 that PCOs can be implemented efficiently in hardware to some extent by using the current hardware DFT and DFD facilities.

The current monitoring techniques for run-time detection and recording of event occurrences in distributed real-time systems can be classified into software monitoring, hardware monitoring and hybrid monitoring. Software monitoring is implemented either by embedding monitoring code inside the application software or by embedding monitoring code inside the system software. Hardware monitoring is implemented either by embedding a monitoring device as a permanent part of the target system, or by using a separate device or coprocessor in the target system. Hybrid monitoring is an intermediary technique using both software and hardware. In the following sections we will discuss these monitoring techniques in depth.

In literature, monitoring in distributed real-time systems is typically used for both performance measurement and software debugging. The major difficulties reside in collecting enough information without causing intolerable monitoring inference, and in determining the relevant events that should be monitored. In general, it is felt that intrusive monitoring is not suitable unless the perturbation caused by intrusion can be precisely quantified and predicted. Non-intrusive monitoring is nearly impossible if monitoring facilities have not been designed in.

6.5.1 Software Monitoring

In software monitoring, the detection and recording of events is performed by software. Software monitoring techniques embed monitoring code either in the target application software (by means of software instrumentation or an additional monitoring process) or in the system software.

Embedding monitoring code in the target application software is very flexible, because a designer can insert dedicated monitoring functions at specific places in the application software. However, this approach is less transparent, because the monitoring code is usually distributed all over the application software. Embedding monitoring code in the operating system is more transparent but less flexible.

The advantages of software monitoring are transparency, flexibility and no needs for additional hardware. However, software monitoring is non-intrusive because the monitoring code requires additional memory space and additional processing time for context switching and execution.

In [CJD91], the application software is annotated to specify the 'observable events' that should be observed during run-time monitoring. The observable events are either label events, which indicate the initiation and completion of a sequence of statements, or transition events, which indicate variable assignments ('watchable variables'). Furthermore, constraints are expressed as invariants on observable events. The application software is instrumented to evaluate the constraints at particular points during execution, and a separate monitoring process is added that continuously monitors some constraints. The software monitor in [CJD91] is extended in [RRJ92, Jah95] for distributed real-time systems that consist of concurrent processes running on multiple processes.

In [JLSU87], a distributed software monitoring system is described to monitor the interprocess communication between concurrent processes. The interprocess communication is monitored by loading a separate interprocess communication protocol that incorporates the monitoring function.

In [TKM89], the ART software monitor is described for monitoring distributed real-time systems. The monitor is embedded as a permanent part of the operating system. The monitor is embedded inside the kernel code that performs process switching. The monitor records changes in process states and it sends the recorded events from the target system to a host system. The event histories are visualized on the host system. The interference of the monitor is predicted during the system analysis phase using a schedulability analyzer for worst case situations.

6.5.2 Hardware Monitoring

In hardware monitoring, the detection and recording of events is performed by dedicated hardware devices. The concept of a hardware monitor is similar to a logic analyzer: run-time information is collected by monitoring the real-time system execution at certain points without intrusion. The monitored data is stored and processed off-line.

A hardware monitoring system typically consists of hardware monitoring devices that are controlled by a central control module. The hardware monitoring devices collect data by snooping signals on control, address and data buses. Relevant events are identified by comparing the monitored data with predefined events such as read/write signals, interrupt signals, specific memory addresses or program instructions.

Hardware monitoring causes no or minimal intrusion in the real-time behavior of the target system. However, hardware monitoring requires additional costs for hardware monitoring devices. Hardware monitoring provides low-level information and is also less flexible and less transparent than software monitoring.

A typical hardware monitoring system is presented in [TFC90, TFCB90]. The hardware monitoring system consists of an interface module and a development module. The interface module is attached to the target system for monitoring and data collection. The development module is a host computer that contains software for initializing the interface module and post-processing the collected data. The interface module copies the internal state of the target processor and records data from buses into memory buffers on predefined trigger conditions. The main feature of the interface module is that it contains a processor that is identical to the target processor. The interface module consists of four functional units:

- The Dual Processor Unit (DPU) contains a processor identical to the target processor. The dual processor mimics the behavior of the target processor.
- The Interface Control Unit (ICU) connects the dual processor to the same buses as the target processor. The ICU prevents the dual processor from writing on the buses.
- The Qualification Control Unit (QCU) samples the buses in the target system on each bus cycle. When the sampled data matches any user-specified conditions, a trigger signal is generated to start recording.
- The High-Speed Buffer Unit (HSBU) is used to store the monitored data from the buses in the target system.

The DPU is synchronized first with the target processor by sending a low-priority interrupt to the target system. The interrupt triggers an interrupt service routine in the target system, which copies the contents of all registers in the target processor to the dual processor. This interrupt is the only interference of the monitoring system in the target system.

After initialization, the dual processor runs synchronously and in parallel with the target processor. When the QCU generates a trigger signal, the dual processor is isolated from the target system and its state is frozen. Hence, the dual processor provides a snapshot of the target processor's state at the beginning of recording. Simultaneously, data from the buses in the target system is recorded in the HSBU. The monitoring activity continues until a stop trigger is generated by the QCU. Finally, the recorded information in the HSBU and the state of the dual processor is transferred to the development module for post-processing. The monitored data provides information on process-level activities (e.g. process creation, process termination, changes in process states, communication and synchronization between processes, interrupts), function-level activities (e.g. procedure calls and returns) as well as instruction-level activities (e.g. step-by-step instruction trace).

6.5.3 Hybrid Monitoring

In hybrid monitoring, both software and hardware are used to perform monitoring [Pla84]. The target software is instrumented to signal the occurrences of relevant events. A hardware monitor is used to detect these signals and to record data. Hybrid monitoring offers the advantages of hardware and software monitoring: it is flexible and transparent, the intrusion of monitoring in the target system is limited, and the costs for hardware monitoring devices are small. However, the intrusion may still be unacceptable and the hardware devices may be too limited or too costly.

In [HW90] a hybrid monitor is presented for a distributed real-time system. Each computing node in the target system contains a Test and Measurement Processor (TMP) for monitoring, recording, and evaluating the node's behavior. The target software on each node is instrumented permanently to generate events which are detected by the TMP. All TMPs are connected via a separate network to a central monitoring station.

The relevant events are either associated to the application software or to the system software. The system software is instrumented to signal dispatcher events (reflecting changes in the states of processes), kernel events (reflecting low-level operations such as initialization of I/O queues), and communication events (reflecting interprocess communication). Instrumentation is performed by inserting *store address, value* instructions at specific points in the software code. The *store* instruction writes through the processor cache and is immediately visible on the system bus, where it is detected by the TMP. The *address* in the store instruction indicates an event class, while the *value* is a parameter which specifies a particular event within the event class. The TMP continuously checks the addresses on the system bus. If an address is within the range of event classes, the values on the address and data bus are stored along with a time stamp. The recorded events are processed first on the TMP and next they are transferred to the central monitoring station. Special synchronization mechanisms are incorporated into the hybrid monitoring system to provide a notion of global time and global state.

In addition, a distributed debugging system is built on top of the hybrid monitoring system. The debugging system consists of local debuggers running on each TMP and a controller running on the central station. The local debuggers cooperate to detect a global breakpoint and to halt the entire system. The central station is informed when the global breakpoint has been reached and the system has been halted. The debugging and monitoring system are used in an incremental test methodology [Hab87].

Related hybrid monitoring systems are the HMON monitoring system [DR92] and the ZM4 monitoring system [H$^+$94a]. These hybrid monitoring systems consist of instrumented target software, local hardware monitoring devices that detect events by bus snooping, and a central processing workstation.

A somewhat different approach to hybrid monitoring has been proposed in [Gor91]. The hybrid monitor is not based on bus snooping, but instead a coprocessor is used to monitor the real-time behavior of software running on a microprocessor. The application software is instrumented with monitoring instructions that are executed by the coprocessor. This concept is analogous to the use of a floating-point coprocessor for executing floating-point instructions. When a monitoring instruction is executed on the microprocessor, the coprocessor is triggered and the coprocessor subsequently performs recording while the target processor proceeds to other tasks. The recorded information consists of time-stamped events (both kernel-level events like process dispatching and application-level events like procedure calls) and possibly additional data.

In [CP95, CP98], a hybrid monitoring system for performance assessment of embedded multiprocessor systems is presented. The MCSE co-design method [Cal93] is used to model the functional behavior of the system as a set of concurrent tasks. The events of interest for monitoring are identified in this functional model, which are basically changes in task states and communication actions between tasks. The hardware or software implementations of the tasks are instrumented to indicate the events. In software tasks, the instruction *capture(i)* is added at locations which correspond to a state modification of a task or communication between tasks. The instruction writes the argument *i* at a specific address to be detected by a hardware monitoring device. The real-time kernel is also instrumented to capture task allocations. In hardware tasks, appropriate statements are added in the VHDL specification. Synthesizer tools are used to implement the instrumented hardware tasks on FPGAs.

Dedicated hardware devices (TransProbe) are used for hardware monitoring. The TransProbe devices monitor events and time stamp them. The monitored data is transferred next to a PC which runs performance analyzer software. The TransProbe devices are connected to the PC by a serial bus (TransBus), which was originally developed for interconnecting transputers on a single bus using a token-ring protocol [CP92]. The bus provides a common reset signal and a global clock signal to all TransProbe devices.

6.6 Discussion

In this chapter we outlined DFT and DFD techniques for both hardware and software. We discussed hardware DFT techniques, hardware DFD techniques, software debugging techniques, and monitoring techniques for distributed real-time systems. In this section we will discuss our findings, focusing on two basic questions: how to implement PCOs in hardware/software and how to implement the infrastructure for accessing the PCOs from an external tester/debugger in the system environment.

In our design for test & debug approach, we insert PCOs in the system specification and we next incorporate them into the hardware/software architecture. The effects of PCO insertion on the system performance can be predicted and dealt with during architecture exploration and the subsequent phases in the design flow. Furthermore, we showed in chapter 5 that we can insert PCOs in a system specification without disturbing the externally observable system behavior. Hence, we eliminate the probe effect already during system specification.

The PCOs may be realized by using test & measurement equipment or the PCOs may be implemented in hardware/software. In this chapter, we focused on the latter. The PCOs are implemented either by dedicated hardware and/or software, or by (re-)using hardware DFT and DFD facilities.

In general, there are two approaches towards test & debug of hardware/software systems: a breakpoint-based approach and a monitoring-based approach. Break-point based test & debug implies running the system for some time and subsequently halting the system. The internal state of the halted state can be observed and controlled, and next the system execution can be continued. Hence, this approach implies that observation and control is performed off-line when the system is halted. Breakpoints and related techniques are used to halt the system on the occurrence of predefined conditions. On the other hand, monitoring-based test & debug implies continuously monitoring the real-time system behavior. This approach allows to observe the internal system behavior at real-time, while the possibilities for controlling the system behavior are limited. The monitored data is usually analyzed off-line.

Both breakpoint-based and monitoring-based techniques are supported by incorporating test &

debug facilities in the hardware/software architecture of the system. These test & debug facilities provide implementations for PCOs and the infrastructure to access PCOs from the system environment.

6.6.1 Breakpoint-Based Test & Debug

Breakpoint-based test & debug implies halting the system on the occurrences of predefined conditions. Subsequently, the internal state of the system can be observed and/or controlled. Breakpoint-based debugging is typically applied in cyclic debugging of sequential software, in which breakpoints can be inserted and removed interactively. In embedded systems, software breakpoints can be inserted by instrumenting the code. However, a more practical and less intrusive approach is to provide hardware breakpoints in which dedicated hardware is used to detect the occurrence of predefined conditions and to halt the system. The concept of breakpoints is not equivalent to the concept of PCO. However, breakpoints can be used to specify on which conditions a PCO should be activated. After all, breakpoints indicate when the system is halted to allow subsequent observation and control activities.

PCOs are directly related to the built-in facilities to observe and control the internal state of a halted system. A cheap solution is to re-use hardware DFT facilities for observation and control, like scan-based debugging and debug modes for reading/writing memories. The test infrastructure of IEEE 1149.1 and 1149.5 test buses, test interfaces and test controllers can be used to access these built-in facilities. Hardware DFT and DFD facilities provide very detailed information on the internal state.

Another powerful debugging technique is single-stepping. By using special debug modes or controlling the internal clock, the system execution can be controlled and observed after every execution cycle. However, single-stepping is a static debugging technique, which cannot be used to debug real-time behavior.

6.6.2 Monitoring-Based Test & Debug

Monitoring-based test & debug implies continuously observing the real-time system behavior. Monitoring is typically performed using Points of Observation (POs) to observe relevant events. The monitored events are transferred to a separate processing station and are analyzed off-line.

Monitoring-based debugging is particularly useful for debugging real-time systems. Modern microprocessors offer monitoring pins on which internal signals can be observed, or background debug modes in which state information is output on the buses in non-used bus cycles during normal operation. In distributed real-time systems, typically monitoring techniques are used that are based on DFD in hardware and/or software.

A cost-effective but limited approach is to use the boundary-scan cells for repeatedly sampling and shifting out the values on IC pins. Boundary-scan cells can be considered as non-intrusive implementations of POs. The IEEE 1149.1 and 1149.5 test infrastructure can be used to transport the observed data to the external environment. However, the resolution of the observed information is rather low, because the sampled data in the boundary-scan cells has to be shifted out serially. Furthermore, the observed information is not very detailed because only the IC pins are observed.

More powerful monitoring-based test & debug techniques have been developed for distributed real-time systems. These monitoring techniques can be classified into software monitoring, hardware monitoring and hybrid monitoring. In software monitoring, the application software and/or the system software is instrumented to detect and record the occurrences of relevant events. Software monitoring is flexible, transparent and inexpensive. However, software instrumentation is non-intrusive and induces the probe effect. In hardware monitoring, the detection and recording of relevant events is performed by dedicated hardware devices that snoop buses. Hardware monitoring is less flexible, less transparent and more expensive than software monitoring. However, hardware monitoring causes no or minimal intrusion and almost eliminates the probe effect. In hybrid monitoring, the software is instrumented to signal the occurrences of relevant events, while the recording of data is performed by dedicated hardware devices. The flexibility, transparency, costs and intrusion of hybrid monitoring are in between software monitoring and hardware monitoring. Both software monitoring, hardware monitoring and hybrid monitoring techniques can be used to implement POs and the infrastructure to access POs from the system environment.

6.7 Summary

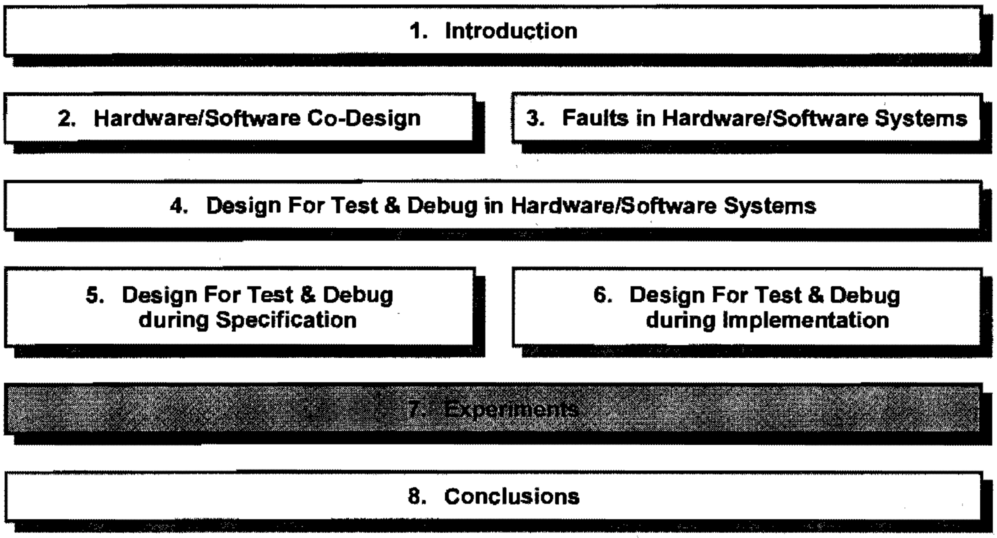
In this chapter we elaborated on design for test & debug during implementation. We outlined the state-of-the-art on hardware DFT and DFD, software debugging, and hardware/software monitoring. We concentrated on two basic questions: how to implement PCOs in hardware/software and how to implement the infrastructure for accessing PCOs from the external system environment.

We argued that there are generally two approaches towards testing and debugging. Breakpoint-based test & debug implies running the system for some time, halting the system on the occurrence of predefined conditions, and subsequently observing and controlling the internal state of the halted system. The concept of breakpoints is not equivalent to the concept of PCO. However, breakpoints can be used to specify on which conditions a PCO should be activated. Monitoring-based test & debug implies continuously monitoring the real-time system behavior. Both breakpoint-based and monitoring-based techniques are supported by incorporating test & debug facilities in the hardware/software architecture of the system.

Hardware DFT facilities on the IC level (e.g. test points, scan cells, boundary-scan cells and test interface elements), and hardware DFD facilities (e.g. scandumps, cachedumps and execution tracing), can reasonably be regarded as low-level hardware implementations of PCOs. The test infrastructure of IEEE 1149.1 on the PCB level and IEEE 1149.5 on the system level provides test buses, test interfaces and test controllers that can be used to access these built-in facilities. In general, hardware DFT and DFD facilities provide very detailed access to the system's internals, however they are somewhat restricted when debugging dynamic, real-time system behavior. Improved control and observation into the internal system operation can be obtained by using dedicated software monitors, hardware monitors or hybrid monitors. These monitors are particularly useful for distributed real-time systems. The monitors can be considered as implementing POs and the infrastructure for accessing the POs from the system environment.

Chapter 7

Experiments


1. Introduction
2. Hardware/Software Co-Design
3. Faults in Hardware/Software Systems
4. Design For Test & Debug in Hardware/Software Systems
5. Design For Test & Debug during Specification
6. Design For Test & Debug during Implementation
7. Experiments
8. Conclusions

In this chapter we present experiments on a case study of an elevator control system. We elaborate on the requirements, the formal specification and the implementation of the elevator control system and we apply our design for test & debug approach. We discuss our experiences and we illustrate the benefits of our design for test & debug approach.

7.1 Introduction

In this chapter, we describe the specification and implementation of an elevator control system (ECS) as a case study to illustrate our design for test & debug approach. The example of the ECS originates from [You89]. We modified and extended Yourdon's description to resemble a realistic system. The ECS may be considered as an embedded system incorporating complex dynamic behavior, a high degree of parallelism, interactions with humans and systems in its environment, and real-time constraints.

This chapter is divided into three parts. In the first part, we give an informal description of the requirements for the ECS. In the second part, we discuss our experiences with the Ward & Mellor method for analysis and design of the ECS. We will apply a simplified version of our design for test & debug approach. In the third part of this chapter, we discuss our experiences with the SHE method for formally specifying the ECS in the POOSL language. We will elaborate on the verification of the specification and on the use of our design for test & debug approach.

7.2 ECS System Requirements

The elevator control system (ECS) controls the operation of four elevators in a building with 40 floors. The elevators transport passengers between the floors. A general requirement is that the ECS should schedule the elevators efficiently and reasonably. For instance, an elevator should not suddenly reverse its direction. We developed a sophisticated algorithm to schedule the four elevators.

7.2.1 System Environment

The ECS interacts with humans and various systems in its environment. A general impression of the ECS environment is shown in figure 7.1.

Prospective passengers that are waiting on a particular floor, can summon an elevator by pressing a summons button. There is a summons panel on each floor with two summons buttons, one marked UP and one marked DOWN. The summons panel on floor 1 has only an UP button, and the summons panel on floor 40 has only a DOWN button. Hence, altogether there are 78 summons buttons (39 UP buttons and 39 DOWN buttons). After a summons button has been pressed, the ECS will schedule an available elevator to service the summons request. Furthermore, the ECS will illumine the pressed summons button to indicate that the summons request has been noticed. Pressing an already lit summons button has no effect. The summons button light is turned off when an elevator arrives.

The interior of each elevator cage is equipped with a destination panel containing 40 destination buttons. There is a destination button for each of the 40 floors. A passenger entering the elevator cage may press a destination button to indicate the floor he wants to be transported to. The ECS illumines a destination button after it has been pressed, to indicate that the destination request has been noticed. Pressing an already lit destination button has no effect. When the elevator cage arrives at the requested floor, the light of the particular destination button is turned off.

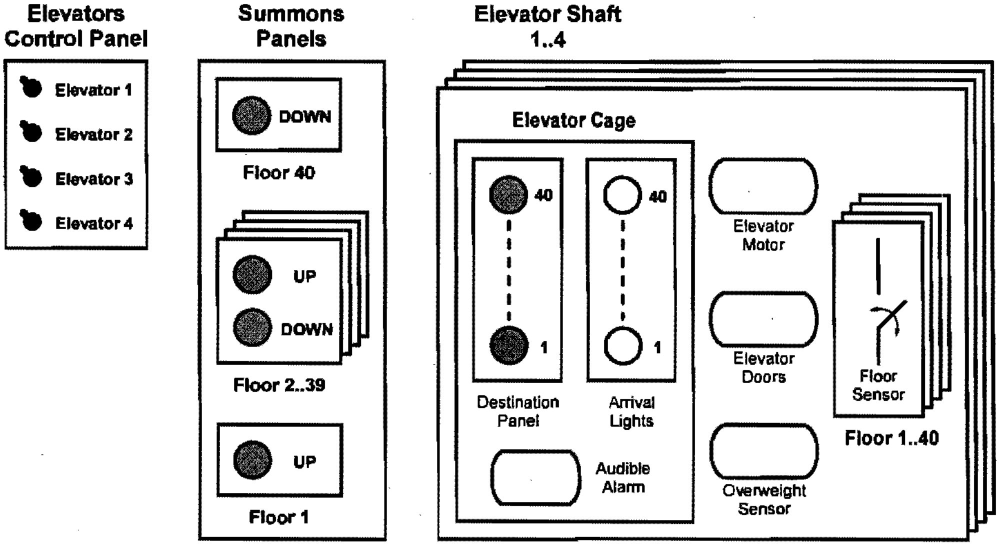


Figure 7.1 Environment of Elevator Control System

The interior of each elevator cage is equipped also with an arrival lights panel, containing one light for each of the 40 floors. The purpose of the panel is to display the current floor number to the passengers in the elevator cage. The ECS turns on an arrival light for a particular floor when the elevator cage arrives at the floor, and turns it off when the elevator arrives at the next floor.

There is a floor sensor on each floor in each elevator shaft. The ECS receives a signal whenever an elevator cage passes a floor sensor. After receiving a signal from a floor sensor, the ECS determines whether the elevator should stop at the floor or pass the floor without stopping.

Each elevator is equipped with an overweight sensor which signals the ECS whenever the elevator cage gets overloaded. The ECS turns on the audible alarm when the elevator cage gets overloaded, and subsequently the elevator cage is not moved. Next, the ECS waits until the overweight sensor signals that the overweight is removed, before turning off the audible alarm.

The ECS controls the motor of each elevator by sending signals to move an elevator cage upwards, downwards, or to stop an elevator cage. There is a separate control system for stopping an elevator cage at the correct position in the elevator shaft. This control system is not part of the ECS.

The doors for each elevator are opened and closed by a separate doors control system, which is not part of the ECS. However, the ECS sends commands to the doors control systems to initiate the opening and closing of elevator doors. A doors control system signals the ECS whenever the elevator doors are fully opened and fully closed. After sending a command to close the doors of a particular elevator, the ECS waits for the doors closed signal. If this signal is not received within 30 seconds, the ECS will turn on the audible alarm. The ECS turns off the audible alarm as soon as the doors closed signal is received.

A system operator controls the operation mode of each individual elevator by means of four switches on the elevators control panel that is connected to the ECS. The operation modes can be normal mode, maintenance mode or disabled. An elevator in normal mode may service both destination requests and summons requests, while an elevator in maintenance mode may service destination requests only. A disabled elevator will not respond to any requests until it is enabled by the operator into normal mode or maintenance mode.

7.2.2 Operation of Elevators

An elevator is either in the disabled, halted, stopped or moving state. When the system operator disables an elevator, the elevator state becomes disabled. An elevator operating in normal mode or maintenance mode may be in the halted, moving or stopped state. Figure 7.2 shows the elevator state transition diagram.

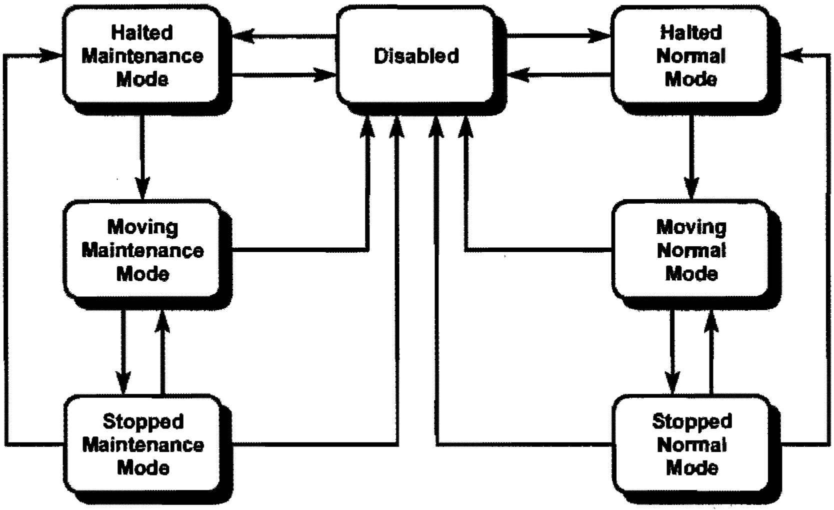


Figure 7.2 Elevator state transition diagram

Initially, all four elevators are disabled. A disabled elevator does not respond to any requests until it is enabled by the system operator at the elevators control panel. When an elevator is enabled, it is switched into normal mode or maintenance mode and its state becomes halted. The ECS next turns on the arrival light in the elevator cage for the current floor.

An elevator in the halted state is parked at a floor with opened doors. A halted elevator is ready to be scheduled to service a destination or summons request in normal mode or a destination request in maintenance mode. An elevator is in the moving state when the elevator cage is actually moving to service a request, or when the elevator cage is about to start moving as the elevator doors are being closed.

An elevator is in the stopped state when it stands still on a floor to service a request. As soon as the elevator goes into the stopped state, the ECS issues the command to open the doors. Subsequently, the ECS waits for an acknowledgement that the doors have been fully opened. If the elevator was scheduled for the floor reached, then the elevator state becomes halted. If the elevator was not scheduled for the floor reached, then the stop is an intermediate stop and after a 10 seconds delay the elevator will start moving again.

The following scenario is performed when a halted or stopped elevator starts moving:

1. The overweight sensor is checked first. The ECS turns on the audible alarm if the elevator is overloaded. Next, the ECS waits until the overweight is removed before turning off the audible alarm.

2. The elevator state is set to moving and next the ECS issues a command to close the elevator doors. The ECS waits for an acknowledgement that the doors have been fully closed. The audible alarm is turned on if the acknowledgement is not received within 30 seconds. The ECS waits until the acknowledgement is received before turning off the audible alarm.

3. Finally, the overweight sensor is checked again because passengers may have entered the elevator during the closing of the elevator doors. The elevator cage starts moving if the elevator is not overloaded. However, if the elevator is overloaded, the doors are opened and subsequently the scenario is restarted from step 1.

The operator may disable an elevator at any moment. The following scenario is performed when the operator disables an elevator:

1. The ECS first checks whether the elevator is scheduled to service a summons request. If this is true, then another available elevator is scheduled to service the summons request.

2. The ECS next checks the elevator state, before setting the elevator state to disabled. If the elevator state is halted or stopped, then the elevator is already parked at a floor with opened doors. If the elevator state is moving, then the elevator is either actually moving between two floors or the elevator is about to start moving (i.e. the doors are being closed). In the first case, the elevator is stopped as soon as it arrives on the next floor and subsequently the doors are opened. In the second case, the elevator doors are opened again and the elevator is parked at the current floor.

3. After the elevator has been parked with its doors opened, all destination requests are removed and the destination button lights as well as the arrival light are turned off.

4. The elevator remains disabled until it is enabled again by the operator. Once an elevator is disabled, it cannot be enabled until step 3 has been performed.

7.2.3 Elevator Scheduling

The purpose of elevator scheduling is to schedule the elevators for servicing destination requests and summons requests. Each elevator has its own destination panel, and therefore a destination request is always mentioned for one particular elevator. On the other hand, a summons request can be serviced by any of the four elevators. When a summons request occurs, all four elevators are examined and an available elevator is scheduled to service the summons request.

Scheduling a particular elevator means that the elevator is directed to move to a particular floor for servicing one particular destination request or summons request. On its way, the elevator may service other, intermediate destination requests and/or summons requests.

Elevator scheduling is performed upon the occurrence of a destination request or a summons request, and upon an elevator state transition.

7.2.3.1 Scheduling and Elevator State Transitions

Elevator scheduling is performed whenever one of the following elevator state transitions occurs (see figure 7.2):

- *Disabled → Halted in Normal Mode*
  Disabled → Halted in Maintenance Mode
  When the operator enables an elevator, the operation mode is set to normal mode or maintenance mode and the elevator state is set to halted. Subsequently, the halted elevator is scheduled. The ECS first checks whether there is a summons request on the current floor.

  a. If there is no summons request on the current floor, then the operation mode is checked. The elevator remains halted at the current floor if the elevator is in maintenance mode. If the elevator is in normal mode, then the elevator is scheduled to service a summons request. The elevator remains halted at the current floor when there are no summons requests pending. Note that there are no destination requests pending, because all destination requests are removed whenever an elevator is disabled.

  b. If there is a summons request at the current floor, then the elevator services this summons request regardless of its operation mode. If there are two summons requests at the current floor, then the elevator services both summons requests simultaneously. Next, the elevator doors remain open for 10 seconds to let passengers enter the elevator and press destination buttons. After the 10 seconds interval expires, the elevator is scheduled to service destination requests. If there are no destination requests and the elevator is in maintenance mode, then the elevator remains halted at the current floor. If there are no destination requests and the elevator is in normal operation mode, then the elevator is scheduled to service a summons request. The elevator remains halted when there are no summons requests pending.

- *Stopped in Normal Mode → Halted in Normal Mode*
  Stopped in Maintenance Mode → Halted in Maintenance Mode
  When an elevator stops at the floor it was scheduled for, then subsequently the elevator state becomes halted. The halted elevator is scheduled as described in section 7.2.4.

- *Stopped in Normal Mode → Disabled*
  Moving in Normal Mode → Disabled
  When an elevator in normal mode is disabled and the elevator was scheduled to service a summons request, then another elevator must be scheduled to service the summons request. Because summons requests are serviced only by elevators in normal mode, the state transitions *Stopped in Maintenance Mode → Disabled* and *Moving in Maintenance Mode → Disabled* do not require elevator scheduling.

7.2.3.2 Scheduling a Destination Request

The ECS receives a destination request for a particular elevator when a passenger presses a destination button for a particular floor. The ECS schedules the elevator to service the destination request only when the elevator state is halted and the elevator is not positioned at the floor of the destination request. In all other situations, no scheduling is required because:

- A disabled elevator will not respond to any destination request.
- A response on a destination request is not required if the elevator is halted or stopped and positioned at the floor of the destination request.
- When the elevator is moving or the elevator is stopped at a floor differing from the floor of the destination request, then the destination request is stored and no explicit scheduling is required. The moving elevator may service the destination request when it passes the floor of the destination request accidentally. However, this does not necessarily happen. Therefore, the elevator is scheduled as soon as the elevator state becomes halted, as described in section 7.2.3.1.

7.2.3.3 Scheduling a Summons Request

The ECS receives a summons request when a prospective passenger at a particular floor presses a summons up button or a summons down button. Explicit scheduling is not required in the following situations:

- There is a halted elevator positioned at the floor of the summons request.
- There is a stopped elevator at the floor of the summons request and the direction of the summons request is the same as the moving direction of the elevator.
- All four elevators are disabled.
- There are no halted elevators available. The summons request is stored and no explicit scheduling is required. The summons request may be serviced when an elevator passes the floor accidentally. However, to guarantee that the summons request will be serviced eventually, scheduling is performed as soon as an elevator state changes to halted.

In all other cases, scheduling is performed whenever a summons request occurs. Scheduling implies selecting a halted elevator in normal mode to service the summons request. When there are multiple halted elevators in normal operation mode, the halted elevator is selected that is nearest to the floor of the summons request, or an arbitrary halted elevator is selected when there are multiple halted elevators at the same distance from the floor with the summons request.

The summons requests are scheduled in a fair way by adopting a FCFS (first-come first-served) strategy. This provides that summons requests are scheduled in the order in which they occur. The FCFS strategy can easily be modeled using a priority counter. When the ECS receives a summons request, it stores the summons request together with the current value of the priority counter. Subsequently the priority counter is increased. The priority counter is decreased every time a summons request is serviced. The ECS always schedules first the summons request with the lowest priority number.

7.2.3.4 Servicing Intermediate Summons Requests

A moving elevator may service summons requests on its way for which the elevator was not scheduled. We refer to such summons requests as intermediate summons requests. Whenever an elevator services an intermediate summons request, the elevator that was originally scheduled for the summons request, must be rescheduled. The rescheduling is performed as soon as this elevator arrives at a floor.

An elevator may not always service an intermediate summons request:

- An elevator in maintenance mode will not service intermediate summons requests.
- An elevator in normal mode that is scheduled to service a destination request, will service an intermediate summons request only if the direction of the intermediate summons request is the same as the moving direction of the elevator.
- An elevator in normal mode that is scheduled to service a summons request, will service an intermediate summons request only if the direction of the intermediate summons request is the same as the moving direction of the elevator, and if the moving direction of the elevator is the same as the direction of the summons request it is scheduled for.

  We will clarify this requirement with the following example. Suppose that an elevator at floor 5 is scheduled to service a summons down request at floor 20. The elevator starts moving upwards to floor 20. After the elevator has reached floor 20, it will probably start moving downwards because passengers entering the elevator at floor 20 will generate destination requests for the lower floors. When the elevator is moving up from floor 5 to floor 20, it will not service intermediate summons requests. Intermediate summons down requests would not tolerate that the elevator is first moving up to floor 20. Intermediate summons up requests would not tolerate that the elevator is moving down after floor 20 has been reached, because passengers may generate destination requests for floor 21 to 40.

When an elevator services an intermediate summons request, the priority counter is decreased by one as explained in the previous section. In addition, the priorities of those pending summons request is decreased whose priority is higher than the priority of the intermediate summons request.

7.2.4 Stopping an Elevator

The ECS receives a signal from a floor sensor whenever an elevator cage arrives at a floor. The ECS next turns off the arrival light in the elevator cage for the previous floor and turns on the arrival light for the reached floor. Subsequently, the ECS decides whether the elevator should stop or not at the reached floor.

First the elevator state is checked, which is either disabled or moving. If the elevator state is disabled, then the elevator stops at the floor reached. The ECS signals the elevator motor to stop the elevator cage and the ECS also signals the doors control system to open the doors. The ECS next waits for an acknowledgement that the doors have been fully opened. Finally, the ECS removes all destination requests and turns off the destination button lights and the arrival light.

If the elevator state is moving, then the ECS decides whether the elevator should stop or not at the reached floor by considering the following four conditions: is the elevator operating in normal mode or maintenance mode; is the elevator scheduled to service a destination request or a summons request; is there a destination request or a summons request at the floor reached which the elevator must service; and, is the rescheduled flag set? The rescheduled flag indicates that the elevator is scheduled to service a summons request, but the summons request has already been serviced by another elevator that passed the floor of the summons request accidentally. These conditions are evaluated in the following scenario:

1. If the elevator is scheduled to service a destination request, then the elevator stops at the floor reached only if there is a destination request and/or a summons request that the elevator may service.

   A destination request at the floor reached is either the destination request that the elevator was scheduled for or an intermediate destination request. In both cases, the ECS stops to service the destination request.

   A summons request at the floor reached is serviced only if the direction of the summons request is the same as the moving direction of the elevator. Furthermore, the elevator should be operating in normal mode or there should also be a destination request at the floor reached. In the latter case, the elevator stops at the floor to service the destination request and it services the summons request at the same time.

   If the elevator stops at the reached floor, the elevator state is set to stopped. The ECS turns off the light of the destination button and/or the summons button that is being serviced. Next, the ECS commands the elevator doors to be opened. After receiving the acknowledgement signal that the doors have been fully opened, the ECS waits for 10 seconds. Subsequently, the elevator is scheduled in the following way:

   (a) The elevator is scheduled for a pending destination request in the current direction.

   (b) When there are no pending destination requests in the current direction, the elevator's moving direction is reversed. The ECS now checks whether there is a summons request at the floor reached in the reversed direction. If this is true, the elevator services the summons request: the summons button light is turned off and the elevator doors stay opened for another 10 seconds. Subsequently, the scheduling is restarted from step (a). Note that the summons request is serviced regardless of the elevator's operating mode.

   (c) When there is no summons request at the floor reached in the reversed direction, the elevator is scheduled for a destination request in the reversed direction.

   (d) When there are no destination requests pending in the reversed direction and the elevator is in normal mode, the elevator is scheduled for a pending summons request. If there is no summons request pending or if the elevator is in maintenance mode, then the elevator state is set to halted.

2. If the elevator is scheduled to service a summons request, then it is operating in normal mode. The subsequent actions depend on the rescheduled flag.

   (a) The following scenario is performed if the rescheduled flag is not set:

- The elevator stops if the elevator is scheduled for the floor reached. The elevator state is set to stopped, the elevator doors are opened, and the doors stay opened for 10 seconds. Next, the elevator is scheduled as described in step 1. If there is also a destination request for the floor reached, then this destination request is serviced at the same time.
- If the elevator is not scheduled for the floor reached, then the elevator stops only if there is a destination request and/or an intermediate summons request at the floor reached. An intermediate summons request is serviced only if the moving direction of the elevator, the direction of the intermediate summons request, and the direction of the scheduled summons request are the same.
- In all other cases, the elevator will not stop at the floor reached.

(b) The following scenario is performed if the rescheduled flag is set:

- If there is a destination request and/or an intermediate summons request at the floor reached, then the elevator will service these requests. An intermediate summons request is serviced only if the moving direction of the elevator is the same as the direction of the summons request. If the elevator services a request, the elevator stops and the elevator state becomes stopped. The elevator doors are opened for 10 seconds and next the elevator is scheduled as described previously in step 1.
- If there is no destination request and no intermediate summons request at the floor reached, then the elevator continues moving if there is a destination request pending in the current direction. Else, the elevator is stopped and the elevator state becomes stopped. The elevator doors are opened for 10 seconds and next the elevator is scheduled as described previously in step 1.

7.3 Specification and Design Using SAD

Initially, we used Ward & Mellor's method [WM86] for Structured Analysis & Design (SAD). We applied Ward & Mellor's analysis method to capture the ECS requirements into a system specification. Next, we implemented the ECS in software using a small operating system. We refer to section 2.6 for a discussion on the advantages and the shortcomings of SAD methods.

7.3.1 ECS Specification

We used Ward & Mellor's analysis method to transform the ECS system requirements as described in section 7.2, into a detailed system specification. Ward & Mellor's analysis method provides an environment-based modeling approach to capture the system behavior in an essential model. The essential model consists of an environmental model, describing objects and events in the system environment, and a behavioral model, describing the system behavior in response to events in the environment.

The environmental model consists of a context diagram, describing the interfaces between the system and objects in the environment, and an event list, describing the external events in the environment to which the system must respond. Figure 7.3 shows the ECS context diagram.

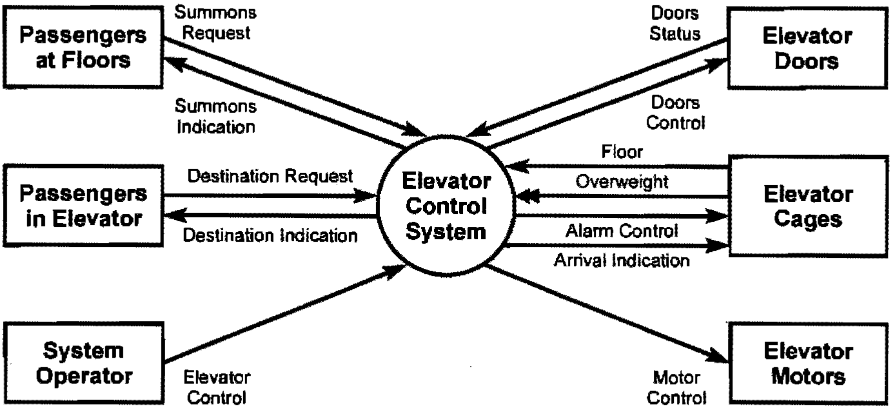


Figure 7.3 ECS context diagram

The behavioral model is a hierarchical model containing transformations (processes), stores, event flows, time-discrete data flows and time-continuous data flows. The behavioral model is created by considering the system behavior in response to external events as described in the event list. Figure 7.4 shows the top level of the behavioral model for the ECS. The process *Handle Passengers & Operator Input* deals with passengers who press destination buttons and summons buttons and with the operator who enables and disables elevators. The process *Schedule Elevators* carries out the elevator scheduling. The process *Control Elevator Motor & Doors* controls the motor and the doors of an individual elevator. The process *Handle Floor Reached* determines whether an elevator should stop or not every time a floor is reached. There are four processes *Control Elevator Motor & Doors* and also four processes *Handle Floor Reached*, one for each individual elevator. The process *Manage Data Access* provides mutual exclusive access to the data stores that contain the state information of the individual elevators.

We discussed the use of event-traces for validation and verification of an essential model in section 4.8. An event-trace describes a part of the system behavior as a sequence of events. An event-trace starts with an external event from the event list, i.e. an event in the system environment, that is input to the system. The event occurs when the system is in a particular state. The event-trace next describes all the subsequent internal events, i.e. events that are produced and consumed by processes in the behavioral model. An event-trace also describes the output events that are sent to objects in the system environment. An event-trace defines the causal relations between the various events. An event-trace typically describes some finite system behavior, starting with an external input event an ending with an output event.

An example of an event-trace is shown in figure 7.6. The event-trace is initiated by a summons request event in the system environment. Upon the occurrence of the event, the system is presumed to be in a state where all four elevators are in the halted state and positioned at floor 1. The summons request models a prospective passenger pressing a button at an arbitrary floor (different from floor 1).

We applied a simplified version of our design for test & debug approach in the ECS behavioral model. We required that each internal event is directly observable in the system environment.

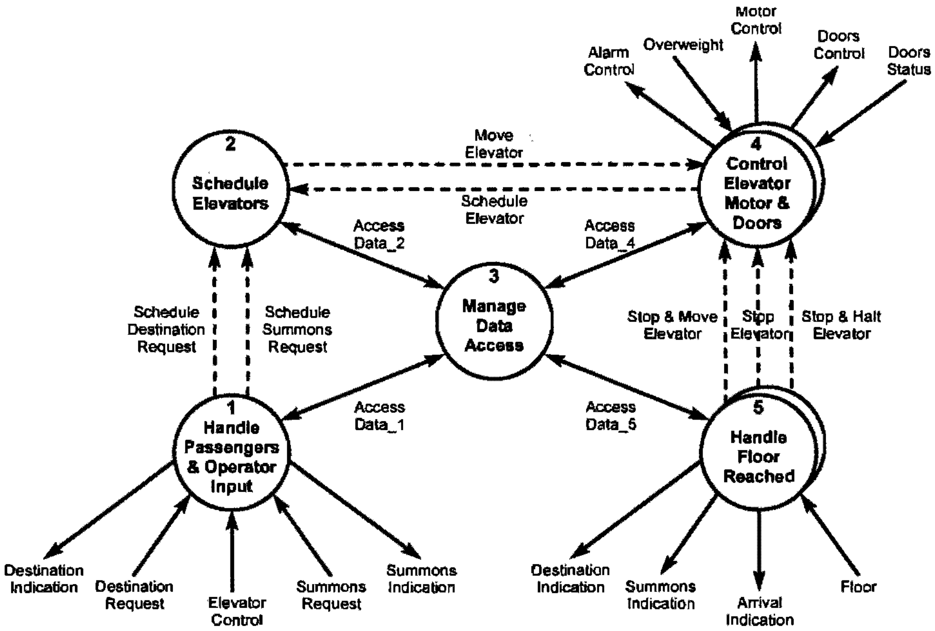


Figure 7.4 ECS behavioral model (top level)

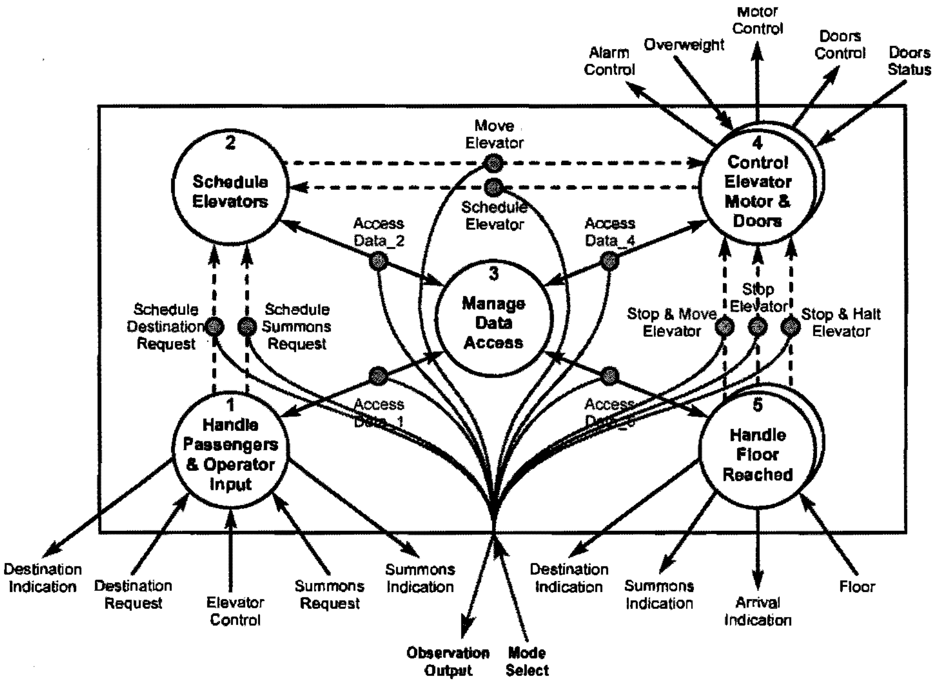


Figure 7.5 ECS behavioral model (top level) with POs

Therefore, we inserted Points of Observation (POs) to monitor all internal events. The modified behavioral model is shown in figure 7.5.

Event	Source	Destination
1. Summons Request	Environment	Handle Passengers & Operator Input
2. Summons Indication	Handle Passengers & Operator Input	Environment
3. Data Access 1	Handle Passengers & Operator Input	Manage Data Access
4. Data Access 1	Manage Data Access	Handle Passengers & Operator Input
5. Schedule Summons Request	Handle Passengers & Operator Input	Schedule Elevators
6. Data Access 2	Schedule Elevators	Manage Data Access
7. Data Access 2	Manage Data Access	Schedule Elevators
8. Move Elevator	Schedule Elevators	Control Elevator Motor & Doors
9. Doors Control	Control Elevator Motor & Doors	Environment
10. Doors Status	Environment	Control Elevator Motor & Doors
11. Data Access 4	Control Elevator Motor & Doors	Manage Data Access
12. Data Access 4	Manage Data Access	Control Elevator Motor & Doors
13. Motor Control	Control Elevator Motor & Doors	Environment

Figure 7.6 Example of an event-trace

7.3.2 ECS Implementation

We derived a software implementation for the ECS from the essential model. The software architecture is depicted in figure 7.7.

The processes in the behavioral model are implemented as application software processes running on a microprocessor. The processes are scheduled by a dispatcher that is part of MBOS (Message Based Operating System). The application processes communicate with each other by writing messages into FIFO queues. A message contains the actual message, the name of the sending process, the name of the destination process and possibly a number of parameters. A priority number is assigned to each queue. The MBOS dispatcher looks for a message in the queues, starting from the queue with the highest priority number. If there is a message present, the dispatcher starts the destination process indicated in the message. An application process is a collection of C functions that is organized as a finite-state machine. Whenever a process receives a message from another process through the dispatcher, the current process state and the message determine which C function is executed. After the function is terminated, the dispatcher is started to read the next message from the queues. Hence, there is a single thread of control in the system, alternately executing the dispatcher and a function in an application process. In addition, an application process can start and stop timers. Whenever a timer expires, the timer writes a message into a queue. Messages from timers are treated in the same way as messages from application processes. A hardware timer provides interrupt signals at a fixed frequency. A timer interrupt invokes the execution of an interrupt handler that decreases the counters in the timers. A timer expires when its counter becomes zero.

As stated, we implemented the ECS in software (C programming language) using a small operating system (MBOS). The processes and flows in the Ward & Mellor behavioral model could be mapped rather easily onto the software architecture imposed by MBOS. We embedded the ECS software implementation into an interactive, software simulator running on a PC. The simulator

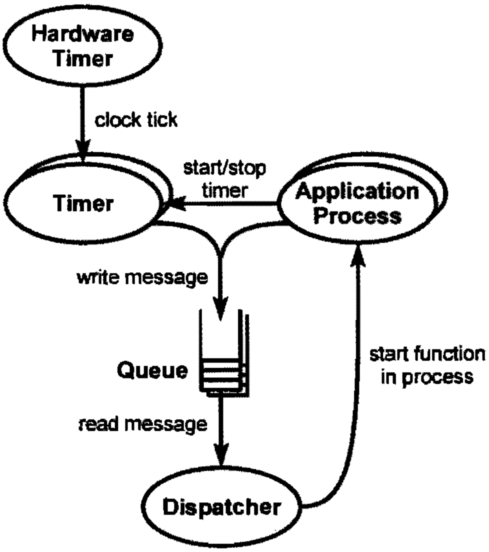


Figure 7.7 ECS software architecture

consists of the ECS application processes, MBOS and a graphical user interface (GUI). The GUI provides a graphical screen showing the movements of the four elevator cages, summons buttons, destination buttons, arrival lights and a menu window. The user can make selections in the menu window by keyboard inputs to simulate passengers pressing buttons, the operator enabling and disabling elevators, and overweight sensors signaling overload situations. The GUI is implemented as just another application process in the MBOS architecture as depicted in figure 7.7. In addition, we added a second application process that models the behavior of objects in the ECS environment, such as the doors control systems and the floor sensors. The simulator provides a powerful interactive environment that allows the user to act as a passenger or the system operator. A limitation of the GUI is that the user can input only one event at a time. For instance, a passenger pressing two buttons simultaneously can only be simulated as two successive user inputs. Another limitation is that the simulation proceeds at real-time, and hence the interactive user has to anticipate to the real-time behavior of the ECS software. It may be difficult to simulate scenarios in which the user inputs should occur at predefined points in time. For instance, the operator may disable an elevator when the elevator is moving between two floors or closing the doors. Simulating this scenario requires that the user inputs the disable message within the time interval that the elevator is moving or closing the doors.

We incorporated the POs as shown in figure 7.5 into the ECS software implementation. As stated, the application processes in the MBOS architecture communicate by sending messages to each other. Sending a message corresponds to writing a message into a FIFO queue, and receiving a message corresponds to the dispatcher removing a message from the FIFO queue and starting the destination process. Hence, every message passes through the dispatcher. We extended the MBOS dispatcher with a simple monitoring function. Every time the dispatcher reads a message from a queue and starts a process, the monitoring function is called which writes the message to a file on disk. Hence, the monitoring function records the event-trace during a simulation session.

The monitoring function provides a simple implementation of all POs in the system specification. The entire software simulation system consists of about 2,400 lines of C code, of which the monitoring function takes about 0.5%. The monitoring function has a minor impact on the system performance, because it is executed only when the dispatcher reads a message from a queue.

The event-traces recorded by the monitoring function during simulation, turned out to be very useful. During simulation, the entire system deadlocked every now and then for no apparent reason. In a deadlock situation, the simulator did not respond to any user inputs and needed to be restarted. Unfortunately, we could not reproduce deadlock situations afterwards when replaying the same scenarios. These symptoms seemed to point to a timing-related error. We already discussed the limitation of the interactive GUI when interacting with the real-time behavior of the ECS software. This limitation impedes to replay scenarios with exactly the same timing of user inputs, which explains why we could not reproduce deadlock situations.

We analyzed the event-traces obtained from simulations that ended in a deadlock situation. We found that the errors were due to race conditions. Initially, we used a single FIFO queue for all messages. A message generated by a user input or a timer expiration corresponds to an external event, which subsequently starts a particular scenario. However, user inputs and timer expirations may occur at arbitrary points in time. Consequently, there may be multiple, concurrent scenarios in the system. For instance, while the system is executing a scenario started by a summons request, additional scenarios may be started such as the scenario for handling a destination request or an elevator arriving on a floor. The application processes that are involved in a scenario, communicate by writing messages into the FIFO queue. Hence, when there are multiple scenarios, the message sequences pertaining to the various scenarios are stored in the FIFO queue in an interleaved order. The dispatcher reads these messages from the FIFO queue and starts the application processes for the various scenarios in the same interleaved order. Errors may occur when the application processes in the various scenarios interfere in each other's behaviors. All application processes access the process *Manage Data Access* that contains data stores for the state information of the individual elevators. When executing a particular scenario A, an application process may read the data stores, perform some computations and send a message to another application process which subsequently updates the data stores. Meanwhile, an application process in another scenario B may already have updated the data stores. Hence, the data of scenario B is overwritten by the data of scenario A. Subsequently, the application processes in scenario B start using incorrect data which causes incorrect behavior. This is a typical example of a race condition as described in section 3.5.4.

We solved the problem of race conditions by using two FIFO queues with different priorities. The messages generated by user inputs and timer expirations, which correspond to external events, are stored in the queue with the lower priority, while all other messages, which correspond to internal communications, are stored in the queue with the higher priority. The modified implementation with two queues provides that there is only one active scenario in the system at a time, which excludes the occurrence of race conditions.

The experiences obtained from the case study clearly demonstrate the advantages of our design for test & debug approach for dealing with timing-related errors. We inserted POs in the system specification and we next incorporated the POs into the software implementation. We could quickly debug timing-related errors due to race conditions by analyzing the monitored event-traces. Standard, interactive debugging tools are inadequate to deal with such timing-related errors in the real-time system behavior.

We applied only a simplified version of our design for test & debug approach. We did not perform a testability analysis to determine the essential communication interactions and state information that should be monitored. Instead, we simply required that all internal events should be monitored. Furthermore, we did not analyze in depth the effects of the POs on the system behavior. We only showed that the monitoring function has a minor impact on the program size and on the system performance. In the next section, we will address these topics more thoroughly.

7.4 Specification and Design Using SHE/POOSL

We also applied Van der Putten & Voeten's SHE method [vdPV97] to the ECS. We already discussed some aspects of the SHE method and the POOSL language in the sections 2.6, 4.8 and 5.3.3. In this section we provide an overview of the SHE method and the POOSL language, we discuss the formal specification of the ECS in the POOSL language and we outline the use of our design for test & debug approach.

7.4.1 SHE and POOSL

SHE (Software/Hardware Engineering) [vdPV97] is a specification method to model the behavior of concurrent, reactive hardware/software systems. SHE provides an activity framework that guides the development of system specifications. The focus is on concurrency, synchronization, communication, scenarios and distribution. SHE uses several views to model a system, such as a behavior view and an architectural view. These separate views are combined into a unified system model that is expressed in the formal specification language POOSL (Parallel Object-Oriented Specification Language) [Voe95a, Voe95b, vdPV97].

POOSL provides process objects and data objects. Data objects are comparable to objects in traditional object-oriented programming languages. A data object is a passive entity that becomes active only when it receives a message. When activated, a data object performs some sequential behavior, possibly outputs a message and next becomes inactive again. Data objects are instantiated from data classes. They incorporate instance variables, local variables and methods. Data objects model data structures and the operations that can be applied to these data structures.

Process objects are instantiated from process classes. They incorporate instance variables, local variables and methods. In contrast to data objects, process objects can exchange messages without becoming passive and they can have infinite, non-terminating behavior. Process objects typically behave like state machines. The behavior of a process object depends on its past and on the receipt of messages. Furthermore, the behavior of a process object can be interrupted or aborted on the receipt of particular messages. The internal data of process objects is represented by data objects. Process objects can communicate by exchanging messages that contain data objects as parameters.

A system specification is modeled in POOSL as a static structure of communicating process objects. The process objects behave as (asynchronous) concurrent, self-contained, autonomous, relatively independent and weakly coupled entities. Process objects are interconnected by channels and they communicate by exchanging messages over the channels. Message exchange is based on the synchronous pair-wise message-passing mechanism of CCS [Mil80, Mil89]. Broadcast communication is supported as well. Hierarchy can be introduced by grouping process objects and channels into clusters.

The interconnection structure and the communication between process objects is visualized in an object instance model. An object instance model consists of message flow diagrams that visualize process objects and communication flows, and instance structure diagrams that visualize process objects and communication channels.

We discussed in section 4.8 the importance of scenarios in the SHE method for creating a system specification. By playing scenarios, the designer can identify objects and their communication flows and reason about behavior, the ordering of events, and the reactions of cooperating processes. A scenario can be visualized in message flow diagrams, showing the process objects that participate in the scenarios and their communications.

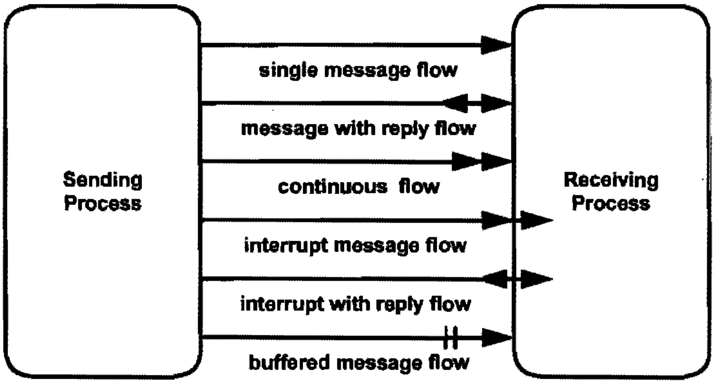


Figure 7.8 Message flows

The communication flows in message flow diagrams are visualized as shown in figure 7.8. A *single message flow* represents one-way, synchronous message passing between two objects, which is the basic communication primitive. The message is passed instantaneously from the sending process to the receiving process in a rendez-vous. The *message with reply flow* represents two subsequent rendez-vous communications. The second rendez-vous follows immediately on the first rendez-vous, representing a reply to the sending process. The *continuous flow* represents a time-continuous flow that has a value at every instant in time. The *interrupt message flow* represents synchronous message passing that can be forced to happen after the receiving process finishes its current atomic process statement. An interrupt message flow in POOSL can be an interrupt or an abort. An interrupt indicates that the receiving process can resume its current behavior after the interrupt behavior is finished. An abort indicates that the current behavior is interrupted and cannot be resumed. The *interrupt with reply flow* represents an interrupt message followed immediately by a reply to the sending process. The *buffered message flow* represents asynchronous communication. Asynchronous communication is modeled by introducing a buffer. The sending process can send a message to the buffer without knowing whether the receiving process is willing to receive a message from the buffer.

The behavior of a process object can be described in POOSL using the process statements listed in figure 7.9. Furthermore, a process object can have infinite, non-terminating behavior. Such infinite behavior is described by methods in the process object that incorporate tail recursion (see also section 5.3.3.1).

Statement	**Meaning**
$S_1; S_2$	Sequential composition of statements S_1 and S_2.
if E *then* S_1 *else* S_2 *fi*	Conditional selection of statement S_1 or statement S_2.
while E *do* S *od*	Repeat statement S while condition E is true.
sel S_1 *or* S_2 *les*	Non-deterministic selection of statement S_1 or statement S_2.
$[E]\ S$	Guarded command; wait until condition E becomes true before executing statement S.
$m(E_1, \ldots, E_n)(p_1, \ldots, p_m)$	Call method m; expressions $E_1, \ldots, E_n$ are evaluated and bound to input parameters of m; the results of m are bound to output parameters $p_1, \ldots, p_m$.
$ch!m(E_1, \ldots, E_n)$	Send message m with the evaluated expressions $E_1, \ldots, E_n$ as parameters on channel ch.
$ch?m(p_1, \ldots, p_m \mid E)$	Receive message m with parameters $p_1, \ldots, p_m$ from channel ch when condition E is true.
S_1 *interrupt* S_2	Enable interruption of statement S_1 with alternative statement S_2.
S_1 *abort* S_2	Enable abortion of statement S_1 with alternative statement S_2.

Figure 7.9 *POOSL process statements*

The formal semantics of POOSL is a computational interleaving semantics, which implies that the behavior of a system is interpreted as a sequential, interleaved execution of all atomic actions. Hence, nothing happens really simultaneously except for the sending and receiving of a message in a rendez-vous. Atomic actions are assumed to take zero time. This semantic model can be considered as using a time scale with a fine enough grain, such that each atomic action can be mapped upon this time scale at an unique point in time. POOSL has been extended with a notion of real-time by means of the *delay* primitive [Gei96].

7.4.2 ECS Specification

We used the SHE method to create a formal system specification of the ECS in the POOSL language. Figure 7.10 shows the ECS instance structure diagram and figure 7.11 shows the ECS message flow diagram. The specification of the ECS consists of nine concurrent process objects. The process *Summons Handler* deals with the summons buttons. The four processes *Individual Elevator Control* are instances from the class *Individual Elevator Control*. Each process corresponds to one particular elevator, which is specified by an instantiation parameter containing the

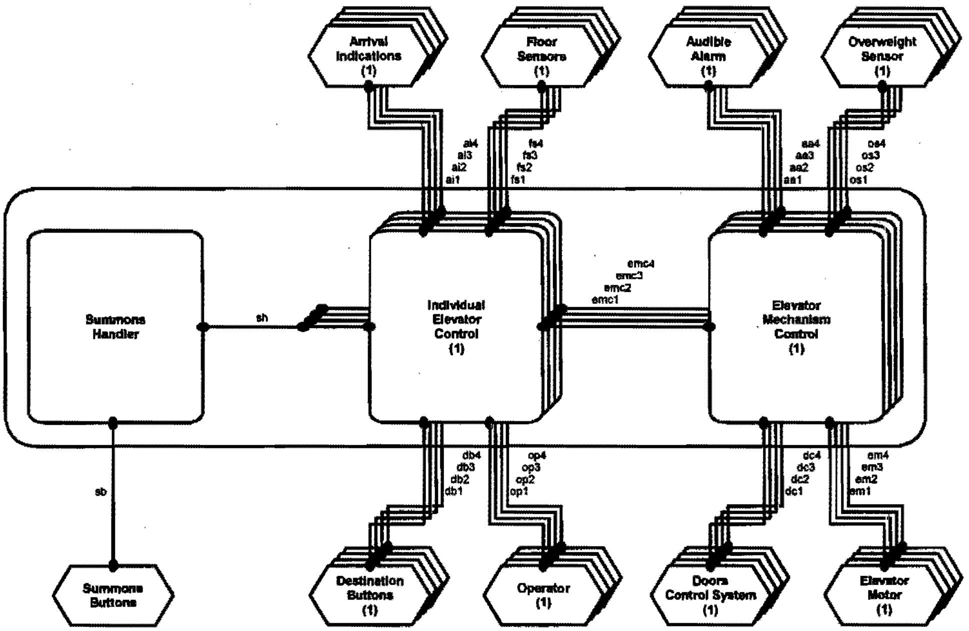


Figure 7.10 ECS instance structure diagram

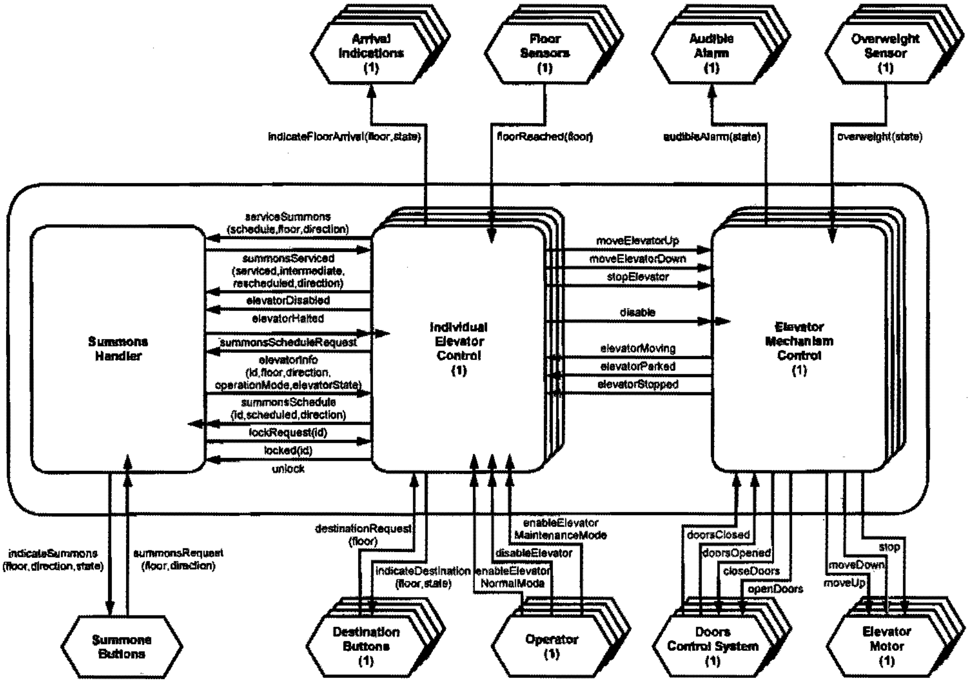


Figure 7.11 ECS message flow diagram

elevator identification number. Hence, the process *Individual Elevator Control (1)* corresponds to the elevator with identification number 1. A process *Individual Elevator Control* deals with the destination buttons, the arrival indication lights, the floor sensors and the operator input for one particular elevator. The four processes *Elevator Mechanism Control* are instances from the class *Elevator Mechanism Control*. Each process corresponds to one particular elevator, indicated by the instantiation parameter containing the elevator identification number. A process *Elevator Mechanism Control* deals with the audible alarm, the overweight sensor, the doors and the motor for one particular elevator. In the following subsections we detail the behavior of these process objects and also the behavior of the process objects in the system environment.

7.4.2.1 Process Objects in the System Environment

The process object *Arrival Indications* models the 40 arrival lights in an elevator cage. There is a separate process for each of the four elevators. The ECS can turn on or off an arrival light in an elevator cage for a particular floor by sending the message *indicateFloorArrival(floor, on)* or *indicateFloorArrival(floor, off)*.

The process object *Floor Sensors* models the 40 floor sensors in an elevator shaft. There is a separate process for each of the four elevators. The ECS receives an interrupt message *floorReached(floor)* whenever an elevator sensor detects the arrival of an elevator cage on a particular floor.

The process object *Audible Alarm* models the audible alarm in an elevator cage. There is a separate process for each of the four elevators. The ECS can turn on or off the audible alarm for a particular elevator by sending the message *audibleAlarm(on)* or *audibleAlarm(off)*.

The process object *Overweight Sensor* models an overweight sensor. There is a separate process for each of the four elevators. The ECS receives an interrupt message *overweight(on)* whenever the overweight sensor detects an elevator cage getting overloaded. The ECS receives an interrupt message *overweight(off)* when the overweight sensor notices that the overweight is removed.

The process object *Elevator Motor* models an elevator motor. There is a separate process for each of the four elevators. The ECS can send the messages *stop*, *moveDown* and *moveUp* to an elevator motor to stop, move down or move up the corresponding elevator cage.

The process object *Doors Control System* models a doors control system. There is a separate process for each of the four elevators. The ECS can send the message *openDoors* or *closeDoors* to initiate the opening or closing of the elevator doors. When the doors are fully opened or closed, the ECS receives the messages *doorsOpened* or *doorsClosed*.

The process object *Operator* models the system operator. Although there is only one system operator, the operator can enable or disable each individual elevator by a separate switch. Therefore, we introduce a separate process *Operator* for each of the four elevators. The ECS can receive the interrupt messages *disableElevator*, *enableElevatorNormalMode* or *enableElevatorMaintenanceMode*.

The process object *Destination Buttons* models the 40 destination buttons in an elevator cage. There is a separate process for each of the four elevators. The ECS receives the message *destinationRequest(floor)* whenever a passenger presses the destination button for a particular floor. The ECS can turn on or off the light of a destination button by sending the messages *indicateDestination(floor, on)* or *indicateDestination(floor, off)*.

The process object *Summons Buttons* models the 78 summons buttons. The ECS receives the interrupt message *summonsRequest(floor, direction)* whenever a prospective passenger presses the summons button on a particular floor in a particular direction. The ECS can turn on or off the light of a summons button by sending the message *indicateSummons(floor, direction, on)* or *indicateSummons(floor, direction, off)*.

7.4.2.2 ECS Process Objects

The process *Individual Elevator Control* models a subsystem of the ECS that deals with the destination buttons, the arrival indication lights, the floor sensors and the operator input for one particular elevator. The entire state information of a particular elevator is stored in the instance variables of the process *Individual Elevator Control*, as indicated in figure 7.12.

Instance Variable	Type	Meaning
floor	1, . . . , 40	position of elevator cage in shaft
direction	*up, down*	moving direction of elevator cage
operationMode	*disabled, normalMode, maintenanceMode*	elevator operating mode
elevatorState	*disabled, halted, moving, stopped*	elevator state
elevatorSchedule	*none, destinationRequest, summonsRequest*	elevator schedule
rescheduled	*true, false*	rescheduled flag (see section 7.2.4)
doorsAreClosing	*true, false*	flag set when elevator doors are closing
doorsAreOpening	*true, false*	flag set when elevator doors are opening

Figure 7.12 *Elevator state information*

The process *Individual Elevator Control* receives the message *destinationRequest(floor)* whenever a passenger presses a destination button. The process next schedules the destination request as described in section 7.2.3.2. The process receives an interrupt message *enableElevatorNormalMode*, *enableElevatorMaintenanceMode* or *disableElevator* whenever the operator changes the operation mode of the elevator. The process handles the operator input as described in the sections 7.2.2 and 7.2.3.1. Finally, the process receives the interrupt message *floorReached(floor)* whenever the elevator cage arrives at a floor. The process handles this message as described in section 7.2.4.

In the scenarios for handling the messages from the operator and the floor sensors, the process *Individual Elevator Control* must check whether there is a summons request on the current floor that the elevator may service. The information on pending summons requests is contained only in the process *Summons Handler*. Therefore, the process *Individual Elevator Control* interacts with the process *Summons Handler* by sending the message *serviceSummons* and receiving the message *summonsServiced*.

The process *Elevator Mechanism Control* models a subsystem of the ECS that deals with the audible alarm, the overweight sensor, the doors and the motor for one particular elevator. As shown in figure 7.10, each process *Elevator Mechanism Control (id)* communicates over a channel with the process *Individual Elevator Control (id)*, where *id* indicates the elevator identification number. When the process *Elevator Mechanism Control* receives the message *moveElevatorUp*, *moveElevatorDown* or *stopElevator*, the process performs the scenario as described in section 7.2.2 to control the audible alarm, the elevator doors and the elevator motor. The process receives the interrupt message *overweight* whenever an overload condition appears or disappears. The process returns the message *elevatorMoving*, *elevatorParked* or *elevatorStopped* as acknowledgement to the process *Individual Elevator Control*.

The process *Elevator Mechanism Control* receives the interrupt message *disable* from the process *Individual Elevator Control* whenever the operator disables the elevator. When the interrupt message is received while the doors are being closed, then the doors are opened again and the elevator is parked at the current floor with doors opened. The message *elevatorParked* is returned to the process *Individual Elevator Control*. However, the process may receive the *disable* interrupt message after the elevator doors have been closed and the message *moveUp* or *moveDown* has been sent to the elevator motor. In this case, the elevator cannot be disabled until the next floor is reached. Now the message *elevatorMoving* is returned to the process *Individual Elevator Control*.

The process object *Summons Handler* models the subsystem of the ECS that deals with the summons buttons. The process receives the interrupt message *summonsRequest(floor, direction)* whenever a prospective passenger presses a summons button. The process next schedules the summons request as described in section 7.2.3.3. Scheduling a summons request requires evaluating and comparing the state information of all four elevators. However, the elevator state information is contained only in the processes *Individual Elevator Control*. Therefore, the process *Summons Handler* broadcasts the interrupt message *summonsScheduleRequest* to each of the four processes *Individual Elevator Control*. These processes return the elevator state information by sending the message *elevatorInfo*. The process *Summons Handler* now schedules the summons request and sends the message *summonsSchedule* to each of the four processes *Individual Elevator Control*.

As indicated in section 7.2.3.1, elevator scheduling is required also whenever the operator disables or enables an elevator. The process *Individual Elevator Control* therefore sends the message *elevatorDisabled* or *elevatorHalted* to the process *Summons Handler*, which subsequently tries to schedule pending summons requests.

The behavior of the process objects could be specified in POOSL rather easily by considering the informal description and narrative scenarios in section 7.2. However, the major difficulty resided in guaranteeing the correct system behavior when there are multiple scenarios being executed simultaneously. The ECS specification consists of nine concurrent processes that may participate in multiple scenarios. For instance, the process *Summons Handler*, all four processes *Individual Elevator Control* and possibly one process *Elevator Mechanism Control* are involved in the scenario for dealing with a summons request. At the same time, a process *Individual Elevator Control* and a process *Elevator Mechanism Control* may be involved in a second scenario for dealing with a destination request. And the process *Summons Handler*, one process *Individual Elevator Control* and one process *Elevator Mechanism Control* may be involved in a third scenario for dealing with an elevator arriving at a new floor. This can easily lead to deadlock situations where processes are circular waiting on each other. For instance, the process *Summons Handler* that is dealing with a *summonsRequest*, can be waiting to interact with all four processes *Individual Elevator Control* for receiving elevator state information. At the same time, some process *Individual Elevator Control* that is dealing with a *destinationRequest*, can be waiting to interact with the process *Summons Handler* for receiving information on summons requests. In an early version of the ECS specification, this situation actually led to a deadlock situation.

We solved this problem by introducing a deadlock-free protocol for communication between the process *Summons Handler* and the four processes *Individual Elevator Control*. The communication protocol is based on priorities for interrupt messages and on hand-shaking by exchanging the messages *lockRequest*, *locked* and *unlock*. Figure 7.13 and figure 7.14 provide outlines of the POOSL specification of the communication protocol in the processes *Summons Handler* and *Individual Elevator Control*.

```
SummonsHandler()

init1()()                                     /* definition of method init1()() */
   init2()()                                  /* method call init2()()          */
   interrupt                                  /* interrupt for method init1()() */
      sel  [!lockRequest[1]]                  /* guard                          */
               sh?lockRequest(id | id=1);     /* receive message                */
               lockRequest[1]:=true           /* variable assignment            */
      or   [!lockRequest[2]]                  /* guard                          */
               sh?lockRequest(id | id=2);     /* receive message                */
               lockRequest[2]:=true           /* variable assignment            */
      or   [!lockRequest[3]]                  /* guard                          */
               sh?lockRequest(id | id=3);     /* receive message                */
               lockRequest[3]:=true           /* variable assignment            */
      or   [!lockRequest[4]]                  /* guard                          */
               sh?lockRequest(id | id=4);     /* receive message                */
               lockRequest[4]:=true           /* variable assignment            */
      les.                                    /* end of selection statement     */

init2()()                                     /* definition of method init2()() */
   loop()()                                   /* method call loop()()           */
   interrupt                                  /* interrupt for method init2()() */
      [!summonsRequest]                       /* guard                          */
      (sb?summonsRequest(floor,direction);    /* receive message                */
       summonsRequest:=true).                 /* variable assignment            */

loop()()                                      /* definition of method loop()()  */
   sel  [lockRequest[1]]                      /* guard                          */
        sh!locked(1);                         /* send message                   */
        ...                                   ...
        sh?unlock;                            /* receive message                */
        lockRequest[1]:=false                 /* variable assignment            */
   or   [lockRequest[2]]                      /* guard                          */
        sh!locked(2);                         /* send message                   */
        ...                                   ...
        sh?unlock;                            /* receive message                */
        lockRequest[2]:=false                 /* variable assignment            */
   or   [lockRequest[3]]                      /* guard                          */
        sh!locked(3);                         /* send message                   */
        ...                                   ...
        sh?unlock;                            /* receive message                */
        lockRequest[3]:=false                 /* variable assignment            */
   or   [lockRequest[4]]                      /* guard                          */
        sh!locked(4);                         /* send message                   */
        ...                                   ...
        sh?unlock;                            /* receive message                */
        lockRequest[4]:=false                 /* variable assignment            */
   or   [summonsRequest                       /* guard                          */
         & !lockRequest[1]
         & !lockRequest[2]
         & !lockRequest[3]
         & !lockRequest[4]]
        sh!*summonsScheduleRequest;           /* send message (broadcast)       */
        ...                                   ...
        summonsRequest:=false                 /* variable assignment            */
   les;                                       /* end of selection statement     */
   loop()().                                  /* tail recursive method call     */
```

Figure 7.13 Outline of process object 'Summons Handler'

```
IndividualElevatorControl(elevatorId)

init1()()                                      /* definition of method init1()() */
   init2()()                                   /* method call init2()()          */
   interrupt                                   /* interrupt for method init1()() */
      (sh?summonsScheduleRequest;              /* receive message                */
       summonsScheduleRequest:=true).          /* variable assignment            */

init2()()                                      /* definition of method init2()() */
   loop()()                                    /* method call loop()()           */
   interrupt                                   /* interrupt for method init2()() */
      [!lockRequestPending]                    /* guard                          */
      sel fs?floorReached(floor);              /* receive message                */
          receivedFloorReached:=true;          /* variable assignment            */
          lockRequestPending:=true;            /* variable assignment            */
          sh!lockRequest(elevatorId)           /* send message                   */
      or  op?disableElevator;                  /* receive message                */
          receivedDisableElevator:=true;       /* variable assignment            */
          lockRequestPending:=true;            /* variable assignment            */
          sh!lockRequest(elevatorId)           /* send message                   */
      or  op?enableElevatorNormalMode;         /* receive message                */
          receivedEnableNormalMode:=true;      /* variable assignment            */
          lockRequestPending:=true;            /* variable assignment            */
          sh!lockRequest(elevatorId)           /* send message                   */
      or  op?enableElevatorMaintenanceMode;    /* receive message                */
          receivedEnableMaintenanceMode:=true; /* variable assignment            */
          lockRequestPending:=true;            /* variable assignment            */
          sh!lockRequest(elevatorId)           /* send message                   */
      les.                                     /* end of selection statement     */

loop()()                                       /* definition of method loop()()  */
   sel [summonsScheduleRequest]                /* guard                          */
       ...                                     ...
       summonsScheduleRequest:=false           /* variable assignment            */
   or  [receivedFloorReached]                  /* guard                          */
       sh?locked(id | id=elevatorId);          /* receive message                */
       ...                                     ...
       sh!unlock;                              /* send message                   */
       receivedFloorReached:=false;            /* variable assignment            */
       lockRequestPending:=false               /* variable assignment            */
   or  [receivedDisableElevator]               /* guard                          */
       sh?locked(id | id=elevatorId);          /* receive message                */
       ...                                     ...
       sh!unlock;                              /* send message                   */
       receivedDisableElevator:=false;         /* variable assignment            */
       lockRequestPending:=false               /* variable assignment            */
   or  [receivedEnableNormalMode]              /* guard                          */
       sh?locked(id | id=elevatorId);          /* receive message                */
       ...                                     ...
       sh!unlock;                              /* send message                   */
       receivedEnableNormalMode:=false;        /* variable assignment            */
       lockRequestPending:=false               /* variable assignment            */
   or  [receivedEnableMaintenanceMode]         /* guard                          */
       sh?locked(id | id=elevatorId);          /* receive message                */
       ...                                     ...
       sh!unlock;                              /* send message                   */
       receivedEnableMaintenanceMode:=false;   /* variable assignment            */
       lockRequestPending:=false               /* variable assignment            */
   or  db?destinationRequest(floor);           /* receive message                */
       ...                                     ...
   les;                                        /* end of selection statement     */
   loop()().                                   /* tail recursive method call     */
```

Figure 7.14 Outline of process object 'Individual Elevator Control'

7.4.2.3 Validation and Verification

We used the POOSL-simulator to simulate the ECS specification together with the process objects in the ECS environment. We introduced hierarchy by grouping process objects and channels into clusters. Figure 7.15 shows the top-level of the ECS simulation model. The cluster *Elevator Control System* contains the process objects *Summons Handler, Individual Elevator Control* and *Elevator Mechanism Control.* Each cluster *Elevator Environment* contains the process objects *Arrival Indications, Floor Sensors, Audible Alarm, Elevator Motor* and *Doors Control System* for one particular elevator. The cluster *Indications* contains process objects that model the lights of destination buttons and summons buttons. The process object *Passengers & Operator* models passengers pressing destination buttons and summons buttons, the operator enabling and disabling the elevators, and the overweight sensors.

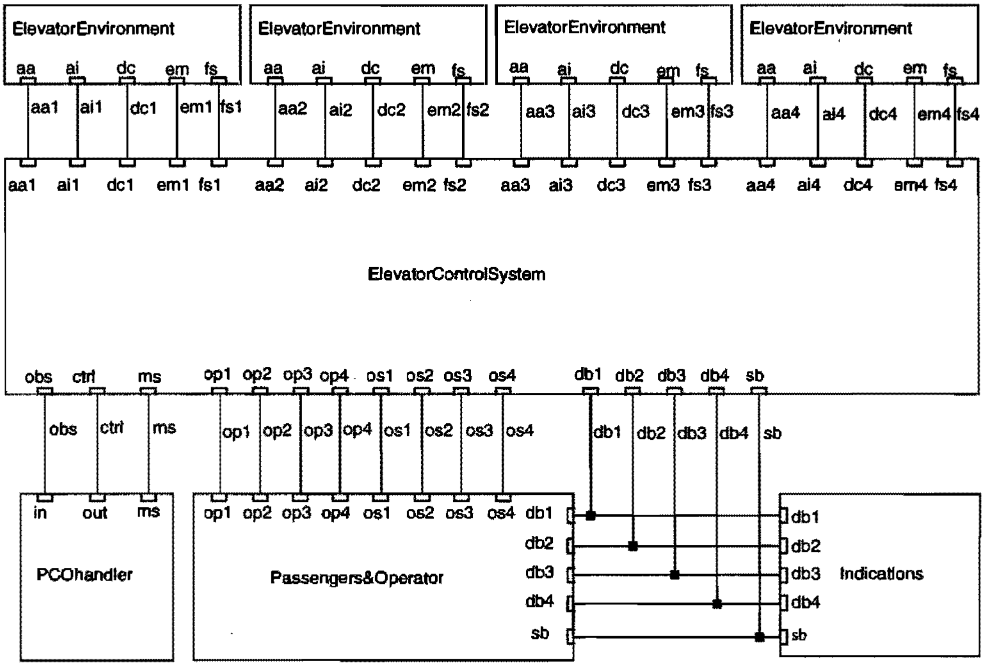


Figure 7.15 ECS simulation model in POOSL-simulator

The POOSL-simulator provides a graphical environment, visualizing the exchange of messages over channels during simulation. We used the POOSL-simulator to validate and verify the ECS specification in various simulation runs. In each simulation run, the process *Passengers & Operator* offers different stimuli to the *Elevator Control System.* We first used branch testing (see section 3.7.2) to verify the behavior of each individual process object. Branch testing implies that in each process, each branch alternative in each method is executed at least once. Next, we simulated individual scenarios, such as the scenario started by a passenger pressing a particular destination button or the scenario started by the operator enabling an elevator. Finally, we simulated the ECS with random behavior of the passengers, the operator and the overweight sensors to verify the system behavior when there are multiple, simultaneous scenarios being executed. Simulation turned out to be very useful for uncovering some subtle specification errors. Furthermore, simulation uncovered the deadlock situations that occurred due to interference of multiple scenarios. As described in the previous section, we eliminated these deadlocks by introducing a deadlock-free communication protocol. Our structured approach to simulation gives confidence that the ECS specification is correct. However, exhaustive simulation, i.e. simulation of all possible combinations of scenarios, is impossible to achieve and hence simulation cannot prove the complete absence of faults.

We used formal verification to mathematically prove that the communication protocol between the processes *Summons Handler* and *Individual Elevator Control*, as described in the previous section, is indeed deadlock-free. We therefore transformed the POOSL specification of these process objects into CCS specifications. We considered only those POOSL statements that are involved in the communication protocol, as outlined in figure 7.13 and 7.14.

A POOSL specification can be transformed rather easily into an equivalent CCS specification. However, the interrupt behavior in POOSL has to be specified explicitly in CCS. Furthermore, data objects in POOSL have to be made explicit in CCS or they should be abstracted from. An example is shown in figure 7.16. In the CCS specification, we specified explicitly the interrupt behavior. We abstracted from the data object *t*, but we explicitly specified the data object *intReceived*, because it is used in guards and affects the control flow.[1] The behavior of the CCS specification in figure 7.16b is:

$$S_0 \overset{\text{def}}{=} (A_0|B_0)\backslash\{\ \mathit{intReceivedFalse},\ \mathit{intReceivedTrue},\ \mathit{intReceivedSetFalse},\ \mathit{intReceivedSetTrue}\ \}$$

$$\begin{array}{lll} S_0 & = & \tau \,.\, S_1 + \tau \,.\, S_2 + \mathit{int?} \,.\, S_0 \\ S_1 & = & \mathit{overflow!} \,.\, S_0 + \mathit{int?} \,.\, S_1 \\ S_2 & = & \mathit{tick!} \,.\, S_0 + \mathit{int?} \,.\, S_2 \end{array}$$

[1] We remark that a precise translation of POOSL specifications into CCS specifications is not as easy as depicted in figure 7.16. This is mainly due to the combination of guarded commands and non-deterministic selection statements. For instance, consider the following POOSL specification and the corresponding CCS specification:

$$\begin{array}{llll} \mathit{sel}\ [E_0]\,\mathit{ch?m_0} & P & = & E_0\mathit{True?} \,.\, \mathit{ch?m_0} \,.\, \mathrm{NIL} \\ \mathit{or}\ \ [E_1]\,\mathit{ch?m_1} & & + & E_1\mathit{True?} \,.\, \mathit{ch?m_1} \,.\, \mathrm{NIL} \\ \mathit{les.} & & + & E_0\mathit{FalseAnd}E_1\,\mathit{False?} \,.\, P \end{array}$$

In the POOSL specification, the actual choice depends on the expressions E_0 and E_1 and also on whether the processes in the environment are willing to communicate by $\mathit{ch!m_0}$ or $\mathit{ch!m_1}$. In the CCS specification, an a priori choice is made by evaluating the expressions E_0 and E_1 only. Subsequently, a deadlock may occur if the processes in the environment are unwilling to communicate the message that is specified in the selected branch.

A more precise definition for translating POOSL into CCS should be based on the formal semantics of POOSL. However, this is beyond the scope of the research described in this thesis.

Our only purpose of translating POOSL into CCS, is to check for deadlocks between communicating processes. Our translation from POOSL into CCS results in CCS specifications that are more non-deterministic than the corresponding POOSL specifications. Consequently, if a CCS specification is deadlock-free, then the corresponding POOSL specification is deadlock-free as well. However, the reverse may not be true in the case of false-negatives. Fortunately, the examples in this section do not suffer from this problem.

a) POOSL Specification

```
(1)  init()()
(2)     t:=0;
(3)     intReceived:=false;
(4)     loop()()
(5)     interrupt
(6)        (ch?int;
(7)         intReceived:=true).

(8)  loop()()
(9)     sel [intReceived]
(10)        t:=0;
(11)        intReceived:=false
(12)    or  [!intReceived]
(13)        t:=t+1;
(14)        if (t>60)
(15)        then t:=0;
(16)             ch!overflow
(17)        else ch!tick
(18)        fi
(19)    les;
(20)    loop()().
```

b) CCS Specification

$(A_0|B_0)\backslash\{$ *intReceivedFalse, intReceivedTrue, intReceivedSetFalse, intReceivedSetTrue* $\}$

$A_0 = \mathrm{intReceivedTrue?} \;.\; A_1$	`(9) [intReceived]`
$\quad + \mathrm{intReceivedFalse?} \;.\; A_2$	`(12) [!intReceived]`
$\quad + \mathrm{int?} \;.\; A_5$	`(6) ch?int`
$A_1 = \mathrm{intReceivedSetFalse!} \;.\; A_0$	`(11) intReceived:=false`
$\quad + \mathrm{int?} \;.\; A_6$	`(6) ch?int`
$A_2 = \tau \;.\; A_3$	`(14) if (t>60) then`
$\quad + \tau \;.\; A_4$	`(17) else`
$\quad + \mathrm{int?} \;.\; A_7$	`(6) ch?int`
$A_3 = \mathrm{overflow!} \;.\; A_0$	`(16) ch!overflow`
$\quad + \mathrm{int?} \;.\; A_8$	`(6) ch?int`
$A_4 = \mathrm{tick!} \;.\; A_0$	`(17) ch!tick`
$\quad + \mathrm{int?} \;.\; A_9$	`(6) ch?int`
$A_5 = \mathrm{intReceivedSetTrue!} \;.\; A_0$	`(7) intReceived:=true`
$A_6 = \mathrm{intReceivedSetTrue!} \;.\; A_1$	`(7) intReceived:=true`
$A_7 = \mathrm{intReceivedSetTrue!} \;.\; A_2$	`(7) intReceived:=true`
$A_8 = \mathrm{intReceivedSetTrue!} \;.\; A_3$	`(7) intReceived:=true`
$A_9 = \mathrm{intReceivedSetTrue!} \;.\; A_4$	`(7) intReceived:=true`
$B_0 = \mathrm{intReceivedFalse!} \;.\; B_0$	`intReceived = false`
$\quad + \mathrm{intReceivedSetFalse?} \;.\; B_0$	
$\quad + \mathrm{intReceivedSetTrue?} \;.\; B_1$	
$B_1 = \mathrm{intReceivedTrue!} \;.\; B_1$	`intReceived = true`
$\quad + \mathrm{intReceivedSetFalse?} \;.\; B_0$	
$\quad + \mathrm{intReceivedSetTrue?} \;.\; B_1$	

Figure 7.16 *Transforming POOSL specification into CCS specification*

In a similar way, we transformed the POOSL specifications of the processes *Summons Handler* and *Individual Elevator Control*, as shown in figure 7.13 and 7.14, into CCS specifications. We used a software tool [vRV94] to calculate the parallel composition of the CCS agent *Summons Handler* and four CCS agents *Individual Elevator Control*. We proved that the resulting monolithic behavior is observational equivalent to:

$$
\begin{aligned}
S_0 = \; & sb?summonsRequest(floor,direction) \,.\, S_0 \\
+ \; & fs1?floorReached(floor) \,.\, S_0 + \ldots + fs4?floorReached(floor) \,.\, S_0 \\
+ \; & op1?disableElevator \,.\, S_0 + \ldots + op4?disableElevator \,.\, S_0 \\
+ \; & op1?enableElevatorNormalMode \,.\, S_0 + \ldots + op4?enableElevatorNormalMode \,.\, S_0 \\
+ \; & op1?enableElevatorMaintenanceMode \,.\, S_0 + \ldots + op4?enableElevatorMaintenanceMode \,.\, S_0 \\
+ \; & db1?destinationRequest(floor) \,.\, S_0 + \ldots + db4?destinationRequest(floor) \,.\, S_0
\end{aligned}
$$

The agent S_0 indicates that the system is continuously ready to receive messages from the summons buttons, the floor sensors, the operator and the destination buttons, which indeed corresponds to the wanted behavior. Furthermore, the agent S_0 clearly is deadlock-free.

7.4.2.4 Design For Test & Debug

We applied our design for test & debug approach to the POOSL specification of the ECS. First, we used scenario-based PCO insertion as described in section 5.5, to determine appropriate places for inserting PCOs. Next, we actually incorporated PCOs at the identified places in the POOSL specification. We applied the transformation functions as described in section 5.6 to preserve the correctness of the specification. Finally, we used the POOSL-simulator to simulate the system specification including PCOs. We will elaborate on these three steps in the following.

Scenario-Based Analysis for PCO Insertion

Scenario-based analysis addresses two basic questions for each scenario: what is the essential information in the system for the particular scenario, and how well can this information be accessed from the system environment? We considered these two questions for each scenario in the POOSL specification of the ECS. We first identified the essential information for each scenario. Next, we analyzed the accessibility of the relevant state information contained in the ECS process objects and the accessibility of the relevant communication interfaces between the ECS process objects. The results of this analysis are outlined below.

The process *Summons Handler* comprises instance variables that store pending summons requests and the summons requests' priority counter. The instance variables for the pending summons requests can be controlled directly in the ECS environment by pressing summons buttons. Furthermore, they can also be observed directly because the summons button light of a pending summons request is illumined. The priority counter is not directly visible in the ECS environment. However, we did not consider the priority counter as essential state information.

Each process *Individual Elevator Control* comprises instance variables that store the state information of a particular elevator, as listed in figure 7.12. The information contained in the instance variables *floor*, *direction*, *operationMode*, *doorsAreClosing* and *doorsAreOpening* is directly controllable and observable in the ECS environment. On the other hand, the information contained in the instance variables *elevatorState*, *elevatorSchedule* and *rescheduled* is not directly visible in the ECS environment. These variables contain information that is essential to explain the behavior of the ECS, and hence inserting a PCO to access these variables is desirable. A process *Individual Elevator Control* also comprises instance variables that store pending destination requests. These instance variables can be controlled directly in the ECS environment by pressing destination buttons. Furthermore, they can be observed directly because the destination button lights of pending destination requests are illumined.

Each process *Elevator Mechanism Control* comprises instance variables that hold state information for the elevator doors, the elevator motor and the overweight sensor. However, this information is directly controllable and observable in the ECS environment and hence a PCO is not required.

The communication interfaces between the ECS process objects consist of the channels on which messages are exchanged. The channels emc_1, emc_2, emc_3 and emc_4 between the processes *Individual Elevator Control* and the processes *Elevator Mechanism Control* are used to communicate commands and acknowledgement signals related to the elevator doors and the elevator motors.

Every message sent by a process *Individual Elevator Control* on a channel *emc* directly results in the process *Elevator Mechanism Control* sending a reply message and sending messages to the *Elevator Motor*, the *Doors Control System* and the *Audible Alarm*. Hence, the messages on the channel *emc* are indirectly visible in the system environment and a PCO is not required.

The channel *sh* between the process *Summons Handler* and the processes *Individual Elevator Control* is used to communicate messages related to the scheduling of elevators. This information is not directly visible in the ECS environment. The communication protocol on the channel is the most complex part of the ECS. An external observer cannot always tell why an elevator starts moving, how an elevator is scheduled, or whether an elevator may service intermediate summons requests or destination requests. Hence, a PCO in the channel *sh* is desirable.

In summary, our scenario-based analysis identified the channel *sh* and the instance variables *elevatorState*, *elevatorSchedule* and *rescheduled* of the processes *Individual Elevator Control* as appropriate places for PCO insertion.

Insertion of PCOs

We incorporated the PCOs next into the POOSL specification. The modified instance structure diagram is shown in figure 7.17. The process object *PCO Controller* in the ECS environment controls the operation mode of the PCOs, receives messages from the PCOs' observation outputs, and sends messages to the PCOs' control inputs.

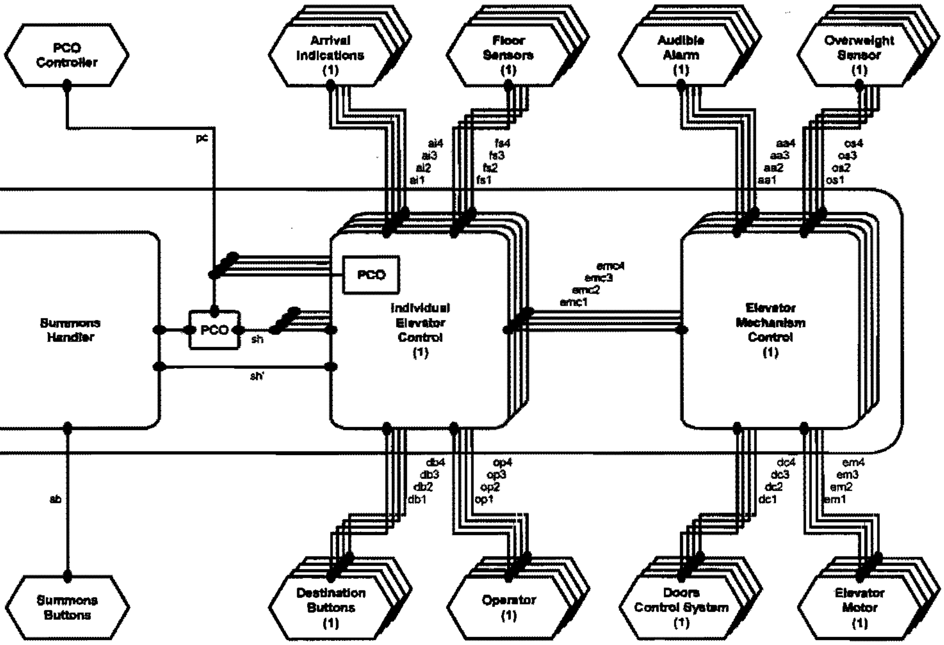


Figure 7.17 ECS instance structure diagram with PCOs

The insertion of PCOs implies a modification of the POOSL specification. Initially, we tried to prove that the original specification is equivalent to the modified specification. We therefore transformed both POOSL specifications into CCS specifications, and we tried to prove that both specifications are observational equivalent using a CCS software tool [vRV94]. However, due to state space explosion we could only prove observation equivalence for a system with one elevator, i.e. a system with a process *Summons Handler*, one process *Individual Elevator Control* and one process *Elevator Mechanism Control*. Observation equivalence could not be proven for systems with two or more elevators, because the requirements on memory space and computing time grew too large.

This problem could be overcome by applying the correctness-preserving transformation functions for PCO insertion, as discussed in section 5.6. The specifications before and after applying the transformation functions are observational equivalent by construction, and hence there is no need for proving observation equivalence afterwards.

An example of applying the transformation functions is as follows. Consider the system in figure 7.18 which consists of two POOSL process objects, A and B, that are connected by the channel ch over which the messages p and q are exchanged. Process object A performs the behavior $(\ldots ch!p; \ldots ch?q; \ldots)$ and process object B performs the behavior $(\ldots ch?p; \ldots ch!q; \ldots)$. These behaviors can be expressed by the following CCS agents:

$$\begin{array}{ll} A_0 = \bar{p}.A_1 & B_0 = p.B_1 \\ A_1 = q.A_0 & B_1 = \bar{q}.B_0 \end{array}$$

Next, we insert a PCO in channel ch to control and observe the messages p and q, as shown in figure 7.19. We obtain the modified agents A'_0 and B'_0 by applying the transformation functions: $A'_0 = \mathcal{F}_{\bar{p}_1}(\mathcal{T}_{\bar{p}}(\mathcal{G}_{q_2}(\mathcal{Z}_q(A_0))))$ and $B'_0 = \mathcal{F}_{\bar{q}_1}(\mathcal{T}_{\bar{q}}(\mathcal{G}_{p_2}(\mathcal{Z}_p(B_0))))$. The resulting agents are:

$$\begin{array}{ll} A'_0 = \bar{p}.A'_1 & B'_0 = p.B'_1 \\ A'_1 = \bar{p}_1.A'_2 & B'_1 = p_2.B'_2 \\ A'_2 = q.A'_3 & B'_2 = \bar{q}.B'_3 \\ A'_3 = q_2.A'_0 & B'_3 = \bar{q}_1.B'_0 \end{array}$$

The PCO behavior is expressed by the agents $PCO_p = p_1.\overline{p_2}.PCO_p$ and $PCO_q = q_1.\overline{q_2}.PCo_q$. The behavior of the PCO in channel ch is defined as $PCO_0 \overset{\text{def}}{=} (PCO_p|PCO_q)$:

$$\begin{array}{l} PCO_0 = p_1.PCO_1 + q_1.PCO_2 \\ PCO_1 = \bar{p}_2.PCO_0 + q_1.PCO_3 \\ PCO_2 = p_1.PCO_3 + \bar{q}_2.PCO_0 \\ PCO_3 = \bar{p}_2.PCO_2 + \bar{q}_2.PCO_1 \end{array}$$

The corresponding POOSL process objects A' and B' perform the behavior
$(\ldots ch_0!p; ch_1!p_1; \ldots ch_0?q; ch_1?q_2; \ldots)$ and $(\ldots ch_0?p; ch_2?p_2; \ldots ch_0!q; ch_2!q_1; \ldots)$.
The messages p and q on channel ch_0 in figure 7.19 can be considered as synchronization messages, while the messages p_1 and p_2 correspond to the original message p in figure 7.18 and similarly the messages q_1 and q_2 correspond to the original message q.

In a similar way we applied the transformation functions to the process objects in the ECS specification to insert a PCO in the channel *sh*.

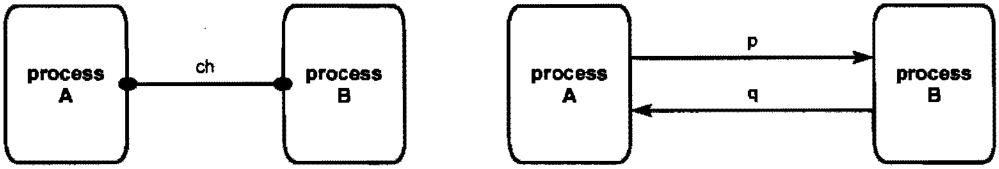


Figure 7.18 *Example instance structure and message flow diagram*

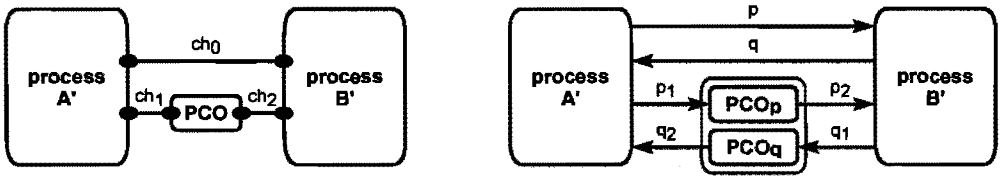


Figure 7.19 *Example instance structure and message flow diagram with PCO*

Simulating the ECS Specification with PCOs

Finally, we simulated the modified ECS specification with the PCOs in the POOSL-simulator as shown in figure 7.15. The process *PCO Handler* in the ECS environment controls the PCO operation modes, it receives the results from the PCO observation outputs and it also sends messages to the PCO control inputs. As expected, the simulated system behavior is equivalent to the initially simulated system behavior without the PCOs.

Obviously, the PCOs are of no direct use in the simulator, because the simulator environment itself provides powerful facilities to view the internal operation of process objects and the messages on communication channels. The PCOs are intended to provide similar facilities in the hardware/software implementation of the system.

We simulated the ECS specification with the PCOs in the POOSL-simulator to demonstrate the use and the benefits of PCOs. When the PCOs are switched into observation mode, the monitored messages on channel *sh* and the monitored state information from the processes *Individual Elevator Control* provide detailed information on the internal behavior of the ECS. This information, together with the messages on the external inputs and outputs in the ECS environment, provides sufficient information to explain and verify the internal system behavior. For instance, we simulated an early version of the ECS specification that contained a deadlock, as described in section 7.4.2.2. It was nearly impossible to identify the cause of the deadlock by just analyzing the messages in the ECS environment. However, by additionally analyzing the monitored data from the PCOs, we could quickly identify that the deadlock was due to an incorrect communication protocol on channel *sh*.

At the beginning of each simulation run, we switched the PCOs into test mode. This allowed us to quickly impose a certain state by writing appropriate values into the instance variables of the processes *Individual Elevator Control*. This considerably shortened the set-up time required for bringing the ECS in a particular initial state. Subsequently, we switched the PCOs into observation mode and monitored the internal behavior as described before.

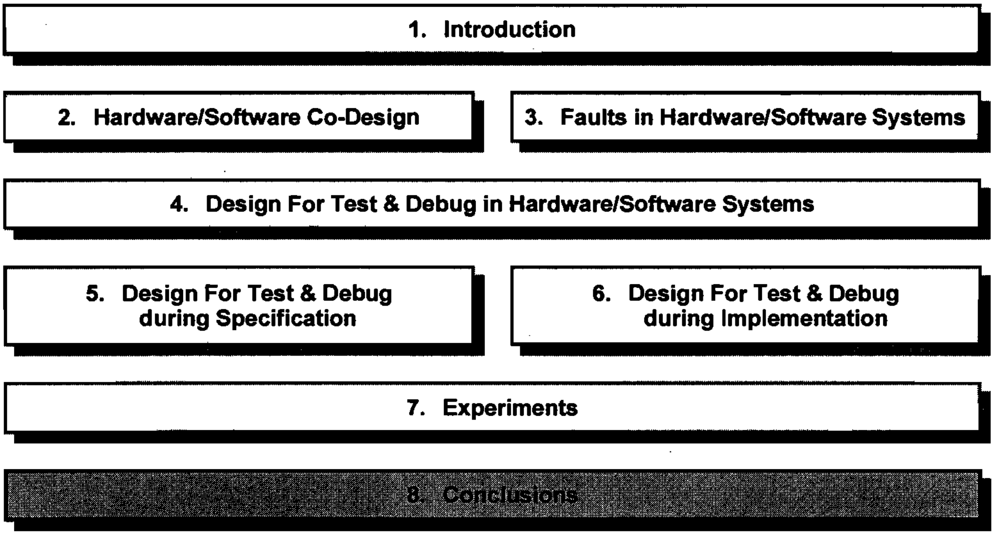
7.5 Summary

In this chapter we discussed the specification and implementation of an elevator control system (ECS) and we applied our design for test & debug approach. We first used the Ward & Mellor analysis method to create a system specification and we subsequently implemented the ECS in software. We applied a simplified version of our design for test & debug approach, which turned out to be invaluable for debugging some complicated, timing-related errors in the software implementation caused by race conditions.

Next, we used the SHE method to create a formal specification in the POOSL language. We used simulation and formal verification to validate and verify the correctness of the specification. We applied our design for test & debug approach, using scenario-based analysis and correctness-preserving transformation functions for PCO insertion. We demonstrated the use of PCOs for testing and debugging by simulating the modified system specification including PCOs.

Chapter 8

Conclusions


1. Introduction
2. Hardware/Software Co-Design
3. Faults in Hardware/Software Systems
4. Design For Test & Debug in Hardware/Software Systems
5. Design For Test & Debug during Specification
6. Design For Test & Debug during Implementation
7. Experiments
8. Conclusions

In this chapter we summarize the conclusions of this thesis and we recommend directions for future research.

8.1 Conclusions

We developed a generic method towards design for test & debug to deal with the problems of system-level testing and debugging in hardware/software systems. The main conclusions are:

- Our method is fully integrated into the system design process and considers design for test & debug in the behavioral specification, the architecture and the hardware/software implementation of a system.

- The major strength of our approach is that we consider design for test & debug already during system specification. We identify communication interfaces and process state information in the functional behavior that are difficult to control and/or to observe in the system environment. Our method allows the insertion of Points of Control and Observation (PCOs) while preserving the correctness of the system behavior. The PCOs provide detailed visibility into the internal system behavior during system-level testing and debugging.

- Our method towards design for test & debug provides a genuine multi-disciplinary, system-level approach that is capable of dealing with the vast complexity of hardware/software systems.

8.1.1 Motivation

We founded our method towards design for test & debug on analyses of co-design methods, hardware/software architectures, and a characterization of faults in hardware/software systems. The results of these analyses are:

- We defined a hardware/softwareco-design flow consisting of system requirements capture, system specification, architecture exploration, architecture refinement, synthesis, and hardware/software integration.

- We concluded that the current co-design methods strongly emphasize verification, using formal verification, (co-)simulation and emulation. However, testing is still an essential necessity to verify the correctness of a hardware/software implementation. Unfortunately, the current co-design methods provide no support to improve the testing and debugging of hardware/software systems and they do not address design for test & debug.

- We identified the limited visibility into the internal system operation as the basic problem of system-level testing and debugging. Access through the external interfaces and auxiliary instruments such as oscilloscopes, logic analyzers and in-circuit emulators, are inadequate to provide detailed controllability and observability of the internal system operation.

- We defined a generic architectural model for hardware/software systems, comprising application software, system software, hardware nucleus, application-specific hardware and various communication interfaces. We concluded that integration testing and system testing should primarily focus on verifying these communication interfaces.

- We showed that faults in communication interfaces of hardware/software systems are typically related to concurrency. We classified these faults into faulty communication and synchronization protocols, faulty mutual exclusive access to shared data or shared resources, faulty process scheduling, deadlocks, race conditions, and faulty interrupt handling.

- We concluded that exhaustive testing for interfacing faults and system-level faults is unfeasible due to the large number of possible sequences of events, the interleaved execution of processes, timing dependencies and non-determinism. Furthermore, these properties cause that faults typically appear as temporary faults at run-time and they usually cannot be reproduced during debugging. In addition, we concluded that there are no effective fault models yet for modeling these interfacing faults.

8.1.2 Design For Test & Debug

We based our method towards design for test & debug in hardware/software systems upon three basic principles:

- Design for test & debug is required to improve integration testing, system testing and debugging of hardware/software systems.
- Design for test & debug should provide visibility into communication interfaces and into state information of software processes and hardware components.
- Design for test & debug should be an elementary part of hardware/software co-design, and should be considered in all steps of the design flow: system specification, architecture exploration, architecture refinement, and synthesis.

The key element of our method is the insertion of Points of Control and Observation (PCOs) to access communication interfaces and process state information. We make a clear distinction between activities related to system specification and activities related to system implementation. We first insert PCOs in the system specification and next we incorporate these PCOs into the hardware/software system architecture during the subsequent stages of the design flow.

Specification of PCOs
During system specification, we insert a balanced number of PCOs to provide observability and controllability of the internal system behavior. We concentrate on two key questions: where should PCOs be inserted in the system specification, and what are the effects of PCO insertion on the system behavior.

- **Guidelines for PCO insertion**

  We propose scenario-based testability analysis to guide PCO insertion. Scenario-based testability analysis first identifies for each scenario the essential communication channels and process state information in a system specification. We showed that scenario-based testability analysis traverses a relevant subset of paths through a system specification in the POOSL language, including loops, tail recursion and interrupts.

  Scenario-based testability analysis next examines how well the relevant communication channels and the relevant process state information can be observed and controlled in the external system environment. A PCO is inserted if improved visibility is required.

  We examined related approaches for PCO insertion in a system specification, considering PCOs in the protocol stacks of communication systems, testability analysis in VLSI circuits, and system-level testability analysis techniques. We concluded that these approaches are all relevant in their own application domain, however none of them is truly applicable to guide PCO insertion in the system specification of hardware/software systems.

- **Effects of PCO insertion**

  Inserting PCOs implies modifying the system specification. However, the interference of PCOs on the system behavior may cause incorrect behavior. We provided a formal discussion on the effects of PCO insertion using CCS process algebra. We showed that the system behavior including PCOs is not necessarily observational equivalent to the system behavior before PCO insertion. However, we proved that a PCO can be inserted in a communication channel while preserving observation equivalence.

  We provided a set of transformation functions that should be applied to those processes that communicate over a channel in which a PCO is to be inserted. We gave a mathematical proof that the modified system behavior, obtained from applying the transformation functions and inserting PCOs, is observational equivalent to the original system behavior. Hence, we proved that we are able of inserting PCOs in a system specification in such a way that – by construction – the interference of PCOs does not induce incorrect behavior.

The system specification provides a system-level view on the functional behavior, consisting of concurrent, communicating processes. The system specification is most suitable to identify those communication interfaces and process state information that should be equipped with PCOs. The PCOs are incorporated next into the hardware/software architecture. Our approach provides that the effects of PCOs on the system architecture can be predicted in advance and appropriate measures can be taken to avoid intolerable side effects such as performance degradation. PCO insertion in the system specification is necessary but not sufficient to obtain full controllability and observability of the internal operation of a hardware/software system. During architecture exploration and architecture refinement, additional PCOs may be inserted in the system software and the application-specific hardware to access implementation-dependent information, such as the interleaved execution of processes and the operation of the system software and control logic.

Implementation of PCOs

The PCOs in the system specification may be realized by using test & measurement equipment, such as a logic analyzer or an in-circuit emulator. However, test & measurement equipment often cannot offer the required controllability and observability and hence PCOs have to be implemented in hardware and/or software. Implementing PCOs can be performed either by dedicated hardware/software or by using the current DFT and DFD techniques in hardware/software. Using dedicated hardware/software implies overhead costs, but this approach is transparent and allows to implement PCOs exactly as they are specified. Using current DFT and DFD techniques may reduce the overhead costs. This requires that already during architecture exploration requirements are stated on the testability and debuggability of off-the-shelf hardware components, the application-specific hardware components, and software components. In traditional design methods these decisions are not made until the synthesis stage.

We argued that there are generally two approaches towards testing and debugging: breakpoint-based and monitoring-based. Both approaches rely on test & debug facilities in the hardware/-software architecture of the system. The concept of breakpoint is not equivalent to the concept of PCO. However, breakpoints can be used to specify on which conditions a PCO should be activated. Monitoring-based test & debug implies the use of POs and off-line analysis of the observed data.

Hardware DFT facilities on the IC level (e.g. test points, scan cells, boundary-scan cells and test interface elements), and hardware DFD facilities (e.g. scandumps, cachedumps and execution tracing), can reasonably be regarded as low-level hardware implementations of PCOs. The test infrastructure of IEEE 1149.1 on the PCB level and IEEE 1149.5 on the system level provides test buses, test interfaces and test controllers that can be used to access these built-in facilities. In general, hardware DFT and DFD facilities provide very detailed access to the system's internals, however they are somewhat restricted when debugging dynamic, real-time system behavior. Improved control and observation into the internal system operation can be obtained by using dedicated software monitors, hardware monitors or hybrid monitors. These monitors are particularly useful for distributed real-time systems. The monitors can be considered as implementing POs and the infrastructure for accessing the POs from the system environment.

Experiments
We performed several experiments by applying our design for test & debug approach in the specification and implementation of an elevator control system. We first used the Ward & Mellor method for structured analysis and we implemented the elevator control system in software. We inserted POs in the system specification and subsequently in the software implementation. The POs turned out to be very useful for debugging some complicated errors in the software implementation caused by race conditions.

Next, we used the SHE method to create a formal specification in the POOSL language. We applied our design for test & debug approach, using scenario-based analysis and correctness-preserving transformation functions for PCO insertion. We demonstrated the use of PCOs for testing and debugging by simulating the system specification including PCOs.

8.2 Recommendations for Future Research

There are many issues in our method towards design for test & debug that need further research:

- The theory in this thesis has been applied successfully to an elevator control system. However, it is required to gain more experience by applying our method into practice on industrial hardware/software systems. This should provide more insight into the costs and the benefits of PCO insertion.

- We outlined the current DFT and DFD techniques in hardware/software and we argued that they can be used to implement PCOs and the infrastructure for accessing PCOs. However, further research and practical experience is required on the mapping of PCOs on hardware/software DFT and DFD facilities.

- This thesis concentrated on design for test & debug, while paying less attention to the problem of test-case generation. We discussed the use of scenarios as system-level test cases.

Further research is required on test-case generation for system-level testing and on coverage metrics. In particular, this requires research on system-level fault models.

An interesting option is to explore the use of automatic test-case generation tools, for instance using the FSM-based techniques for protocol conformance testing. Related to this is the question whether test-case generation can be facilitated by taking into account the PCOs for additional observation and control. At the moment, test-case generation techniques regard the system as a black box and hence control and observation is only considered at the primary inputs and outputs.

- During testing and debugging, large amounts of data have to be analyzed. The evaluation of test responses can be automated to a large extent by comparing the observed responses with the expected responses. However, debugging requires a detailed analysis of the test results, which is in general a very complex task. This can be facilitated by visualizing the test stimuli and test responses with appropriate representation techniques, e.g. Message Sequence Charts.

- We defined transformation functions for correctness-preserving PCO insertion using process algebra. An interesting research question is to define similar transformation functions for other formalisms and languages.

  We showed that our transformation functions can be applied reasonably to POOSL specifications, because the basic communication mechanism in POOSL is the one-way, synchronous message passing mechanism originating from CCS. We indicated that a more precise definition of transformation functions in POOSL should be based on the formal semantics of POOSL.

  In languages that lack formal semantics (e.g. VHDL) it may be rather difficult to formally define correctness-preserving transformation functions for PCO insertion.

- We concluded that exhaustive path analysis is unfeasible, which largely impedes testability analysis. We provided an informal discussion on how scenario-based testability analysis may circumvent this problem. However, a more precise and preferably formal theory on path analysis should provide more insight into this problem field.

Appendix A

Formal Proof for PCO Insertion

In this appendix we give a formal proof for inserting a PCO in a communication channel while preserving the externally observable system behavior. The proof is based on the process algebra CCS [Mil80, Mil89]. This appendix provides:

- Formal definitions of transformation functions that are applied to agents that communicate over a channel in which a PCO is inserted.
- A formal proof that the modified system, in which a PCO has been inserted and in which transformation functions have been applied to the relevant agents, is observational equivalent to the original system when the PCO is operating in transparent mode or in observation mode.

A.1 Definitions

We extend the process algebra CCS, as defined in [Mil89], with two new symbols: θ and ζ. The semantics are defined by $\theta.P \xrightarrow{\theta} P$ and $\zeta.P \xrightarrow{\zeta} P$ for $P \in \mathcal{P}$, where $\mathcal{P}$ is the new set of agents. We further extend the definition of observation equivalence in such a way that the symbols θ and ζ are interpreted as silent actions.

Definition 1

(1) We define the parameterized set of agents $\mathcal{R}(abc) \subseteq \mathcal{P}$ as follows:
(Note that $\mathcal{R}(abc)$ is a short notation for $\mathcal{R}(a,b,c)$.)

(i) $\mathbf{0} \in \mathcal{R}(abc)$

(ii) if $P \in \mathcal{R}(abc)$ then $\alpha.P \in \mathcal{R}(abc)$ if $\alpha \neq a, \bar{a}, b, \bar{b}$

(iii) if $P, Q \in \mathcal{R}(abc)$ then $P + Q \in \mathcal{R}(abc)$

(iv) if $P, Q \in \mathcal{R}(abc)$ then $P|Q \in \mathcal{R}(abc)$

(v) if $P \in \mathcal{R}(abc)$ then $P[f] \in \mathcal{R}(abc)$ if $\begin{cases} f(l) = a & \text{iff } l = a \\ f(l) = b & \text{iff } l = b \\ f(l) = c & \text{iff } l = c \end{cases}$

(vi) if $P \in \mathcal{R}(abc)$ then $P \backslash L \in \mathcal{R}(abc)$ if $a, b, c \notin L$

(vii) if $P \in \mathcal{R}(abc)$ then $A \in \mathcal{R}(abc)$ if $A \overset{\text{def}}{=} P$

In this definition: A is an agent constant, f is a relabelling function, and $L \subseteq \mathcal{L}$. An agent $P \in \mathcal{R}(abc)$ cannot perform the actions $a, \bar{a}, b, \bar{b}$.

(2) We define the parameterized set of agents $\mathcal{A}_\theta(a,b,c) \subseteq \mathcal{P}$, or shortly $\mathcal{A}_\theta(abc)$, as follows:

$$
\begin{array}{lll}
\text{(i)} & \mathbf{0} \in \mathcal{A}_\theta(abc) & \\
\text{(ii)} & \text{if } P \in \mathcal{A}_\theta(abc) \text{ then } & \left\{ \begin{array}{lll} \theta.P \in \mathcal{A}_\theta(abc) & \text{if} & P \overset{\theta}{\not\rightarrow} \\ \alpha.P \in \mathcal{A}_\theta(abc) & \text{if} & P \overset{\theta}{\not\rightarrow} \text{ and } \alpha \neq a, \bar{a}, b, \bar{b}, \bar{c} \\ \bar{c}.\theta.P \in \mathcal{A}_\theta(abc) & \text{if} & P \overset{\theta}{\not\rightarrow} \end{array} \right. \\
\text{(iii)} & \text{if } P, Q \in \mathcal{A}_\theta(abc) \text{ then } P + Q \in \mathcal{A}_\theta(abc) \text{ if } P \overset{\theta}{\not\rightarrow} \text{ and } Q \overset{\theta}{\not\rightarrow} & \\
\text{(iv)} & \text{if } P, Q \in \mathcal{A}_\theta(abc) \text{ then } P|Q \in \mathcal{A}_\theta(abc) & \\
\text{(v)} & \text{if } P \in \mathcal{A}_\theta(abc) \text{ then } P[f] \in \mathcal{A}_\theta(abc) \text{ if } & \left\{ \begin{array}{lll} f(l) = a & \text{iff} & l = a \\ f(l) = b & \text{iff} & l = b \\ f(l) = c & \text{iff} & l = c \end{array} \right. \\
\text{(vi)} & \text{if } P \in \mathcal{A}_\theta(abc) \text{ then } P\backslash L \in \mathcal{A}_\theta(abc) \text{ if } a, b, c \notin L & \\
\text{(vii)} & \text{if } P \in \mathcal{A}_\theta(abc) \text{ then } A \in \mathcal{A}_\theta(abc) \text{ if } A \overset{\text{def}}{=} P \text{ and } P \overset{\theta}{\not\rightarrow} &
\end{array}
$$

An agent $P \in \mathcal{A}_\theta(abc)$ cannot perform the actions $a, \bar{a}, b, \bar{b}$. Furthermore, whenever agent P performs an action $\bar{c}$, subsequently the agent will eventually perform the action θ.

(3) We define the parameterized set of agents $\mathcal{B}_\zeta(a,b,c) \subseteq \mathcal{P}$, or shortly $\mathcal{B}_\zeta(abc)$, as follows:

$$
\begin{array}{lll}
\text{(i)} & \mathbf{0} \in \mathcal{B}_\zeta(abc) & \\
\text{(ii)} & \text{if } P \in \mathcal{B}_\zeta(abc) \text{ then } & \left\{ \begin{array}{lll} \zeta.P \in \mathcal{B}_\zeta(abc) & \text{if} & P \overset{\zeta}{\not\rightarrow} \\ \alpha.P \in \mathcal{B}_\zeta(abc) & \text{if} & P \overset{\zeta}{\not\rightarrow} \text{ and } \alpha \neq a, \bar{a}, b, \bar{b}, c \\ c.\zeta.P \in \mathcal{B}_\zeta(abc) & \text{if} & P \overset{\zeta}{\not\rightarrow} \end{array} \right. \\
\text{(iii)} & \text{if } P, Q \in \mathcal{B}_\zeta(abc) \text{ then } P + Q \in \mathcal{B}_\zeta(abc) \text{ if } P \overset{\zeta}{\not\rightarrow} \text{ and } Q \overset{\zeta}{\not\rightarrow} & \\
\text{(iv)} & \text{if } P, Q \in \mathcal{B}_\zeta(abc) \text{ then } P|Q \in \mathcal{B}_\zeta(abc) & \\
\text{(v)} & \text{if } P \in \mathcal{B}_\zeta(abc) \text{ then } P[f] \in \mathcal{B}_\zeta(abc) \text{ if } & \left\{ \begin{array}{lll} f(l) = a & \text{iff} & l = a \\ f(l) = b & \text{iff} & l = b \\ f(l) = c & \text{iff} & l = c \end{array} \right. \\
\text{(vi)} & \text{if } P \in \mathcal{B}_\zeta(abc) \text{ then } P\backslash L \in \mathcal{B}_\zeta(abc) \text{ if } a, b, c \notin L & \\
\text{(vii)} & \text{if } P \in \mathcal{B}_\zeta(abc) \text{ then } A \in \mathcal{B}_\zeta(abc) \text{ if } A \overset{\text{def}}{=} P \text{ and } P \overset{\zeta}{\not\rightarrow} &
\end{array}
$$

An agent $P \in \mathcal{B}_\zeta(abc)$ cannot perform the actions $a, \bar{a}, b, \bar{b}$. Furthermore, whenever agent P performs an action c, subsequently the agent will eventually perform the action ζ.

□

Definition 2

(1) We define the parameterized function $\mathcal{T}_{\bar{c}} : \mathcal{R}(abc) \rightarrow \mathcal{A}_\theta(abc)$ as follows:

(i) $\mathcal{T}_{\bar{c}}(\mathbf{0}) \equiv \mathbf{0}$

(ii) $\mathcal{T}_{\bar{c}}(\alpha.P) \equiv \begin{cases} \alpha.\mathcal{T}_{\bar{c}}(P) & \text{if } \alpha \neq \bar{c} \\ \bar{c}.\theta.\mathcal{T}_{\bar{c}}(P) & \text{if } \alpha = \bar{c} \end{cases}$

(iii) $\mathcal{T}_{\bar{c}}(P + Q) \equiv \mathcal{T}_{\bar{c}}(P) + \mathcal{T}_{\bar{c}}(Q)$

(iv) $\mathcal{T}_{\bar{c}}(P|Q) \equiv \mathcal{T}_{\bar{c}}(P) \mid \mathcal{T}_{\bar{c}}(Q)$

(v) $\mathcal{T}_{\bar{c}}(P[f]) \equiv \mathcal{T}_{\bar{c}}(P)[f]$

(vi) $\mathcal{T}_{\bar{c}}(P \backslash L) \equiv \mathcal{T}_{\bar{c}}(P) \backslash L$

(vii) $\mathcal{T}_{\bar{c}}(A) \equiv B$ where $B \overset{\text{def}}{=} \mathcal{T}_{\bar{c}}(P)$ if $A \overset{\text{def}}{=} P$

The function application $\mathcal{T}_{\bar{c}}(P)$ returns an agent that is syntactically identical to agent P except that every occurrence of action $\bar{c}$ in P is replaced by $\bar{c}.\theta$ in $\mathcal{T}_{\bar{c}}(P)$.

(2) We define the parameterized function $\mathcal{Z}_c : \mathcal{R}(abc) \rightarrow \mathcal{B}_\zeta(abc)$ as follows:

(i) $\mathcal{Z}_c(\mathbf{0}) \equiv \mathbf{0}$

(ii) $\mathcal{Z}_c(\alpha.P) \equiv \begin{cases} \alpha.\mathcal{Z}_c(P) & \text{if } \alpha \neq c \\ c.\zeta.\mathcal{Z}_c(P) & \text{if } \alpha = c \end{cases}$

(iii) $\mathcal{Z}_c(P + Q) \equiv \mathcal{Z}_c(P) + \mathcal{Z}_c(Q)$

(iv) $\mathcal{Z}_c(P|Q) \equiv \mathcal{Z}_c(P) \mid \mathcal{Z}_c(Q)$

(v) $\mathcal{Z}_c(P[f]) \equiv \mathcal{Z}_c(P)[f]$

(vi) $\mathcal{Z}_c(P \backslash L) \equiv \mathcal{Z}_c(P) \backslash L$

(vii) $\mathcal{Z}_c(A) \equiv B$ where $B \overset{\text{def}}{=} \mathcal{Z}_c(P)$ if $A \overset{\text{def}}{=} P$

The function application $\mathcal{Z}_c(P)$ returns an agent that is syntactically identical to agent P except that every occurrence of action c in P is replaced by $c.\zeta$ in $\mathcal{Z}_c(P)$.

□

Definition 3

(1) We define the class of agents $\mathcal{A}_n(abc) \subseteq \mathcal{A}_\theta(abc)$, or shortly $\mathcal{A}_n$, for $n \in \mathbb{N}$, as follows:

(i) $\mathcal{A}_0 = \{\, P \in \mathcal{A}_\theta(abc) \mid P \overset{\theta}{\nrightarrow} \,\}$

(ii) $\mathcal{A}_{n+1} = \{\, P \in \mathcal{A}_\theta(abc) \mid P \overset{\theta}{\rightarrow} P'$ for some $P' \in \mathcal{A}_n \,\}$

Agents in the set $\mathcal{A}_n$ can perform a sequence of n successive θ actions.

(2) We define the class of agents $\mathcal{B}_n(abc) \subseteq \mathcal{B}_\zeta(abc)$, or shortly $\mathcal{B}_n$, for $n \in \mathbb{N}$, as follows:

(i) $\mathcal{B}_0 = \{\, P \in \mathcal{B}_\zeta(abc) \mid P \overset{\zeta}{\nrightarrow} \,\}$

(ii) $\mathcal{B}_{n+1} = \{\, P \in \mathcal{B}_\zeta(abc) \mid P \overset{\zeta}{\rightarrow} P'$ for some $P' \in \mathcal{B}_n \,\}$

Agents in the set $\mathcal{B}_n$ can perform a sequence of n successive ζ actions.

□

Definition 4

(1) We define the parameterized function $\mathcal{F}_{\bar{a}} : \mathcal{A}_{\theta}(abc) \to \mathcal{P}$ as follows:

$$
\begin{array}{ll}
\text{(i)} & \mathcal{F}_{\bar{a}}(\mathbf{0}) \equiv \mathbf{0} \\
\text{(ii)} & \mathcal{F}_{\bar{a}}(\alpha.P) \equiv \begin{cases} \alpha.\mathcal{F}_{\bar{a}}(P) & \text{if } \alpha \neq \theta \\ \bar{a}.\mathcal{F}_{\bar{a}}(P) & \text{if } \alpha = \theta \end{cases} \\
\text{(iii)} & \mathcal{F}_{\bar{a}}(P + Q) \equiv \mathcal{F}_{\bar{a}}(P) + \mathcal{F}_{\bar{a}}(Q) \\
\text{(iv)} & \mathcal{F}_{\bar{a}}(P|Q) \equiv \mathcal{F}_{\bar{a}}(P) \mid \mathcal{F}_{\bar{a}}(Q) \\
\text{(v)} & \mathcal{F}_{\bar{a}}(P[f]) \equiv \mathcal{F}_{\bar{a}}(P)[f] \\
\text{(vi)} & \mathcal{F}_{\bar{a}}(P\backslash L) \equiv \mathcal{F}_{\bar{a}}(P)\backslash L \\
\text{(vii)} & \mathcal{F}_{\bar{a}}(A) \equiv B \ \text{ where } \ B \overset{\text{def}}{=} \mathcal{F}_{\bar{a}}(P) \ \text{ if } \ A \overset{\text{def}}{=} P
\end{array}
$$

The function application $\mathcal{F}_{\bar{a}}(P)$ returns an agent that is syntactically identical to agent P except that every occurrence of θ in P is replaced by $\bar{a}$ in $\mathcal{F}_{\bar{a}}(P)$.

(2) We define the parameterized function $\mathcal{G}_b : \mathcal{B}_{\zeta}(abc) \to \mathcal{P}$ as follows:

$$
\begin{array}{ll}
\text{(i)} & \mathcal{G}_b(\mathbf{0}) \equiv \mathbf{0} \\
\text{(ii)} & \mathcal{G}_b(\alpha.P) \equiv \begin{cases} \alpha.\mathcal{G}_b(P) & \text{if } \alpha \neq \zeta \\ b.\mathcal{G}_b(P) & \text{if } \alpha = \zeta \end{cases} \\
\text{(iii)} & \mathcal{G}_b(P + Q) \equiv \mathcal{G}_b(P) + \mathcal{G}_b(Q) \\
\text{(iv)} & \mathcal{G}_b(P|Q) \equiv \mathcal{G}_b(P) \mid \mathcal{G}_b(Q) \\
\text{(v)} & \mathcal{G}_b(P[f]) \equiv \mathcal{G}_b(P)[f] \\
\text{(vi)} & \mathcal{G}_b(P\backslash L) \equiv \mathcal{G}_b(P)\backslash L \\
\text{(vii)} & \mathcal{G}_b(A) \equiv B \ \text{ where } \ B \overset{\text{def}}{=} \mathcal{G}_b(P) \ \text{ if } \ A \overset{\text{def}}{=} P
\end{array}
$$

The function application $\mathcal{G}_b(P)$ returns an agent that is syntactically identical to agent P except that every occurrence of ζ in P is replaced by b in $\mathcal{G}_b(P)$.

□

Definition 5

We define a new equivalence relation $\backsim$ based on the following notion of (θ, ζ)-bisimulation. $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ is a (θ, ζ)-bisimulation if the following holds for each $(P, Q) \in \mathcal{S}$:

$$
\begin{array}{ll}
\text{(i)} & \text{if } P \xrightarrow{\theta,\zeta} P' \text{ and } Q \overset{\theta,\zeta}{\nrightarrow} \text{ then } (P', Q) \in \mathcal{S} \\
\text{(ii)} & \text{if } P \overset{\theta,\zeta}{\nrightarrow} \text{ and } Q \xrightarrow{\theta,\zeta} Q' \text{ then } (P, Q') \in \mathcal{S} \\
\text{(iii)} & \text{if } P \xrightarrow{\theta,\zeta} P' \text{ and } Q \xrightarrow{\theta,\zeta} Q' \text{ then } (P', Q') \in \mathcal{S} \\
\text{(iv)} & \text{if } P \overset{\theta,\zeta}{\nrightarrow} \text{ and } Q \overset{\theta,\zeta}{\nrightarrow} \text{ then, for all } \alpha \in Act \begin{cases} \text{if } P \xrightarrow{\alpha} P' \text{ then } Q \xrightarrow{\alpha} Q' \text{ and } (P', Q') \in \mathcal{S} \\ \text{if } Q \xrightarrow{\alpha} Q' \text{ then } P \xrightarrow{\alpha} P' \text{ and } (P', Q') \in \mathcal{S} \end{cases}
\end{array}
$$

Two agents P and Q are (θ, ζ)-equivalent, written as $P \backsim Q$, if $(P, Q) \in \mathcal{S}$ for some (θ, ζ)-bisimulation $\mathcal{S}$.

Here *Act* is the set of actions defined by Milner in [Mil89], and hence $\theta, \zeta \notin Act$. The notation $P \xrightarrow{\theta,\zeta} P'$ indicates that either $P \xrightarrow{\theta} P'$ or $P \xrightarrow{\zeta} P'$, while the notation $P \not\xrightarrow{\theta,\zeta}$ implies that P can neither perform action θ nor action ζ.

It is not difficult to prove that $\leadsto$ satisfies the following proposition. Let $P_1 \leadsto P_2$, then

(1) $\alpha.P_1 \leadsto \alpha.P_2$
(2) $P_1|Q \leadsto P_2|Q$
(3) $P_1\backslash L \leadsto P_2\backslash L$
(4) $P_1[f] \leadsto P_2[f]$

It is not necessarily true that $P_1 + Q \leadsto P_2 + Q$.
In fact, (θ, ζ)-equivalence $\leadsto$ is stronger than observation equivalence $\approx$, but it is weaker than strong equivalence $\sim$. We introduce (θ, ζ)-equivalence to prove the validity of our proposition 1.

□

A.2 Lemmas

Lemma 1

(1) (i) If $P \in \mathcal{A}_n(abc)$ and $P \xrightarrow{\alpha} P'$ for $\alpha \neq \bar{c}, \theta$, then $P' \in \mathcal{A}_n(abc)$
(ii) If $P \in \mathcal{A}_n(abc)$ and $P \xrightarrow{\bar{c}} P'$, then $P' \in \mathcal{A}_{n+1}(abc)$
(iii) If $P \in \mathcal{A}_{n+1}(abc)$ and $P \xrightarrow{\theta} P'$, then $P' \in \mathcal{A}_n(abc)$ and $P \leadsto P'$

(2) (i) If $P \in \mathcal{B}_n(abc)$ and $P \xrightarrow{\alpha} P'$ for $\alpha \neq c, \zeta$, then $P' \in \mathcal{B}_n(abc)$
(ii) If $P \in \mathcal{B}_n(abc)$ and $P \xrightarrow{c} P'$, then $P' \in \mathcal{B}_{n+1}(abc)$
(iii) If $P \in \mathcal{B}_{n+1}(abc)$ and $P \xrightarrow{\zeta} P'$, then $P' \in \mathcal{B}_n(abc)$ and $P \leadsto P'$

Proof of lemma 1

(1) Let $P \in \mathcal{A}_n(abc)$ and $P \xrightarrow{\alpha} P'$. We will refer to $\mathcal{A}_n(abc)$ by the short notation $\mathcal{A}_n$. The proof is by transition induction on the inference of $P \xrightarrow{\alpha} P'$.

(a) Let $P \xrightarrow{\alpha} P'$ because $P \equiv \alpha.P'$.
If $\alpha \neq \bar{c}, \theta$ then by definition $P \in \mathcal{A}_0$, $P' \not\xrightarrow{\theta}$ and $P' \in \mathcal{A}_0$.
If $\alpha = \bar{c}$ then by definition $P \equiv \bar{c}.\theta.P''$, $P' \equiv \theta.P''$, and $P'' \not\xrightarrow{\theta}$.
Hence, $P \in \mathcal{A}_0$ and $P' \in \mathcal{A}_1$.
If $\alpha = \theta$ then by definition $P \in \mathcal{A}_1$ and $P' \in \mathcal{A}_0$. Clearly, $P \leadsto P'$.

(b) Let $P \xrightarrow{\alpha} P'$ because $P \equiv Q + R$ and $Q \xrightarrow{\alpha} P'$.
Since $Q \not\xrightarrow{\theta}$ and $R \not\xrightarrow{\theta}$, it follows that $\alpha \neq \theta$ and $P \in \mathcal{A}_0$.
If $\alpha \neq \bar{c}$ then $Q \in \mathcal{A}_0$ and by induction $P' \in \mathcal{A}_0$.
If $\alpha = \bar{c}$ then $Q \in \mathcal{A}_0$ and by induction $P' \in \mathcal{A}_1$.
Let $P \xrightarrow{\alpha} P'$ because $P \equiv Q + R$ and $R \xrightarrow{\alpha} P'$.

Analogous, symmetric case.

(c) Let $P \xrightarrow{\alpha} P'$ and $P \equiv Q|R$.
Since $P \in \mathcal{A}_n$, it follows that $P \xrightarrow{\theta} P_{n-1} \xrightarrow{\theta} P_{n-2} \xrightarrow{\theta} \cdots \xrightarrow{\theta} P_0 \overset{\theta}{\not\rightarrow}$.
Then, for some $Q_{k-1}, \ldots, Q_0$ and $R_{\ell-1}, \ldots, R_0$, with $n = k + \ell$,
$P \equiv Q|R \xrightarrow{\theta} Q_{k-1}|R \xrightarrow{\theta} \cdots \xrightarrow{\theta} Q_0|R \xrightarrow{\theta} Q_0|R_{\ell-1} \xrightarrow{\theta} \cdots \xrightarrow{\theta} P_0 \equiv Q_0|R_0 \overset{\theta}{\not\rightarrow}$,
or any other interleaved sequence of $Q_i|R_j \xrightarrow{\theta} Q_{i-1}|R_j$ or $Q_i|R_j \xrightarrow{\theta} Q_i|R_{j-1}$.
Hence, $Q \xrightarrow{\theta} Q_{k-1} \xrightarrow{\theta} Q_{k-2} \xrightarrow{\theta} \cdots \xrightarrow{\theta} Q_0 \overset{\theta}{\not\rightarrow}$ and $Q \in \mathcal{A}_k$,
and $R \xrightarrow{\theta} R_{\ell-1} \xrightarrow{\theta} R_{\ell-2} \xrightarrow{\theta} \cdots \xrightarrow{\theta} R_0 \overset{\theta}{\not\rightarrow}$ and $R \in \mathcal{A}_\ell$.
Let $P \xrightarrow{\alpha} P'$ because $P \equiv Q|R$, $Q \xrightarrow{\alpha} Q'$ and $P' \equiv Q'|R$.
If $\alpha \neq \bar{c}, \theta$ then by induction $Q' \in \mathcal{A}_k$, and it is easy to see that $P' \equiv Q'|R \in \mathcal{A}_n$.
If $\alpha = \bar{c}$ then by induction $Q' \in \mathcal{A}_{k+1}$, and it is easy to see that $P' \equiv Q'|R \in \mathcal{A}_{n+1}$.
If $\alpha = \theta$ then by induction $Q' \in \mathcal{A}_{k-1}$ and $Q \rightsquigarrow Q'$. It is easy to see that
$P' \equiv Q'|R \in \mathcal{A}_{n-1}$ and $P \rightsquigarrow P'$.
Let $P \xrightarrow{\alpha} P'$ because $P \equiv Q|R$, $R \xrightarrow{\alpha} R'$ and $P' \equiv Q|R'$.
Analogous, symmetric case.

(d) Let $P \xrightarrow{\alpha} P'$ because $P \equiv Q \backslash L$, $Q \xrightarrow{\alpha} Q'$ and $P' \equiv Q' \backslash L$.
If $\alpha \neq \bar{c}, \theta$ then by induction $Q' \in \mathcal{A}_n$. It is easy to see that $P' \equiv Q' \backslash L \in \mathcal{A}_n$.
If $\alpha = \bar{c}$ then by induction $Q' \in \mathcal{A}_{n+1}$. It is easy to see that $P' \equiv Q' \backslash L \in \mathcal{A}_{n+1}$.
If $\alpha = \theta$ then by induction $Q' \in \mathcal{A}_{n-1}$ and $Q \rightsquigarrow Q'$. It is easy to see that
$P' \equiv Q' \backslash L \in \mathcal{A}_{n-1}$ and $P \rightsquigarrow P'$.

(e) Let $P \xrightarrow{\alpha} P'$ because $P \equiv Q[f]$, $Q \xrightarrow{\beta} Q'$, $f(\beta) = \alpha$ and $P' \equiv Q'[f]$.
If $\alpha \neq \bar{c}, \theta$ then $\beta \neq \bar{c}, \theta$ and by induction $Q' \in \mathcal{A}_n$.
It is easy to see that $P' \equiv Q'[f] \in \mathcal{A}_n$.
If $\alpha = \bar{c}$ then $\beta = \bar{c}$ and by induction $Q' \in \mathcal{A}_{n+1}$.
It is easy to see that $P' \equiv Q'[f] \in \mathcal{A}_{n+1}$.
If $\alpha = \theta$ then $\beta = \theta$ and by induction $Q' \in \mathcal{A}_{n-1}$ and $Q \rightsquigarrow Q'$.
It is easy to see that $P' \equiv Q'[f] \in \mathcal{A}_{n-1}$ and $P \rightsquigarrow P'$.

(f) Let $P \xrightarrow{\alpha} P'$ because $P \equiv A$, $A \overset{\text{def}}{=} B$, $B \xrightarrow{\alpha} B'$ and $P' \equiv B'$.
By definition 1.2, $B \overset{\theta}{\not\rightarrow}$.
If $\alpha \neq \bar{c}$ then by induction $B' \in \mathcal{A}_n$. It is easy to see that $P' \equiv B' \in \mathcal{A}_n$.
If $\alpha = \bar{c}$ then by induction $B' \in \mathcal{A}_{n+1}$. It is easy to see that $P' \equiv B' \in \mathcal{A}_{n+1}$.

(2) In a similar way, the proof of lemma 1.2 is by transition induction on the inference of $P \xrightarrow{\alpha} P'$, with θ replaced by ζ, $\bar{c}$ replaced by c, and $\mathcal{A}_n$ replaced by $\mathcal{B}_n$.

□

Lemma 2

(1) If $P \in \mathcal{A}_\theta(abc)$ then

(i) $P \xrightarrow{\alpha\neq\theta} P'$ iff $\mathcal{F}_{\bar{a}}(P) \xrightarrow{\alpha\neq\bar{a}} \mathcal{F}_{\bar{a}}(P')$

(ii) $P \xrightarrow{\theta} P'$ iff $\mathcal{F}_{\bar{a}}(P) \xrightarrow{\bar{a}} \mathcal{F}_{\bar{a}}(P')$

(2) If $P \in \mathcal{B}_\zeta(abc)$ then

(i) $P \xrightarrow{\alpha\neq\zeta} P'$ iff $\mathcal{G}_b(P) \xrightarrow{\alpha\neq b} \mathcal{G}_b(P')$

(ii) $P \xrightarrow{\zeta} P'$ iff $\mathcal{G}_b(P) \xrightarrow{b} \mathcal{G}_b(P')$

Proof of lemma 2

The proof is an easy transition induction.

□

Lemma 3

$P \approx \mathcal{T}_{\bar{c}}(P)$ and $P \approx \mathcal{Z}_c(P)$ for $P \in \mathcal{R}(abc)$.

Proof of lemma 3

It can easily be shown that $\{\langle P, \mathcal{T}_{\bar{c}}(P)\rangle \mid P \in \mathcal{R}(abc)\}$ and $\{\langle P, \mathcal{Z}_c(P)\rangle \mid P \in \mathcal{R}(abc)\}$ are (weak) bisimulations.

□

A.3 Propositions

See figure A.1.
Let $P, Q \in \mathcal{R}(abc)$, $\mathcal{T}_{\bar{c}}(P) \in \mathcal{A}_\theta(abc)$, $Z_c(Q) \in \mathcal{B}_\zeta(abc)$, and $\mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(P)), \mathcal{G}_b(Z_c(Q)) \in \mathcal{P}$.
We will prove that $(P|Q)\backslash c$ is observational equivalent to $(\mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(P))|B_0|\mathcal{G}_b(Z_c(Q)))\backslash\{a, b, c\}$.

The agent B_0 represents a PCO operating in transparent mode or in observation mode. Its behavior is defined as $B_0 \overset{\text{def}}{=} a.B_1$ and $B_1 \overset{\text{def}}{=} \bar{b}.B_0$. It can easily be seen that the agent B_0 represents a PCO in transparent mode. The agent B_0 may also represent the composed behavior of a PCO operating in observation mode and an external observer. The behavior of the PCO in observation mode is defined as $C_0 \overset{\text{def}}{=} a.C_1$, $C_1 \overset{\text{def}}{=} \overline{obs}.C_2$ and $C_2 \overset{\text{def}}{=} \bar{b}.C_0$. The behavior of the external observer E_0 is defined as $E_0 \overset{\text{def}}{=} obs.E_0$. It can easily be shown that $B_0 \approx (C_0|E_0)\backslash obs$. Hence, the agent B_0 may be considered as representing a PCO in transparent mode or in observation mode.

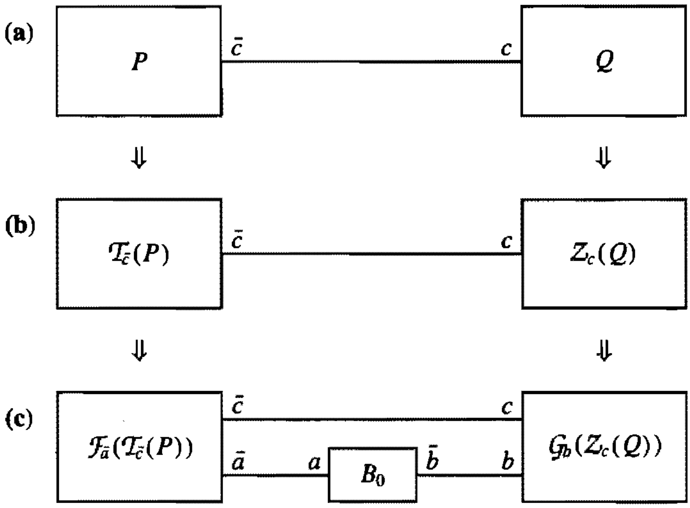


Figure A.1 Transforming $(P|Q)\backslash c$ while preserving observational equivalence

Proposition 1

Let $P, Q \in \mathcal{R}(abc)$.
Then $(\mathcal{T}_{\bar{c}}(P)|Z_c(Q))\backslash c$ is observational equivalent to $(\mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(P))|B_0|\mathcal{G}_b(Z_c(Q)))\backslash\{a, b, c\}$.

Proof of proposition 1

For readability, we define $R \equiv \mathcal{T}_{\bar{c}}(P)$ and $S \equiv Z_c(Q)$.
Hence, we have to prove that $(R|S)\backslash c$ is observational equivalent to $(\mathcal{F}_{\bar{a}}(R)|B_0|\mathcal{G}_b(S))\backslash\{a, b, c\}$.

We will show that $\mathcal{S}$ is a bisimulation up to $\approx$ (see [Mil89] and accompanying errata[1]), where

$$\begin{array}{rcl} \mathcal{S} & = & \{ \langle (R|S)\backslash c, (\mathcal{F}_{\bar{a}}(R)|B_0|\mathcal{G}_b(S))\backslash\{a,b,c\}\rangle \mid R \in \mathcal{A}_n(abc), S \in \mathcal{B}_n(abc) \} \\ & \cup & \{ \langle (R|S)\backslash c, (\mathcal{F}_{\bar{a}}(R)|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\}\rangle \mid R \in \mathcal{A}_n(abc), S \in \mathcal{B}_{n+1}(abc) \} \end{array}$$

For our case, it suffices to show that for each $(X, Y) \in \mathcal{S}$, the following holds:

(i) Whenever $X \xrightarrow{\alpha} X'$ then, for some Y', $Y \overset{\hat{\alpha}}{\Rightarrow} Y'$ and $X' \rightsquigarrow \mathcal{S} \approx Y'$.

(ii) Whenever $Y \xrightarrow{\alpha} Y'$ then, for some X', $X \overset{\hat{\alpha}}{\Rightarrow} X'$ and $X' \approx \mathcal{S} \rightsquigarrow Y'$.

The proof proceeds by the applied cases on the inference of $X \xrightarrow{\alpha} X'$ and $Y \xrightarrow{\alpha} Y'$.

(a) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_n$.
Let $X \equiv (R|S)\backslash c \xrightarrow{\alpha} X'$ because $R \xrightarrow{\alpha} R', \alpha \neq \bar{c}, \theta$ and $X' \equiv (R'|S)\backslash c$.
Then it follows from lemma 1.1 that $R' \in \mathcal{A}_n$, and from lemma 2.1 that $\mathcal{F}_{\bar{a}}(R) \xrightarrow{\alpha \neq \bar{a}} \mathcal{F}_{\bar{a}}(R')$.
Hence, $Y \equiv (\mathcal{F}_{\bar{a}}(R)|B_0|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\alpha} Y'$ where $Y' \equiv (\mathcal{F}_{\bar{a}}(R')|B_0|\mathcal{G}_b(S))\backslash\{a,b,c\}$ and $R' \in \mathcal{A}_n, S \in \mathcal{B}_n$. And thus, $\langle X', Y'\rangle \in \mathcal{S}$.

(b) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_n$.
Let $X \equiv (R|S)\backslash c \xrightarrow{\alpha} X'$ because $S \xrightarrow{\alpha} S', \alpha \neq c, \zeta$ and $X' \equiv (R|S')\backslash c$.
Then it follows from lemma 1.2 that $S' \in \mathcal{B}_n$, and from lemma 2.2 that $\mathcal{G}_b(S) \xrightarrow{\alpha \neq b} \mathcal{G}_b(S')$.
Hence, $Y \equiv (\mathcal{F}_{\bar{a}}(R)|B_0|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\alpha} Y'$ where $Y' \equiv (\mathcal{F}_{\bar{a}}(R)|B_0|\mathcal{G}_b(S'))\backslash\{a,b,c\}$ and $R \in \mathcal{A}_n, S' \in \mathcal{B}_n$. And thus, $\langle X', Y'\rangle \in \mathcal{S}$.

(c) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_n$.
Let $X \equiv (R|S)\backslash c \xrightarrow{\theta} X'$ because $R \xrightarrow{\theta} R'$ and $X' \equiv (R'|S)\backslash c$.
Then it follows from lemma 1.1 that $R' \in \mathcal{A}_{n-1}$ and $R \rightsquigarrow R'$.
It follows from lemma 2.1 that $\mathcal{F}_{\bar{a}}(R) \xrightarrow{\bar{a}} \mathcal{F}_{\bar{a}}(R')$.
Hence, $Y \equiv (\mathcal{F}_{\bar{a}}(R)|B_0|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\tau} Y'$ where $Y' \equiv (\mathcal{F}_{\bar{a}}(R')|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\}$ and $R' \in \mathcal{A}_{n-1}, S \in \mathcal{B}_n$. And thus, $\langle X', Y'\rangle \in \mathcal{S}$.

(d) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_n$.
Let $X \equiv (R|S)\backslash c \xrightarrow{\zeta} X'$ because $S \xrightarrow{\zeta} S'$ and $X' \equiv (R|S')\backslash c$.
Then it follows from lemma 1.2 that $S' \in \mathcal{B}_{n-1}$ and $S \rightsquigarrow S'$.
It follows from lemma 2.2 that $\mathcal{G}_b(S) \xrightarrow{b} \mathcal{G}_b(S')$.
Because $S \xrightarrow{\zeta} S'$ implies that $n \geq 1$, it follows that $R \xrightarrow{\theta} R'$.
Hence, $X' \equiv (R|S')\backslash c \xrightarrow{\theta} X''$ because $R \xrightarrow{\theta} R'$ and $X'' \equiv (R'|S')\backslash c$.
Then it follows from lemma 1.1 that $R' \in \mathcal{A}_{n-1}$ and $R \rightsquigarrow R'$.
It follows from lemma 2.1 that $\mathcal{F}_{\bar{a}}(R) \xrightarrow{\bar{a}} \mathcal{F}_{\bar{a}}(R')$.

[1] $\mathcal{S}$ is a (weak) bisimulation up to $\approx$ if $P\mathcal{S}Q$ implies, for all α,

(i) Whenever $P \overset{\alpha}{\Rightarrow} P'$ then, for some Q', $Q \overset{\hat{\alpha}}{\Rightarrow} Q'$ and $P' \approx \mathcal{S} \approx Q'$,
or alternatively, whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \overset{\hat{\alpha}}{\Rightarrow} Q'$ and $P' \sim \mathcal{S} \approx Q'$.

(ii) Whenever $Q \overset{\alpha}{\Rightarrow} Q'$ then, for some P', $P \overset{\hat{\alpha}}{\Rightarrow} P'$ and $P' \approx \mathcal{S} \approx Q'$,
or alternatively, whenever $Q \xrightarrow{\alpha} Q'$ then, for some P', $P \overset{\hat{\alpha}}{\Rightarrow} P'$ and $P' \approx \mathcal{S} \sim Q'$.

Hence, $Y \equiv (\mathcal{F}_{\bar a}(R)|B_0|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\tau} (\mathcal{F}_{\bar a}(R')|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\tau} Y'$
where $Y' \equiv (\mathcal{F}_{\bar a}(R')|B_0|\mathcal{G}_b(S'))\backslash\{a,b,c\}$ and $R' \in \mathcal{A}_{n-1}, S' \in \mathcal{B}_{n-1}$.
Because $\langle X'', Y'\rangle \in \mathcal{S}$ and $X' \leadsto X''$, it follows that $\langle X', Y'\rangle \in \leadsto \mathcal{S} \approx$.

(e) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_n$.

Let $X \equiv (R|S)\backslash c \xrightarrow{\tau} X'$ because $R \xrightarrow{\ell} R', S \xrightarrow{\bar\ell} S'$ and $X' \equiv (R'|S')\backslash c$.
If $\ell = \bar c$ then it follows from lemma 1 that $R' \in \mathcal{A}_{n+1}$ and $S' \in \mathcal{B}_{n+1}$.

From lemma 2 it follows that $\mathcal{F}_{\bar a}(R) \xrightarrow{\bar c} \mathcal{F}_{\bar a}(R')$ and $\mathcal{G}_b(S) \xrightarrow{c} \mathcal{G}_b(S')$.
Hence, $Y \equiv (\mathcal{F}_{\bar a}(R)|B_0|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\tau} Y'$ where
$Y' \equiv (\mathcal{F}_{\bar a}(R')|B_0|\mathcal{G}_b(S'))\backslash\{a,b,c\}$ and $R' \in \mathcal{A}_{n+1}, S' \in \mathcal{B}_{n+1}$.
And thus, $\langle X', Y'\rangle \in \mathcal{S}$.

If $\ell \neq \bar c$ then it follows from lemma 1 that $R' \in \mathcal{A}_n$ and $S' \in \mathcal{B}_n$.

From lemma 2 it follows that $\mathcal{F}_{\bar a}(R) \xrightarrow{\ell} \mathcal{F}_{\bar a}(R')$ and $\mathcal{G}_b(S) \xrightarrow{\bar\ell} \mathcal{G}_b(S')$.
Hence, $Y \equiv (\mathcal{F}_{\bar a}(R)|B_0|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\tau} Y'$ where
$Y' \equiv (\mathcal{F}_{\bar a}(R')|B_0|\mathcal{G}_b(S'))\backslash\{a,b,c\}$ and $R' \in \mathcal{A}_n, S' \in \mathcal{B}_n$.
And thus, $\langle X', Y'\rangle \in \mathcal{S}$.

(f) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_n$.

Let $Y \equiv (\mathcal{F}_{\bar a}(R)|B_0|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\alpha} Y'$ because $\mathcal{F}_{\bar a}(R) \xrightarrow{\alpha} \mathcal{F}_{\bar a}(R'), \alpha \neq \bar c, \bar a$
and $Y' \equiv (\mathcal{F}_{\bar a}(R')|B_0|\mathcal{G}_b(S))\backslash\{a,b,c\}$.
Then it follows from lemma 2.1 that $R \xrightarrow{\alpha} R'$, and from lemma 1.1 that $R' \in \mathcal{A}_n$.
Hence, $X \equiv (R|S)\backslash c \xrightarrow{\alpha} X'$ where $X' \equiv (R'|S)\backslash c$ and $R' \in \mathcal{A}_n, S \in \mathcal{B}_n$.
And thus, $\langle X', Y'\rangle \in \mathcal{S}$.

(g) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_n$.

Let $Y \equiv (\mathcal{F}_{\bar a}(R)|B_0|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\alpha} Y'$ because $\mathcal{G}_b(S) \xrightarrow{\alpha} \mathcal{G}_b(S'), \alpha \neq c, b$
and $Y' \equiv (\mathcal{F}_{\bar a}(R)|B_0|\mathcal{G}_b(S'))\backslash\{a,b,c\}$.
Then it follows from lemma 2.2 that $S \xrightarrow{\alpha} S'$, and from lemma 1.2 that $S' \in \mathcal{B}_n$.
Hence, $X \equiv (R|S)\backslash c \xrightarrow{\alpha} X'$ where $X' \equiv (R|S')\backslash c$ and $R \in \mathcal{A}_n, S' \in \mathcal{B}_n$.
And thus, $\langle X', Y'\rangle \in \mathcal{S}$.

(h) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_n$.

Let $Y \equiv (\mathcal{F}_{\bar a}(R)|B_0|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\tau} Y'$ because $\mathcal{F}_{\bar a}(R) \xrightarrow{\bar a} \mathcal{F}_{\bar a}(R')$
and $Y' \equiv (\mathcal{F}_{\bar a}(R')|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\}$.
Then it follows from lemma 2.1 that $R \xrightarrow{\theta} R'$.
It follows from lemma 1.1 that $R' \in \mathcal{A}_{n-1}$ and $R \leadsto R'$.
Hence, $X \equiv (R|S)\backslash c \xrightarrow{\theta} X'$ where $X' \equiv (R'|S)\backslash c$ and $R' \in \mathcal{A}_{n-1}, S \in \mathcal{B}_n$.
And thus, $\langle X', Y'\rangle \in \mathcal{S}$.

(i) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_n$.

Let $Y \equiv (\mathcal{F}_{\bar a}(R)|B_0|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\tau} Y'$ because $\mathcal{F}_{\bar a}(R) \xrightarrow{\ell} \mathcal{F}_{\bar a}(R'), G(S) \xrightarrow{\bar\ell} G(S')$
and $Y' \equiv (\mathcal{F}_{\bar a}(R')|B_0|\mathcal{G}_b(S'))\backslash\{a,b,c\}$.
If $\ell = \bar c$ then it follows from lemma 2 that $R \xrightarrow{\bar c} R'$ and $S \xrightarrow{c} S'$.

It follows from lemma 1 that $R' \in \mathcal{A}_{n+1}$ and $S' \in \mathcal{B}_{n+1}$.
Hence, $X \equiv (R|S)\backslash c \xrightarrow{\tau} X'$ where $X' \equiv (R'|S')\backslash c$ and $R' \in \mathcal{A}_{n+1}, S' \in \mathcal{B}_{n+1}$.
And thus, $\langle X', Y'\rangle \in \mathcal{S}$.

If $\ell \neq \bar{c}$ then it follows from lemma 2 that $R \xrightarrow{\ell} R'$ and $S \xrightarrow{\bar{\ell}} S'$.

From lemma 1 it follows that $R' \in \mathcal{A}_n$ and $S' \in \mathcal{B}_n$.

Hence, $X \equiv (R|S)\backslash c \xrightarrow{\tau} X'$ where $X' \equiv (R'|S')\backslash c$ and $R' \in \mathcal{A}_n$, $S' \in \mathcal{B}_n$.

And thus, $\langle X', Y' \rangle \in \mathcal{S}$.

(j) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_{n+1}$.

Let $X \equiv (R|S)\backslash c \xrightarrow{\alpha} X'$ because $R \xrightarrow{\alpha} R', \alpha \neq \bar{c}, \theta$ and $X' \equiv (R'|S)\backslash c$.

Then it follows from lemma 1.1 that $R' \in \mathcal{A}_n$, and from lemma 2.1 that $\mathcal{F}_{\bar{a}}(R) \xrightarrow{\alpha \neq \bar{a}} \mathcal{F}_{\bar{a}}(R')$.

Hence, $Y \equiv (\mathcal{F}_{\bar{a}}(R)|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\alpha} Y'$ where $Y' \equiv (\mathcal{F}_{\bar{a}}(R')|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\}$ and $R' \in \mathcal{A}_n, S \in \mathcal{B}_{n+1}$. And thus, $\langle X', Y' \rangle \in \mathcal{S}$.

(k) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_{n+1}$.

Let $X \equiv (R|S)\backslash c \xrightarrow{\alpha} X'$ because $S \xrightarrow{\alpha} S', \alpha \neq c, \zeta$ and $X' \equiv (R|S')\backslash c$.

Then it follows from lemma 1.2 that $S' \in \mathcal{B}_{n+1}$, and from lemma 2.2 that $\mathcal{G}_b(S) \xrightarrow{\alpha \neq b} \mathcal{G}_b(S')$.

Hence, $Y \equiv (\mathcal{F}_{\bar{a}}(R)|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\alpha} Y'$ where $Y' \equiv (\mathcal{F}_{\bar{a}}(R)|B_1|\mathcal{G}_b(S'))\backslash\{a,b,c\}$ and $R \in \mathcal{A}_n, S' \in \mathcal{B}_{n+1}$. And thus, $\langle X', Y' \rangle \in \mathcal{S}$.

(l) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_{n+1}$.

Let $X \equiv (R|S)\backslash c \xrightarrow{\zeta} X'$ because $S \xrightarrow{\zeta} S'$ and $X' \equiv (R|S')\backslash c$.

Then it follows from lemma 1.2 that $S' \in \mathcal{B}_n$ and $S \rightsquigarrow S'$.

It follows from lemma 2.2 that $\mathcal{G}_b(S) \xrightarrow{b} \mathcal{G}_b(S')$.

Hence, $Y \equiv (\mathcal{F}_{\bar{a}}(R)|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\tau} Y'$ where $Y' \equiv (\mathcal{F}_{\bar{a}}(R)|B_0|\mathcal{G}_b(S'))\backslash\{a,b,c\}$ and $R' \in \mathcal{A}_n, S \in \mathcal{B}_n$. And thus, $\langle X', Y' \rangle \in \mathcal{S}$.

(m) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_{n+1}$.

Let $X \equiv (R|S)\backslash c \xrightarrow{\theta} X'$ because $R \xrightarrow{\theta} R'$ and $X' \equiv (R'|S)\backslash c$.

Then it follows from lemma 1.1 that $R' \in \mathcal{A}_{n-1}$ and $R \rightsquigarrow R'$.

It follows from lemma 2.1 that $\mathcal{F}_{\bar{a}}(R) \xrightarrow{\bar{a}} \mathcal{F}_{\bar{a}}(R')$.

Because $R \xrightarrow{\theta} R'$ implies that $n \geq 1$, it follows that $S \xrightarrow{\zeta} S'$.

Hence, $X' \equiv (R'|S)\backslash c \xrightarrow{\zeta} X''$ because $S \xrightarrow{\zeta} S'$ and $X'' \equiv (R'|S')\backslash c$.

Then it follows from lemma 1.2 that $S' \in \mathcal{B}_n$ and $S \rightsquigarrow S'$.

It follows from lemma 2.2 that $\mathcal{G}_b(S) \xrightarrow{b} \mathcal{G}_b(S')$.

Hence, $Y \equiv (\mathcal{F}_{\bar{a}}(R)|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\tau} (\mathcal{F}_{\bar{a}}(R)|B_0|\mathcal{G}_b(S'))\backslash\{a,b,c\} \xrightarrow{\tau} Y'$ where $Y' \equiv (\mathcal{F}_{\bar{a}}(R')|B_1|\mathcal{G}_b(S'))\backslash\{a,b,c\}$ and $R' \in \mathcal{A}_{n-1}, S' \in \mathcal{B}_n$.

Because $\langle X'', Y' \rangle \in \mathcal{S}$ and $X' \rightsquigarrow X''$, it follows that $\langle X', Y' \rangle \in \rightsquigarrow \mathcal{S} \approx$.

(n) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_{n+1}$.

Let $X \equiv (R|S)\backslash c \xrightarrow{\tau} X'$ because $R \xrightarrow{\ell} R', S \xrightarrow{\bar{\ell}} S'$ and $X' \equiv (R'|S')\backslash c$.

If $\ell = \bar{c}$ then it follows from lemma 1 that $R' \in \mathcal{A}_{n+1}$ and $S' \in \mathcal{B}_{n+2}$.

From lemma 2 it follows that $\mathcal{F}_{\bar{a}}(R) \xrightarrow{\bar{c}} \mathcal{F}_{\bar{a}}(R')$ and $\mathcal{G}_b(S) \xrightarrow{c} \mathcal{G}_b(S')$.

Hence, $Y \equiv (\mathcal{F}_{\bar{a}}(R)|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\tau} Y'$ where
$Y' \equiv (\mathcal{F}_{\bar{a}}(R')|B_1|\mathcal{G}_b(S'))\backslash\{a,b,c\}$ and $R' \in \mathcal{A}_{n+1}, S' \in \mathcal{B}_{n+2}$.

And thus, $\langle X', Y' \rangle \in \mathcal{S}$.

If $\ell \neq \bar{c}$ then it follows from lemma 1 that $R' \in \mathcal{A}_n$ and $S' \in \mathcal{B}_{n+1}$.

From lemma 2 it follows that $\mathcal{F}_{\bar{a}}(R) \xrightarrow{\ell} \mathcal{F}_{\bar{a}}(R')$ and $\mathcal{G}_b(S) \xrightarrow{\bar{\ell}} \mathcal{G}_b(S')$.

Hence, $Y \equiv (\mathcal{F}_{\bar{a}}(R)|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\tau} Y'$ where
$Y' \equiv (\mathcal{F}_{\bar{a}}(R')|B_1|\mathcal{G}_b(S'))\backslash\{a,b,c\}$ and $R' \in \mathcal{A}_n, S' \in \mathcal{B}_{n+1}$.
And thus, $\langle X', Y' \rangle \in \mathcal{S}$.

(o) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_{n+1}$.
Let $Y \equiv (\mathcal{F}_{\bar{a}}(R)|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\alpha} Y'$ because $\mathcal{F}_{\bar{a}}(R) \xrightarrow{\alpha} \mathcal{F}_{\bar{a}}(R'), \alpha \neq \bar{c}, \bar{a}$ and $Y' \equiv (\mathcal{F}_{\bar{a}}(R')|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\}$.
Then it follows from lemma 2.1 that $R \xrightarrow{\alpha} R'$, and from lemma 1.1 that $R' \in \mathcal{A}_n$.
Hence, $X \equiv (R|S)\backslash c \xrightarrow{\alpha} X'$ where $X' \equiv (R'|S)\backslash c$ and $R' \in \mathcal{A}_n, S \in \mathcal{B}_{n+1}$.
And thus, $\langle X', Y' \rangle \in \mathcal{S}$.

(p) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_{n+1}$.
Let $Y \equiv (\mathcal{F}_{\bar{a}}(R)|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\alpha} Y'$ because $\mathcal{G}_b(S) \xrightarrow{\alpha} \mathcal{G}_b(S'), \alpha \neq c, b$ and $Y' \equiv (\mathcal{F}_{\bar{a}}(R)|B_1|\mathcal{G}_b(S'))\backslash\{a,b,c\}$.
Then it follows from lemma 2.2 that $S \xrightarrow{\alpha} S'$, and from lemma 1.2 that $S' \in \mathcal{B}_{n+1}$.
Hence, $X \equiv (R|S)\backslash c \xrightarrow{\alpha} X'$ where $X' \equiv (R|S')\backslash c$ and $R \in \mathcal{A}_n, S' \in \mathcal{B}_{n+1}$.
And thus, $\langle X', Y' \rangle \in \mathcal{S}$.

(q) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_{n+1}$.
Let $Y \equiv (\mathcal{F}_{\bar{a}}(R)|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\tau} Y'$ because $\mathcal{G}_b(S) \xrightarrow{b} \mathcal{G}_b(S')$ and $Y' \equiv (\mathcal{F}_{\bar{a}}(R)|B_0|\mathcal{G}_b(S'))\backslash\{a,b,c\}$.
Then it follows from lemma 2.2 that $S \xrightarrow{\zeta} S'$.
It follows from lemma 1.2 that $S' \in \mathcal{B}_n$ and $S \rightsquigarrow S'$.
Hence, $X \equiv (R|S)\backslash c \xrightarrow{\zeta} X'$ where $X' \equiv (R|S')\backslash c$ and $R \in \mathcal{A}_n, S' \in \mathcal{B}_n$.
And thus, $\langle X', Y' \rangle \in \mathcal{S}$.

(r) Let $R \in \mathcal{A}_n$ and $S \in \mathcal{B}_{n+1}$.
Let $Y \equiv (\mathcal{F}_{\bar{a}}(R)|B_1|\mathcal{G}_b(S))\backslash\{a,b,c\} \xrightarrow{\tau} Y'$ because $\mathcal{F}_{\bar{a}}(R) \xrightarrow{\ell} \mathcal{F}_{\bar{a}}(R'), G(S) \xrightarrow{\bar{\ell}} G(S')$ and $Y' \equiv (\mathcal{F}_{\bar{a}}(R')|B_1|\mathcal{G}_b(S'))\backslash\{a,b,c\}$.
If $\ell = \bar{c}$ then it follows from lemma 2 that $R \xrightarrow{\bar{c}} R'$ and $S \xrightarrow{c} S'$.
It follows from lemma 1 that $R' \in \mathcal{A}_{n+1}$ and $S' \in \mathcal{B}_{n+2}$.
Hence, $X \equiv (R|S)\backslash c \xrightarrow{\tau} X'$ where $X' \equiv (R'|S')\backslash c$ and $R' \in \mathcal{A}_{n+1}, S' \in \mathcal{B}_{n+2}$.
And thus, $\langle X', Y' \rangle \in \mathcal{S}$.
If $\ell \neq \bar{c}$ then it follows from lemma 2 that $R \xrightarrow{\ell} R'$ and $S \xrightarrow{\bar{\ell}} S'$.
From lemma 1 it follows that $R' \in \mathcal{A}_n$ and $S' \in \mathcal{B}_{n+1}$.
Hence, $X \equiv (R|S)\backslash c \xrightarrow{\tau} X'$ where $X' \equiv (R'|S')\backslash c$ and $R' \in \mathcal{A}_n, S' \in \mathcal{B}_{n+1}$.
And thus, $\langle X', Y' \rangle \in \mathcal{S}$.

□

Proposition 2

Let $P, Q \in \mathcal{R}(abc)$.
$(P|Q)\backslash c$ is observational equivalent to $(\mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(P))|B_0|\mathcal{G}_b(\mathcal{Z}_c(Q)))\backslash\{a, b, c\}$.

Proof of proposition 2

Because $P, Q \in \mathcal{R}(abc)$ and $\mathcal{T}_{\bar{c}}(P) \in \mathcal{A}_\theta(abc), \mathcal{Z}_c(Q) \in \mathcal{B}_\xi(abc)$, it follows from lemma 3 that $P \approx \mathcal{T}_{\bar{c}}(P)$ and $Q \approx \mathcal{Z}_c(Q)$. Hence, $(P|Q)\backslash c \approx (\mathcal{T}_{\bar{c}}(P)|\mathcal{Z}_c(Q))\backslash c$.
From proposition 1 it follows that $(\mathcal{T}_{\bar{c}}(P)|\mathcal{Z}_c(Q))\backslash c \approx (\mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(P))|B_0|\mathcal{G}_b(\mathcal{Z}_c(Q)))\backslash\{a, b, c\}$.
From the transivity of the equivalence relation $\approx$, it now follows that
$(P|Q)\backslash c \approx (\mathcal{F}_{\bar{a}}(\mathcal{T}_{\bar{c}}(P))|B_0|\mathcal{G}_b(\mathcal{Z}_c(Q)))\backslash\{a, b, c\}$.

□

This concludes the proof that a PCO can be inserted in a communication channel while preserving the externally observable system behavior.

References

[AB85] M.S. Abadir and M.A. Breuer, *A Knowledge-Based System for Designing Testable VLSI Chips*, IEEE Design & Test of Computers **2** (1985), no. 4, 56–68.

[ABF90] M. Abramovici, M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design*, revised printing ed., IEEE Press, 1990.

[Abr87] S. Abramsky, *Observation Equivalence as a Testing Equivalence*, Theoretical Computer Science **53** (1987), 225–241.

[Ada95] *International Standard ISO/IEC 8652:1995 Ada95 Reference Manual: Language and Standard Libraries*, 1995, URL:http://www.adahome.com/rm95.

[AHLTR96] G. Al-Hayek, Y. Le-Traon, and C. Robach, *Considering Test Economics in the Process of Hardware/Software Partitioning*, Proceedings EUROMICRO Conference, IEEE, 1996, pp. 28–34.

[AHTR97] G. Al-Hayek, Y. Le Traon, and C. Robach, *Impact of System Partitioning on Test Cost*, IEEE Design & Test of Computers **14** (1997), no. 1, 64–74.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[AKS93a] V.D. Agrawal, C.R. Kime, and K.K. Saluja, *A Tutorial on Built-In Self-Test, Part 1: Principles*, IEEE Design & Test of Computers **10** (1993), no. 1, 73–82.

[AKS93b] V.D. Agrawal, C.R. Kime, and K.K. Saluja, *A Tutorial on Built-In Self-Test, Part 2: Applications*, IEEE Design & Test of Computers **10** (1993), no. 2, 69–77.

[AL86] A. Avižienis and J.-C. Laprie, *Dependable Computing: From Concepts to Design Diversity*, Proceedings of the IEEE **74** (1986), no. 5, 629–638.

[AM82] V.D. Agrawal and M.R. Mercer, *Testability Measures – What Do They Tell Us?*, Digest of Papers International Test Conference, IEEE, 1982, pp. 391–396.

[AM89] P.N. Anirudhan and P.R. Menon, *Symbolic Test Generation for Hierarchically Modeled Digital Systems*, Proceedings International Test Conference, IEEE, 1989, pp. 461–469.

[AM94] P. Ashar and S. Malik, *Implicit Computation of Minimum-Cost Feedback-Vertex Sets for Partial Scan and Other Applications*, Proceedings Design Automation Conference, ACM/IEEE, 1994, pp. 77–80.

[Anc86] F. Anceau, *The Architecture of Microprocessors*, Addison-Wesley, 1986.

[ANS94] ANSI/IEEE, *ANSI/IEEE Std 1044-1994, Standard Classification for Software Anomalies*, 1994.

[ARM95a] ARM (Advanced RISC Machines), *ARM7TDMI Data Sheet*, 1995, URL:http://www.arm.com/Documentation/UserMans.

[ARM95b] ARM (Advanced RISC Machines), *The ARM7TDMI Debug Architecture*, 1995, Application Note 28, URL:http://www.arm.com/Documentation/AppNotes.

[ARM96a] ARM (Advanced RISC Machines), *Using EmbeddedICE*, 1996, Application Note 31, URL:http://www.arm.com/Documentation/AppNotes.

[ARM96b] ARM (Advanced RISC Machines), *Using the ARM7TDMI's Debug Communication Channel*, 1996, Application Note 38, URL:http://www.arm.com/Documentation/AppNotes.

[AT96] J.K. Adams and D.E. Thomas, *The Design of Mixed Hardware/Software Systems*, Proceedings Design Automation Conference, ACM/IEEE, 1996, pp. 515–520.

[B$^+$86] F.P.M. Beenker et al., *Macro Testing: Unifying IC and Board Test*, IEEE Design & Test of Computers **3** (1986), no. 4, 26–32.

[B$^+$92a] G.v. Bochmann et al., *Fault Models in Testing*, Protocol Test Systems, IV (J. Kroon, R.J. Heijink, and E. Brinksma, eds.), Elsevier, 1992, IFIP, pp. 17–30.

[B$^+$92b] F. Bouwman et al., *Macro Testability; The Results of Production Device Applications*, Proceedings International Test Conference, IEEE, 1992, pp. 232–241.

[B$^+$93] H. Bouwmeester et al., *Minimizing Test Time by Exploiting Parallelism in Macro Test*, Proceedings International Test Conference, IEEE, 1993, pp. 451–460.

[BA82] P. Banerjee and J.A. Abraham, *Fault Characterisation of VLSI MOS Circuits*, Proceedings International Conference on Circuits and Computers, IEEE, 1982, pp. 564–568.

[Bat89] P. Bates, *Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior*, ACM SIGPLAN Notices **24** (1989), no. 1, 11–22, Proceedings ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging.

[BB91] A. Beneviste and G. Berry, *The Synchronous Approach to Reactive and Real-Time Systems*, Proceedings of the IEEE **79** (1991), no. 9, 1270–1282.

[BBT95] F.P.M. Beenker, R.G. Bennetts, and A.P. Thijssen, *Testability Concepts for Digital ICs: The Macro Test Approach*, Kluwer, 1995.

[BCMD90] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill, *Sequential Circuit Verification Using Symbolic Model Checking*, Proceedings Design Automation Conference, ACM/IEEE, 1990, pp. 46–51.

[BCO96] G. Borriello, P. Chou, and R. Ortega, *Embedded System Co-Design: Towards Portability and Rapid Integration*, In Micheli and Sami [MS96], pp. 243–264.

[BDSvdS90] F. Beenker, R. Dekker, R. Stans, and M. van der Star, *Implementing Macro Test in Silicon Compiler Design*, IEEE Design & Test of Computers **7** (1990), no. 1, 41–51.

[BE94] D.K. Bhavsar and J.H. Edmondson, *Testability Strategy of the Alpha AXP 21164 Microprocessor*, Proceedings International Test Conference, IEEE, 1994, pp. 50–59.

[BE97] D.K. Bhavsar and J.H. Edmondson, *Alpha 21164 Testability Strategy*, IEEE Design & Test of Computers **14** (1997), no. 1, 25–33.

[Bei90] B. Beizer, *Software Testing Techniques*, second ed., Van Nostrand Reinhold, 1990.

[Ben94a] S. Bennett, *Real-Time Computer Control: An Introduction*, Prentice Hall, 1994.

[Ben94b] R.G. Bennetts, *Progress in Design for Test: A Personal View*, IEEE Design & Test of Computers **11** (1994), no. 1, 53–59.

[Ben96] L. Benders, *System Specification and Performance Analysis*, Ph.D. thesis, Eindhoven University of Technology, 1996.

[BFS96a] A. Balboni, W. Fornaciari, and D. Sciuto, *Co-synthesis and Co-simulation of Control-Dominated Embedded Systems*, Design Automation for Embedded Systems **1** (1996), no. 3, 257–289.

[BFS96b] A. Balboni, W. Fornaciari, and D. Sciuto, *TOSCA: A Pragmatic Approach to Co-Design Automation of Control-Dominated Systems*, In Micheli and Sami [MS96], pp. 265–294.

[BHLM94] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, *Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*, International Journal of Computer Simulation **4** (1994), 155–182.

[BKM95] R.S. Boyer, M. Kaufmann, and J.S. Moore, *The Boyer-Moore Theorem Prover and its Interactive Enhancement*, Computers & Mathematics with Applications (1995), 27–62.

[Bol96] I. Bolsens, *A Systematic approach for HW/SW Co-design of systems on silicon*, Proceedings ProRISC/IEEE Workshop on Circuits, Systems and Signal Processing, 1996, pp. 1–7.

[Bou90] B. Bourbon, *System-level design*, Computer Design **29** (1990), no. 3, 19–21.

[Bri88] E. Brinksma, *A theory for the derivation of tests*, Protocol Specification, Testing, and Verification VIII, IFIP, Elsevier, 1988, pp. 63–74.

[BS91a] M. Bottazzi and C. Salati, *Processes, Threads, Parallelism in Real-Time Systems*, Proceedings CompEuro, IEEE, 1991, pp. 103–107.

[BS91b] F. Boussinot and R. De Simone, *The ESTEREL Language*, Proceedings of the IEEE **79** (1991), no. 9, 1293–1304.

[BSS87] E. Brinksma, G. Scollo, and C. Steenbergen, *LOTOS Specifications, Their Implementations and Their Tests*, Protocol Specification, Testing, and Verification VI, IFIP, Elsevier, 1987, pp. 349–360.

[BSV95] K. Buchenrieder, A. Sedlmeier, and C. Veith, *CODES: A Framework for Modeling Heterogeneous Systems*, In Rozenblit and Buchenrieder [RB95], pp. 378–393.

[BT93] D. Brière and P. Traverse, *AIRBUS A320/A330/A340 Electrical Flight Controls - A Family of Fault-Tolerant Systems*, Digest of Papers International Symposium on Fault-Tolerant Computing, IEEE, 1993, pp. 616–623.

[BvdEdJ93] H. Bleeker, P. van den Eijnden, and F. de Jong, *Boundary-Scan Test: A Practical Approach*, Kluwer, 1993.

[C$^+$94a] P. Camurati et al., *System-Level Modeling and Verification: a Comprehensive Design Methodology*, Proceedings European Design and Test Conference, IEEE, 1994, pp. 636–640.

[C$^+$94b] M. Chiodo et al., *Hardware-Software Codesign of Embedded Systems*, IEEE Micro **14** (1994), no. 4, 26–36.

[CA90] K.-T. Cheng and V.D. Agrawal, *A Partial Scan Method for Sequential Circuits with Feedback*, IEEE Transactions on Computers **39** (1990), no. 4, 544–548.

[Cal93] J.P. Calvez, *Embedded Real-Time Systems*, Wiley, 1993.

[CB85] T.-H. Chen and M.A. Breuer, *Automatic Design for Testability Via Testability Measures*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **4** (1985), no. 1, 3–11.

[CBA94] S.T. Chakradhar, A. Balakrishnan, and V.D. Agrawal, *An Exact Algorithm for Selecting Partial Scan Flip-Flops*, Proceedings Design Automation Conference, ACM/IEEE, 1994, pp. 81–86.

[CCMP94] P. Camurati, F. Corno, M. Meo, and P. Prinetto, *A new Functional Fault Model for System-Level Descriptions*, Proceedings VLSI Test Symposium, IEEE, 1994, pp. 214–219.

[CCP93a] P. Camurati, F. Corno, and P. Prinetto, *An efficient toolfor system-level verification of behaviors and temporal properties*, Proceedings European Design Automation Conference, IEEE, 1993, pp. 124–129.

[CCP93b] P. Camurati, F. Corno, and P. Prinetto, *System-Level Fault Modeling and Test Pattern Generation with Process Algebras*, Proceedings European Test Conference, IEEE, 1993, pp. 47–56.

[Chi93] V. Chickermane, *Design and Synthesis for Testability Using Architectural Descriptions*, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1993.

[Cho78] T.S. Chow, *Testing Software Design Modeled by Finite-State Machines*, IEEE Transactions on Software Engineering **SE-4** (1978), no. 3, 178–187.

[CJD91] S.E. Chodrow, F. Jahanian, and M. Donner, *Run-Time Monitoring of Real-Time Systems*, Proceedings Real-Time Systems Symposium, IEEE, 1991, pp. 74–83.

[CKL96] W-T. Chang, A. Kalavade, and E.A. Lee, *Effective Heterogeneous Design and Co-simulation*, In Micheli and Sami [MS96], pp. 187–212.

[CKS94] C.-H. Chen, T. Karnik, and D.G. Saab, *Structural and Behavioral Synthesis for Testability Techniques*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **13** (1994), no. 6, 777–785.

[CLP92a] V. Chickermane, J. Lee, and J.H. Patel, *A Comparative Study of Design for Testability Methods Using High-Level and Gate-Level Descriptions*, International Conference on Computer-Aided Design, IEEE/ACM, 1992, pp. 620–624.

[CLP92b] V. Chickermane, J. Lee, and J.H. Patel, *Design for Testability Using Architectural Descriptions*, Proceedings International Test Conference, IEEE, 1992, pp. 752–761.

[CM89] C.-H. Chen and P.R. Menon, *An Approach to Functional Level Testability Analysis*, Proceedings International Test Conference, IEEE, 1989, pp. 373–380.

[COB92] P. Chou, R. Ortega, and G. Borriello, *Synthesis of the Hardware/Software Interface in Microcontroller-based Systems*, Proceedings International Conference on Computer-Aided Design, IEEE, 1992, pp. 488–495.

[COB95a] P. Chou, R. Ortega, and G. Borriello, *The Chinook Hardware/Software Co-Synthesis System*, Proceedings International Symposium on System Synthesis, IEEE, 1995, pp. 22–27.

[COB95b] P. Chou, R.B. Ortega, and G. Borriello, *Interface Co-Synthesis Techniques for Embedded Systems*, Proceedings International Conference on Computer-Aided Design, IEEE/ACM, 1995, pp. 280–287.

[CP92] J.P. Calvez and O. Pasquier, *A Transputer Interconnection Bus For Hard Real-Time Systems*, Transputer'92, 1992, pp. 273–283.

[CP95] J.P. Calvez and O. Pasquier, *Performance Assessment of Embedded Hw/Sw Systems*, Proceedings International Conference on Computer Design, IEEE, 1995, pp. 52–57.

[CP98] J.P. Calvez and O. Pasquier, *Performance Monitoring and Assessment of Embedded Hw/Sw Systems*, Design Automation for Embedded Systems (to appear), 1998.

[CPC94] A.L. Crouch, M. Pressly, and J. Circello, *Testability Features of the MC68060 Microprocessor*, Proceedings International Test Conference, IEEE, 1994, pp. 60–69.

[CR95] G. Castelli and G. Ragazzini, *EOS: A Real-Time Operating System Adapts to Application Architectures*, IEEE Micro **15** (1995), no. 5, 41–49.

[Cro89] A. Cron, *IEEE* 1149.1 *Use in Design for Verification and Testability at Texas Instruments*, Proceedings Annual ASIC Seminar and Exhibit, IEEE, 1989, pp. P4–1/1–5.

[CS92] C.-H. Chen and D.G. Saab, *Behavioral Synthesis for Testability*, International Conference on Computer-Aided Design, IEEE/ACM, 1992, pp. 612–615.

[CS93] C.-H. Chen and D.G. Saab, *A Novel Behavioral Testability Measure*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **12** (1993), no. 12, 1960–1970.

[CW96] R. Camposano and J. Wilberg, *Embedded System Design*, Design Automation for Embedded Systems **1** (1996), no. 1–2, 5–50.

[CWS91] C.-H. Chen, C. Wu, and D.G. Saab, *BETA: Behavioral Testability Analysis*, International Conference on Computer-Aided Design, IEEE/ACM, 1991, pp. 202–205.

[Dac93] M.C. Daconta, *C Pointers and Dynamic Memory Management*, QED, 1993.

[Dan92] W.T. Daniel, *Design Verification of a High Density Computer Using IEEE* 1149.1, Proceedings International Test Conference, IEEE, 1992, pp. 84–90.

[DASC93] K. Drira, P. Azema, B. Soulas, and A.M. Chemali, *Testability of a communicating system through an environment*, TAPSOFT'93: Theory and Practice of Software Development, Springer-Verlag, 1993, Lecture Notes in Computer Science 668, pp. 529–543.

[Dij59] E.W. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik **1** (1959), 269–271.

[DR92] P.S. Dodd and C.V. Ravishankar, *Monitoring and Debugging Distributed Real-time Programs*, Software—Practice and Experience **22** (1992), no. 10, 863–877.

[Dus78] J.A. Dussault, *A Testability Measure*, Semiconductor Test Conference, Digest of Papers, IEEE, 1978, pp. 113–116.

[Eck93] W. Ecker, *Using VHDL for HW/SW Co-Specification*, Proceedings European Design Automation Conference with EURO-VHDL, IEEE, 1993, pp. 500–505.

[Edw93] K. Edwards, *Real-Time Structured Methods - Systems Analysis*, Wiley, 1993.

[EH92] W. Ecker and M. Hofmeister, *The Design Cube - A New Model for VHDL Designflow Representation*, Proceedings of the EURO-VHDL, 1992, pp. 752–757.

[ESA96] ESA (European Space Agency), *Ariane 5 Flight 501 Failure: Report by the Inquiry Board*, 1996,
URL:http://www.esrin.esa.it/tidc/press/press96/ariane5rep.html.

[F+91] S. Fujiwara et al., *Test Selection Based on Finite State Models*, IEEE Transactions on Software Engineering **17** (1991), no. 6, 591–603.

[Fav94] C. Favre, *Fly-by-wire for commercial aircraft: the Airbus experience*, International Journal of Control **59** (1994), no. 1, 139–157.

[Flo62] R.W. Floyd, *Algorithm 97: shortest path*, Communications of ACM **5** (1962), no. 6, 345.

[FM90] P. Fleming and D. McClean, *Scan-Based Design Verification - An Alternative Approach*, 1990, ATE and Instrumentation Conference West, URL:http://www.ti.com/sc/docs/psheets/appnote.htm.

[Fuj85] H. Fujiwara, *Logic Testing and Design for Testability*, MIT Press, 1985.

[FW88] P.G. Frankl and E.J. Weyuker, *An Applicable Family of Data Flow Testing Criteria*, IEEE Transactions on Software Engineering **14** (1988), no. 10, 1483–1498.

[G+96] G. Goossens et al., *Programmable Chips in Consumer Electronics and Telecommunications: Architecture and Design Technology*, In Micheli and Sami [MS96], pp. 135–164.

[Gei96] M.C.W. Geilen, *Real-Time Concepts for Software/Hardware Engineering*, Master's thesis, Eindhoven University of Technology, 1996.

[GK83] D.D. Gajski and R.H. Kuhn, *Guest editor's introduction: new VLSI tools*, IEEE Computer **16** (1983), no. 12, 11–14.

[GM93] M.J.C. Gordon and T.F. Melham (eds.), *Introduction to HOL: A theorem proving environment for higher order logic*, Cambridge University Press, 1993.

[GM96] R.K. Gupta and G. De Micheli, *A Co-Synthesis Approach to Embedded System Design Automation*, Design Automation for Embedded Systems **1** (1996), no. 1–2, 69–120.

[GMG90] H.H.S. Gundlach and K.-D. Müller-Glaser, *On Automatic Testpoint Insertion in Sequential Circuits*, Proceedings International Test Conference, IEEE, 1990, pp. 1072–1079.

[Gol79] L.H. Goldstein, *Controllability/Observability Analysis of Digital Circuits*, IEEE Transactions on Circuits and Systems **26** (1979), no. 9, 685–693.

[Gom93] H. Gomaa, *Software Design Methods for Concurrent and Real-Time Systems*, Addison-Wesley, 1993.

[Gön70] G. Gönenç, *A Method for the Design of Fault Detection Experiments*, IEEE Transactions on Computers **19** (1970), no. 6, 551–558.

[Gor91] M.M. Gorlick, *The Flight Recorder: An Architectural Aid for System Monitoring*, ACM SIGPLAN Notices **26** (1991), no. 12, 175–183, Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging.

[GR93] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[Gra79] J. Grason, *TMEAS, a Testability Measurement Program*, Proceedings Design Automation Conference, ACM/IEEE, 1979, pp. 156–161.

[Gra86] J. Gray, *Why do computers stop and what can be done about it?*, Symposium on Reliability in Distributed Software and Database Systems, IEEE, 1986, pp. 3–12.

[Gra90] J. Gray, *A Census of Tandem System Availability Between 1985 and 1990*, IEEE Transactions on Reliability **39** (1990), no. 4, 409–418.

[GT80] L.H. Goldstein and E.L. Thigpen, *SCOAP: Scandia Controllability/Observability Analysis Program*, Proceedings Design Automation Conference, ACM/IEEE, 1980, pp. 190–196.

[Gup95] R.K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, Kluwer, 1995.

[GV95] D.D. Gajski and F. Vahid, *Specification and Design of Embedded Hardware-Software Systems*, IEEE Design & Test of Computers **12** (1995), no. 1, 53–67.

[GVNG94] D.D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, Prentice Hall, 1994.

[H$^+$90a] D. Harel et al., *STATEMATE: A Working Environment for the Development of Complex Reactive Systems*, IEEE Transactions on Software Engineering **16** (1990), no. 4, 403–414.

[H$^+$90b] P.N. Hilfinger et al., *DSP specification using the SILAGE language*, Proceedings International Conference on Acoustics, Speech and Signal Processing, IEEE, 1990, pp. 1057–1060.

[H$^+$94a] R. Hofmann et al., *Distributed Performance Monitoring: Methods, Tools, and Applications*, IEEE Transactions on Parallel and Distributed Systems **5** (1994), no. 6, 585–598.

[H$^+$94b] K. Holdbrook et al., *microSPARC^{TM}: A Case-Study of Scan Based Debug*, Proceedings International Test Conference, IEEE, 1994, pp. 70–75.

[HA95] H. Hao and R. Avra, *Structured Design-for-Debug - the SuperSPARC^{TM} II Methodology and Implementation*, Proceedings International Test Conference, IEEE, 1995, pp. 175–183.

[Hab87] D. Haban, *DTM - A Method for Testing Distributed Systems*, Proceedings Symposium on Reliability in Distributed Software and Database Systems, IEEE, 1987, pp. 66–73.

[HB95] H. Hao and K. Bhabuthmal, *Clock Controller Design in SuperSPARC^{TM} II Microprocessor*, Proceedings International Conference on Computer Design, IEEE, 1995, pp. 124–129.

[HK92] O.F. Haberl and T. Kropf, *HIST: A Methodology for the Automatic Insertion of a Hierarchical Self Test*, Proceedings International Test Conference, IEEE, 1992, pp. 732–741.

[HK94] O.F. Haberl and T. Kropf, *Self Testable Boards with Standard* 1149.5 *Module Test and Maintenance (MTM) Bus Interface*, Proceedings European Design & Test Conference, IEEE, 1994, pp. 220–225.

[HO96] R. Helaihel and K. Olukotun, *Emulation and Prototyping of Digital Systems*, In Micheli and Sami [MS96], pp. 339–366.

[Hoa85] C. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[Hol91] G.J. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall, 1991.

[HP87] D.J. Hatley and I.A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, 1987.

[HPSS87] D. Harel, A. Pneuli, J. Schmidt, and R. Sherman, *Statecharts: A Visual Formalism for Complex Systems*, Science of Computer Programming **8** (1987), 231–274.

[HR96a] G. Al Hayek and C. Robach, *From Specification Validation to Hardware Testing: A Unified Approach*, Proceedings International Test Conference, IEEE, 1996, pp. 885–893.

[HR96b] G. Al Hayek and C. Robach, *On the Adequacy of Deriving Hardware Test Data from the Behavioral Specification*, Proceedings EUROMICRO Conference, IEEE, 1996, pp. 337–342.

[HS88] J. Henshall and S. Shaw, *OSI Explained: End-To-End Computer Communication Standards*, Horwood, 1988.

[HS89] R.V. Hudli and S.C. Seth, *Testability Analysis of Synchronous Sequential Circuits Based On Structural Data*, Proceedings International Test Conference, IEEE, 1989, pp. 364–372.

[HSBP95] G. Hetherington, G. Sutton, K.M. Butler, and T.J. Powell, *Test Generation and Design for Test for a Large Multiprocessing DSP*, Proceedings International Test Conference, IEEE, 1995, pp. 149–156.

[HVTL94] C. Hunter, E.K. Vida-Torku, and J. LeBlanc, *Balancing Structured and Ad-hoc Design for Test: Testing of the PowerPC 603^{TM} Microprocessor*, Proceedings International Test Conference, IEEE, 1994, pp. 76–83.

[HW90] D. Haban and D. Wybranietz, *A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems*, IEEE Transactions on Software Engineering **16** (1990), no. 2, 197–211.

[HYC89] A. Halliday, G. Young, and A. Crouch, *Prototype Testing Simplified by Scannable Buffers and Latches*, Proceedings International Test Conference, IEEE, 1989, pp. 174–181.

[IAJ94] T. Ben Ismail, M. Abid, and A. Jerraya, *COSMOS: A CoDesign Approach for Communicating Systems*, Proceedings Third International Workshop on Hardware/Software Codesign, IEEE, 1994, pp. 17–24.

[IEE90] IEEE, *Standard 1149.1, IEEE Standard Test Access Port and Boundary-Scan Architecture*, 1990.

[IEE95] IEEE, *Standard 1149.5, IEEE Standard for Module Test and Maintenance Bus (MTM-Bus) Protocol*, 1995.

[IJ95] T. Ben Ismail and A.A. Jerraya, *Synthesis Steps and Design Models for Codesign*, IEEE Computer **28** (1995), no. 2, 44–52.

[Int93] Intel, *Pentium^{TM} Processor User's Manual*, 1993.

[IOJ94] T. Ben Ismail, K. O'Brien, and A. Jerraya, *Interactive System-level Partitioning with PARTIF*, Proceedings European Design and Test Conference, IEEE, 1994, pp. 464–468.

[Ism96] T. Ben Ismail, *Synthèse au Niveau Système et Conception de Systèmes Mixtes Logiciels/Matériels (System-Level Synthesis and Design of Mixed Hardware/Software Systems)*, Ph.D. thesis, TIMA-INPG, Grenoble, France, 1996.

[ISO89] ISO/IEC, *ISO/IEC 7498-4:1989 Information Processing Systems – Open Systems Interconnection – Basic Reference Model – Part 4: Management Framework*, 1989, (ITU-T Recommendation X.700).

[ISO92] ISO/IEC, *ISO/IEC 10040:1992 Information Technology – Open Systems Interconnection – Systems Management Overview*, 1992, (ITU-T Recommendation X.701).

[ISO94a] ISO/IEC, *ISO/IEC 10164-12:1994 Information Technology – Open Systems Interconnection – Systems Management: Test Management Function*, 1994, (ITU-T Recommendation X.745).

[ISO94b] ISO/IEC, *ISO/IEC 7498-1:1994 Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*, 1994, (ITU-T Recommendation X.200).

[ISO94c] ISO/IEC, *ISO/IEC 9646:1994 Information Technology – Open Systems Interconnection – Conformance Testing Methodology and Framework*, 1994, (ITU-T Recommendation X.290-X.296).

[ISO96] ISO/IEC, *ISO/IEC 10164-14:1996 Information Technology – Open Systems Interconnection – System Management: Confidence and Diagnostic Test Categories*, 1996, (ITU-T Recommendation X.737).

[J$^+$92] I. Jacobson et al., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

[JA90] B.N. Jain and A.K. Agrawala, *Open Systems Interconnection: Its Architecture and Protocols*, Elsevier, 1990.

[Jah95] F. Jahanian, *Run-Time Monitoring of Real-Time Systems*, Advances in Real-Time Systems (S.H. Son, ed.), Prentice Hall, 1995, pp. 435–460.

[JCDZ86] W. Jian-Chao and W. Dao-Zheng, *A New Testability Measure for Digital Circuits*, Proceedings International Test Conference, IEEE, 1986, pp. 506–512.

[JDA93] D.D. Josephson, D.J. Dixon, and B.J. Arnold, *Test Features of the HP PA7100LC Processor*, Proceedings International Test Conference, IEEE, 1993, pp. 764–772.

[JK93] M. Jamoussi and B. Kaminska, *Controllability and Observability Measures for Functional-Level Testability Evaluation*, Proceedings VLSI Test Symposium, IEEE, 1993, pp. 154–157.

[JLSU87] J. Joyce, G. Lomow, K. Slind, and B. Unger, *Monitoring Distributed Systems*, ACM Transactions on Computer Systems **5** (1987), no. 2, 121–150.

[JO95] A.A. Jerraya and K. O'Brien, *SOLAR: An Intermediate Format for System-Level Modeling and Synthesis*, In Rozenblit and Buchenrieder [RB95], pp. 145–175.

[Joh89] B.W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley, 1989.

[Kat94] J. Katz, *A Case-Study in the use of Scan in microSPARC^{TM} testing and debug*, Proceedings International Test Conference, IEEE, 1994, pp. 456–460.

[KL93] A. Kalavade and E.A. Lee, *A Hardware-Software Co-Design Methodology for DSP Applications*, IEEE Design & Test of Computers **10** (1993), no. 3, 16–28.

[Kni93] K.G. Knightson, *OSI Protocol Conformance Testing: IS 9646 Explained*, McGraw-Hill, 1993.

[Koo91] C.J. Koomen, *The Design of Communicating Systems: A System Engineering Approach*, Kluwer, 1991.

[KSFM95] J. Kumar, N. Stradar, J. Freeman, and M. Miller, *Emulation Verification of the Motorola 68060*, Proceedings International Conference on Computer Design, IEEE, 1995, pp. 150–158.

[Kum97] J. Kumar, *Prototyping the M68060 for Concurrent Verification*, IEEE Design & Test of Computers **14** (1997), no. 1, 34–41.

[L$^+$91] J.A. Lyon et al., *Testability Features of the 68HC16Z1*, Proceedings International Test Conference, IEEE, 1991, pp. 122–130.

[L+95] M.E. Levitt et al., *Testability, Debuggability, and Manufacturability Features of the UltraSPARC^{TM}-I Microprocessor*, Proceedings International Test Conference, IEEE, 1995, pp. 157–166.

[LA90] P.A. Lee and T. Anderson, Dependable Computing and Fault-Tolerant Systems, vol. 3 Fault Tolerance: Principles and Practice, ch. 2, Prentice-Hall, second ed., 1990.

[Lap92a] J.-C. Laprie, *Dependability Modeling and Evaluation of Computing Systems*, Hardware and Software Fault Tolerance in Parallel Computing Systems (D.R. Avresky, ed.), Ellis Horwood, 1992.

[Lap92b] J.-C. Laprie, *Dependable Computing and Fault-Tolerant Systems*, vol. 5 Dependability: Basic Concepts and Terminology, Springer-Verlag, 1992.

[Lee94] P.A. Lee, *Software-Faults: The Remaining Problem in Fault Tolerant Systems?*, Hardware and Software Architectures for Fault Tolerance: Experiences and Perspectives (M. Banatre and P.A. Lee, eds.), Springer-Verlag, 1994, Lecture Notes in Computer Science 774, pp. 171–181.

[Lef90] M.F. Lefebvre, *Functional Test and Diagnosis: A Proposed JTAG Sample Mode Scan Tester*, Proceedings International Test Conference, IEEE, 1990, pp. 294–303.

[Lev97] M.E. Levitt, *Designing UltraSparc for Testability*, IEEE Design & Test of Computers **14** (1997), no. 1, 10–17.

[LHC93] S.L.A. Lo, N.C. Hutchinson, and S.T. Chanson, *Architectural Considerations in the Design of Real-Time Kernels*, Proceedings Real-Time Systems Symposium, IEEE, 1993, pp. 138–147.

[LP93] J. Lee and J.H. Patel, *Testability Analysis Based on Structural and Behavioral Information*, Proceedings VLSI Test Symposium, IEEE, 1993, pp. 139–146.

[LR90] D.H. Lee and S.M. Reddy, *On Determining Scan Flip-Flops in Partial-Scan Designs*, Proceedings International Conference on Computer-Aided Design, 1990, pp. 322–325.

[LSVH96] L. Lavagno, A. Sangiovanni-Vincentelli, and H. Hsieh, *Embedded System Co-Design: Synthesis and Verification*, In Micheli and Sami [MS96], pp. 213–242.

[LSW88] E.L. Lloyd, M.L. Soffa, and C.-C. Wang, *On Locating Minimum Feedback Vertex Sets*, Journal of Computer and System Sciences **37** (1988), 292–311.

[LV94] B. Lin and S. Vercauteren, *Synthesis of Concurrent System Interface Modules with Automatic Protocol Conversion Generation*, Proceedings International Conference on Computer-Aided Design, ACM, 1994, pp. 101–108.

[LVM96] B. Lin, S. Vercauteren, and H. De Man, *Embedded Architecture Co-Synthesis and System Integration*, Proceedings International Workshop on Hardware/Software Co-Design (Codes/CASHE'96), IEEE, 1996, pp. 2–9.

[M+96] H. De Man et al., *Co-Design of DSP Systems*, In Micheli and Sami [MS96], pp. 75–104.

[MA80] T.E. Mangir and A. Avizienis, *Failure Modes for VLSI and their Effect on Chip Design*, Proceedings International Conference on Circuits and Computers, IEEE, 1980, pp. 685–688.

[Mad95] V. Madisetti, *System-Level Synthesis and Simulation VHDL: A Taxonomy and Proposal Towards Standardization*, Proceedings VHDL International Users' Forum, 1995.

[Mag93] S. Maguire, *Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Programs*, Microsoft Press, 1993.

[Mal87] W. Maly, *Realistic Fault Modelling for VLSI Testing*, Proceedings Design Automation Conference, ACM/IEEE, 1987, pp. 173–180.

[Mar95] N. Maretti, *Mechanized Implementation Verification*, Ph.D. thesis, Technical University of Denmark, 1995.

[Mau93] C. Maunder, *A Universal Framework for Managed Built-In Test*, Proceedings International Test Conference, IEEE, 1993, pp. 21–29.

[MCC96] MCC (Microelectronics and Computer Technology Corp.), Austin, Texas, and OMI (Open Microprocessor Initiative), Brussels, Belgium, *Joint MCC/OMI Hardware/Software Codesign Study Report*, 1996.

[McR95] R. McRee, *Testing and Diagnosing Managed Networks*, IEEE Design & Test of Computers **12** (1995), no. 4, 68–80.

[ME95] V.K. Madisetti and T.W. Egolf, *Virtual Prototyping of Embedded Microcontroller-Based DSP Systems*, IEEE Micro **15** (1995), no. 5, 9–21.

[Mel94] N. Mellergaard, *Mechanized Design Verification*, Ph.D. thesis, Technical University of Denmark, 1994.

[MEMMR95] T.E. Marchok, A. El-Maleh, W. Maly, and J. Rajski, *Complexity of Sequential ATPG*, Proceedings European Design & Test Conference, IEEE, 1995, pp. 252–261.

[MH88] B.T. Murray and J.P. Hayes, *Hierarchical Test Generation Using Precomputed Tests for Modules*, Proceedings International Test Conference, IEEE, 1988, pp. 221–229.

[MH89] C.E. McDowell and D.P. Helmbold, *Debugging Concurrent Programs*, ACM Computing Surveys **21** (1989), no. 4, 593–622.

[MH91] B.T. Murray and J.P. Hayes, *Test Propagation Through Modules and Circuits*, Proceedings International Test Conference, IEEE, 1991, pp. 748–757.

[Mic94] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[Mic96] G. De Micheli, *Hardware/Software Co-Design: Application Domains and Design Technologies*, In Micheli and Sami [MS96], pp. 1–28.

[Mil80] R. Milner, *A Calculus of Communicating Systems*, Springer-Verlag, 1980, Lecture Notes in Computer Science 92.

[Mil85] G.J. Milne, *CIRCAL and the Representation of Communication, Concurrency, and Time*, ACM Transactions on Programming Languages and Systems **7** (1985), no. 2, 270–298.

[Mil89] R. Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[Mil91] G.J. Milne, *The Formal Description and Verification of Hardware Timing*, IEEE Transactions on Computers **40** (1991), no. 7, 811–826.

[MKW93] E.J. Marinissen, K. Kuiper, and C. Wouters, *Testability and Test Protocol Expansion in Hierarchical Macro Testing*, Proceedings European Test Conference, IEEE, 1993, pp. 28–36.

[MRW92] A.D. Malony, D.A. Reed, and H.A.G. Wijshoff, *Performance Measurement Intrusion and Perturbation Analysis*, IEEE Transactions on Parallel and Distributed Systems **3** (1992), no. 4, 433–450.

[MS96] G. De Micheli and M. Sami (eds.), *Hardware/Software Co-Design*, Proceedings NATO Advanced Study Institute on Hardware/Software Co-Design, NATO ASI Series E: Applied Sciences, vol. 310, Kluwer, 1996.

[MT90] C.M. Maunder and R.E. Tulloss, *The Test Access Port and Boundary Scan Architecture*, IEEE Computer Society Press, 1990.

[MU84] M.R. Mercer and B. Underwood, *Correlating Testability with Fault Detection*, Proceedings International Test Conference, IEEE, 1984, pp. 697–704.

[Mur94] B.T. Murray, *Hierarchical Testing Using Precomputed Tests for Modules*, Ph.D. thesis, University of Michigan, 1994.

[NAS88] NASA (National Aeronautics and Space Administration), *NSTS Shuttle Reference Manual*, 1988, URL:http://shuttle.nasa.gov/reference.

[NG94a] S. Narayan and D.D. Gajski, *Protocol Generation for Communication Channels*, Proceedings Design Automation Conference, ACM/IEEE, 1994, pp. 547–552.

[NG94b] S. Narayan and D.D. Gajski, *Synthesis of System-Level Bus Interfaces*, Proceedings European Design and Test Conference, IEEE, 1994, pp. 395–399.

[NG95] S. Narayan and D. Gajski, *Interfacing Incompatible Protocols using Interface Process Generation*, Proceedings Design Automation Conference, ACM/IEEE, 1995, pp. 468–473.

[NG96] W. Needham and N. Gollakota, *DFT Strategy for Intel Microprocessors*, Proceedings International Test Conference, IEEE, 1996, pp. 396–399.

[NH84] R. De Nicola and M.C.B. Hennessy, *Testing Equivalences for Processes*, Theoretical Computer Science **34** (1984), 83–133.

[NT81] S. Naito and M. Tsunoyama, *Fault Detection for Sequential Machines by Transition-Tours*, Digest of Papers International Symposium on Fault-Tolerant Computing, IEEE, 1981, pp. 238–243.

[OIJ93] K. O'Brien, T. Ben Ismail, and A.A. Jerraya, *A Flexible Communication Modelling Paradigm for System-Level Synthesis*, Handouts International Workshop on Hardware-Software Co-Design, 1993.

[P$^+$96a] P. Paulin et al., *Trends in Embedded System Technology: An Industrial Perspective*, In Micheli and Sami [MS96], pp. 311–337.

[P$^+$96b] C. Pixley et al., *Commercial Design Verification: Methodology and Tools*, Proceedings International Test Conference, IEEE, 1996, pp. 839–848.

[PA92] S. Park and S.B. Akers, *A Graph Theoretic Approach to Partial Scan Design by K-Cycle Elimination*, Proceedings International Test Conference, 1992, pp. 303–311.

[Par92] K.P. Parker, *The Boundary-Scan Handbook*, Kluwer, 1992.

[PDW95] M. Potkonjak, S. Dey, and K. Wakabayashi, *Design-For-Debugging of Application Specific Designs*, Proceedings International Conference on Computer-Aided Design, IEEE/ACM, 1995, pp. 295–301.

[PH94] D.A. Patterson and J.L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 1994.

[Pla84] B. Plattner, *Real-Time Execution Monitoring*, IEEE Transactions on Software Engineering **10** (1984), no. 6, 756–764.

[Pra95] V. Pratt, *Anatomy of the Pentium Bug*, TAPSOFT '95: Theory and Practice of Software Development (P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, eds.), Springer-Verlag, 1995, Lecture Notes in Computer Science 915, pp. 97–107.

[PS93] D.E. Perry and C.S. Stieg, *Software Faults in Evolving a Large, Real-Time System: a Case Study*, Proceedings European Software Engineering Conference, IEEE, 1993, pp. 48–67.

[Pur94] Pure Software Inc., Sunnyvale, CA, *Purify User's Guide*, 1994, URL:http://www.pure.com.

[PY93] R. Patel and K. Yarlagadda, *Testability Features of the SuperSPARC^{TM} Microprocessor*, Proceedings International Test Conference, IEEE, 1993, pp. 773–781.

[R$^+$91] J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[RASa] *RASSP VHDL Modeling Terminology and Taxonomy*, URL:http://rassp.scra.org.

[RASb] *RASSP Information Server*, URL:http://rassp.scra.org.

[RAvG96] J. Rooijmans, H. Aerts, and M. van Genuchten, *Software Quality in Consumer Electronics Products*, IEEE Software **13** (1996), no. 1, 55–64.

[RB95] J. Rozenblit and K. Buchenrieder (eds.), *Codesign: Computer-Aided Software/-Hardware Engineering*, IEEE Press, 1995.

[RMM84] C. Robach, P. Malecha, and G. Michel, *CATA: A Computer-Aided Test Analysis System*, IEEE Design & Test of Computers **1** (1984), no. 2, 68–79.

[Row94] J.A. Rowson, *Hardware/Software Co-Simulation*, Proceedings Design Automation Conference, ACM/IEEE, 1994, pp. 439–440.

[RRJ92] S.C.V. Raju, R. Rajkumar, and F. Jahanian, *Monitoring Timing Constraints in Distributed Real-time Systems*, Proceedings Real-Time Systems Symposium, IEEE, 1992, pp. 57–67.

[RVBM96] K. Van Rompaey, D. Verkest, I. Bolsens, and H. De Man, *CoWare—A design environment for heterogeneous hardware/software systems*, Proceedings EURO-DAC, 1996.

[RW85] S. Rapps and E.J. Weyuker, *Selecting Software Test Data Using Data Flow Information*, IEEE Transactions on Software Engineering **11** (1985), no. 4, 367–375.

[Sar93] B. Sarikaya, *Principles of Protocol Engineering and Conformance Testing*, Ellis Horwood, 1993.

[Sav83] J. Savir, *Good Controllability and Observability Do Not Guarantee Good Testability*, IEEE Transactions on Computers **32** (1983), no. 12, 1198–1200.

[SB92] J.S. Sun and R.W. Brodersen, *Design of System Interface Modules*, Proceedings International Conference on Computer-Aided Design, IEEE, 1992, pp. 478–481.

[SB94] H.P. Sharangpani and M.L. Barton, *Statistical Analysis of Floating Point Flaw in the Pentium Processor (1994)*, Intel Corporation, November 1994, URL:http://support.intel.com/procs/support/pentium/fdiv/white11/index.htm.

[SC91] M. Sullivan and R. Chillarege, *Software Defects and their Impact on System Availability - A Study of Field Failures in Operating Systems*, Proceedings International Symposium on Fault-Tolerant Computing, IEEE, 1991, pp. 2–9.

[SC92] M. Sullivan and R. Chillarege, *A Comparison of Software Defects in Database Management Systems and Operating Systems*, Proceedings International Symposium on Fault-Tolerant Computing, IEEE, 1992, pp. 475–484.

[Sch93a] S.E. Schultz, *An Overview of System Design*, ASIC & EDA (1993), 12–21.

[Sch93b] W. Schütz, *The Testability of Distributed Real-Time Systems*, Kluwer, 1993.

[SD88] K. Sabnani and A. Dahbura, *A Protocol Test Generation Procedure*, Computer Networks and ISDN Systems **15** (1988), no. 4, 285–297.

[SG76] J.E. Stephenson and J. Grason, *A Testability Measure for Register Transfer Level Digital Circuits*, Proceedings International Symposium on Fault-Tolerant Computing, IEEE, 1976, pp. 101–107.

[SGP83] R. Spillman, N. Glaser, and D. Peterson, *Development of a General Testability Figure-of-Merit*, International Conference on Computer-Aided Design, Digest of Technical Papers, IEEE/ACM, 1983, pp. 34–35.

[SIA94] SIA Semiconductor Industry Association, *The National Technology Roadmap for Semiconductors*, 1994.

[SNH95] D. Soni, R.L. Nord, and C. Hofmeister, *Software Architecture in Industrial Applications*, Proceedings International Conference on Software Engineering, IEEE, 1995, pp. 196–207.

[Spu94] D.A. Spuler, *C++ and C Debugging, Testing and Reliability: the Prevention, Detection, and Correction of Program Error*, Prentice Hall, 1994.

[SS91a] J.W. Sheppard and W.R. Simpson, *A Mathematical Model for Integrated Diagnostics*, IEEE Design & Test of Computers **8** (1991), no. 4, 25–38.

[SS91b] W.R. Simpson and J.W. Sheppard, *System Complexity and Integrated Diagnostics*, IEEE Design & Test of Computers **8** (1991), no. 3, 16–30.

[SS92a] J.W. Sheppard and W.R. Simpson, *Applying Testability Analysis for Integrated Diagnostics*, IEEE Design & Test of Computers **9** (1992), no. 3, 65–78.

[SS92b] W.R. Simpson and J.W. Sheppard, *System Testability Assessment for Integrated Diagnostics*, IEEE Design & Test of Computers **9** (1992), no. 1, 40–54.

[SS93a] J.W. Sheppard and W.R. Simpson, *Performing Effective Fault Isolation in Integrated Diagnostics*, IEEE Design & Test of Computers **10** (1993), no. 2, 78–90.

[SS93b] W.R. Simpson and J.W. Sheppard, *Fault Isolation in an Integrated Diagnostic Environment*, IEEE Design & Test of Computers **10** (1993), no. 1, 52–66.

[SS94a] W.R. Simpson and J.W. Sheppard, *System Test and Diagnosis*, Kluwer, 1994.

[SS94b] M. Singhal and N.G. Shivaratri, *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems*, McGraw-Hill, 1994.

[Sta92] W. Stallings, *Operating Systems*, MacMillan, 1992.

[SW75] G.W. Smith, Jr and R.B. Walford, *The Identification of a Minimal Feedback Vertex Set of a Directed Graph*, IEEE Transactions on Circuits and Systems **22** (1975), no. 1, 9–15.

[T^{+}87] A. Tevanian et al., *Mach Threads and the Unix Kernel: The Battle for Control*, Proceedings of the USENIX Summer Conference, 1987, pp. 185–197.

[TA80] S.M. Thatte and J.A. Abraham, *Test Generation for Microprocessors*, IEEE Transactions on Computers **C-29** (1980), no. 6, 429–441.

[TA89] K. Thearling and J. Abraham, *An Easily Computed Functional Level Testability Measure*, Proceedings International Test Conference, IEEE, 1989, pp. 381–390.

[Tan95] A.S. Tanenbaum, *Distributed Operating Systems*, Prentice-Hall, 1995.

[Tar72] R.E. Tarjan, *Depth First Search and Linear Graph Algorithms*, SIAM Journal of Computing **1** (1972), no. 2, 146–160.

[TAS93] D.E. Thomas, J.K. Adams, and H. Schmit, *A Model and Methodology for Hardware-Software Codesign*, IEEE Design & Test of Computers **10** (1993), no. 3, 6–15.

[Tex96] Texas Instruments, *IEEE Std* 1149.1 *(JTAG) Testability Primer*, 1996.

[TFC90] J.J.P. Tsai, K.-Y. Fang, and H.-Y. Chen, *A Noninvasive Architecture to Monitor Real-Time Distributed Systems*, Computer **23** (1990), no. 3, 11–23.

[TFCB90] J.J.P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi, *A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging*, IEEE Transactions on Software Engineering **16** (1990), no. 8, 897–916.

[THR96] Y. Le Traon, G. Al Hayek, and C. Robach, *Testability-Oriented Hardware/ Software Partitioning*, Proceedings International Test Conference, IEEE, 1996, pp. 725–731.

[TKM89] H. Tokuda, M. Kotera, and C.W. Mercer, *A Real-Time Monitor for a Distributed Real-Time Operating System*, ACM SIGPLAN Notices **24** (1989), no. 1, 68–77, Proceedings ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging.

[TM96] N.A. Touba and E.J. McCluskey, *Test Point Insertion Based on Path Tracing*, Proceedings VLSI Test Symposium, IEEE, 1996, pp. 2–8.

[TR95a] Y. Le Traon and C. Robach, *Form hardware to software testability*, Proceedings International Test Conference, IEEE, 1995, pp. 710–719.

[TR95b] Y. Le Traon and C. Robach, *Towards a Unified Approach to the Testability of Co-Designed Systems*, Proceedings International Symposium on Software Reliability Engineering, IEEE, 1995, pp. 278–285.

[TSV94] M. Theißinger, P. Stravers, and H. Veit, *Castle: An Interactive Environment for HW-SW Co-Design*, Proceedings Third International Workshop on Hardware/Software Codesign, IEEE, 1994, pp. 203–209.

[TY95] J.J.P. Tsai and S.J.H. Yang (eds.), *Monitoring and Debugging of Distributed Real-Time Systems*, IEEE Computer Society Press, 1995.

[V+95a] C.A. Valderrama et al., *A Unified Model for Co-simulation and Co-synthesis of Mixed Hardware/Software Systems*, Proceedings European Design and Test Conference, IEEE, 1995, pp. 180–184.

[V+95b] K.A. Vissers et al., *Architecture and programming of two generations video signal processors*, Microprocessing and Microprogramming **41** (1995), 373–390.

[VAA92] P. Vishakantaiah, J. Abraham, and M. Abadir, *Automatic Test Knowledge Extraction From VHDL (ATKET)*, Proceedings Design Automation Conference, ACM/IEEE, 1992, pp. 273–278.

[VCI90] S.T. Vuong, W.W.L. Chan, and M.R. Ito, *The UIOv-Method for Protocol Test Sequence Generation*, Protocol Test Systems (J. De Meer, L. Mackert, and W. Effelsberg, eds.), Elsevier, 1990, IFIP, pp. 161–175.

[vdPV97] P.H.A. van der Putten and J.P.M. Voeten, *Specification of Reactive Hardware/Software Systems: The method Software/Hardware Engineering (SHE)*, Ph.D. thesis, Eindhoven University of Technology, 1997.

[vdPVS95] P.H.A. van der Putten, J.P.M. Voeten, and M.P.J. Stevens, *Object-Oriented Co-Design for Hardware/Software Systems*, Proceedings of EUROMICRO, IEEE, 1995, pp. 718–726.

[VIJK94] M. Voss, T. Ben Ismail, A.A. Jerraya, and K.-H. Kapp, *Towards a Theory for Hardware/Software Codesign*, Proceedings International Workshop on Hardware/Software Codesign, IEEE, 1994, pp. 173–180.

[Voe95a] J.P.M. Voeten, *POOSL: An Object-Oriented Language for the Analysis and Design of Hardware/Software Systems*, EUT Report 95-E-290, Eindhoven University of Technology, 1995.

[Voe95b] J.P.M. Voeten, *Semantics of POOSL: An Object-Oriented Language for the Analysis and Design of Hardware/Software Systems*, EUT Report 95-E-293, Eindhoven University of Technology, 1995.

[Vra94] H.P.E. Vranken, *A Structured Approach towards System-Level Testability of Hardware/Software Systems*, Tech. report, Eindhoven University of Technology, Institute for Continuing Education, 1994.

[VRBM96] D. Verkest, K. Van Rompaey, I. Bolsens, and H. De Man, *CoWare - A Design Environment for Heterogeneous Hardware/Software Systems*, Design Automation for Embedded Systems **1** (1996), no. 4, 357–386.

[vRBMV97] G.J. van Rootselaar, F. Bouwman, E.J. Marinissen, and M. Verstraelen, *Debugging of Systems on a Chip: Embedded Triggers*, International High Level Design Validation and Test Workshop (HLDVT), IEEE, 1997.

[vRV94] A. van Rangelrooij and J.P.M. Voeten, *CCSTOOL2: An Expansion, Minimization, and Verification Tool for Finite State CCS Descriptions*, EUT Report 94-E-284, Eindhoven University of Technology, 1994.

[VSS96] H.P.E. Vranken, M.P.J. Stevens, and M.T.M. Segers, *High Level DFT for Hardware/Software Co-Testing*, International High Level Design Validation and Test Workshop (HLDVT), IEEE, (handouts), 1996.

[VSS97] H.P.E. Vranken, M.P.J. Stevens, and M.T.M. Segers, *Design-For-Debug in Hardware/Software Co-Design*, Proceedings International Workshop on Hardware/Software Codesign (CODES/CASHE), IEEE, 1997, pp. 35–39.

[VSSvR94] H.P.E. Vranken, M.P.J. Stevens, M.T.M. Segers, and J.H.M.M. van Rhee, *System-Level Testability of Hardware/Software Systems*, Proceedings International Test Conference, IEEE, 1994, pp. 134–142.

[VVA93] K.V. Varma, P. Vishakantaiah, and J.A. Abraham, *Generation of Testable Designs from Behavioral Descriptions Using High Level Synthesis Tools*, Proceedings VLSI Test Symposium, IEEE, 1993, pp. 124–130.

[VvdPS96] J.P.M. Voeten, P.H.A. van der Putten, and M.P.J. Stevens, *Behaviour-Preserving Transformations in SHE: A Formal Approach to Architecture Design*, Proceedings of EUROMICRO, IEEE, 1996, pp. 19–27.

[VWvW96] H.P.E. Vranken, M.F. Witteman, and R.C. van Wuijtswinkel, *Design for Testability in Hardware-Software Systems*, IEEE Design & Test of Computers **13** (1996), no. 3, 79–87.

[vWW95] R.C. van Wuijtswinkel and M.F. Witteman, *Testing Using Telecommunications Management*, Protocol Test Systems VII, IFIP, Chapman & Hall, 1995, Proceedings International Workshop on Protocol Test Systems.

[W$^+$94] P. Willekens et al., *Algorithm specification in DSP station using data flow language*, DSP Applications **3** (1994), no. 1, 8–16.

[Whe92] L. Whetsel, *A Proposed Method of Accessings 1149.1 in a Backplane Environment*, Proceedings International Test Conference, IEEE, 1992, pp. 202–216.

[WM86] P.T. Ward and S.J. Mellor, *Structured Development for Real-Time Systems*, vol. 1: Introduction & Tools, 2: Essential Modeling Techniques, 3: Implementation Modeling Techniques, Prentice Hall, 1986.

[Wol94] W.H. Wolf, *Hardware-Software Co-Design of Embedded Systems*, Proceedings of the IEEE **82** (1994), no. 7, 967–989.

[WvW94] M.F. Witteman and R.C. van Wuijtswinkel, *ATM Broadband Testing Using the Ferry Principle*, Protocol Test Systems VI, IFIP, Elsevier, 1994, Proceedings International Workshop on Protocol Test Systems, pp. 125–138.

[Y+95] J.-T. Yen et al., *Overview of PowerPC^{TM} 620 Multiprocessor Verification Strategy*, Proceedings International Test Conference, IEEE, 1995, pp. 167–174.

[Yeh96] Y. Yeh, *Triple-Triple Redundant 777 Primary Flight Computer*, Proceedings Aerospace Applications Conference, vol. 1, IEEE, 1996, pp. 293–307.

[You89] E. Yourdon, *Modern Structured Analysis*, Prentice Hall, 1989.

[ZCS89] H.X. Zeng, S.T. Chanson, and B.R. Smith, *On Ferry Clip Approaches in Protocol Testing*, Computer Networks and ISDN Systems **17** (1989), 77–88.

[ZDH88] H.X. Zeng, X.F. Du, and C.S. He, *Promoting the "Local" Test Method with the New Concept "Ferry Clip"*, Protocol Specification, Testing, and Verification VIII, IFIP, Elsevier, 1988, Proceedings Symposium on Protocol Specification, Testing, and Verification, pp. 231–241.

[ZR86] H.X. Zeng and D. Rayner, *The Impact of the Ferry Concept on Protocol Testing*, Protocol Specification, Testing, and Verification V, IFIP, Elsevier, 1986, Proceedings Symposium on Protocol Specification, Testing, and Verification, pp. 519–531.

Index

Curriculum Vitae

Harald Vranken was born on June 6, 1969 in Maastricht, the Netherlands, and he grew up in a lovely village nearby, Oost-Maarland. From 1981 to 1987 he attended the Scholengemeenschap Jeanne d'Arc in Maastricht. Subsequently, he studied Information Technology at the Department of Electrical Engineering, Eindhoven University of Technology. His graduation project dealt with RTL architectures for programmable Digital Signal Processors. He received his Master's degree on August 27, 1992.

From September 1992 to December 1994 he attended the postgraduate designer course Information and Communication Technology at the Eindhoven Institute for Continuing Education (now Stan Ackermans Institute). He received the degree Master of Technological Design on December 8, 1994. His thesis concerned system-level testability of hardware/software systems, describing initial ideas that lie at the base of this Ph.D. thesis.

He continued his research on system-level testability from January 1995. He worked towards his doctor's degree as a research assistant in the Information and Communication Systems group at the Eindhoven University of Technology. His work was financed by Philips Electronic Design & Tools/Test. He will receive the doctor's degree for the work described in this thesis on June 2, 1998.

Since February 1, 1998, he is working as a member of the VLSI Design Automation & Test group in the IC Design sector at Philips Research Laboratories in Eindhoven.

Stellingen

behorende bij het proefschrift
Design For Test & Debug in Hardware/Software Systems
door H.P.E. Vranken

1. In de huidige ontwerpmethoden voor hardware/software co-design wordt nauwelijks aandacht besteed aan design for test & debug. Het testen en debuggen van aldus ontworpen hardware/software systemen blijft derhalve uitermate problematisch.
   [Dit proefschrift, hoofdstuk 2.]
2. Design for test & debug in hardware/software systemen dient zich met name te richten op het vergroten van de observeerbaarheid en de controleerbaarheid van communicatie interfaces en toestandsinformatie van processen in hardware en software.
   [Dit proefschrift, hoofdstuk 4.]
3. Design for test & debug voor hardware/software systemen dient een elementair onderdeel te zijn in alle ontwerp fasen, nl. systeem specificatie, architectuur onderzoek, architectuur verfijning en synthese.
   [Dit proefschrift, hoofdstuk 4.]
4. Het is mogelijk om het invoegen van PCOs uit te voeren als een correctheid behoudende transformatie in een transformationeel ontwerp systeem.
   [Dit proefschrift, hoofdstuk 5 en appendix A.]
5. De ontwerper belast met design for test & debug op systeem niveau, dient eenzelfde kennisniveau te hebben van het systeem gedrag en de systeem architectuur als de ontwerpers belast met systeem specificatie en architectuur definitie.
6. De mislukte lancering van de eerste Ariane 5 raket leert dat foutentolerantie in hardware/software systemen niet verkregen wordt door het gebruik van redundante, identieke componenten met ontwerpfouten of software fouten.
   [Dit proefschrift, hoofdstuk 3.]
7. Het 'debuggen' van hardware/software systemen kan helaas niet meer worden uitbesteed aan de gemeentelijke ongedierte bestrijdingsdienst.
8. Waarom reist de reiziger? Hij is op zoek naar het Andere. Alles op aarde is in wezen hetzelfde. Maar het zijn de kleine verschillen die ons het oude en bekende als nieuw doen zien.
   [Bertus Aafjes, *Dag van gramschap in Pompeji.*]
9. Het feit dat men in regionale kranten regelmatig onjuistheden kan aanwijzen in de berichtgeving over gebeurtenissen waarvan men persoonlijk de achtergrond kent, geeft ernstig te denken over de betrouwbaarheid van deze kranten.

10. Door haar gedoogbeleid op diverse gebieden draagt de Nederlandse overheid zelf bij aan de vervaging van normen en waarden.

11. In de industrie is een research succes pas een echt succes als het leidt tot een business succes. Aan de universiteiten daarentegen is een research succes pas een succes als het leidt tot een groot aantal hoogstaande publicaties.

12. De meeste promovendi zullen over hun promotiewerk zeggen: "That's one small step for mankind, but a giant leap for a man."
[Vrij naar Neil Armstrong bij het zetten van de eerste voet op de maan.]

13. De stellingname dat stellingen bij een proefschrift beproefbaar dienen te zijn, stelt eisen aan de stelligheid waarmee stellingen gesteld worden.

14. Tegen alle verwachting zal pagina 215 van dit proefschrift vaak geraadpleegd worden.