MASTER

Performance modeling of hardware systems with UML

van Ewijk, A.J.

Award date:
2001

Link to publication
Performance modeling of hardware systems with UML

Author: A.J. van Ewijk
Supervisor Alcatel Bell: dr. ir. A.M.M. Niemegeers
Supervisor ICS: ir. B.D. Theelen, dr. ir. J.P.M. Voeten
Professor: prof. ir. M.P.J. Stevens
Date: August 2001
Abstract

The field of real-time embedded systems is growing fast in complexity and designers of these systems have to look for methods that can cope with the complexity of today's and tomorrow's hardware systems. The project discussed in this thesis is part of the research project performed by Alcatel and the Section of Information and Communications Systems (ICS) of the faculty of Electrical Engineering of the Eindhoven University of Technology that aims at the development of a design flow suitable for designing embedded hardware-software systems. This thesis investigates how the Unified Modeling Language (UML) can be used for performance modeling of hardware systems. Based on the characteristics of hardware systems, formulated requirements and experiences in modeling of a network processor with UML, an UML modeling framework is presented that can be used in the Architecture Design Phase of the design flow. An existing performance modeling framework, used to create executable performance models, is extended in order to make it possible to create executable performance models based on models created with the UML modeling framework. The Software Hardware Engineering (SHE) method is used to assess the developed modeling framework.

Furthermore, simulation output analysis methods, based on confidence intervals, are evaluated. Two promising methods for confidence analysis and simulation run-length determination, the method of regenerative cycles and uncorrelated batches, are worked out. A test for confidence analysis methods based on coverage analysis is discussed. Algorithms involving the investigated confidence methods are introduced. The confidence methods are evaluated in simulations involving the network processor.

Keywords: Performance modeling, performance simulation, hardware, UML, confidence analysis, confidence intervals, simulation run-length, coverage analysis
## Contents

1 Introduction ......................................................................................................................... 1

1.1 Context of the project ........................................................................................................ 1

1.2 Problem statement and project description ...................................................................... 1

1.3 Outline of this thesis ......................................................................................................... 2

**PART I: UML in hardware design flow** .............................................................................. 3

2 Design flow of hardware systems ...................................................................................... 3

2.1 Sanford DataFlow network processor ............................................................................. 3

2.2 DataFlow design flow ....................................................................................................... 6

2.3 Performance modeling in the Architecture Design Phase ............................................. 9

2.4 UML in the design flow .................................................................................................. 12

3 Hardware modeling methods .............................................................................................. 14

3.1 Introduction ..................................................................................................................... 14

3.2 Properties of hardware systems ....................................................................................... 14

3.3 Requirements of modeling methods and frameworks .................................................... 16

3.4 Modeling languages, methodologies and tools ............................................................... 17

3.5 UML diagrams ............................................................................................................... 20

3.6 UML for Real-Time ........................................................................................................ 29

3.7 UML Tools ...................................................................................................................... 31

4 Modeling hardware systems with UML ........................................................................... 33

4.1 UML and hardware performance modeling .................................................................. 33

4.2 Specific hardware modeling challenges ........................................................................ 33

4.3 Developed performance modeling framework and heuristics .................................. 36

4.4 Requirements modeling ................................................................................................ 38

4.5 Performance Modeling ................................................................................................. 39

4.6 Performance modeling with developed modeling framework .................................. 42

4.7 Conclusions ..................................................................................................................... 45

5 Performance simulation of hardware systems .................................................................. 46

5.1 Abstractions in performance modeling of DataFlow ...................................................... 46

5.2 Currently used modeling framework .............................................................................. 47

5.3 Extended modeling framework ...................................................................................... 49

5.4 Extended modeling framework in C++ ....................................................................... 52

5.5 Extended modeling framework in UML-RT ................................................................. 57

5.6 Assessment extended modeling framework based on defined requirements ............. 64

5.7 Conclusions ..................................................................................................................... 65

6 Brief comparison of the extended modeling framework and SHE .................................. 66

6.1 Selected UML diagrams versus SHE diagrams ............................................................ 66

6.2 Synchronous models created with POOSL ................................................................. 67

6.3 A comparison between the extended modeling framework and POOSL .................. 71

**PART II: Performance simulation output analysis** ............................................................ 72

7 Simulation output analysis: confidence intervals ............................................................ 72

7.1 Introduction ..................................................................................................................... 72

7.2 Stochastic simulation and performance analysis ........................................................ 72

7.3 Simulation output analysis ............................................................................................. 75

7.4 Benchmarking the methods: coverage analysis ............................................................ 80

7.5 Fixed sample length simulation .................................................................................... 85

8 Simulation output analysis: confidence intervals ............................................................ 87

8.1 Sequential procedures for steady-state simulation ..................................................... 87

8.2 Batch means ................................................................................................................. 88

8.3 Method of regenerative cycles .................................................................................... 90

Alcatel Bell N.V.
8.4 Method of uncorrelated batches ........................................................................................................ 94
8.5 Detection of the initial transient ........................................................................................................ 101
8.6 Discussion explored solutions ............................................................................................................. 106
9 Application of confidence analysis in hardware simulation ................................................................. 108
  9.1 PA library ........................................................................................................................................ 108
  9.2 Hardware performance simulations .................................................................................................. 109
  9.3 Conclusions regarding confidence analysis ..................................................................................... 114
10 Conclusions and recommendations .................................................................................................... 116
Bibliography ............................................................................................................................................... 119
Abbreviations ........................................................................................................................................... 123
Acknowledgements ................................................................................................................................. 124
Appendix A: Deterministic Random Number Generators ........................................................................ 125
  A1 Need for good random number generators ..................................................................................... 125
  A2 Linear Congruential Generators ....................................................................................................... 126
  A3 Other RNGs ...................................................................................................................................... 127
  A4 Empirical tests on random number generators ................................................................................ 127
  A5 Influence of the seed ........................................................................................................................ 128
  A6 Conclusions and recommendations .................................................................................................. 128
Appendix B: UML modeling of DataFlow subsystem .............................................................................. 130
Appendix C: Extended modeling framework in C++ .............................................................................. 169
  C1 Ports in the C++ modeling framework ............................................................................................. 169
  C2 Model created with extended modeling framework in C++ ............................................................. 172
Appendix D: Extended modeling framework in RoseRT ...................................................................... 176
Appendix E: Performance analysis library in UML ............................................................................... 178
Appendix F: Derivation of the classical estimator for ratios of expectations ........................................ 181
Appendix G: Algorithm for detection of the initial transient (Schreuben algorithm) ........................... 183
1 Introduction

1.1 Context of the project

The project discussed in this thesis is performed as part of the cooperation agreement between Alcatel and the Section of Information and Communications Systems (ICS) of the faculty of Electrical Engineering of the Eindhoven University of Technology. This cooperation involves the development of a design flow suitable for designing embedded hardware-software systems. The project described in this thesis is performed in the hardware Architecture Design Group (WA10) of the Carrier Internetworking Division (CID) at Alcatel Bell Antwerp.

Alcatel builds next generation telecommunication networks (NGN). The system and hardware architecture is a key differentiator in Alcatel’s systems. Since the field of real-time embedded systems is growing fast in complexity, designers of these systems are urged to look for methods that can cope with the design problems they have to face. Moreover, these designers have to deal with the decreasing timeframe they get due to time to market of telecommunication systems. To deal with the complexity of hardware system a tremendous need emerges for methods that support the making of design decisions at an early stage in the design flow.

1.2 Problem statement and project description

The term “system-level design” is often used to refer to the process of definition of behavior and components of a system, based on specific requirements at an early stage in the design flow. It is known that performance modeling and analysis can be of great support in taking design decisions regarding system-level design. Evaluation of architectures, optimization of algorithms, determination of system dimensions and thresholds, detection of bottlenecks and rapid evaluation of system operation are known advantages of performance modeling. Performance analysis is mainly based on simulations performed using performance models. Performance models and performance simulation results are input for discussion among designers. The currently used performance modeling methods (see [N100a]) produce models that are difficult to use as input for discussions. There is need for methods that can be used to produce clear models that can be used to answer performance questions (performance analysis) regarding the system under design. These methods must support the creation of executable models, needed for performance analysis of the system, and support, next to the discussions regarding the system, also easy documentation of the system. Furthermore, an important aspect of performance analysis is how is dealt with the output of performance simulations. Performance analysis requires proper simulation output analysis.

The project described in this thesis investigates how object-oriented analysis (OOA), together with performance analysis can support the design of hardware systems. Investigated is how the Unified Modeling Language (UML) [UML1.3], a language based on the object-oriented paradigm, can support the creation of performance models. UML is a language for expressing the constructs and relationships of complex systems. It has gained vast popularity in software development areas, since the Object Modeling Group (OMG) introduced its first standard in 1997. The main goal is to evaluate the use of UML together with the methods that are currently used to model and design hardware systems. Due to its graphical presentations, UML is expected to be very useful in discussions among designers and documentation of a design. Since UML has already application in software development, modeling of hardware systems in UML can bring software and hardware designs and designers together. Another goal of the project is to develop guidelines on how UML can be used to model hardware systems. A method that can be used for modeling of real-time systems is the Software Hardware Engineering (SHE) method [PV97], which incorporates the POOSL language. This method is used as a reference for developed modeling frameworks and guidelines.

The project furthermore focuses on performance simulation output analysis. In order to be able to make constructive decisions about an investigated design, the precision of the output of simulation must be considered. The goal is to find methods that can help in evaluating the precision of simulation output.
The pilot system for this project is the Sanford DataFlow network processor system, which is for example used in Alcatel's Xantium high-speed IP routers. The functionality of this DataFlow system is implemented in an Application Specific Integrated Circuit (ASIC). The DataFlow system is a well-suited choice to investigate the possibilities of using a modeling language like UML for system level design purposes, since it is a representative design for the 'style' and philosophy of design that is applied by Alcatel for its hardware systems.

1.3 Outline of this thesis

This thesis exists of two parts. The first part concerns the performance modeling of complex hardware systems with UML. The second part discusses and introduces methods to analyze the output of performance simulation.

PART I

In chapter 2 the Sanford DataFlow network processor is discussed and an indication and justification is given for the use of UML in a part of the design flow of this system. In chapter 3 basic aspects of performance modeling are discussed. Characteristics and requirements of performance modeling methods involving hardware systems are given. Several modeling methods and languages are evaluated. In chapter 4, a newly created performance-modeling framework using UML is presented and discussed. In chapter 5, the currently used method for building executable performance modeling is extended to match it to the models created with UML. Furthermore a link is made to existing UML tools to support the building of these executable models. Chapter 6 presents a brief assessment of the modeling framework and compares it with the SHE method and the POOSL language.

PART II

Chapter 7 discusses requirements for methods for analyzing results of performance simulation (random number generators, simulation run-length determination). In chapter 8, two promising confidence analysis methods are worked out and benchmarked. Algorithms are introduced that can be used to implement the methods. In chapter 9, the confidence analysis methods discussed in chapter 8 are applied to performance simulation models created with the framework of PART I and confidence analysis results are presented.

In chapter 10, conclusions and recommendations are given.
PART I: UML in hardware design flow

The first part of this thesis focuses on the modeling of hardware systems for the purpose of performance analysis. The Sanford DataFlow network processor is used as an example design, which is a system completely implemented in an Application Specific Integrated Circuit (ASIC).

2 Design flow of hardware systems

In this chapter an explanation is given of the complex hardware system that is used as an example design. Furthermore, the design flow that was employed by the designers of the DataFlow system is presented. This design flow is investigated to evaluate the possibilities of using UML in the various stages of the design flow.

2.1 Sanford DataFlow network processor

The Xantium IP router is Alcatel’s next generation networks IP core router. The Xantium router is basically an input/output buffered switch [TA96] with a flow control mechanism. The system, used as example design throughout this thesis, the DataFlow subsystem, is part of the Sanford network processor chipset, which is used in the Xantium router. The Sanford chipset is part of a termination link system (TLK), which main purpose is IP/MPLS [TA96] packet forwarding to and from a Multi-Path Self-Routing (MPSR) switch core in the Xantium router. The Sanford system is used in two modes in the TLK system: ingress mode (input mode) and egress (output mode). From a functional point of view, the Sanford chipset is a network processor chipset that performs IP frame processing and forwarding at line speeds up to OC192c (10Gbps) level. The three chips Sanford exists of, are: DataFlow, EPC and Scheduler.

DataFlow performs IP packet buffering and bandwidth expansion to and from the MPSR switch. For buffering of frame data, the DataFlow chip uses the DataStore memory. The Scheduler chip schedules the appropriate frames for transmission, based on Quality of Service (QoS) priority classes and virtual connections that are setup through the MPSR switch core. EPC stands for Embedded Processor Complex. The EPC has multiple internal, parallel processors that perform router calculations on MPLS and IP packets.

In ingress mode of operation (frames traveling towards the MPSR switch), PPP frames [TA96] are sent to DataFlow. The incoming PPP packets are received on the incoming interface of the DataFlow chip and frame data is written in the DataStore memory. Frame data is after readout from the DataStore memory embedded by DataFlow in PRIZMA cells [VR00a] and sent towards the MPSR switch. In egress mode of operation, the DataFlow system receives PRIZMA cells from the MPSR switch. Frames are reassembled by DataFlow with support of the storing capacity of DataStore memory (frame data is stored in DataStore) and transmitted into the network.

2.1.1 DataFlow internal architecture

The internal architecture of the DataFlow system was, like the other chips of the Sanford chipset, still under design during the writing of this thesis. The architecture presented in this paragraph and used throughout this document represents the design that was known from an existing C++ model (see [NI00b]).

Fig 2.1 shows a block diagram of the internal architecture of DataFlow and the interfaces it has to its environment. The main functional modules in DataFlow are: Memory Arbiter, Receiver, Transmitter, EPC controller, Frame Control Block (FCB) memory, Buffer Control Block (BCB) memory, Buffer Acceptance (BA), DRAM controller (DataStore memory) and Scheduler.

Frames arrive in the Receiver module of DataFlow. The DataFlow system has input preparation FIFOs, called Preparation Area, or PrepArea, used to buffer frames in ingress mode of operation or to buffer PRIZMA cells in egress mode of operation.

Alcatel Bell N.V.
Key resource for DataFlow is the memory that will be denoted as ‘DataStore’, which is an external memory that exists of 6 DRAM (dynamic ram) memory chip which are denoted here as memory slices. The memory is needed to temporarily store arriving frame data until it can be forwarded through DataFlow’s output. The memory is a resource that is shared by the Receiver, Transmitter and the EPCController. To schedule the read/write accesses to DataStore, DataFlow is provided with the Memory Arbiter, which has a central role in DataFlow.

The Receiver issues write request to the Memory Arbiter, in order to write the received data to DataStore. Frame data is written to DataStore in blocks of 64 Bytes, called a buffer. Each buffer consists of 4 parts of 16 Bytes. The 16 Bytes parts are called quad word, in short ‘qw’. The four quad words in a buffer correspond to four banks in a memory slice. Per slice the DataStore memory can handle read or write accesses of total 64 Bytes in 11 cycles of 6ns. The relevant cycle time regarding the DataStore memory is therefore 66ns, also denoted as WindowCycle. Each buffer of frame data stored in DataStore is addressed by one or multiple Buffer Control Blocks (BCBs). A BCB holds information on the position of data within the DataStore memory and a pointer to the next BCB of the same frame. The BCBs in the BCB memory are written by the Arbiter at request of the Receiver to write frame data to DataStore. The BCB memory contains a collection of linked lists of BCBs (one linked list per frame) corresponding to the stored buffers of frames in DataStore. At writing of the last data of a certain frame to DataStore, the Receiver writes the context of the frame to the FCB memory, in the form of a Frame Control Block (FCB). Per frame present in the DataFlow system an FCB is present in the FCB memory. A
FCB holds the first BCB of that frame and the number of BCBs the frame exists of. The first BCB corresponds to
the first data of that frame in DataStore memory.

Each buffer of frame data written to DataStore is represented by one or two BCB entries in the BCB memory,
with exception of the first BCB of a frame, which is stored in the FCB of a frame. The organization of the
DataStore memory slices is on a quad word basis. Consequently read accesses to DataStore are, like write
accesses, done on a quad word basis (4 quad words in a buffer). In an access to one memory slice 4 different
quad words can be read or written in different banks of the memory slice.

For every complete frame, written to DataStore, the Receiver writes an FCB to FCB memory and enters an
address to this FCB, FCB address (FCBA), in the GQueue of the EPCController. FCBA and frame data is
dispatched to the EPC, which processes a frame in one of its internal processors. After processing of a frame,
the EPC returns frame context to DataFlow. FCBAs of frames that are finished processing in the EPC and
EPCController are sent to Buffer Acceptance, which manages frame type and priority accounting. FCBAs in
Buffer Acceptance are after processing sent to the Scheduler. After selection of a frame for transmission based
on priority, the Scheduler in its turn sends FCBAs to the TargetPortQueue in the Transmitter. FCBAs in the
TargetPortQueue represent a frame that is ready for transmission to the MPSR switch. The Transmitter will issue
read requests to the Memory Arbiter for read access to DataStore depending on the FCBAs in the
TargetPortQueue, in order to obtain the data belonging to the frames. The Memory Arbiter grants the
Transmitter requests at any time. Exception on this is the refresh cycle of the DataStore memory slices. The slices
all need two refresh cycles every N WindowCycles. Every N cycles of 66ns the DataStore DRAM’s need a
refresh period of 2 WindowsCycles. In this cycle no read/write is possible to a memory slice that is refreshing.
The Transmitter has top priority in reading to DataStore.

2.1.2 Embedded Processor Complex Controller

Fig 2.2 shows a block diagram of the internal structure of the EPCController. In the EPC Controller the
functional modules are: GQueue (manager), Request queuing logic (RQL), Message Control Block Read (MCBRead),
Message Control Block Write (MCBWrite). For every new frame that is written to DataStore, the
Receiver enters a pointer to this FCB, FCB address (FCBA), in the GQueue of the EPCController. On credit
provided by the EPC to the GQueue, the GQueue writes the FCBA to Message Control Block Read (MCBRead). MCBRead holds messages for read requests to DataStore. After receiving a FCBA from GQueue,
MCBRead will issue a read request for read to DataStore to the Request Queuing Logic (RQL). At receiving grants for read access by the Memory Arbiter, MCBRead will read the first BCB of the frame from FCB memory. If a frame involves multiple BCBS then also the BCB memory will be accessed by MCBRead. This, in order to obtain further BCBS of the frame and hence to read further data of the frame from DataStore memory. The EPC checks the status of the messages in MCBRead. If the status of the next available message in MCBRead satisfactory, the frame pointer FCBA and required frame data will be dispatched to the EPC. The EPC will then process the frame in one of its parallel processors. After processing of a frame, the frame context FCBA is passed by the EPC to Message Control Block Write (MCBWrite). MCBWrite passes a write request to RQL. At receiving write grants concerning this request from the Memory Arbiter, the RQL will issue a confirmation to MCBWrite. After this MCBWrite can do a write access to DataStore. Messages in MCBWrite, which have finished their write access to DataStore are ready for dequeue to the Transmitter. At dequeue of these messages, the corresponding FCBA's are sent to Buffer Acceptance (see fig. 2.2). Messages of MCBRead and MCBWrite in the RQL represent requests for access for read and write of these modules to the DataStore Memory. The priority of MCBRead and MCBWrite DataStore access messages are based on the age of the messages, where age means in this case the time the messages are stored in RQL.

2.2 DataFlow design flow

Fig 2.3 gives an overview of the design flow that was employed for the design of the DataFlow system. The flow is divided into three main stages by the following milestones in the diagram (indicated by the triangles): design request (DR1), system design review (SDR) together with feasibility report available (FRA) and top-level design completed (TLDC). The three stages can be described as follows:

1. System Architecture Design phase and pre-architecture design phase activities: all activities performed before the DR1 milestone belong to this phase.
2. Architecture Design Phase: all activities between the DR1 and SDR milestones.
3. Top Level Design: all activities between SDR and TLDC milestones.

2.2.1 System Architecture Design phase

The top of the flows of fig. 2.3 shows the System Architecture Design process. In this process, the overall design at system level of the Xantium IP router is created. In the Requirements Specification process, both the technical and non-technical specification of the Xantium router are made. These requirements are made at a global scale and are informal regarding detailed components as the Sanford hardware subsystem.

The outcome of this process is recorded in a set of documents. These documents are the Architecture and Requirements Document. The Architecture document describes the architecture of the Xantium system. The requirements clarify the initial product and system requirements. It explains what behavior is expected from the system without specifying how this will be achieved. Next to this, the document describes informal requirements regarding the Sanford subsystem, involving the functionality, speed and performance, it also specifies requirements concerning area (die size), pinning, power.
Architect (WAX) can issue CR to Architect (WA10)

System Architect
Responsible for complete design.

Architect (WA10) can issue CR for TRS WAX.

Architecture CR TRS CR

Fig 2.3: The DataFlow design flow
2.2.2 Architecture design phase

The Architecture Design Phase (ADP) involves the architecture design, feasibility and performance process, performed by the Architecture Design Group (WA10). Output of the ADP is the TRS/HW-TRS (Hardware-Technical Requirements Specification), the Architecture document and the Feasibility reports. The Architecture Design Process concerns the technical studies and investigations that must be performed before the TRS and HW-TRS can be created. This includes definition of the requirements on hardware level, finding optimal partitioning of the system into hardware building blocks, an identification of the most important interfaces between these building blocks, investigating technological bottlenecks and risks, and making an estimation of the most important attributes such as power consumption, area, cost, pinning, etc.

This ADP can be subdivided into three processes:

1. Non-functional design or Architecture Design + Resource Selection
2. Functional Architecture design
3. Performance Modeling and Evaluation

Architecture Design + Resource Selection

In the Non-functional and Architecture Design + Resource Selection process (see ① in fig. 2.3) the resources for the system are selected, e.g. the used type of memory the system uses. Moreover the non-functional aspects of the design as chip area, chip cost, chip production yield are considered. The created architecture of selected resources is sometimes called Logical Architecture. The logical architecture is documented in the architecture document by using diagrams and text in the architecture document.

Functional architecture design

The Functional Architecture Design process (see ② in fig. 2.3) concerns the design of a system’s functional behavior (technology independent), called Functional Architecture. The Functional Architecture design process involves the design of all functionality of the system. The design is documented by making block diagrams accompanied with textural explanation of the functionality in the architecture document.

Performance modeling and evaluation

Performance modeling and evaluation (see ③ in fig. 2.3) provides the feedback between the logical and functional architecture decisions. It is used to check performance of algorithms with a given set of hardware resources. The Performance analysis of design and algorithm alternatives is done using an executable performance model. The model is created to be able to answer critical questions concerning design alternatives. This executable performance model has all the functional properties that are needed to be able to answer particular performance questions about the system. These performance questions can for instance be: acquired throughput, memory occupation, and packet drop. Only those properties that are relevant for answering the performance questions are included in the model. As in general a performance model includes merely a part of the complete system specification, it costs less effort than creating a model with all aspects of the system included. The performance model is preferred above a full functional specification since the creation of a performance model is considered to be less labor intensive to build.

Feasibility report: The Feasibility report is the output of a technical feasibility study. The technical feasibility study focuses on the implementation feasibility of the system or a part of the system in a given development discipline: ASIC design, Hardware dependent Software design (HdS), Physical Design.

TRS/HW-TRS document: The TRS/HW-TRS document gives the technical requirements specification concerning functionality, speed and performance. Example of a TRS is for instance [VR00b], defining:

- Block diagrams of top level configuration and partitioning of functionality.
- Functionality definition of the system.
• Non-functional requirements as power, area (die size), pinning, etc.

**Architecture document:** The outcome of the Architecture Design Process is used as input for the Architecture document. The architecture document gives a detailed description of the architecture designed during the ADP.

### 2.2.3 Top level design

The architecture document and the HW-TRS/TRS documents are used as input for the designers of the Hardware dependent Software (HdS) and ASIC Design Specification. The outcome of the HdS and ASIC design specification process are top-level design (TLD) documents for both the HdS and ASIC group. These documents give a detailed specification of the hardware and software to be created.

To evaluate some specific design choices made during the ASIC/HdS design specification process implementation specific simulations are performed. The model used for these simulations is the one that was employed for performance analysis in the Architecture design phase extended with the necessary implementation specific design properties. If the detailed design specification is approved, the actual design of the DataFlow system is started.

### 2.2.4 Actors and responsibilities

The diagram of fig. 2.3 shows two important actors: the Systems Architect and the (Hardware-) Architect WA 10.

**System architect:** The System Architect is responsible for the complete design of a system. It is an actor with expertise on many areas involving the design of the Xantium IP router. As can be seen in fig. 2.3, the System Architect actor has arrows pointing to several processes and documents in the design flow. An arrow and its direction indicate the participation of the actor in the process. The direction of the arrow is not intended to indicate the flow of knowledge between the actor and the process since this can be considered as bi-directional. Next to the arrow, a textual description is given on how the actor participates in the flow. The System Architect <<performs>> the System Architecture Design task, <<approves>> the Requirement Document and <<supervises>> the Architecture design, feasibility and performance process.

**(Hardware-) architect:** The (hardware-) Architect is an actor with knowledge and experience regarding the design of hardware systems. The (hardware-) Architect has a central role in the Architecture design, feasibility and performance process (<<performs>> participation). This actor has also a role as advisor in the top-level design stage (TLD) of the design flow (<<advises>> participation).

### Back loops in the design flow

During the Architecture design phase a request for changing the requirements can be issued by one of the WA10 department architects (see (a) in fig. 2.3). This requirements change request can involve both functional and non-functional requirements. An example of these change requests is for instance the number of multiple instances of adjacent instances of the DataFlow system are supported by the DataFlow design. At disapproval of the detailed design specification an architecture design change (see (b) in fig. 2.3) or requirements change can be requested (see (c) in fig. 2.3). An architecture change request can lead to a partial reconsideration of the designed architecture.

### 2.3 Performance modeling in the Architecture Design Phase

A closer look is taken at performance modeling in the Architecture Design Phase. Input for the ADP design phase is a set of informal requirements, both technical as non-technical. The technical requirements involve definition of expected functionality, block diagrams of the global top level architecture showing the position of the system under design in its environment and non-functional requirements of the system to be designed as allowable power consumption, chip area, pinning, thermal and EMC (Electromagnetic compatibility) properties. The non-technical requirements involve e.g. chip cost and time to market.
The architecture design phase focuses on hardware system and architecture definition. The two main processes in the ADP are:

- Functional Architecture design (algorithms, behavior)
- Resource Selection & Logical Architecture design (memories, buffer space, logical blocks)

The mediator between these two processes is Performance modeling and Analysis. To evaluate whether the required functionality can be realized with the created logical architecture design, an executable performance model is created of the design known at that point, incorporating all functional and logical aspects of the design that are needed to answer performance questions. These questions involve e.g. the throughput and latency of the system, subsystems or the systems components and the occupancy of buffers and memories under influence of input data and internal behavior.

Performance modeling has known advantages regarding the supporting of the design process in the ADP (see [N100a]). Through performance modeling the bottlenecks and shortcomings in a design solution can be detected. Algorithms and architectures can be explored and assessed through rapid evaluation of algorithms and architecture alternatives. Through performance modeling algorithms (for instance developed in the research department) can be tuned to a hardware design. Dimensions, thresholds and settings of the system can be determined. Moreover, performance modeling supports rapid evaluation with little effort of system operation and system aspects of interest and it supports the learning process involving the system by modeling the system.

During its design, executable models were created of the DataFlow system for performance analysis purposes. This used modeling framework used for the modeling of DataFlow the object-oriented characteristics of the C++ language. The modeling framework and the models created with it of DataFlow have been used as starting point for the evaluation of UML regarding the modeling of hardware system.

A performance model covers only a part of a systems' logical and functional design. Only those aspects of the functional design and the selected resources (logical architecture) that are needed to create an executable model that can be used to answer the performance questions about the design are included in the model. In [TH00a] the term adequacy is used. Adequacy means that all relevant aspects of a design necessary for properly answering performance questions must be included in the performance model. Abstractions are made in order to free a model from design details that are irrelevant for the answering the performance questions. A balance is made between abstraction and adequacy in a model.

Furthermore, the performance model is an evolving model, changing with the proposed design solutions. Fig. 2.4a and b give an abstract illustration on this. At an early stage of the design, the functional design and logical architecture provide some basic conceptual solutions to the requirements given in the requirements specification (see fig. 2.4a). In the final stages of the system level design process, when design solutions have evolved to a mature design, the performance model includes more aspects of the final

![Fig. 2.4: (a) Limited performance model at early stage of design; (b) More detailed model at final stage of the design](image-url)
functional and logical architecture of the hardware implementation (see fig. 2.4b) than in the early stages of a design.

Based on discussion given in [TH00a] a performance modeling flow can be presented. The performance modeling and evaluation process involves the creation of performance models based on design solutions and requirements. Based on requirements that are input to the ADP, which are informal by character, design alternatives are created. These design alternatives are conceptual design solutions. The conceptual design solutions involve created functional architectures (algorithms, behavior) and logical architectures (selected resources as memories, logical blocks). Requirements that match the conceptual design solutions are created. These requirements have more precise character than the initial requirements provided to the ADP. Conceptual design solutions and requirements are partly based on design experience and design solutions from the past. Design solutions are recorded in the Architecture document, requirements are recorded in the TRS document. Both documents are draft documents during the ADP and are finalized at ADP end. Design solutions and requirements are described in these documents by schematic pictures and textual descriptions. The process of recording design solutions and requirements in the documents, involves a process of formulation.

Fig. 2.5 shows the modeling flow involving performance modeling. Based on created conceptual design solutions and requirements a conceptual model and performance questions can be created. The process of creating the conceptual model and performance questions is a process of formulation. From conceptual solutions a conceptual model is created. Requirements are used to formulate performance questions. Purpose of the formulation process is to create models and performance questions that can be used in discussions among designers and can support the evaluation of performance issues in further stage of the performance modeling flow. The Conceptual model is a model created by using pictures and text that merely incorporate those implementation aspects that are needed to answer the performance questions. The performance questions are also formulated in pictures and text and involve the design aspects that need to be assessed in the performance modeling process.
Note that the formulation process involving the Architecture and TRS document, created to provide a view on the design solution and requirements that is as complete as possible, has overlap with the formulation process that is used in the performance modeling flow, which focuses on the performance questions that need to be answered. Since the focus is here on performance modeling, merely the formulation process that is used in performance modeling is discussed.

The conceptual model and performance questions are used to create an executable performance model and describe formal performance properties that need to be evaluated. The process of creating the executable performance model and describing the performance properties is a process of formalization. Based on the Conceptual model, which has a rather informal character, an executable model is created using a language with defined syntax and semantics, which is used to perform performance simulations. Formal performance properties are described in formula form (formal). These properties can be latency, throughput and buffer occupation, expressed in formula form, using parameters measured in the executable model. The process of formalization for the executable performance model and the formal performance properties can be performed in parallel. A clear distinction is made between the executable model and the formal performance properties to keep the executable model free from formal descriptions (formulas) that are used to evaluate the formal performance properties. In practice this can be difficult, and an executable model needs to be extended in order to be able to analyze the performance properties.

Validation involves the questions whether the executable performance model is an acceptable model of actual system under study. The validation process concerns the question whether the formal model represents the conceptual models created in the formulation process.

The next step involves the evaluation of the formal performance properties by performing simulations using the executable performance model. The aim of the simulations is to answer the performance questions and to verify whether the performance properties satisfy the conceptual requirements. The evaluation results are the output of the performance modeling and evaluation activity. Based on these results, design decisions can be made regarding functional and logical architecture design in the ADP (involving the Architecture Design + Resource Selection process and functional architecture design process).

2.4 UML in the design flow

Based on the evaluation of the design flow of the DataFlow system, an indication is given, regarding the use of UML in the design flow.

The Unified Modeling Language (UML) is a general-purpose modeling language for specifying, visualizing, constructing and documenting the artifacts of software systems as well as so-called business systems and other non-software modeling systems (see UML standard 1.3 [UML1.3]). Artifacts are artificial or man-made objects. In the context of UML, artifacts are described as valuable pieces of information, consisting of an aim, semantics, a supporting notation and templates.

This thesis investigates the possibility of using UML for modeling in the Architecture Design Phase. Special focus is on the performance modeling and evaluation of hardware systems. Arguments for using UML in this process in the design flow of hardware systems are:

- Performance models and performance evaluation (simulation) results are important input for further stages in the design flow. UML is expected to be useful for clear documentation of the performance model and hence for communication of design decisions among designers.
- Discussion of performance models and performance simulation results among designers play an important role in the ADP. Application of UML is expected to ease these discussions.
- Performance model are based on a modeling framework, which is merely understood by model designers, rather than by whole design teams. Performance models are written in plain C++ code.
Documentation is labor intensive and models cannot easily be read and understood by designers not working on the models.

- UML modeling of hardware can link hardware and software designs and designers. UML has already application involving the modeling of software. UML models created in the ADP can both be used as input for hardware designers as software designers of the HdS (Hardware dependent Software) group in the Top Level Design phase (see fig. 2.4).

Important questions involving using UML in performance modeling is how to create UML models of hardware systems. Performance modeling requires executable models that can be used to analyze performance aspects of a system, as is discussed in section 2.4. Next to the question regarding how to model hardware systems with UML, this thesis also investigates how to create UML models at various abstraction levels. Performance models are abstract in the beginning of a design. Performance models are furthermore used to evaluate implementation specific design aspects in the Specific Design Evaluation process (see fig. 2.2). Overall, the main question is how UML can be used for performance modeling and how UML can be combined with existing methodologies used for performance modeling.
3 Hardware modeling methods

In this chapter, a closer look is taken at specific aspects concerning the modeling of hardware systems. Presented and assessed are some, existing modeling languages, methods and tools.

3.1 Introduction

In order to be able to answer the main question concerning how UML can be used for performance modeling involving hardware systems and how UML can be combined with existing methodologies used for performance modeling, specific aspects regarding the modeling of hardware systems must be considered. An investigation is made of properties of hardware systems that are relevant for performance modeling and existing methods for modeling of hardware systems are evaluated. Prior to looking into the various aspects of modeling methods involving hardware systems, it is useful to define what is meant by the term method. The following distinction can be made between a modeling method, a language, modeling heuristics and tools (see also [SG94]):

- **Method**: A method covers the process of model design, a modeling language and modeling heuristics.
- **Modeling language (notation)**: A modeling language provides the basic language to capture the system properties to be modeled. The UML language is used as modeling language in this thesis.
- **Heuristics**: Modeling heuristics are informal guidelines on how the modeling language can be used in a particular situation (rules of thumb).
- **Tool**: A tool supports the creation of models based on a modeling language. A tool can provide a modeling environment that supports the process of model design and model heuristics.

3.2 Properties of hardware systems

Hardware systems exhibit specific aspects that need to be considered.

3.2.1 Architecture and behavior

Complex hardware systems like the DataFlow system consist of modules (components) that communicate with each other through lines and busses. Hardware modules or components have ports that are used to communicate with their outside world. The components of hardware systems can be distributed, meaning that the system consists typically of a set of distributed modules that communicate through communication channels that are available between their ports. The hardware modules encompass typical aspects as behavior and data (memory). Behavior execution is a reaction on external/internal events or external/internal data and can be dependent on internal data (memory).

3.2.2 Timing

The DataFlow system is a real-time system. A critical aspect of modeling real-time systems is how the concept time is handled. Timing and delay of behavior is an essential feature of hardware systems and a modeling framework should be able to capture this and must have notion of time. In [PV97], real-time is defined as a property of a system to fulfill timing requirements for reactions on external and internal events. In this thesis the term timeliness is used. Timeliness of system behavior is behavior meeting time constraints, such as a deadline. Deadlines can be hard or soft:

**Hard real-time deadline**: the correctness of a system response depends on the fact whether response has occurred at the specified time (deadline). A late response is considered as system malfunction. An example of a hard deadline is the WindowCycle time in which the memory Arbiter has to finish its functionality.

**Soft real-time deadline**: soft timing requirements are preferred to be met, but if occasionally an event is processed late, it is usually not considered as system malfunction. On the other hand a consistently late processing of events or execution of system behavior can result in system failures.
3.2.3 Concurrency and communication
A thread is a set of operations that is executed in a sequential fashion [DO99]. Concurrency is the simultaneous execution of multiple (n≥2) parallel threads [SG94]. The concurrent threads can dynamically depend on each other in order to fulfill their individual objectives.

Synchronous versus asynchronous concurrency
In synchronous systems, the times of processing and communication of distributed modules are related. A synchronous concurrent system that is directed by a common clock has modules of which the execution of operations must be finished within a fixed time interval. A system is asynchronous if the modules in the system have no notion of their relative speed of processing and communication.

Synchronous versus asynchronous communication
Two primitive forms of interaction between threads or processes are: synchronization and communication. Synchronization involves adjusting the timing of the execution of operations in a thread based on the execution state of other threads. Communication is typically the passing of information between two parallel threads. If information exchange (communication) and synchronization through this communication occurs simultaneously, the type of communication is called synchronous communication. With asynchronous communication, information exchange does not directly imply synchronization between modules. In this case, for instance, input queues are used to buffer messages at the receiving side of the communication. Note that synchronization between asynchronous concurrent modules can be achieved through synchronous communication.

3.2.4 Locus of control
A thread can be a sequence of instructions contained in one process or task. It can also be a sequence of operations that are executed in multiple autonomous processes under influence of a control mechanism or control process. In a synchronous concurrent design, the execution of operations within modules is typically under control of a central process, e.g. a common clock provided by a clock module or a central control process. In the latter case, the central control process maintains a locus of control over various processes in the system. Fig 3.1 gives a brief explanation involving threads in a synchronous design. The shown control process is initiator of a sequence of operations in both Process 1 and Process 2 (shown as gray boxes). Initiation of either Process 1 or 2 means handing over the focus in the locus of control by the control process to the involving process. After execution finishes in Process 1 or Process 2 the focus is returned to the Central Process. The central process is in control over the sequence in which functionality in processes 1 and 2 is executed. Hardware systems as the Dataflow system can have multiple parallel threads. In essence, the internal modules of the DataFlow system can be seen as several concurrent, autonomous hardware processors. A locus of control is maintained by the Memory Arbiter through these processors and involves the Receiver, EPCController and Transmitter.

3.2.5 Synchronous design of hardware systems
The DataFlow system has multiple concurrent hardware modules, as e.g. the Receiver and EPCController. To deal with complexity, hardware designers prefer making time-synchronous designs. Designs of systems like DataFlow are therefore made synchronously concurrent. A super cycle is controlled by the memory Arbiter that makes the modules of the DataFlow system operate in synchronous concurrent fashion. The execution of

Alcatel Bell N.V. 15
operations in modules of DataFlow must be finished within the fixed cycle time of the Arbiter. The modules in the DataFlow system therefore have to meet hard real-time deadlines. The cycle time of the system is made focused on the 66ns cycles that are typically needed for accesses involving 64 bytes of data to the DataStore memory. The hardware blocks are designed to cope with this cycle time. Modules that have latencies larger than this cycle time are pipelined in the implementation.

### 3.3 Requirements of modeling methods and frameworks

In order to be a good mediator between functional design and the logical architecture under design, a performance modeling paradigm should have at least have the following features:

Requirements regarding modeling methods (see also [N100a,TH00a]):

1. **Abstraction**: a modeling method should support a wide range of abstraction levels, meaning the creation of models that omit certain details. Models should be able to represent conceptual design solutions adequately at the right abstraction level. Abstraction is needed to be free from implementation details at an early stage in the design, since these details are often not yet known. Furthermore, abstraction makes it easier to reason about a design. On the other hand, a model should adequately represent system characteristics in order to answer performance questions. Note that benefit of abstraction is improved simulation speed.

2. **Well-defined language**: Executable model should be written in a language with well-defined syntax and semantics. A language is called formal if its semantics is based on a mathematical framework.

3. **Clear modeling language that supports discussions**: performance models created in a modeling language should give a clear representation of the conceptual solutions presented in the functional design and logical architecture. Models must be clear and easy readable and support discussion among designers.

Requirements regarding modeling languages:

4. **Encapsulation and modularity**: objects encapsulate their attributes and behavior and hide them from external objects through a shell. External objects do not know the internal structure of the objects and communicate with them through well-defined interfaces. Encapsulation can involve multiple objects, sometimes referred to as nesting (see [PV97]). Nesting is used to represent system structure. It supports the creation of hierarchical models. Subsystems can be represented by a top-level representation that 'nests' the subsystems. Through hiding of attributes and behavior, systems can be divided in modules and subsystems. Modules and subsystems can be changed without changing the system (modularity).

5. **Reusability**: it should be easy to reuse created models and model components in newly created models to support easy model building.

6. **Scalability**: models should be scalable, meaning that the number of instances of model parts or objects should easily be scaled. This in order to evaluate the influence of the number of instances on system performance.

7. **Distinction between processes and data objects**: process objects model hardware modules. Behavior of hardware modules is modeled in the process objects. Data objects model internal context (state) and memory of hardware modules and data transferred between process objects.

8. **Inheritance**: classes inherit properties and behavior from a base class and form specializations of the base class. By using inheritance data and process objects can inherit properties from more general objects.

9. **Polymorphism**: objects of which the type is determined at compile created from a (abstract) base class.
Requirements regarding modeling tools:

10. **Separation between model and scheduler**: A performance-modeling tool should support the separation of the model, representing a conceptual design solution and the execution schedule used to execute the model. Furthermore, the evaluation of formal performance properties must be separated from the executable performance model. Processes that evaluate performance properties should be separated from the executable performance model.

11. **Debugging**: Tools used to build models should have good debugging capabilities to ease model building and to support validation.

12. **Maintainability and manageability**: Since models built for performance analysis are likely to change often during design, modeling tools should support easy model building and incremental modeling. It should be easy to make modifications to a model. Models must be maintainable and manageable.

3.4 Modeling languages, methodologies and tools

3.4.1 UML

UML is the product of a fusion of the modeling languages and concepts of Booch, OMT and OOSE [BR99]. UML focuses on object oriented modeling. Contributors to the UML modeling language, which has a running standard, collaborate in the Object Management Group (OMG). OMG has many members that participate in the specification of UML.

UML provides a set of general purpose modeling language concepts that is applicable in the previously mentioned system domains. More precisely, the UML standard defines the following:

- A semi-formal semantic meta-model. This meta-model provides a general modeling paradigm that can be specialized for more particular design areas. With this meta-model UML defines basic modeling concepts, like objects and classes (the UML meta-model is the model of UML itself, expressed in UML).
- UML provides graphical notation for its modeling concepts. There are 9 different diagram types provided: Use Case diagram, Class diagram, Object diagram, Activity diagram, State Chart diagram, Collaboration diagram, Message Sequence Chart diagram, Deployment diagram, Component diagram.
- UML provides rules that are expressed as formal constraints in the Object Constraint Language (OCL). The OCL can be used to specify invariants (statements that should be valid during a computation), computation pre- and post-conditions and guards.

UML is a modeling language not a methodology. It standard provides information on how to use its diagrams to visualize certain aspects of a system. It does not provide information on when to use which diagram or concept in which stage of a design flow. UML does not include a standard design process or meta-design flow. UML can, as will be shown in this and the next chapter cover the properties involving modeling languages of section 3.3.

A wide variety exists of tools that offer UML modeling capabilities. A short list of evaluated tools and their characteristics is given here:

- Rational Rose [Rose]: General UML modeling tool that focuses on the RUP method (section 3.4.7). Rose provides no particular support for modeling of real-time systems.
- I-Logix Rhapsody [Rhapsody]: UML modeling tool that incorporates the view on UML modeling presented in [DO99]. Rhapsody supports modeling of real-time software.
- Artisan [Artisan]: UML tool with similar capabilities as Rhapsody. Artisan uses additional code (e.g. C++ or Java) to create real-time software.
- Telelogic TAO [Telelogic]: UML modeling tool that combines UML with SDL as a specification language.
3.4.2 UML-RT
UML for Real-Time (UML-RT) [RS00] combines the UML 1.3 standard modeling concepts with the Real-Time Object-Oriented Modeling language (ROOM) [SG94]. UML-RT is a visual modeling language that is optimized for specifying, visualizing, documenting and the construction of event-driven and potentially distributed real-time systems. UML-RT is discussed in more detail in section 3.6. The Rational Rose RealTime (RoseRT) modeling tool supports UML-RT and RUP (section 3.4.7).

3.4.3 SHE
Software Hardware Engineering (SHE) [PV97] is a methodology developed for system level object-oriented hardware/software co-specification, modeling, analysis and design. SHE consists of a method framework and formal specification language POOSL (Parallel Object Oriented Specification Language). The method involves clear, but informal graphical representations of systems that are used in the analysis and design phase. The graphical presentations strongly support discussions among designers. The used diagrams are UML compliant [TH00b]. SHE uses the POOSL language, which is based on a formal, mathematical framework to unambiguously specify system behavior. POOSL has powerful concepts to deal with the aspects as concurrency, timing and communication. Furthermore, POOSL can be used to model hard real-time constraints. With its accompanying tools, SHESim and Rotalumis, executable POOSL models can be created for performance analysis purposes. SHE supports abstraction and POOSL is a well-defined language that supports encapsulation, modularity, reusability, scalability, inheritance, and polymorphism. SHESim and Rotalumis: incorporate the SHE methodology and supports debugging and maintainability. Model and scheduler are separated.

3.4.4 OCTOPUS
OCTOPUS [WS96] is a method intended for modeling and design of real-time software. OCTOPUS is based on the combination of the object-oriented method OMT (Object Management Technology) and Fusion. OCTOPUS/UML is the UML variant of OCTOPUS, using the UML language. OCTOPUS focuses mainly on the development of software that enables modeling soft real-time constraints. Hard deadlines are not implicitly approached. OCTOPUS/UML provides a design flow, giving heuristics on where to use what type of diagram regarding system functionality and structure. OCTOPUS/UML specifies a framework that can be used to manage modeling and design. The framework involves a pattern following which a design project is invited to configure its own design sequence. OCTOPUS uses three modeling views to visualize system behavior and architecture: Structural model, Functional model and Dynamic model. The Structural model defines the structure of the system, depicting classes and objects and their relations. The Functional model specifies operations provided in objects. The Dynamic model defines events that can work on a system and accompanying actions that are taken. OCTOPUS uses event diagrams, state charts and message sequence charts to depict the behavior. The design framework defined in OCTOPUS focuses on design of software. OCTOPUS can be used with any UML tool that supports real-time modeling.

3.4.5 BOOM
BOOM (Behavioral Object Oriented Modeling) [ME00] is a methodology that aims at software systems with soft real-time constraints. It uses UML and the object-oriented programming language Java for software prototyping of complex software systems. BOOM provides modeling heuristics. BOOM is, to the authors opinion well suited to describe software systems, but not suitable to describe a system with hard real-time constraints (like DataFlow). Concurrency is modeled using multiple threads provided by the Java virtual machine. This implies that the behavior of a created model depends on the behavior and the multitasking capabilities of the virtual machine and the used operating system. BOOM can be used with tools that support UML modeling. As BOOM does not support modeling hard-real-time deadlines, it is considered unsuitable for modeling hardware systems with such requirements.
3.4.6 MSCE
Embedded Systems CoDesign Methodology (MSCE) [CA98] specifies the description model and the heuristics designers can follow to develop hardware/software solutions. The method presents a generic (meta-) design methodology. It provides a graphical modeling framework for system level design. This framework can be used to visualize the functional design, behavior of components and the logical architecture. The graphical framework can furthermore be used to visualize the mapping of functionality on the physical architecture after hardware/software partitioning. The graphical notation used in MSCE is different than the UML diagrams, although some similarities exist. The notation covers the requirements involving modeling languages as given in section 3.3. MSCE supports abstraction and through its notation discussion among designers. The MSCE tools provide the possibility to create executable models that can be used for performance analysis purposes.

3.4.7 RUP
The Rational Unified Process (RUP) [RA00] is the result of an attempt to create a design process that can be used on a wide variety of systems that are not necessarily real-time. It incorporates UML. RUP describes the ‘what’, ‘when’ and ‘how’ concerning the design of a system (design process and heuristics). RUP is strongly focused on the design and development management of software systems. RUP gives three approaches to system design: use-case driven, architecture centric and guidance of software development. In the use-case driven approach the focus is on the functionality offered to external user(s) of the system. Architecture centric focuses on the system architecture. Guidance of software development is a software development management approach. RUP is mentioned in this document since the tools that were used (Rational Rose and RoseRT) for the UML modeling of DataFlow support RUP. The modeling process defined by RUP is not considered with the modeling of DataFlow, since RUP is not focused on hardware systems with real-time constraints. RUP is supported by the Rose and RoseRT tools (section 3.4.1 and 3.4.2).

3.4.8 ROPES
Rapid Object-Oriented Process for Embedded Systems (ROPES) is an embedded systems design method presented in [ROPES]. It provides information on the roles of developers and designers in projects and their activities, the project schedule and project artifacts. Furthermore ROPES presents a model involving the life cycle of a project. It in fact promotes a spiral form project lifecycle.
ROPES is applicable to real-time embedded systems. Most recordings of the ROPES UML modeling process [DO98] focus on the modeling of software systems that are deployed on (potentially distributed) hardware systems, rather than modeling hardware/software systems together. Although ROPES includes the design of hardware in its process, it considers the hardware (logical) architecture a requirement to its UML modeling process. ROPES promotes a separation between functional and logical architecture, in order to be able to change the logical architecture, without having to change the functional architecture. Reason for this is that a created functional architecture can then be reused and deployed on different logical architectures. ROPES is supported by the Rhapody (see section 3.4.1) UML modeling tool.

3.4.9 Discussion on the found methods
With exception of SHE and MSCE, all the presented methods focus on the modeling and design of real-time software systems. This statement is supported by [HE99, JM00]. Also, no references of previous work on the modeling of hardware systems with UML have been found. Exception on this is the SHE method, which incorporates diagrams that closely relate to UML [TH00b]. UML-RT focuses on the modeling of real-time systems. In practice, however, the tool that implements the UML-RT language is focused on modeling and development of real-time software rather than hardware. The remaining of this thesis is focused on modeling of hardware systems with the UML and UML-RT language. Furthermore, the SHE method is used to compare developed modeling heuristics involving the modeling of hardware with UML. Moreover, tools that are used for performance modeling with UML are compared with the SHESim tool.
3.5 UML diagrams

UML can be considered as a rather open modeling language. Next to the diagrams, in the UML standard some concepts are defined on what diagram to use in which view on a design. A view can be considered as a set of diagrams depicting some specific aspect of a design such as architecture and behavior. Examples of this are:

- System structure views: static view, use case view, implementation view and deployment view
- System dynamic views: state machine view, activity view, interaction view
- Model management: model management view

These views all involve one or more of the UML diagrams. These distinctions made between diagrams through views on the design of a system are rather loose. For effective use in system level design of hardware system clear guidelines are needed. Without clear guidelines, a wildwood of models are likely to emerge. After all, an important goal of using a language as UML is to obtain clear and unambiguous models. Therefore, the attempt is made to create guidelines for the modeling of hardware systems. These guidelines provide heuristics on how to model hardware systems with the UML diagrams. UML has a rich set of notations. A set of 9 diagrams is available to model a system.

3.5.1 Use Case diagram

The Use Case diagram is used to model the functionality of a system, as it is perceived by outside users. These outside users are typically called actors and can be human or for instance another (sub-) system. Fig 3.2 shows a part of the Use Case diagram that was created involving the DataFlow system. Use Case diagrams are intended to capture a broad view of the primary functionality of a system in a way that is easily understood by non-experts. The shown ovals are the use cases in the use case diagram, depicting functionality that can be used or applied to the system by the external actors. Next to the ovals the definition of the use case is given in text (e.g. “Send IP frames for streams up to 10Gbps”). The external actors are the small puppets in the diagram. The interaction between actor and use case is defined through a unidirectional or bi-directional association. Use cases are of two types in the given diagram: it can depict something that is done by the external actor to the system. An example of this is: “Send IP frames for streams up to 10Gbps”. This is indicated by the arrowhead pointing towards the use case (unidirectional association). The other type of use case is the one that offers functionality or information (e.g. data) to the actor. This is depicted with a bi-directional association. The indications between << >> indicate a so-called stereotype. Stereotypes are an extension mechanism to UML, allowing emphasizing a characteristic of model items in a diagram and creation of new

Fig. 3.2: Use Case diagram for a part of the DataFlow functionality
model elements derived from existing modeling elements provided in the UML meta-model. All modeling entities in UML can be stereotyped. The unidirectional association between EPCIngress and DataFlow has the <<realize>> stereotype. This stereotype indicates that the EPCIngress is the one to perform the functionality belonging to this use case. In other words EPCIngress does the following to the DataFlow system: “Read frame pointer and retrieve frame context”.

Other UML stereotypes are <<uses>>, <<extends>> and <<communicate>>. A <<uses>> stereotype indicates a generalization in which one use case depends on the functionality of another use case. <<extends>> indicates an extension of a use case. A <<communicate>> stereotype indicates that an association between an actor and a use case involves a transfer of data between the two. Note that in UML extra notes can be added to models for explanation purposes. These notes are linked to model items through a dotted line. Some important features of Use Case diagrams are:

- Use Case Diagrams show functionality at a high conceptual level, without showing communication interfaces or messages or the internal structure of the system. It is therefore well suited to visualize directly the specification of the system and is well suited as communication means to management.
- Use Cases themselves can be decomposed into other use cases and be used to show scenarios that show detailed sequences of object interaction.

Use Cases are mostly used to describe black-box system functionality.

Scenarios are in UML terms instances of Use Cases. Scenarios represent individual histories of system interaction with its environment or internal system operation.

### 3.5.2 Class diagram

The Class diagram is used to depict the classes in a UML model and their static relations. Classes typically have attributes and operations. The class diagram is purposed to visualize the static structure of a system being modeled, depicting all possible instances of classes in a system altogether. Class diagrams provide a generalization by giving the universal relations between classes. Fig. 3.3 gives a subsection of the class diagrams that are created for DataFlow. A class may appear in multiple Class diagrams. In a class diagrams one is not forced to define all classes, attributes or operations.

Class diagrams can depict active and passive classes. Active classes are classes that autonomously performs actions, controls the actions in passive classes or generates events or messages. Passive classes provide passive control (e.g. an interface), provide service to active classes or are data classes which represent data or data storage (memory).

The shown Class diagram exists of two parts. One part (a) shows a class diagram that is created from an existing model of the DataFlow Receiver and the other (b) a class diagram that applies to the complete DataFlow system. In a Class diagram, the classes are shown as rectangles with three compartments. These compartments show the class name, the class attributes and the class operations. Note that the association between class ‘Receiver’ and ‘PrepAreaArray’ is called an aggregate relationship. This is visualized by the diamond shape. The diamond indicates that the Receiver class ‘has one’ or ‘has multiple’ attribute(s) of the type PrepAreaArray. In this case the multiplicity of variables of type PrepAreaArray is one. This is indicated by the ‘1’ shown at the PrepAreaArray end of the aggregate relationship.
Fig. 3.3: An example of a Class diagram

The name indication next to the aggregate relationship, prefixed with a +, indicates the role the class PrepAreaArray plays in the Receiver class. In this case the PrepAreaArray class is used in Receiver through an attribute called PrepAreaArray. The diamond in the aggregate relationship comes in two appearances. The shown black diamond indicates that the PrepAreaArray class is only used in the Receiver class (non-shared aggregation relationship). An white colored diamond would indicate that the PrepAreaArray class is also used by another class. The arrowhead in the aggregate relationship means that the relation is a so-called client-server association. This means that the Receiver class (the client) has knowledge of the existence of the PrepAreaArray class, but that the PrepAreaArray class (the server) has no knowledge of the existence of the Receiver class. The PrepAreaArray cannot navigate the attributes or operations of the Receiver class. The PrepAreaArray has an aggregate relationship with the shown PrepArea class. This aggregate relationship is accompanied by the so called multiplicity on both ends of the relationship. The PrepAreaArray has a multiplicity of 1 (indicated by the ‘1’ next to the relationship), meaning that at runtime one PrepAreaArray object will exist in the relation with PrepArea. On the PrepArea class end the multiplicity is indicated with an *, indicating that the number of PrepArea objects that will exist at runtime is determined at runtime. Other possible multiplicity indications are: 0, 1, 0..1, 0..n and 1..n.

The second shown class diagram shows a general class ‘DataFlow’ and two specialized classes ‘DataFlow Ingress’ and ‘DataFlow Egress’. The relation between the general class and the specialized classes is a generalization specialization relationship. The general class has attributes and operations that are inherited by the two specialized classes. The relation is a so-called ‘is a’ relation ship. DataFlow Ingress is a DataFlow.
3.5.3 Object diagram

Object diagrams are used to show instances of classes. This means that they can be used to depict the system structure at a certain stage of system operation or can be used to depict a system structure involving a scenario. Fig. 3.4 shows a part of an object diagram created for DataFlow. The shown objects represent hardware blocks in the DataFlow design that are considered to be active objects. It is possible though to show also passive objects in an Object diagram. The shown links between the objects represent the associations that exist at the instance of the system. The names of the blocks are underlined and typically denoted as: InstanceName:ClassName. The underlining stresses that the shown block is an object. The instance name is the name the object carries and the class name is the name the object is an instance of.

Because of its powerful way of showing the structure of a system regarding its objects, it seems to be a good idea to use this diagram as a block diagram to depict the system structure of hardware systems.

3.5.4 State Chart

UML State Charts or State diagram (these terms are synonyms) combine finite state modeling (like with finite state machines) with the concepts of nested, hierarchical states and concurrency. An object whose behavior may be captured with a State Chart is said to be reactive [DO99], meaning that the object’s behavioral space is divided into disjoint and non-overlapping conditions called states. Transition from one state to another is a response to an internal or external event. State diagrams model the possible life histories of an object of a class [UML1.3]. Fig. 3.5 shows a State diagram created for the DataFlow Receiver. State Charts can have states within super states. The inner states are called sub states. Furthermore, states can be divided into multiple orthogonal regions, meaning that a state can have multiple sub-state machines that operate in parallel. In UML these regions are indicated by using a dotted line that divides a state that has multiple orthogonal regions. Fig 3.5 gives a super state called ReceiverProcess. The ReceiverProcess state has two orthogonal regions, i.e. two sub state machines, that are entered when the ReceiverProcess state is entered. The orthogonal regions are depicted here by two state machines inside the ReceiverProcess state that have the same initial transition with a ‘true’ transition condition. No dotted line was available in the used Rational Rose (see section 3.4.1) drawing tool.

This reveals a gap between the UML standard and many UML implementing tools. Although the UML standard provides possibilities to create concurrent state machines, many tools lack implementation of this. Tools that do implement concurrency (see e.g. [Rhapsody][Artisan]) base their implementation on the real-time capabilities of the used operating system or a ‘real-time’ layer on top of the operating system.
In the state diagram, some dedicated UML states are used. Some dedicated states useful for the modeling of hardware are given in Table 3.1.

**Table 3.1: Dedicated states**

<table>
<thead>
<tr>
<th>Dedicated state</th>
<th>Purpose:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state</td>
<td>The initial default state within a state context.</td>
</tr>
<tr>
<td>End state</td>
<td>The termination of a local state. When in the outermost level of context of a state machine this state indicates the destruction of the object the state belongs to.</td>
</tr>
<tr>
<td>History state</td>
<td>The initial default state if the state exists and no history exists. Stores the last active state when leaving state context. Last active state when leaving a state context becomes the default state (through the state in history). Two types of history: Shallow history: applies only to the immediate context, not to sub states or sub state machines. Deep history: applies to immediate context and all nested states and state machines.</td>
</tr>
<tr>
<td>Conditional connector</td>
<td>Is used for selection of branching transitions based on guards.</td>
</tr>
</tbody>
</table>

Transitions are taken to and from either a super state or a sub state. When a transition is taken from a super state (from ReceiverProcess), it means that the transition applies to all contained sub states. States may have both entry actions, executed on state entry, and exit actions, executed at state exit. Furthermore states may have activities that are executed as long as a state is active. Transitions are triggered by an event, which may have parameters.
For the modeling of real-time systems (and hardware systems in particular) time is an important aspect (see section 3.2). The UML standard itself leaves the implementation of timing open. Two recorded implementations of timing events are: tm(interval) [DO99, Rhapsody] and informAt(interval) [SG94, RoseRT] both are timeout events that can be used in transitions. The informAt(time) event models a time event generated at a specific moment in time. Recorded events that are commonly used in UML are:
- Signal events (asynchronous events).
- Call event (event due to the execution of an operation within an object).
- Time event (event due to the elapse of an interval of time).

The operation of the Receiver State diagram can be described as follows. At creation of the Receiver object, the transition between the initial state and the ReceiverProcess state is taken, executing the init() action (operation). The two concurrent sub state machines are then operational. At arrival of the WriteFrameInPrepArea(Frame) event the accompanying transition is taken. In this case the WriteFrameInPrepArea(Frame) event is a direct operation call (function call), with Frame as parameter. A event ArbiterReadRequests() causes a change of state (from the Idle state to ArbiterRequests). At this change of state an event is generated by the receiver, meaning that request data (Requests) is sent to the Arbiter. At reception of the ArbiterAcknowledgement(Grants) event the transition between the ArbiterRequests state and the Idle state is taken, causing the WriteToDataStore(Grants) event to be generated. Note that the order is which WriteFrameInPrepArea(Frame) and ArbiterAcknowledgement(Grants) events occur is unspecified.

At destruction of the receiver object (~ReceiverObject()) the ReceiverProcess top state is left, and the transition to the end state is taken. Notice that, in this case, the behavior of the Receiver as described in the State diagram is completely determined by external events. How internal behavior is modeled will be explained later.

### 3.5.5 Activity diagram

The Activity diagram is a variant of the State diagram that is purposed to show activities that are performed involving a certain workflow or performance of a computation [UML1.3]. Activity diagrams can describe both sequential as concurrent activities. The diagram can be used to show how an algorithm is executed in a system. Fig. 3.6 shows an Activity diagram depicting the activity flow of a short algorithm. Each activity shown

![Activity diagram](image-url)
represents in this case a single operation. The arrows between the activities represent the sequencing of sub-activities and are unconditional. The two swim lanes represent two separate threads. The horizontal bars shown represent synchronization between the threads. The lane with activities 1, 3, 4 is in operation simultaneously with activity 2. After the initialization activity, the activities in the two lanes are started.

Activity 1 is started after occurrence of Event 1. The diamond in swim lane 1 is a decision point of which the branching is protected by guards ([ok], [nok]). The lower shown horizontal bar depicting synchronization between the two lanes will only be passed after the activities in both swim lanes are finished. The section between the two synchronization bars models synchronous concurrency. After both swim lanes have finished execution, the overall behavior ends with the end activity. The end activity can represent for instance the end of an algorithm or the end of a subroutine (for instance the return statement in a C++ algorithm).

**Fig. 3.7:** Message Sequence chart concerning DataFlow

### 3.5.6 Sequence diagram

(Message) Sequence diagrams or Message Sequence charts are used to depict the interactions between objects with the focus on their order in time. Fig. 3.7 shows a sub section of a message sequence chart created regarding the DataFlow system, depicting both the external actors of the system as the DataFlow system itself. Messages are drawn in the Message Sequence Diagram as horizontal lines between the vertical lifeline of the originator object and the target object of the message. The vertical boxes drawn on the lifelines depict the so-called focus of control (FOC)[UML1.3]. The FOC or activation shows the period during which an object is performing actions. It represents both the duration of the action and the control relationship between activation and its callers. Messages can have a stereotype indicating the type of communication between the objects. Messages can also be numbered to indicate the order. The message between Framer ingress and DataFlow ‘SendFramesInStreamsUpto10Gbps’ is indicated with a event identifier ‘a’. At sending of this message the Framer has focus of control meaning it is active. DataFlow will develop activity as a consequence of the message, meaning that both object will operate concurrently and that DataFlow has FOC regarding this message temporarily. The arrow head of a conveyed message is used to indicate the type of communication. The type of communication is indicated with a UML stereotype that is iconized through a message arrowhead.
Some important stereotypes regarding messages are given in table 3.2. All modeling entities in UML can be stereotyped with the risk of creating non-portable models. Not all stereotypes have an iconic representation.

Table 3.2: Some message stereotypes

<table>
<thead>
<tr>
<th>Message stereotype</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;simple&gt;&gt;</td>
<td></td>
<td>No specific information on the communication is known.</td>
</tr>
<tr>
<td>&lt;&lt;synchronous&gt;&gt;</td>
<td></td>
<td>Sender sends synchronous message. Sender and receiver synchronize at communication.</td>
</tr>
<tr>
<td>&lt;&lt;call&gt;&gt;</td>
<td>or</td>
<td>Sender performs synchronous procedure call to receiver.</td>
</tr>
<tr>
<td>&lt;&lt;asynchronous&gt;&gt;</td>
<td></td>
<td>Sender of message does not wait for receiver to accept message.</td>
</tr>
</tbody>
</table>

3.5.7 Collaboration diagram

The Collaboration diagram is used to model, similar to the Message Sequence diagram, the objects and links that are meaningful in an interaction [UML1.3]. In the Message Sequence diagram the focus is on the subsequent (in time) messages that occur in a system (one thread of messages). The key elements in Collaboration diagrams are: objects, relationships joining the objects that exchange messages and messages with: sequence numbers, identifiers and message direction and stereotypes.

The Collaboration diagram is used to provide a view on the communication patterns among a set of objects with respect to the system structure and connection topology. In fact there are two forms in which the diagram is used: both at a level of specification, showing classifiers, association roles and messages, or at an instance level, showing objects (instances of classes), links and messages. A sub section of a collaboration diagram for the DataFlow system is shown in fig. 3.8.

![Collaboration diagram for the DataFlow system](image)

3.5.8 Component diagram

Components are static or dynamic design artifacts [UML1.3], used to represent static artifacts, e.g. a document or a file or dynamic artifacts, e.g. executable libraries or subsystems.

Often, Components are used to represent deployable units, which in the software world are e.g. databases, dynamic link libraries (dlls). Focus with Components is on reusability of design artifacts. The Component diagram is used to depict the structure and relationships among components in a design. Fig. 3.9 shows how...
components can be used to model the DataFlow system (drawn as a component) in relation to other systems. The shown components are provided with interfaces (the lollipops connected to the components), which, in UML terms, represent a coherent set of services. An interface connected to a component (through a solid line) provides the services offered by that component to the outside world. For instance, some services provided by the DataFlow Ingress component are promoted by the interface ‘EPC Interface’.

Fig. 3.9: Component diagram depicting dynamic artifacts

The EPC component uses the services offered through the EPCIngress interface. To use the services offered by the EPCIngress, the EPC uses its own interface ‘DataFlow Interface’. In the same way, services of the EPC component are used by the DataFlow component. This bi-directional uses of services is depicted by the dashed arrow in two directions between the two interfaces.

3.5.9 Deployment diagram

Deployment diagrams can be used to visualize hardware architecture on which system functionality executes. The diagram is used to represent logical system architecture. Multiple Deployment diagrams can exist. Fig. 3.10 gives a Deployment diagram showing a part of the internal architecture if the DataFlow system. The primary entities in Deployment diagrams are nodes and connections. UML defines some stereotypes of nodes: <<processor>> and <<device>> (see also fig. 3.10). Nodes can be used at a variety of abstraction levels [UML1.3]. Here the following abstraction is used: a processor is typically a hardware component capable of executing functionality that is part of the system. A device is a node (hardware component) with that does not execute functionality that is part of the system. A device can however have its own functionality. Examples of a device are for instance memory or a standalone device. A connection can represent some type of hardware

Fig. 3.10: Deployment diagram
coupling between two nodes, like a data or control bus. Connections are usually considered as bi-directional (representing e.g. a processor bus). Tasks and components are added to a node to link functionality to the physical hardware. Processor nodes can be shown containing active classes or objects, but usually contain components or packages (which both in fact contain passive and active objects).

3.5.10 Packages
Packages are a generalized grouping mechanism used to group elements into logically cohesive structures. Packages add no expressive meaning to a model. They are merely used to structure a model. The difference with components is that packages provide a generalized grouping mechanism. Where, on the other hand, components are either a static or dynamic design artifact.

3.6 UML for Real-Time

UML-RT provides a set of extensions to UML. It extensively uses the UML concept of Role modeling [UML1.3]. Role modeling is used to capture the structural communication patterns between model components, like objects (instances of classes). Role models can be seen as a more general form of an instance object model. The term object model is here for the collection of Object diagrams an UML model has. Role models provide more information than an object model since it is independent of the scenarios of a system. In fact, it is used to visualize all of the instances of a system altogether. Object models provide a view on a system that is specialized for a certain instance of the system or scenario, whereas role models combine all the instances of the system. Class models (the collection of Class diagrams) are a high level generalization, since they give the universal relationships of a set of classes to each other. Class diagrams cover all model instances and give the static structure of a system. Role models match better to the block diagrams that are usually drawn during a design, since in hardware design block diagrams are usually not scenario dependent. (Please note that role models do sustain scenarios). The diagrams showing the role models will be called Structure diagram in this document.

UML-RT adds three concepts (extensions) to UML: Capsules, Ports and Connectors [SE99]. It claims to support the modeling of real-time system aspects. Although these three concepts do not cover any timing aspects, this claim has to be interpreted in such way that capsules, ports and connectors are concepts that match aspects that are typically found in real-time (hardware) systems. The extensions of UML-RT are purposed to model system structure. A hardware design typically is composed of components that communicate through ports that are interconnected through lines and busses. Models made in UML-RT exist of components (capsules) that communicates through ports. Ports are interconnected through connectors.

Capsules are defined [RS00] as complex, possibly distributed, active objects that communicate through their message based interface ports. A port is part of the capsule and mediates the interactions of the capsule and its environment. To ports are associated with a protocol that defines the valid flow of information between connected ports of capsules. This information is represented by UML signals. By forcing capsules to communicate through ports, encapsulation of its internal behavior and data is accomplished (information hiding). Moreover, it makes the active objects (capsules) reusable. Ports act as UML interfaces (although they do not directly map to interfaces). Ports are is part of the Capsule structure and are of the type that is defined by the protocol they support (their protocol role). Ports of different capsules are interconnected through connectors, which are UML signal-based communication channels. In Class and Collaboration diagrams, connectors are presented by association roles.
Capsules can be nested. Two types of ports exist to support this: relay ports and end ports. These ports are indistinguishable from the outside of a capsule. Relay ports are ports that simply pass all signals through to capsules that are nested inside the capsule the relay port belongs to. End ports are ports that communicate messages from other capsules to the inner state machine of the capsule the end port belongs to. The internal behavior of a capsule is modeled with a State Chart. Messages received by end ports trigger the internal State Chart. Messages to send capsules can be either synchronous or asynchronous. Fig. 3.11a gives a Structure diagram of the DataFlow system in a test environment, showing Capsules, ports and connectors. Fig. 3.11b shows a State Chart belonging to the indicated Capsule. Fig. 3.11c shows the capsule class, which is a stereotype of the standard UML class. In capsule class diagrams, the ports of a capsule are listed in a labeled list compartment. UML allows to add specific named compartments [UML1.3]. Furthermore, the attributes are private and the operations are protected. A capsule class merely communicates through its ports. Each port is associated with a protocol that defines the valid signals. Ports that are connected together by a connector play complementary but compatible roles in a protocol (base protocol role versus conjugated protocol role).

Fig. 3.11: UML-RT: (a) Structure diagram showing Capsules connected through their Ports; (b) State Chart triggered through messages received by its capsules’ ports; (c) Stereotyped Capsule class.

Capsule behavior is modeled through State Charts of which transition are triggered by external events, i.e. reception of a message on a particular port. Transitions taken after an event have run-to-completion semantics. Note: fig. 3.11a shows a relay port with cardinality > 1, meaning that this port has multiple instances.
3.7 UML Tools

In order to be able to use UML effectively good tools are needed. A brief comparison of tools is given here:

- **Microsoft Visio 2000** [Visio] offers a rich toolset to create UML models.
  
  **Model consistency:** consistency check involves the classes and their associations. Consistency and syntax are checked involving messages, operations and attributes defined in existing classes and used in other diagrams.

- **Rational Rose** [Rose] is purely a visual modeling tool that supports the creation of models, using plain UML. The tool strongly focuses on RUP. Disadvantage of the tool it provides no explicit way of creating active classes. Threads must be created in the implementation language (e.g. C++, Java) that is needed to create executable models. Rose provides no particular support for modeling of real-time systems. Rose supports both synchronous as asynchronous UML messages. In the implementation (Rose generates class structures in the implementation language) UML messages are implemented as simple procedure calls. Rose provides little code generation based on the visual models.
  
  **Model consistency:** Rose can be used to perform consistency checks involving created models. Consistency checked involves associations between classes in all UML diagrams. Consistency and syntax checking involving behavior and scenarios defined in Collaboration diagrams, State Charts and Message Sequence Charts merely involves the checkup whether used messages and operations and attributes are defined in class diagrams.

- **I-Logix Rhapsody** [Rhapsody] focuses on creating real-time software. Rhapsody has knowledge of passive and active (concurrent) UML classes. Messages can be both synchronously as asynchronously in the visual models, but in the implementation, messages are procedure calls. Rhapsody provides code generation based on the visual models and additional code added in the environment. Implementation code is based on State Charts (see chapter 4).
  
  **Model consistency:** Model consistency checks with Rhapsody involve class consistency in all diagrams. Behavior defined in State Charts can merely be checked through compilation and execution of created executable models. Consistency check involving behavior described in Message Sequence diagrams and Collaboration diagrams, merely involves the checkup whether messages used in these diagrams exist in the classes.

- **Artisan** [Artisan] allows the creation of passive and active classes. The implementation of the classes is based on the implementation language. Scheduling of active classes is based on the used implementation language and operating system.
  
  **Model consistency:** Artisan has UML syntax check that is similar to Rose.

- **Rational RoseRT** supports UML-RT. It allows the creation of executable UML-RT models, without adding much implementation code. This has consequences for the UML diagrams that can be drawn in RoseRT. RoseRT focuses strongly on object-oriented design (OOD) and code generation, rather than object-oriented analysis (OOA). With OOA, one is interested in drawing class diagrams, showing classes, passive and active, their attributes, cardinality and relationships in a common notation. In RoseRT all active objects must be capsules. Capsules in RoseRT implicitly carry a lot of design/implementation baggage that supports the creation of executable models. A capsule cannot have a generic association to other capsules on the class diagram, since the association that is allowed between capsules is the communication through ports and connectors. RoseRT merely allows aggregation and/or composition between capsules. With models created in RoseRT one is obliged to decide which classes are active or passive. When changing classes from passive to active all relationships to other classes are lost. Next to generic associations between capsules in class diagrams, it is also impossible to depict the associations and messages in collaboration diagrams that are transferred between capsules.

**Model consistency:** Due to its design-oriented character and capabilities to create executable models, RoseRT can offer a better consistency check than the other evaluated tools. RoseRT does a complete pass over the model looking for model inconsistencies and unresolved references involving
capsules, classes, messages, etc. Behavior described in State-Charts and Message Sequence diagrams can be checked with respect to used messages, signals, protocols and ports. Compilation and execution of models is performed in the RoseRT environment.
4 Modeling hardware systems with UML

In this chapter modeling heuristics and a modeling framework incorporating UML that can be used for the modeling of hardware is presented. The attempt is made to combine existing hardware performance-modeling methods and heuristics with an UML modeling framework.

4.1 UML and hardware performance modeling

From the previous chapter is known that UML is a combination of visual languages and attempts to address the modeling of a wide variety of systems. The notation provides a rich set of conceptual diagrams that are not necessarily orthogonal. Various diagrams can be used to depict the same aspects of a system. An example of this is that an Object diagram can be used to visualize system structure and the Collaboration diagram that essentially can be used to depict the same system aspect. The UML set of nine diagrams is large. For the modeling of hardware systems it is therefore desired to find a coherent subset of UML diagrams. Furthermore, it is not clear which diagram must be used to during the design of a system. UML provides merely a notation rather than a modeling framework or a set of guidelines to model systems. The UML modeling language doesn’t detail any modeling or design process. Therefore, guidelines are needed to create clear and consistent models.

UML provides syntax, but is semantically weak. UML is a visual modeling language and not intended as a visual programming language in the sense of having all necessary visual and semantic support to replace a programming language. Code generation from UML models is with some of the evaluated tools possible by adding object-oriented code. It is clear that for the modeling of hardware systems is it therefore desired to develop heuristics in order to enable the creation of executable UML models, which can be used for performance analysis.

A hardware design starts in the ADP with ideas that are drawn in block diagrams that are accompanied by text. Requirements are in this phase of the design flow specified and involve conceptual block diagrams, accompanied by requirements written in text. To enable the use of UML for performance modeling in this phase of the design flow, three questions have to be answered regarding UML in the performance modeling of hardware:

- How can the (conceptual-) requirements be captured with UML?
- How can UML be used to model conceptual design solutions involving performance modeling?
- How can UML be used to create executable performance models?

4.2 Specific hardware modeling challenges

4.2.1 Modeling system structure

Class diagrams are a powerful way of showing the static structure of a system. The UML standard distinguishes active and passive classes. Active classes are those that control or execute actions, produce data or provide interfacing capabilities to the environment of a system. For the modeling of hardware systems active classes are of importance since they can be used to model the hardware blocks (components) in a design. Active objects form the roots of threads and invoke the services (behavior) of passive objects. Passive classes typically supply behavior and data storage to other objects. Passive objects can be used to model data in a hardware model.

Classes are in most UML tools (see e.g. [Rhapsey][Rose]) implemented in an object-oriented language using e.g. Java or C++. Messages are in most UML tools implemented as procedures. This implies that communication through messages between classes are implemented using remote procedure calls (RPCs), which essentially means synchronization between objects and invocation of behavior of one object by another. Decoupling of operations and messages is usually unavailable in these tools. The UML standard does however specify the use of so-called signals. Signals are messages that can be communicated between objects without directly calling procedures in objects. This type of communicating is usually used with active objects. Objects
themselves are responsible for the handling of received signals (signals can of course lead to procedure calls). There are some tools that support signals in this manner [Artisan, Rhapsody]. Since the concept of signals is not directly supported by sequential object-oriented languages as C++ and Java, these tools provide their own scheduling implementation to handle signals (such as behavior execution scheduling). The issue concerning how signals must be handled and how active objects should interface is still subject of discussion for the expected UML2.0 standard. This aspect of UML is important though for the modeling of hardware systems as will be shown later.

Fig. 4.1 shows a sub section of the class diagram created involving DataFlow. Two external actors associated with the system are shown. The communication between these actors is achieved through interfaces, which provide a coherent set of services. Communication between the classes of the system itself is done through the association between the classes. Although UML provides the possibility of using messages with signal stereotype, in reality the implementation of communication will in most cases attained through RPCs. The messages shown in the operator box in the classes of fig. 4.1 are without exception operations. Meaning that every message between classes is a synchronous RPC. Interfaces can be used to provide services in order to implement the concepts of asynchronous communication. Please note that the need of using interfaces in order to model the various communication concepts is not a shortcoming of UML, but merely an UML implementation issue. The used tool Rational Rose does not implement asynchronous messages and has no knowledge of active classes. As mentioned, exceptions to this are e.g. [Rhapsody, Artisan] which both implement RPC and signals and use them intermixed. Even if messages with signal stereotype are used, there is still no clear distinction between the internal operations of a class and the interfaces to its environment. The distinction can be made through using public and private member operations of a class (see e.g. class Receiver which has public and private member operations), making internal behavior private and external messages public. A better solution would be however, providing every active class with interfaces in order to encapsulate its internal behavior. This can be done by using the UML-RT extensions. By using clear interfaces with active

---

**Fig. 4.1: Class diagram showing relations among external actors through interfaces**

Alcatel Bell N.V.
objects that are connected through channels both encapsulation, modularity as distribution are better supported. The UML interfaces stereotypes are less adequate to model this since they would lead to large, unreadable diagrams. Therefore the concepts of UML-RT are adopted. Tools implement UML communication concepts and messages in different ways. To adopt the UML-RT communication concepts is merely a choice. Despite the drawbacks discussed in section 3.7, RoseRT [RoseRT] is used for performance modeling, since the extensions of UML-RT are found suitable to model hardware systems. Diagrams that cannot be drawn with RoseRT are drawn with Rose [Rose].

4.2.2 System timing and behavior
State diagrams provide a static view on the entire state space of a system. What a State diagram doesn’t show are typical paths through the state space as the system is in operation. Therefore State diagrams are not suitable to depict system scenarios. State diagrams are used in many tools (see e.g. [RoseRT, Rhapsody, Artisan]) to describe object behavior and support the generation of executable models. Events are in some UML implementations modeled as UML signals [RoseRT, Rhapsody], other implementations of use procedure calls (RPCs) to implement events. As mentioned, important aspect in hardware modeling are parallelism and concurrency. Tools that implement the concept of orthogonal regions (e.g. [Rhapsody]) implement this by the grace of the used operating system that is obliged to provide the needed real-time scheduling.

State Charts in theory
Object (hardware component) behavior can be modeled using UML State Charts. Fig. 4.2 gives a State Chart that created to model the behavior of the EPC created with the I-Logix Rhapsody real-time UML tool. The behavior of the EPC is described in the procedure EPCProcess(). This duration of the EPC behavior (algorithm) is modeled with a time interval ‘EPCtime’ seconds. State Charts are based on the synchronous reactive paradigm [DO99]: at time t a flow of data and/or events arrives, all operations are started and must end before time t+1, i.e. when a new flow of data or events will arrive. The theory is based on the hypothesis that the reaction of a system is instantaneous (In fact sufficiently fast to fit in the time steps and much faster then its environment). The synchronous system has a unique clock that sends at each time t a work-event to all the objects which can then exchange data or events and execute local operations. The execution of operations of the objects must end before the next t+1. Transitions are modeled in State Chart as taking approximately zero time to execute (classical state machines assume zero time transitions). State machine execution proceeds in discrete time units called run to completion (RTC) model steps. An RTC step is the period of time in which events are accepted and acted upon. The processing of an event always has to complete within a single model step, including exiting the source state, executing any associated actions and entering the target state. Assumption is made that the systems is much faster than its environment, meaning that the RTC time steps are much shorter than the arrival times of external data or events. The required behavior regarding modeling based

Fig. 4.2: (a) EPC behavior modeled through State diagram created with I-Logix Rhapsody; Timing diagram of modeled EPC behavior: (b) Required: zero algorithm execution time, delay behavior modeled with abstract time interval; (c) Scenario 1: execution scenario based on C++ implementation.
on the synchronous hypothesis is given in fig. 4.2b: behavior takes no time to execute compared to arrival times of data and events (EPCProcess() is executed in zero time); modeled delay is exactly executed and can be abstracted from real-time. This means that delay in a model is not related to real execution time. With behavior that is executed in zero model time and the ability to model delay exactly in model time, behavior can be modeled exactly (see e.g. [DO99]).

### State Charts in practice

The major difference between the required behavior of State Charts and the implementation offered by available tools is that execution of behavior is not performed in zero time and that delay modeled is directly related to real-time (no abstract time). This means that the assumption made that behavior is executed in zero time in relation to arrival times of data or events is invalid. Since the arrival time of frames to the DataFlow system is within range of the cycle time of the system this leads to the following problem:

- Execution of EPCProcess() takes no zero time. Delay of behavior is dependent on algorithm execution time and operating system behavior. The length (in time) of RTC steps is determined by the execution time of algorithms and can vary during model execution.
- Delay is modeled in real-time. No abstract model time is available. A direct relation exists between the modeled delay and the offered real-time capabilities of the used operating system. Modeled delays can be longer due to non-real-time scheduling of the operating system.

Since timing of modeled delay and execution of algorithms is directly related and dependent of the operating system, it is difficult to model both behavior and delay of a hardware system exactly. A possible scenario of the reality is given in fig. 4.2c (Scenario1), based on a C++ implementation. The entry of states takes time, indicated in fig. 4.2c as 'State initiation period'. After the init() operation has finished executing in non-zero time, the timeout delay timer tm(EPCtime) is started. After this timer fires, the EPCProcess() algorithm is executed on the transition from state 'idle' to state 'EPCProcess'. Execution of EPCProcess() takes an arbitrary amount of time. When the algorithm is executed the State Chart changes state from EPCProcess to idle state directly, since in UML transitions without specification are considered null transitions which are always immediately taken at entry of its accompanying state.

To overcome the problem of long execution times of algorithms, long algorithms can be broken down into larger State Charts with possibly hierarchical states. This way the cycle time and the length of RTC model steps can be decreased to create steps that have more equal length and improve the needed difference in range between the frame arrival time and the system cycle time (RTC steps). This is considered bad practice and therefore StateCharts are not explicitly used to model behavior.

As is shown in chapter 5, State Charts don’t match the modeling methods used for performance modeling of hardware systems.

### 4.3 Developed performance modeling framework and heuristics

Based on UML models created of the DataFlow hardware system, a performance modeling framework and heuristics are created. The created modeling framework is presented in fig. 4.3. Fig. 4.3 shows the same processes that are presented in fig. 2.4 in chapter 2 of the System architecture design and Architecture design phase and their relations. The processes are drawn as gray boxes.

- From the System architecture design phase: System Architecture Design and Requirements specification process.

Next to the processes, fig. 4.3 shows the artifacts from the two involved phases of the design flow: the Architecture document and the HW-TS/TRS document. In fig. 4.3 these artifacts are drawn as white boxes. Note that the performance model is considered an artifact in the design process.
**System Architecture Design:** The System Architecture Design process involves the design of the overall system of which the hardware system (e.g. DataFlow) is a sub system. In the case of DataFlow, it involves the Architectural design of the Xantium IP router and at a lower architectural level, the architectural design of the system the Sanford chipset is used in. This System Architecture Design activity is, as described in chapter 2, the initiator of the design of the Sanford chipset and its subsystem the DataFlow chip. Requirements involving the system architecture designed in the System Architecture design process are created in a process, here called Requirements Specification. Outcomes of this process is the requirements specification which describes both technical and non-technical requirements of the design of the Sanford chipset. These requirements are input to the processes of the ADP (indicated by the arrows between Requirements Specification and the ADP).
Subsystem Technical Requirements Specification: The Subsystem Technical Requirements Specification process involves the specification of technical requirements concerning the hardware subsystem (DataFlow). These requirements are recorded in the TRS document. The Subsystem Technical Requirements Specification process is performed in close relation with the Functional design and Architecture design process (indicated by the arrows between the processes).

Functional design and Architecture design + resource selection: The Functional design and Architecture design + resource selection processes together represent the actual design process of the hardware subsystem. The processes use requirements and operate in close relation with the Technical Requirements Specification process. The two activities interact in order to create a design that covers the required functionality with a certain physical architecture. Furthermore, these processes are influenced by the technical and non-technical requirements specification created in the System Architecture design phase. The Functional design and Architecture design processes can issue requirements change requests to the Requirements specification process of the System design phase.

The Functional design activity involves the design of the Functional architecture a subsystem. The Architecture design + Resource Selection activity involves the design of the Logical architecture. Functional and Logical architecture are here considered artifacts in the ADP.

Performance modeling: The Performance Modeling process covers aspects that are considered in both the both Functional design and Architecture design process and a performance model covers aspects of the Functional and Logical architecture. The Performance Modeling process provides feedback between both design processes through performance analysis. Furthermore, performance results generated by the Performance Modeling process and created performance models (with UML) are documented and used in later stages of the design flow.

4.4 Requirements modeling

The requirements provided in the Technical Requirements Specification of a subsystem like DataFlow are provided can be described as follows:

- System top-level specification: describing the position of the subsystem within a designed architecture and defining the interactions of the subsystem with its environment.
- Subsystem requirements: describing the desired functionality and technical requirements regarding the subsystem that has to be designed.

UML can cover merely functional, behavioral and structural aspects of a system. Other technical requirements have to be captured in text. UML requirements modeling is done based on the distinction between Structural, Functional and Dynamical aspects of requirements (see fig. 4.3). From the OCTOPUS method (see section 3.4.4) the distinction between structural aspects and behavioral aspects of a design is adopted. These distinctions are made to categorize aspects of a system design. They do not represent separate models, but make it simpler to discuss and document a design. The three models created involving requirements are: ‘Functional model’, ‘Dynamic model’ and ‘Structural model’.

4.4.1 Functional model

The Functional model involves the functional requirements and aspects that are offered to the systems environment in general. It covers the overall system functionality (independent of scenarios). Use Case diagrams are used to depict this functionality.

Use Case diagram: The Use Case diagram is well suited to capture system functionality that is specified during the requirements specification phase of a design. In the TRS document (see e.g. [VR00b]) functional requirements are typically given through text. These requirements can be defined using the Use Case diagram. However accompanying text will in most designs be needed in addition to the diagrams. The Use Case
Diagram cannot be used to capture the non-functional and non-technical requirements of a system. To capture these requirements text remains essential.

Functionality that is mode of operation dependent is grouped in separate packages (e.g. ingress, egress mode of operation of DataFlow). In that case Use Case diagrams are used to cover functionality that is offered in certain system scenarios (functionality that is system mode).

**Context diagram:** Context diagrams are used to depict a system in its context. Context diagrams [DO99] are analogous to Collaboration diagrams and are used to show the system under design in its real environment and identify the input/output messages of the system. The diagram shows the system as one object and its external actors with actor stereotype. The associations between the system under design and the external actors are defined and the messages belonging to the association are listed. Closely related to the Use Cases are the messages shown in the Context diagram. Since DataFlow is a subsystem of the Sanford chipset and a subsystem in the Xantium architecture, the interaction between DataFlow and its environment are known from the System Architecture design phase. The messages involving these interactions are shown in the Context diagram.

### 4.4.2 Dynamic model

The Dynamic model covers functional requirements and aspects that are dynamic (typically scenario dependent). Message Sequence Charts are used to depict interaction scenarios between a system and its environment.

**Message Sequence diagram:** The Message Sequence diagram is considered as an important diagram in the design of hardware systems. Scenarios of a system can be depicted, such as the flow of a packet through a system. In Message Sequence diagrams, information can be given on events, timing constraints, state marks and sub processes. Message Sequence diagrams are used to show the sequence of the interaction messages provided in the Context diagram. Furthermore requirements are captured regarding timing and events.

### 4.4.3 Structural model

The Structural model provides a view on the physical position of a hardware subsystem within its environment with focus on the architectural structure. Deployment diagrams are used.

**Deployment diagram:** Deployment diagrams are used to depict the position of the system that is under design within the hardware architecture. Deployment diagrams can be used to visualize the physical hardware architecture. The nodes in the Deployment diagram depict the hardware systems in the environment of the sub-system. This can e.g. involve the other chips in the chipset of which the subsystem is part. The links between the nodes in the Deployment diagram are the busses and lines between the hardware sub-system and its environment.

### 4.5 Performance Modeling

Regarding performance modeling the distinction is made between functional/dynamic and structural aspects of conceptual design solution that needs to be modeled (see chapter 2). The performance model exists of a Functional/Dynamic model and Structural model and covers aspects of the Functional and Logical architectures created in the main processes of the ADP (Functional design and Architecture design + Resource selection). As mentioned in chapter 2, merely those aspects of the Functional and Logical architecture are covered that are relevant for answering the considered performance questions. The justification for making a distinction between functional aspects and structural aspects of a design is similar to the one given in the previous section. The Performance modeling block in fig. 4.3 shows the various UML diagrams that are created to cover a design. Although a split is made between Functional/Dynamic and Structural model, the diagrams represent the conceptual design solution and the executable performance model coherently. The Performance model involves a coherent and consistent set of diagrams.
The relations, indicated by the arrows, between the diagrams shown in the Functional/Dynamic represent the information exchange involving used objects and messages. In the Structural model it involves the structure of the system concerning Functional and Logical architecture. Note that the arrows between diagrams are not intended as a modeling sequence. It merely provides a framework of diagrams and their relations.

Furthermore, the executable performance model is given. As mentioned in chapter 2, this performance model is the mediator between the main design activities in the ADP. The executable performance model is completely covered by the UML model of the design. The executable performance model is created with help of the various UML diagrams and provides feedback to the UML model. Moreover it provides feedback on the performance of the design between the Functional and Logical architecture through performance analysis. The performance model covers both behavioral as structural aspects of the design.

The Functional design and Architecture design + Resource Selection processes create a running Architecture document. Created UML diagrams of the design can be used to document the design in the Architecture document. Next to the created diagrams, textual explanation is needed.

Algorithms that are provided in the Requirement Specification are used in the executable performance models. The performance model implements behavior specified through Collaboration diagrams and Message Sequence Charts and algorithms in C++.

Results from performance simulation are documented. The UML model created of the design can be used in the Architecture document. Next to the created diagrams, explanations in text must be provided.

The UML models are created to support the design with respect to the performance model and cover merely those aspects that are needed to do so. However, the UML diagrams could also be used to cover the design as fully as needed to be able to correctly document it. The performance model would in that case implement a subset of the UML diagrams.

4.5.1 Structural model
A design usually starts with the drawing of block diagrams and discussion. Based on the requirements given and experience from previous design conceptual design solutions are defined and recorded using block diagrams. These block diagrams can be created by using an Object (UML) or Structure diagram (UML-RT).

The Deployment diagram depicts the physical (actual hardware blocks) that are defined in the Architecture design + Resource selection process. This strongly relates to the Structure diagrams drawn. The Structure diagrams covers the structure in the Functional architecture of a design, whereas the Deployment diagram covers the structure of the Logical architecture. The relation between these diagrams links up the designs made in both design processes in the ADP.

The following gives an overview of the proposed usage of UML diagrams regarding system structure and architecture.

Object diagram and Structure diagram: Object diagrams are used to depict systems Functional architecture at an instance and are therefore used as type of block diagram. The choice is made to use the Object diagram to solely depict active objects. This means that only objects that are capable of executing own behavior based on external or internal events are shown. Data objects are not drawn in the Objects diagram. In hardware terms, hardware processes in a chip, memory etc. will be shown in Object diagrams. The active objects in the Object diagram are assumed to be concurrent. The Structure diagrams are input to the executable performance model that is created during the design to check design alternatives in logical architecture with the functional design and vice versa. The Structure of Object diagram depicts objects (blocks) and their mutual relations, showing system structure at run time. The shown blocks can represent e.g. the VHDL components in a hardware design, but also functionality that is common to hardware components in a design.
like e.g. an initialization block. With this latter example, the initialization will eventually be implemented in a hardware block, possibly as a separate process or thread.

**Deployment diagram:** Deployment diagrams are created to depict the logical architecture (selected resources) of the design. In contradiction to many software development flows [DO99, WS96], the deployment diagram is used earlier in the modeling process. It is used to depict the relation between the created performance model, which in fact depicts a part of the Functional architecture, and the Logical architecture. Deployment diagram represents the physical structure or Logical architecture of a system. In hardware design the diagram can be used to depict the systems components as hardware processors, memories etc. The instance structure, represented by the Objects will eventually have to be mapped on the hardware structure presented by the Deployment diagram. This is possible by packing objects together in UML components, provide the Components with interfaces and deploy these components onto the nodes of a Deployment diagram. This seems too farfetched for modeling hardware. A direct mapping of the Object diagram to Deployment diagram is used.

**Class diagram:** In order to create executable performance models, behavior and algorithms are defined in procedures. Furthermore, data objects and attributes belonging to classes are defined. Class diagrams are used to depict both active as passive classes and show their associations. Furthermore, association classes, interfaces and external actors are shown in Class diagrams. Class diagrams represent the static structure of the performance model.

**Component diagram:** The Component diagram can be used to model hardware components. UML Components can have clear interfaces through which encapsulation, modularity and reusability is accomplished. However, component diagrams should be used to depict model organization rather than modeling a system itself. Meaning that models or part of models should be packed into Components that can be reused in other models. Component diagrams are not used to model the components of hardware system, but to organize models.

Based on the requirements the Functional and Logical architecture are created. The structure of these designs is input to the Structural model of the performance model. Requirements modeled with UML involve system structure of the top level system. Information regarding structure is input to the Structural performance model.

The nodes in the Deployment diagram are represented by active objects in the Object (Structure) diagram. The links between the nodes represent lines and busses, which are essentially channels through which the nodes communicate. The Structure represented by the Deployment diagram is used to create the Object (Structure) diagram. The capsules (active objects) in the Structure diagrams are interconnected through ports and connectors that represent channels that interconnect the capsules. Protocols are in UML-RT created to map messages that can be transferred between capsules onto these channels (UML-RT ports support these protocols). The links defined in the Deployment diagram are used to create the same channel structure in the Structure diagram (channels in the Structure diagram match the channels in the Deployment diagram). Based on the designs created with Objects (Structure) and Deployment diagrams the Class diagrams are created in order to build the executable performance model. The Class diagram lists the attributes, operations, ports and messages (signals) that are used in the executable performance model.

4.5.2 Functional/dynamic model

The functional and dynamic model involves the creation of Collaboration diagrams and Message Sequence Charts:

**Collaboration diagram:** Collaboration diagrams are capable of showing the same aspects of a system as Message Sequence diagrams in a different way. The diagram can be useful to show interaction involving a scenario in hardware systems with focus on collaboration in the system structure. A disadvantage of the
Collaboration diagram is that if the structure of collaboration changes (which is likely to happen during design), it requires in the evaluate tools more work to maintain the diagrams than e.g. Message Sequence charts. Collaboration diagrams are used to depict system the individual messages and the collaboration between the active objects defined in the Structure (Object) diagrams, without specifying channels between the objects. The message numbering is not used in the diagrams since the sequence of messages is depicted in created Message Sequence diagrams. Focus is on the type of messages (type of communication) that is employed between existing objects (synchronous, asynchronous, etc.) and on the messages with their parameters. This type of communication depicted in the Collaboration diagram has to be retraceable in the executable performance model.

The Collaboration diagram uses information from the Use Case and Context diagram in the Requirements model. It depicts how a use case is realized by the design (performance model) in a scenario.

(Message) Sequence Diagram: Message Sequence diagrams are used to depict how use cases are realized in certain scenarios of collaborating objects in the design. Aspects as timing constraints, message order and event succession are defined in Message Sequence Charts. They are also used as feedback between the executable model and design made on paper (model validation). Message sequence diagrams can be generated during execution of the executable performance model and compared to Message Sequence diagrams depicting required behavior (diagrams made on paper).

Collaboration diagrams and Message Sequence Charts depict the similar message scenarios. Here the focus in Collaboration diagrams is on the availability and type of messages that are transferred. The sequence of messages is shown in Message Sequence diagrams.

Activity diagram: Activity diagrams are powerful way of describing how activities are executed in e.g. a hardware architecture. For instance, the execution of calculations in an algorithm performed by a pipeline with multiple hardware blocks can be visualized. The activities performed in multiple parallel operating hardware blocks can be visualized with 'swim lanes'. Activity diagrams are not implicitly used, although they can be applied to depict algorithm execution by various subsystems in the design (especially at parallel execution of algorithm operations).

4.6 Performance modeling with developed modeling framework

In chapter 2, a performance modeling flow is presented. The performance modeling framework of fig. 4.3 can be used in the performance modeling process. Two models were created to demonstrate the performance modeling framework in two different phases of the ADP: a high level model of DataFlow (phase 1) and an implementation oriented model of the EPCController (phase 2). The models were created in Rational Rose and RoseRT and use the execution scheme presented in chapter 5. Appendix B provides the UML models created (figures with description fig. B.x).

The models created are presented at three hierarchical levels [NJ99]: System level, intermediate level and module level. Fig 4.4 gives the three hierarchical levels in a design according to this view. Notice that this way of structuring a model, does not imply a modeling flow. The scope of each hierarchical level and the used diagrams that are used on each level are also given in fig. 4.4.
### Fig. 4.4: Three hierarchical levels of performance models

#### 4.6.1 Requirements modeling

In fig. 4.1 this phase involves the Subsystem Technical Requirements Specification activity. In this activity the position regarding functionality and structure of the system to be designed within its environment is defined. The context of a system within its environment is defined with Use cases and scenarios. Externally visible interfaces and messages, events and actions are defined with Context diagrams and Message Sequence diagrams. The system under design is treated as a black box and shown as a node in a Deployment diagram. The diagrams created regarding this activity are all created at a System level scope on the system (see fig 4.16).

Functional requirements as described in TRS documents (see e.g. [VR00b]) are modeled with UML. See table 4.1 for the created diagrams. The modeled requirements given here are based on finalized TRS/HW-TRS documents.

<table>
<thead>
<tr>
<th>Table 4.1:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structural model:</strong></td>
</tr>
<tr>
<td><strong>Functional model:</strong></td>
</tr>
</tbody>
</table>
Dynamic model: Dynamic model is created to visualize system functionality scenario dependent. Message Sequence Charts depict instances of the Use Cases in the Functional model as a packet walk through the top-level system. Timing, latency etc. can be modeled. Fig. B.6 shows a packet walk through the system.

4.6.2 Formulation of design solutions

Formulation involves the creation of models of conceptual design solutions. The result of formulation is a description of design solutions made using UML diagrams accompanied by text. The purpose is to support discussion regarding the design solutions. With Deployment diagrams the actual (current) design solution (the selected hardware resources) is depicted. Structure of the system is identified by creation of Object diagrams or UML-RT Structure diagrams. Class diagrams can be used to depict system/subsystem structure, using aggregation relationships. With Collaboration diagrams (without sequence numbering), the inter-object behavior through simple UML messages between objects is depicted. Formulation is mainly performed at System level and Intermediate level.

Phase1:
The first phase (early stage in the design) the focus is on DataFlow Receiver operation. Models are created to evaluate the Receiver bandwidth to DataStore memory:

- The Object diagram (fig. B.7) shows the formulation of the design solution created in the first phase. Model analysis involves the identification of objects.
- Deployment diagram (fig. B.8) shows the current design that has to be evaluated. Main design aspect playing a major role in the performance model is the DataStore memory, for which 66ns DDR ram is chosen. Other design aspects are still rather loose.
- The Collaboration diagram (fig. B.9) depicting the "<<simple>>" messages exchange between the objects in the design (essentially the data exchange between the objects). Collaboration diagram defines messages based on ideas triggered by the requirements specification.
- Class diagrams (fig. F.80) depict structure regarding part to whole relationships. Furthermore, the Class diagrams depict the static associations between classes known at that point.

Phase2:
The second phase the interest is aimed at the algorithms and architectural solutions regarding the hardware modules. Diagrams created: Object diagram (fig. B.15), Deployment diagram (fig. B.16), Class diagram (fig. B.17), Collaboration diagram (fig. B.18).

4.6.3 Formalization of design solutions

Formalization involves the creation of executable models, aid of conceptual the models created in the formulation step that can be used to attain answers to performance questions. Performance models are described using the diagrams at intermediate level and an object-oriented language (e.g. C++) at module level. At system level, the performance model consists mainly of the environment modeled as a test-bench, basically including a traffic generator and -analyzer, and the system under investigation. Diagrams of the formulated model are used. Class diagrams are created to support the creation of the executable model. Collaboration diagrams are extended by identifying the type and parameters of messages. Message Sequence diagrams support the validation of model behavior and expected behavior.

Phase1:
In this case the Receiver bandwidth to DataStore is of interest.

- The Structure diagram. Fig. B.11 depicts the structure of the executable performance model at System level. Fig. B.12 gives the system structure at intermediate level. Information used to construct these diagrams is based on the models created in the formulation step. Abstractions made regarding time and data are incorporated in the model. The execution schedule is provided by the clocking module as presented in chapter S (fig. B.11).

44
• Collaboration diagram (Fig. B.13) provides the Collaboration diagram depicting the messages conveyed between the capsules in the model. Note that the type of the messages is specified.

Pase2: In phase 2, the total operation and the EPCController in particular is of interests. Created diagrams are:
- Structure diagram (fig. B.19), Class diagrams (fig. B.20), Collaboration diagram (fig. B.21), Structure diagram (fig. B.22), Class diagrams (fig. B.23).

4.6.4 Validation
Validation involves the question whether a given model is an acceptable model of the actual system under study. It involves the comparison between the executable performance model and the models created in the formulation process. Generated message sequence diagrams and debugging tools can be used to validate model operation.

Pase1: Message sequence diagrams are generated with the model and compared with the assumption of system behavior. Fig. B.14 shows the Message Sequence diagram that is created with the executable performance model.

Pase2: Fig. B.24 shows the Message Sequence diagrams used to validate the executable performance model.

4.7 Conclusions
Based on performance models created of the DataFlow hardware system a set of six UML diagrams is selected. A modeling framework is introduced for performance modeling. With the developed modeling framework, performance models are created and simulation are performed using the execution scheme given in chapter 5. With these models the focus is on formulation of a design solution and formalizing the model to a executable performance model.

It can be concluded that the UML language needs heuristics in order to be able to use the language effectively for performance modeling. Furthermore, additional code in an object-oriented language is needed to create executable models.

With the developed modeling framework UML performance models can be created. There are however a few drawbacks. For performance simulation the UML models need to be executable models. It is shown that some tools offer possibilities for the creation of executable models through use of State Charts. This solution can however not be used for the modeling of hardware systems. Furthermore, modeling behavior with State Charts as provided with the evaluated tools does not support currently used performance modeling methods.

The evaluated UML tools offer possibilities for UML syntax and consistency checking. The tools are however not mature yet. The Rhapsody and Artisan tools support implementation for both synchronous UML signals (RPCs) as asynchronous UML signals, but intermix UML signals with normal procedures. It is expected that in the UML2.0 standard the problem of interfacing between classes will be addressed. Preliminary documents involving the UML2.0 standard [UML2.0] show that extension to UML will be provided that allow the separation between messages and procedures. RoseRT supports the UML-RT extensions and makes a clear distinction between UML messages (signals) and procedures owned by a class through its concept of ports and protocols.

The evaluated tool RoseRT is however implementation oriented. RoseRT is suitable to create the Use Case, Structure, Deployment and Message Sequence diagram. Other diagrams could not be created properly with RoseRT and a second tool, Rational Rose, had to be used to create the Class, Collaboration and Object diagrams. This way, it is possible to create all needed diagrams, but the syntax and consistency checking cannot be performed completely. This makes it hard to develop consistent models. Since an important aspect of performance modeling is the creation of an executable model, it is advised to use at least the RoseRT tool.
5 Performance simulation of hardware systems

The aim of performance simulation as done on DataFlow is architecture exploration, algorithm exploration and performance analysis. In the case of DataFlow the main interest is aimed at the statistical access to the DataStore memory. Performance models are currently created in C++ using the framework presented in [N100a]. The main goal of this chapter is to present extensions created for the currently used modeling framework and to demonstrate that these extensions can be combined with the UML modeling framework of chapter 4.

5.1 Abstractions in performance modeling of DataFlow

Fig. 5.1 shows a block diagram of DataFlow system. In the block diagram the data path and control path are given that are relevant for the performance modeling regarding the statistical DataStore accesses. The data path (drawn in black) represents the flow of data through the system. Arriving frame data in the Receiver is written to DataStore memory (1) and read by the Transmitter (4) in order to send it to the Sink. Furthermore, the EPC acquires read (2) and write (3) accesses the frame data in DataStore memory in order to perform frame processing. The DataStore memory is a resource shared among the Receiver, Transmitter and EPC. A control path (shown in gray) in DataFlow is maintained by the Memory Arbiter, which controls the accesses to DataStore by Receiver, Transmitter and EPC. An important aspect of the DataFlow performance simulations is to investigate the behavior of data flows (data path) through the system under influence of the control algorithms (control path).

The performance models that were created of DataFlow are control-based models. The main interest with these models is the evaluation and exploration of design alternatives regarding architecture and algorithms for controlling data flow through the system, rather than the contents of data. Abstractions are made of data by representing data by tokens with proper annotations. The content of data is considered irrelevant in the performance model. The smallest relevant data granularity is the quad word (see chapter 2).

![Fig 5.1: Data path and control path in the DataFlow system](image-url)
Furthermore, abstraction is made of time. The smallest relevant time window in the DataFlow system is 66ns (time needed to read or write buffers of 64 bytes to a slice in the DataStore memory). Time is modeled in the performance model as equidistant (in simulated time) time events. Every time event represents the passing of a 66ns simulated real time window in the system. The time window is denoted as WindowCycle. The simulated time increases with steps with a length equal to a WindowCycle. At every time event in the model behavior of the modules in the model is executed. The behavior of the modules executed at these time events mimics the behavior of the modules in a complete window of simulated time. The right abstractions regarding time and data have to be chosen in order to model system behavior adequately.

5.2 Currently used modeling framework

Fig. 5.2 gives a brief explanation on the currently used modeling framework. It uses C++ to describe the system to be modeled. The diagram shows a part of the DataFlow system. A system is described as a set of communicating modules. Communication between modules is described in terms of master/slaveness (master initiates behavior in the slave). A communication channel is described as a master port (shown as a black small box owned by the module) on the initiating module connected to a slave port (small white box) on the slave module. Exchanged messages between masters and slaves can be without data passing or have content. Note that a description specifying a master port and slave already implies an implementation choice. A black colored box indicates an initiator of behavior (master), a white box indicates a slave. The arrows between master and slave indicate the direction of data in a communication relation.

A module consists of state and behavior. In the modeling framework, C++ classes implement the modules. Communication between modules is performed through remote procedure calls (RPCs), accessing the state (context) of the module. These calls are invocations of public operations of the C++ classes. Behavior of a module is considered to have zero-delay.

Fig. 5.2: Model based on the currently used modeling framework

Fig. 5.2 shows a shared resource (e.g. DataStore memory) that is accessed by both a Receiver and Transmitter module. The shown Arbiter controls the accesses. As mentioned in chapter 3, shared resources are most of the time accompanied by an Arbiter in the designs made. Reason for this is that accesses of concurrent modules to a shared resource need arbitration. The Arbiter performs this arbitration. The Receiver writes data after process to the shared resource and the Transmitter accesses data in the shared resource in order to process it and send it on its output. There is merely one physical thread (sequential description), maintained by a clock module in the models. Multiple threads are modeled through queues and two available clocks in the model. Two clocks that are provided by a clock module exist in the model. These clocks, Process clock (p) and Propagation clock (c) subsequently trigger behavior. The process clock triggers the Arbiter. Both Receiver and Transmitter are

Alcatel Bell N.V.
slaved to the Arbiter. The shared resource is slaved to the propagation clock. Each sequence of process-clock and propagation clock is considered a clock cycle in the system. Since the relevant cycle in this case is the window cycle, this cycle will be called window cycle.

Fig. 5.3 shows a Message Sequence Chart that clarifies the execution schedule of the model of fig. 5.2.

The Message Sequence Charts shows two window cycles in the system (Window clock period n and n+1). The data in the system at cycle n is also shown in fig. 5.2. As mentioned, each clock cycle involves a process clock and a propagation clock. At process clock each module subsequently executes its behavior. In this case the Arbiter triggers behavior in Receiver and Transmitter. As can be seen in the Message Sequence Chart, the messages that are communicated among the modules are all RPCs. Message numbers shown correspond to the messages in fig. 5.2.

The Receiver and Transmitter are modeled as synchronously concurrent executing modules. This behavior is obtained in the following way. By means of message (p) to the Arbiter, the Arbiter behavior is executed in Windows cycle n. The Arbiter retrieves through message (3a) the requests of the Transmitter for read access to the shared resource. By means of message (2a), the Arbiter determines the Receiver requests for write access to the shared resource.

With message (3b) the Arbiter grants the read requests for read to the Transmitter and invokes Transmitter behavior. The Arbiter provides this way a process trigger to the Transmitter. This behavior invocation is in fact the call of a Transmitter operation. The execution of this operation involves the read access (message (8)) to the shared resource through which R(X(n-1)) is obtained. This data is made eligible in the previous cycle of the
model. In the same operation the data is processed and outputted (message (4)) in the form of \( T(R(X[n-1])) \). Note that \( X[n-1], R(X[n-1]) \) and \( T(R(X[n-1])) \) are data tokens in the model.

The write grants (message (3b)) sent to the Receiver by the Transmitter result in similar behavior. The message is a call to a Receiver operation and is essentially a trigger for the Receiver process. The invoked Receiver operation writes the \( R(X[n]) \) data to the shared resource. The data can only made eligible for dispatch to the Transmitter by the propagation clock. This means that data written to the shared resource is at that moment not yet available to the Transmitter, but is made available next cycle. At that point the data from a previous cycle \( R(X[n-1]) \) is already read by the Transmitter from the shared resource. The data token \( R(X[n]) \) written to the shared resource is made available for Transmitter-read through the propagation trigger (message (c)) sent to the shared resource.

At window cycle \( n+1 \), the data \( R[n] \) is available for read by the Transmitter. Note that the next window cycle (cycle \( n+1 \)) provides similar behavior regarding the model scheduling as the actual cycle (cycle \( n \)).

Note that clock triggers (p) and (c) in fig. 5.2 may not be used intermixed in order to maintain data consistency. Furthermore, the data in the shared resource is made eligible by setting a flag.

### 5.3 Extended modeling framework

To improve the modeling framework extensions are introduced. The new modeling framework is called “Extended modeling framework”. The developed modeling framework extensions utilizes also two clocks, but with a distinct purpose in model execution.

#### 5.3.1 Introduction

From the example given in fig. 5.2 and fig. 5.3 can be concluded that models created according the currently used modeling framework are synchronous concurrent models. There are a few drawbacks. To demonstrate the problems that can be encountered when using the currently used modeling framework, fig. 5.4 gives a model of the DataFlow system that is created with the framework. In fig. 5.4, the process clock (p1) triggers the Arbiter. Receiver and Transmitter process are in the model triggered by the DataStore access grants send by the Arbiter. The propagation clock triggers behavior of the Traffic generator, the EPC and the Sink. The propagation clock is intended to propagate data through the model: (c1) triggers the Traffic generator to send data to the Receiver, which is directly written in the Receiver’s buffers. The Receiver writes at Arbiter write grant not only data to the DataStore memory, but also writes a context pointer (FCB) to this data to the EPC. The EPC is modeled here as a FIFO queue. At triggering by propagation clock (c2) the EPC makes the context pointer from previous window cycles eligible for dispatch and writes these pointers to the Transmitter. The triggering of the Sink (c3) makes the Sink read data from the Transmitter.

The question with a system like in fig. 5.4 is: where must which clock trigger used? There is no clear split made between purpose of process and propagation clock in models created with the currently used modeling framework. Although the propagation clock is originally purposed to merely maintain data consistency in the model and not to play a role in the actual model operation, it is not how it is used. Behavior of modules is in fig. 5.4 triggered both by process and propagation trigger. This way, using the currently used modeling framework can become a complex issue with large models, especially when modules have sub-modules.

The order of Propagation clocks (c) is in fig. 5.4 of no importance, however the order of Process clocks (p) needs proper scheduling. In the example the scheduling is performed by the Arbiter. The Arbiter maintains the control path in the DataFlow system. Its execution cycle is tuned to the relevant window time with respect to the DataStore memory. The chosen abstraction made regarding time matches the cycle time of the control path maintained by the Arbiter and forces the modules it controls to be finished within one cycle.
Fig. 5.4: Model of the DataFlow system created with C++ modeling framework

From the previous discussion one can conclude that extension is needed of the currently used modeling framework. There is need for:

- Clear definition of clock triggers in a model and therefore clear model execution schedule. Process triggers should be used to initiate module behavior, propagation triggers merely to transport data.
- Possibilities to build hierarchy in models: modules can have sub modules.
- Uniform description of models. At all levels in the model (system level, intermediate level and module level), the provided execution schedule should be clear, for building large models.

5.3.2 Developed extensions

In the extended modeling framework concurrency of modules is modeled by using a FIFO queue between modules. This FIFO queue needs to be emptied at triggering by the propagation clock (c), whereas the modules are triggered by the process clock (p). Data transferred between two modules by sending messages via a FIFO queue from one module to another is available to a receiving module next window cycle (propagation trigger of FIFO follows the process clock that triggers the process that causes the message to be sent). Message communication is buffered. The sender of a message does not have to wait for the receiving module to be ready to receive the message. Furthermore the sender is not blocked by the sending operation and can immediately continue operation after sending. The sending of messages causes not directly the execution of behavior in a receiving module. Moreover, the sender can send more than one message in one window cycle, however due to finite length of the FIFO queue messages can get lost caused by FIFO overflow. This type of communication will be defined asynchronous communication here (see fig. 5.5). The modules do not synchronize behavior at communication. Asynchronous transferred data is always available in a next elementary model cycle to a receiving in the model. (Data crosses the window cycle boundary). It is important to note that asynchronous communication used in a model does not imply that the actual implementation will use this type of communication (abstraction).

Fig. 5.5: Buffered asynchronous communication
Data transferred between modules without using a FIFO queue means synchronization between the behaviors of the modules at the abstraction level (see fig. 5.6). A sender of a synchronous message calls a procedure in the receiving module. RPCs between modules are used to model this. The communication e.g. between Arbiter and Receiver is an example of synchronous communication. The Arbiter controls the accesses to the shared resource of Receiver, Transmitter and EPCCController within one elementary model cycle. The order in which requests and grants are communicated between Arbiter and these modules is fixed (first requests, then grants). This means that synchronous communication is needed in order to accomplish in one cycle. The type of communication which uses RPC in the modeling framework will be defined here as (unbuffered) synchronous communication.

Fig. 5.6: Unbuffered synchronous communication

Note that the communication between the Arbiter and the three accessing modules in fig. 5.4 is synchronous at the used abstraction level since the Arbiter algorithm must be finished in one window cycle.

Fig. 5.7 explains the operation of the new extended modeling framework in general. In this figure the internal ‘structure’ of module and the possible types of communication with its outside world is shown. The modules used in the framework are classes. The module classes have ports that are the interfaces used to communicate. Both synchronous and asynchronous communication is available. The ports are member attributes of the module classes. Module classes have attributes and operations. The attributes are the data objects of the module and the operations model module behavior. Every module in a model is provided with both process (p) and propagation clock (c) through a dedicated port.

Fig. 5.7: Module used in the new modeling framework

- A provided process clock (p) to the module is a RPC to an operation ‘Process()’ which models the actual behavior of the module. The Process() operation represents an algorithm as it is also used in the currently used framework. It represents module behavior for a period equal to process clock period. During execution of this behavior, RPCs or asynchronous messages to other modules can be sent through master ports. Example of a RPC is the data sent to DataStore (see fig. 5.5). An asynchronous
message is for instance sent to the EPC. (Note that the Process() operation can use sub-operations and attributes to attain its objectives).

- Synchronous input messages are RPCs to operations of the module. Care must be taken in using RPC since it means direct interference with internal behavior of the module. Data transfer between modules is done on a deep-copy basis.

- The propagation clock (c) is used to handle asynchronous messages. These messages are in this case merely data. Asynchronous messages, for instance data sent by the shown traffic generator are enqueued in the internal FIFO of the slave port. After reception of the propagation clock, the Propagation() operation is executed which dequeues the data in the FIFO and writes it to the module's attributes. The data stored in the attributes is then available to module behavior next process clock. Note that asynchronous messages are merely used for transfer of data here. It is however possible to transfer asynchronous messages that initiate behavior. There is however no need to use this mode of operation in the performance simulations discussed in this document.

The Process(), Propagation() and RPC operations can influence each other through attributes and sub-operations. It is the responsibility of the designer to watch over the consistency of data and behavior in the model.

Fig. 5.8a shows a State Chart of the basic operation of modules in the new modeling framework (here Receiver module). After module creation (receiver()), initialization is performed and the module reaches an idle state. Process and propagation triggers cause the module to execute Process() and Propagation() operations respectively. An RPC message to the module directly calls the appropriate module operation (see self transition). RPC calls are directly related to other modules and the scheduling of RPC is the responsibility of the designer, which is highly unwanted since different schedules of RPCs could lead to different model operation and hence to different performance simulation results. Therefore, RPCs should be used with care. Fig 5.8b shows the schedule of the process and propagation triggers in a model with multiple modules. One sequence of (p)'s an (c)'s represents one clock cycle in the system. (p)'s an (c)'s should not be intermixed. In this case all of the modules have the same clock cycle period. In section 5.6.3 explanation will be given on multiple cycle periods (clock domains) in one model.

![State Chart](image)

**Fig. 5.8:** (a) State Chart depicting module behavior caused by process and propagation trigger and RPCs; (b) Execution schedule of process and propagation triggers

### 5.4 Extended modeling framework in C++

A first evaluation of the extended modeling framework is performed in C++. Fig. 5.9 shows a part of the models created in C++. The modules introduced in section 5.4 are here C++ classes. Appendix C gives the C++ extensions made and models that were created according the extended C++ framework.
Fig. 5.9: Model of synchronous and asynchronous communicating modules in C++

The following lists the characteristics of the developed C++ extensions:

- The modules are instances C++ classes with at least the functions Process() and Propagation(). The attributes of modules are also instances of C++ classes.
- Modules have attributes, which are instances of data classes. Data classes can be (re-)used in multiple modules. Modules encapsulate their internal instances of data classes.
- The shown ports are objects instances of the class PORT and are owned by a module class. Both synchronous as asynchronous communication is solely performed through these ports. The ports of a module are the only public objects of a module. Other attributes and operations are private. Three types of ports are defined. Their role-names and purpose match that of the ports defined in UML-RT (chapter 3): base, conjugated and relay. Base ports (provided with <<base>> UML stereotype) are here defined as initiator in a communication scenario (master port). Conjugate ports (provided with <<conjugated>> UML stereotype) are slave ports. Relay ports merely transfer (relay) data from a sending port to a receiving port. This receiving port can again be a relay port.
- All data transfer between modules is done on deep copy basis (No pointers to data are transferred). All data types used by the modules can be transferred between modules through their ports.
- Modules are connected through their ports at top level. In the shown example, the connection between Receiver (PORT_Arb1) and Arbiter (PORT_Rx1) is made in the constructor of the Toplevel object. The connection between ports is made through pointers, where a base port (e.g. PORT_Rx1) has a pointer to a conjugated port (e.g. PORT_Arb1) or a relay port. A connection between ports is unidirectional and adds no behavior.
- Synchronous communication is performed through RPCs. The Arbiter module in fig. 5.9 uses RPCs to obtain Receiver requests and grant these requests. A RPC is performed by calling the Invoke() operation of a master ports with proper function arguments (see Process()) operation in Arbiter module). The base (master) port uses its pointer to the appropriate conjugated (slave) port to call the RPC procedure that belongs to this conjugated (slave) port. The RPC procedure of the slave port executes appropriate procedures of the receiving (slave) module (the module procedures are known by the slave port at compile time). An RPC can have return data as shown in the Arbiter module. Furthermore, all data transfer through RPC is done on deep copy basis. At compile time it is checked whether all data types that are transferred through RPC match.
RPCs can have direct influence on data and behavior of modules. Dependency (see Receiver module) between data used in RPC operations and in the behavior described in Process() operation can exist. Scheduling RPCs is responsibility of the model designer.

Example of synchronous communication is the PORT_Rx1_Invoke() call in the Arbiter Process() operation to the RPCReq() operation in the Receiver module.

- Asynchronous communication is performed by the calling of a base port Send() operation. Data sent through this operation is enqueued in the FIFO queue of the conjugated port. The propagation operation (triggered by the propagation clock (c)) empties the port’s queue and writes the data into the appropriate module attributes (for instance a data object or a queue). Data is retrieved from a conjugated port by calling its GetPortData() operation. Asynchronous communication has no returning data.

Fig. 5.10 explains how synchronous and asynchronous communication operates between ports and the role a relay port plays in a communication scenario. Note: communication between modules is unidirectional, with exception of the return objects in a RPC. Relay ports do not introduce extra clock cycles delay.

**Fig. 5.10: Asynchronous (a) and synchronous (b) communication with relay ports**

Fig. 5.10a shows that in a process() operation an asynchronous send() operation can be performed with an arbitrary (limited by the used compiler) number of data objects as argument. A set of data objects transferred can be of various types and sizes (within limits of execution capacity): \{ASignal1, ASignal2, \ldots\}. In the example the conjugated port (slave port) belongs to a module, which is encapsulated by another module. The conjugated port receives the data from the relay port. To the conjugated port, the relay ports acts if it’s the base port. In this way, the encapsulated module is completely shielded off from the exterior of the encapsulating module. The sending module and its base port have no knowledge about receiving ports other than that a port is connected to it that can accept the data. A connected (receiving) port can be a conjugated or relay port.

Data asynchronously sent to a conjugated port is stored in the FIFO of the conjugated port. At execution of the propagation() operation, the FIFO of the conjugated port is emptied and the contents written to the appropriate attributes. Asynchronously sent data is available in the next windows cycle.
Fig. 5.10b shows that in a process() operation an synchronous RPC (Invoke() operation) can be performed to an encapsulated module. The base port uses its pointer to a connected port to make a RPC. In this case the pointer indicates the relay port. The relay port redirects this pointer to the encapsulated conjugated port. Through this, the operation called in the sending module (owning the base port) directly calls the appropriate operation in the receiving module (owning the conjugated port). The called operation (in the receiving module) can return data which is directly available in the calling module (essentially bi-directional communication). Data transferred through synchronous RPC is directly available, but directly interferes with internal data and operations, which can lead to module data inconsistencies.

Ports are dedicated to one type of communication (synchronous or asynchronous) and a set of communication data-objects, which is known at compile time (no sharing of connections). Conjugated ports used for RPC are connected to one operation in the receiving module.

With the extensions the models created with the currently used framework (see fig. 5.4) were rebuilt. In appendix C an example is presented. Fig. 5.11 shows the impact on a simplified version of the DataFlow model. Fig. 5.12 shows the accompanying Message Sequence diagram.

In this model of fig. 5.11, a clear distinction is made between asynchronously and synchronous messages. Each of the modules is provided with the process/propagation clock (p/c). The (p/c) triggers are provided to the modules by a separate clocking module, which performs the appropriate scheduling of the triggers. The token annotation given, represent the data present in the system at window cycle n. The frame data, represented by token [n+1], is asynchronously sent to the Receiver by the Traffic generator. ([n+1] means in this case that the data is available for process in the system at cycle n+1). [n+1] represents the frame data sent by the Traffic generator within one cycle of the system, which can involve a single or multiple frames dependent on the frame length and the cycle time. The Message Sequence diagram of fig. 5.12 shows the behavior of the model in fig. 5.10. Notice that the distinction between synchronous and asynchronous messages is also shown in the diagram. The Message Sequence diagram shows that due to the scheduling mechanism provided, first the processes of Receiver, Transmitter and EPC are executed. Both Receiver and Transmitter determine here their requests for access to DataStore. The EPC behavior involves, in this case, the handling of the data token FCB([n-1]) and the asynchronously sending of this token (6) to the Transmitter. The assumption is made that the execution of EPC behavior takes no extra window cycles (total delay of tokens through the EPC is here one cycle, as the arriving messages in the EPC are asynchronous). After the Receiver, Transmitter and EPC process are executed, the Arbiter process is executed. The Arbiter communicates through
RPCs with the Receiver and Transmitter and controls their accesses to DataStore. The Arbiter obtains the requests of Transmitter and Receiver through RPCs (3a) and (2a). The Arbiter determines through its algorithm, the grants for access and provides these grants through an RPC. Firstly the grants are sent to the Transmitter by means of an RPC (message (3b)). This RPC directly causes the Transmitter to access the DataStore memory through another RPC (message (7)). Through this RPC, the Transmitter obtains data presented by token fd[n-2] ('fd' stands for frame data, [n-2] indicates that the data was stored two cycles before). DataStore is here a shared resource modeled with zero access latency. The Transmitter requests for read access to DataStore are based on the received FCBs from the EPC. The data fd[n-2] was read because the Transmitter had received FCB[n-2] from the EPC in the previous cycle (previous propagation clock has made FCB[n-2] available in the Transmitter's internal target port queue). Notice that the assumption is made that that size of the data portions written to DataStore by the Receiver are of the same size as the data read by the Transmitter per processed FCB.

![Diagram](image)

**Fig. 5.12: Message Sequence diagram of rebuilt model**

The Transmitter processes the read frame data and sends it through asynchronous message (4) to the Sink. This sent data is represented by token T(FCB(\(f[n-2]+fd[n-2]\))), where T(.) represents the processing done by the Transmitter. After Transmitter operation has finished its operations, the Arbiter continues its algorithm. Access grants are sent to the Receiver by the Arbiter through a RPC. The Receiver writes received frame data, represented by fd[n] to the granted memory space in DataStore. At reception of the end of a frame, a data token FCB(\(f[n]\)) is asynchronously sent to the EPC. It is assumed that every windows cycle a frame end is received by the Receiver. The FCB token represents the context of a frame in the system. Notice that due to the use of RPC, the order of execution of process triggers can be of importance if Receiver and Transmitter are allowed to access the same data within one cycle. The Arbiter algorithm must schedule the accesses of Receiver and Transmitter to DataStore to prevent this. The Arbiter process has to be triggered after Receiver and Transmitter, since the Receiver and Transmitter must have determined their request before the Arbiter can grant Receiver and Transmitter request.
After the sequence of process triggers has finished, the propagation triggers are provided to the system. At execution of the Receiver propagation operation (triggered by the (c) trigger), the data \( f[n] \) sent in the current cycle to the conjugated port of the Receiver is retrieved from this port and written in the Receiver PrepArea (see chapter 2 for explanation). Propagation trigger to the Transmitter causes the propagation operation to transfer the FCBs in the conjugated port’s FIFO to the Target port queue in the Transmitter. Triggering of the EPC transfers data from its conjugated input port to the internal FIFO.

The diagram of fig. 5.11 can be presented in UML by using UML-RT Structure diagrams and UML Collaboration diagrams. The UML Collaboration diagram can be used to show the individual messages that are conveyed between modules. The types of messages (synchronous or asynchronous) is specified with UML stereotypes. The modules in fig. 5.11 are modeled with UML-RT capsules. The Structure diagram is used to show the model structure. Messages defined in the Collaboration diagram are mapped to protocols and ports in the Structure diagram (Multiple message sent between modules are grouped in protocols). In order to be able to use UML-RT to create models that match the extended modeling framework, the operation of the tool that supports UML-RT executable model building, must be investigated.

### 5.5 Extended modeling framework in UML-RT

In the tool Rational RoseRT executable UML-RT models can be made. It is investigated how this modeling tool can be used to create executable models that match the extended modeling framework.

#### 5.5.1 Executable models in RoseRT

UML-RT models created in RoseRT need additional code (for instance C++) to make them executable. Created executable models interface with a provided abstract layer called the RoseRT virtual machine [RoseRT]. The virtual machine provides the necessary services, like scheduling, timing, communication between capsules, etc. needed for executing a model. This virtual machine is situated between the model and the target environment. The provided services and the real-time capabilities of the virtual machine depend on the target environment. For the models created for this project, the operating system WindowsNT is used as a target environment.

**Capsules and threads**

UML-RT capsules in RoseRT each have their own logical thread (of control) provided by the virtual machine. Physical threads are the threads provided by the operating system. Logical threads or a group of logical threads can be mapped to physical threads. The scheduling of thread execution is performed by the virtual machine. The virtual machine relies regarding the scheduling of physical threads on the used operating system. In the case of a single model mapped on a single physical thread, the scheduling of threads is completely maintained by the virtual machine. As shown in chapter 3, behavior of capsules is modeled with state charts that solely are reactive to events (messages) provided through ports.

Messages have priorities. A message priority is interpreted as the relative importance of an event with respect to all other unprocessed messages on a logical thread. The scheduling of logical threads by the virtual machine is based on run-to-completion (RTC) semantics of transitions between states in the state machine of a capsule. With the run-to-completion approach, handling of the current event (message) does not allow any interruptions, even by the arrival of higher priority events (messages). The advantage of this approach is simplicity. Logical threads mapped on different physical threads may be executed interleaved by the operating system. A used operating system can perform preemption on physical thread execution.

The time of executing behavior (in C++ code) is operating system dependent. Furthermore, two primitive timing instructions to model delay in real-time are available: `InformAt(timemoment)` and `InformIn(delayperiod)`. The timeliness of these instructions is operating system dependent.
Asynchronous and synchronous messages

The basic communication between capsules is message passing. Messages are sent asynchronously or synchronously and are scheduled by the virtual machine, which uses priority queues to maintain the messages. A receiving capsule in a communication scenario must accept a message (synchronous or asynchronous) or can defer it. A message sender has no knowledge about the readiness of a receiver capsule to receive the message. Essentially messages can be lost. Messages that are not expected by a receiving capsule are considered a model error (resulting in an error notification). Fig. 5.13 briefly shows how message passing is used in the modeling of DataFlow. The shown algorithms that perform message transfer are executed at transition between states.

Asynchronous messages are communicated through the primitive: `aPort.send(signal, priority, dataObject)` operation if `aPort` is the name of an end port. `Signal` is the symbolic name of the UML message that contains the `dataObject`, which is involved with the communication. `Priority` indicates the required priority level of the message.

Synchronous communication involves a controlled exchange of messages between sender and receiver. The sender dispatches a message with the primitive: `retMessagePtr = aPort.invoke(signal, dataObject)`. This message is conveyed by the underlying communication service (involving priority queuing) to the destination capsule in the same way as in the case of asynchronous communication. The message is then dispatched (after scheduling) at the destination where it triggers a transition in the destination behavior (if a suitable transition exists in the current state). The capsule associated with this transition must perform a special `reply()` operation, which returns a message back to the sender through the same port on which the invocation message was received. This `msg->reply(signal)` operation marks the end of the synchronous communication. The reply message always takes precedence over any message that may have been queued at the sender, regardless of priority. Threads that have to reply to an invoke operation cannot be interrupted. This way, the synchronous message passing operates as a RPC. When a reply is received and dispatched, the sender is unblocked and continues with the next instruction following the synchronous invoke.

Fig. 5.13: Message passing as used in the UML-RT models of DataFlow
Message scheduling
The virtual machine scheduler uses simple priority scheduling so that messages at a particular priority level are not processed until all higher-priority messages on that controller have been processed. Within a given priority level, the virtual machine guarantees that messages will be processed in the order of arrival regardless on what port in a model they arrive (priority FIFO operation). Message priorities do not imply interruption of the processing of the current event even if a newly arrived message is of a higher priority. This is due to the run-to-completion semantics of transitions as described in the previous section. Fig. 5.14 shows how messages are scheduled with a single physical thread (one logical thread per capsule). Assume that the relative order of arrival of each message is specified by its index (arrival times: t(1)<t(2)<t(n)). The shown messages are the messages that arrived during the processing of a certain event in a capsule. For simplicity reasons, only two of the available priority levels in RoseRT are considered here: p1 and p2, where p1 is the highest priority. Furthermore, all of the shown capsules operate on the same physical thread. A message starting with a 's' is asynchronous, with an 'i' synchronous (RPC).

**Fig. 5.14: Message scheduling involving two modules and using a single physical thread**

The order of messages handling be changed through priorities as can be seen in fig. 5.14 (overtaking of events by priority). Fig. 5.14 also shows how the execution focus is shared between the two logical threads through handling of messages.

5.5.2 Extended modeling framework in RoseRT
RoseRT offers the same communication primitives (synchronous and asynchronous) as are used in the extended modeling framework. With a logical abstract time basis and delay primitives in RoseRT it would be possible to let RoseRT schedule a performance model. Unfortunately this logical time basis is unavailable in RoseRT. Moreover, RoseRT cannot provide the scheduling as needed for the creation of models according the extended modeling framework. Therefore, the scheduling mechanism defined in the extended modeling framework must be added to UML-RT models created in this tool. This violates the requirement that model and scheduling mechanism must be separated (see requirements chapter 3). However, this shortcoming must be accepted if UML-RT models must be made executable for hardware performance modeling.
Fig. 5.15: Extended modeling framework in RoseRT

Fig 5.15 shows the same example as modeled in fig 5.9 in RoseRT. The following is needed to create a model that matches the extended modeling framework:

- All logical threads are mapped on a single physical thread. This makes models operating system independent; it makes the scheduling in the model predictable and eases the debugging of a model.
- A clock module is needed to schedule the model. Capsule behavior in RoseRT is modeled with state charts. For reasons discussed in section 4.2.2 State Charts are not explicitly used in the modeling of hardware behavior.
  Each capsule is connected to the clock module. Capsules in the model are provided with a standard (re-usable) state machine that handles the docking schedule provided by the clock module.
  - The timing primitives of RoseRT are not used for reasons discussed in the same section. By applying scheduling through a docking module, model schedule and model time are decoupled from execution real-time. It makes the model operating system independent.

Fig. 5.16 gives the Message Sequence diagram, which describes the scheduling of the model in fig. 5.15. The Message Sequence diagram shows that the clock module subsequently sends synchronous messages to the capsules in the model. These synchronous messages (CLK()) activate the behavior of each of the capsules (process clock). Due to the scheduling, the processes in the capsules are executed in a fixed order. The clock module determines the order in which the capsules processes are executed. During the execution of behavior, the capsules can send messages (both synchronous as asynchronous) to other capsules. Due to scheduling, the virtual machine allows only the synchronous messages to be handled directly by receiving capsules as long as the process cycle of the clock module is executed. The synchronous call through signal Signal1() of the Arbiter to the Receiver is therefore immediately executed and acknowledged by the Receiver with the signal AcReq(Req). Asynchronously sent messages are queued and handled after the clock module process finishes...
executing the current cycle. The message intSig(FCBData) is not dispatched to the EPC capsule until the cycle of the clock module algorithm is ready. At the end of a cycle of the clock module, the clock module re-triggers itself through an asynchronous (self-) message. The virtual machine then handles the (asynchronous) messages in its priority queue. One run of the clocking process equals one window cycle.

![Diagram](image)

**Fig. 5.16: Scheduling provided by using the extended modeling framework in RoseRT**

Notice that no separate propagation trigger is provided to the capsules. A capsule which receives a synchronous message (behavior of the capsule is invocated) can only receive asynchronous messages of the same priority after it has replied to the synchronous message (see for example the AckCLK() reply to the CLK() message). Dispatch of asynchronous messages is postponed to the end of the process cycle. This way a propagation trigger is implicitly created. The scheduling of messages is explained with fig. 5.17. At creation of the capsules in the model, all capsules receive an initialization message (Initial()). The clock module receives focus from the virtual machine. The clock module starts executing its clocking procedure. In this procedure it sends in a fixed sequence synchronous CLK() messages to the other capsules and waits until it has received all AckCLK() reply messages. The CLK() ensures that the other capsules execute their behavior (process clock). During this execution of behavior messages can be exchanged between the capsules. The synchronous Signal1() messages of the Arbiter to the Receiver is directly handled by the virtual machine (asynchronous messages are handled directly). The clock capsule can only continue executing when the AcReq(Req) reply message has been returned by the Receiver to the Arbiter (no reply is a model error). The CLK() message to the Receiver causes the Receiver to send an asynchronous message intSig(FCBData) to the EPC. This message is entered at the back-end of the queue and will only be handled if the clock module has finished executing its clocking procedure (this procedure only finishes when the last AckCLK() is received). After receiving the last AckCLK(), the clock module sends at the end of its clocking operation an asynchronous self-message (Self_Message()), which is enqueued in the priority queue. Since all asynchronous messages (e.g. intSig(FCBData)) sent during the clock cycle are in front of the Self_Message(), these messages are dispatched to the appropriate capsules first. This is essentially the propagation clock. The Self_Message() message re-triggers the clock procedure in the clock module and starts a new clock cycle.

The synchronous CLK() messages to the capsules are scheduled according to the number in the cardinality of the CLK port of the clock module. Cardinality is used to group the ports of the clock module, which provide the clock triggers to the clocked capsules. The clock module is explained in more detail in appendix D.
5.5.3 Specific modeling issues

Multiple clock domains
Multiple clock domains can be modeled with the extended modeling framework and hence also in RoseRT, by using a clock module, which provides the triggers for the clock domains. The clock triggers have different cycle times, but are all derive from the same base clock and are correlated. Appendix D gives an example of a clock module used in simulations involving DataFlow. The DataFlow system has a clock period of 66ns and the clock period of the modeled CellInterface is 36ns. The ratio between these clocks is 11:6. Every 11th cycle of the clock module, the DataFlow modules receive a clock trigger and every 6th cycle the CellInterface will be triggered. Communication between the clock domains needs interfacing through an adapter which handles the conversion between time and data abstractions in the clock domains. The assumption is made that the fact that different clock domains in the real-life system can have little correlation can be neglected. The adapter between two clock domains needs to be a rate converter (using e.g. a FIFO queue). If different data abstractions are used in the two clock domains and data is transferred across clock domain boundaries a conversion of data is needed. The abstractions and the assumptions made in this situation lead to model errors. A danger is the synchronization between modules in the model that represent hardware modules that in reality are not synchronized at all. It is the responsibility of the model designer to take these errors in account (e.g. by using the random latency FIFO presented in chapter 9). In the models built, the different clock triggers are provided by the clock module through different ports (each with cardinality dependent on the connected clocked capsules).

Modeling latency
Messages that are exchanged through RPCs are considered to take zero model time. Fig. 5.18a shows a capsule to which data is written through a RPC. The data is immediately available for process and can for instance be read by another capsule in the same window cycle. Notice that the order in which (1), (2) and (p/c) are applied, can be of great importance for the consistency of data. Fig. 5.18b gives a resource accessed asynchronously. Data written to the resource is available in the next window cycle and can hence be transmitted (4) in the next cycle. The latency is here one window cycle.
In order to model latencies larger than one cycle, latency FIFO queues can be used. These FIFOs can be used in two ways:

- Data retrieved at propagation trigger is stored in these FIFOs and is provided with a timestamp. Every new process-clock cycle (p), the timestamps of the data in the latency FIFOs is updated. Data at the front of the queue of which the timestamp indicates that the data has undergone the correct latency is dequeued and written in the appropriate attribute.
- Data is retrieved at propagation trigger and immediately written to the appropriate attributes. At process trigger, output data is generated and written to latency FIFO queues with a timestamp. Every process-clock cycle, the timestamps of the output data in the FIFO queues are updated. When the data has undergone the correct latency it can be written to the output port.

The timestamp is decreased every process clock cycle. The modeled latency equals the number of process triggers that are needed before the timestamp is decreased to zero.

**Model hierarchy**

Hierarchical models can be created through the nesting of capsules and the use of relay ports as can be seen in fig. 5.18d. The latency between message (7) and (8) is two cycles. In hierarchical models there is need for channel splitting (e.g. splitting of channels that provide the clocking to the modules). Currently RoseRT (RoseRT 2001) only supports binary protocols [RS00], which involve just two participants in communication of messages (no broadcast). Merely two protocol roles are supported in a communication scenario. Channel splitting (e.g. needed for broadcast) is in the used version not supported. Channel splitting would support complete encapsulation of nested capsules, since channels can then be split to provide clocking triggers to sub-capsules. In current version of RoseRT, sub-capsules have to be triggered separately, which violates encapsulation. Cardinality can be used to group ports that share a common protocol, which is the case with the clocking ports. The cardinality factor on a port determines the number of instances of a port. Capsules are scalable and so are ports with cardinality.
5.6 Assessment extended modeling framework based on defined requirements

RoseRT supports the building of executable UML-RT models according the extended modeling framework. UML models made according the UML modeling framework of chapter 4 can be made executable. Furthermore, modeling done according the extended modeling framework using UML-RT and RoseRT matches with currently used methods of performance modeling.

With aid of the requirements regarding performance modeling methods and frameworks defined in chapter 4 (section 4.4) the extended modeling framework can be assessed. These requirements can be used to verify if the presented extended modeling framework is suitable to be used for performance modeling in system level design. The assessment performed can be in some cases merely qualitative.

1. Abstraction:
Abstraction can made easily of data through tokens, presented by data objects. Since scheduling is provided in the extended modeling framework by a clocking module, abstraction of time can be made.

2. Well-defined language:
UML/UML-RT is an informal language, but has a well-defined syntax through its meta-model. Executable models are created with additional C++ code.

3. Clear modeling language that supports discussions:
Models created in UML/UML-RT are clear and easy readable and can support discussions if they are made in a consistent way. Consistent models can be made, using the modeling framework presented in chapter 4.

4. Encapsulation and modularity:
The concept of ports used in the extended modeling framework assures encapsulation and modularity of data objects and internal procedures (behavior) owned by modules. In UML-RT ports provide a coherent set of services defined using UML signals. Encapsulation is violated if RPCs are allowed to directly access internal data objects. It is the responsibility of the model designer to assure data encapsulation and to maintain data consistency. Nesting is possible through use of relay ports, however UML-RT lacks possibilities for channel splitting and therefore encapsulation is not completely obtained with the extended modeling framework. Possible solutions are: use of cardinality or provide scheduling mechanism in the capsule that provides the nesting (provide the nesting capsule with merely one clock trigger).

5. Reusability:
The models created in UML are reusable. Models or model parts can be combined into packages, which can be reused in other models. Packages of generic model elements are created with the models of DataFlow. Examples are the scheduling mechanism used in the models and the performance analysis components (library).

6. Scalability:
Scalability is directly supported with respect to the models in UML-RT (Cardinality is an attribute of the capsule role). With the C++ extensions (extended modeling framework deployed in C++) scalability has to be provided through dynamic creation of modules.

7. Distinction between processes and data objects:
UML makes distinction between active and passive objects through stereotypes. UML-RT capsules are active classes that encapsulate passive classes.

8. Inheritance:
Inheritance is supported by UML-RT regarding capsules and passive (data) objects. It is currently merely used with data objects.

9. Polymorphism:
Covered by UML/UML-RT, but not used.

10. Separation between model and scheduler:
The extended modeling framework uses a clock module, which provides the execution scheme. Separation between model and execution scheme is not achieved in the extended modeling framework.
11. **Debugging:**
   Observation and debugging of a running model can be done through tracing of messages, injection of messages and inspection of a model:
   - Inspection of triggered transitions and active states in the state diagram monitors.
   - Inspection of the Structure diagram animated in the structure monitor.
   Message sequence of messages passed in the system through inspected through probes (RoseRT generates diagrams that show message sequences).

12. **Maintainability and manageability:**
   Models can easily be changed in the used UML tools. UML tools used support easy model building concerning extended modeling framework models. UML-RT models can be created with little effort in RoseRT, since hardly any overhead code has to be written. Capsules, ports and connectors are created through drawing the Structure diagram. Compilation of the model can be done with standard C++ compilers. Model building issues like, memory management, portability of code and other model building issues that play a role in building a C++ model are responsibility of the used tool.

5.7 **Conclusions**

The design of the DataFlow system has a control path maintained by the Arbiter, which has a fixed cycle time period. The models created of DataFlow exist of synchronous concurrent operating modules. A window cycle time exists in the model and the modeled deadlines are hard. The currently used modeling framework matches the design of the DataFlow system. The concept of two clock triggers in a model has proven to work well in the performance modeling of DataFlow. It is however not always clear where and when which clock trigger must be used. Therefore the modeling framework is changed and extended with ports that can handle both synchronous as asynchronous communication between modules: the extended modeling framework. In the extensions all communication between modules is performed through ports.

Through the newly introduced modules and ports in the extended modeling framework firstly a clear distinction is made between clock triggers in the model. Therefore the execution schedule has become more obvious. Secondly, a uniform description is created of the modules in a model. Thirdly, hierarchical models can be created and at hierarchical levels in a model (System level, intermediate level and module level) the execution schedule is clear. Fourthly, through the introduced ports encapsulation of internal module data and behavior is achieved. With the extended modeling framework modules can easily be reused.

By using UML-RT and the RoseRT tool, models that are created with the performance-modeling framework of chapter 4 can be made executable. Models created in RoseRT according the extended modeling framework need an additional scheduling mechanism. A clock module needs to be added to the models.

The extended modeling framework meets all of the requirements regarding performance modeling methods, except for the encapsulation and separation between model and scheduler requirements. Regarding encapsulation, this is mainly caused by the lack of channel splitting in the RoseRT tool. Separation between model and scheduler cannot be achieved in the extended modeling framework, since a clock module is necessary in the model created by the model designer.
6 Brief comparison of the extended modeling framework and SHE

6.1 Selected UML diagrams versus SHE diagrams

The SHE method (see section 3.4.3) presents a coherent set of diagrams that can be used to model real-time hardware/software systems at system level. SHE consists of a method framework that involves clear, but informal graphical representations that are UML compliant. However, unlike with the UML language, the SHE diagrams form a more consistent set of diagrams that can be used to create orthogonal views of a design. A comparison is made here between the diagrams selected for the extended modeling framework and the diagrams defined in SHE.

Extended modeling framework selected

UML diagrams
- Deployment diagrams
- Collaboration Diagrams
- Class diagrams
- Use case diagrams

SHE diagrams
- Analysis
  - Architecture structure diagrams
  - Message flow diagrams
  - Object class diagrams
  - Functional requirements captured in requirements catalogue

Simulation, verification, validation
- Structure diagrams
- Sequence diagrams
- C++ based executable model

Fig. 6.1: Comparison between UML and SHE diagrams

Relations between the diagrams:
1. SHE Architecture structure diagrams depict the physical structure and distribution of a design. The UML Deployment diagram in the extended modeling framework is used to depict the same aspect of a design.
2. SHE Message flow diagrams depict the messages conveyed between process objects and clusters without specifying channels between the process objects. SHE defines a set of message flow types. Messages can be grouped and can be used to define a POOSL channel in the Instance Structure diagram (see 5). The UML Collaboration diagrams can be used to model the same aspect. In UML stereotypes can be defined that identify the SHE message flow types.
3. Class diagrams are used in UML to depict the classes of a model and their associations. Classes in UML define messages, attributes and operations. UML makes distinction between messages and operations, however in the evaluated tools, this distinction is not implemented. In SHE a clear distinction is made between messages and operations. In the SHE Object class diagrams, attributes and messages are defined. Furthermore aggregate and specialization/generalization are specified. In SHE the choice is made to show the associations between process objects implicitly through the POOSL channels in the Instance structure diagram, whereas in the performance modeling framework this is also done in Class diagrams.
4. Use Case diagrams are used in the modeling framework of chapter 4 to capture functional requirements. In SHE functional requirements are captured in the requirements catalogue. SHE leaves to choice of using Use Case diagrams for functional requirements specification up to the user.

5. SHE Instance Structure diagrams offer a powerful graphical way of depicting processes (process objects) that communicate using ports, connected through channels. Just as with the UML Object and Structure diagrams in extended modeling framework, the Instance Structure diagram depicts the active objects of a design. The channels in the Instance Structure diagram can represent the physical channels (line and busses) between hardware resources depicted in the Architecture structure diagram.

6. SHE Message Sequence charts depict the messages conveyed in a POOSL model and their relative order (in time). Message Sequence charts can be generated by the SHE supporting tools and are a powerful way of validating models. Message Sequence diagrams in UML are usually merely used to specify behavior. In the used tool, RoseRT, Message Sequence diagrams are generated while running a model.

7. The relation between POOSL and the extended modeling framework is discussed in section 6.2 and 6.3.

6.2 Synchronous models created with POOSL

6.2.1 The POOSL language

The SHE method uses the modeling language POOSL (Parallel Object-Oriented Specification Language). POOSL is a very expressive, compact system level modeling language that can be used to model system structure and behavior. POOSL is based on a mathematical framework. POOSL consists of a process part and a data part. The process part of POOSL involves process objects. The process objects operate concurrently and independent (asynchronous concurrency). Between the concurrent process objects, POOSL uses blocking synchronous message (and data) passing as communication primitive. Messages and data are transferred between process objects over static channels. Sending processes wait for a receiving process to be ready to receive a message. Receiving processes have to accept messages that are sent.

The process objects encapsulate data objects (the data part of POOSL), which can be transferred between process objects. The behavior of process objects is described with methods and per process object one methods is called at the initiation of the object. Behavior is described with a set of process statements. It can include operations on data objects. These process statements are created using the POOSL language primitives defined in [POOSL]. Behavior described in the process object can be parameterized by initialization of instance parameters.

Process objects can be grouped using the concept of clusters. A cluster is used to group a set of process objects and other clusters to create hierarchical structure in a model. A cluster hides internal process objects, clusters and channels. Process objects, clusters and channels are well suited to describe system architecture (architectural structure), topology and implementation boundaries.

POOSL data objects can be used to model process information or resources. Data objects in POOSL are based on the object-oriented paradigm and involve the definition of methods (operational procedures). The methods can be used to perform operations on the data objects.

Execution of POOSL code takes no model time. POOSL has a delay primitive that can be used to describe time in the model and process delay. Execution of a POOSL model is based on a two-phase execution model (see [VP98]). One phase involves the asynchronous execution of POOSL code. This execution involves both communication between process objects as the processing of data. The other phase involves the synchronous passing of time, which is described in the model with the POOSL delay primitive.

In POOSL, no extra code is needed to provide the scheduling for executing the model. A POOSL model itself defines through its semantics how it should be executed.
6.2.2 Modeling DataFlow with POOSL

Fig. 6.2 shows a part of a POOSL model of the synchronous design of DataFlow. The model is created in the synchronous concurrent manner as done with the extended modeling framework. This, to discuss the models of DataFlow created in the extended modeling framework in relation to asynchronous DataFlow models created in POOSL. Fig. 6.3 shows the POOSL code of the Receiver, Arbiter and EPC process objects. The behavior described of these processes basically exists of one main procedure. The procedures exist of two stages:

- The behavior of the processes and the synchronous communication of requests and grants (triggered by process clock in extended modeling framework)
- The communication of messages (triggered by propagation clock)

All of the modules shown have knowledge of the process clock. Through an instance parameter WindowCycleTime, the cycle time of the system is known to all process objects. All of the process objects have an initialization procedure init() which is called immediately when a process is created at runtime. The init() operations call the main procedure of the modules CLK(). In the CLK() the POOSL delay statements are used to subdivide the process in the two phases. The first delay statement models the actual process delay of the behavior of process objects. It delays behavior for a period of WindowCycleTime of model time. The second delay statement models zero delay. Both statements are used to synchronize the execution of the stages in the CLK() procedure for each separate proces. By nature POOSL process objects operate asynchronously concurrent and using the delay statement forces process execution to synchronize. The CLK() procedures use tail recursion to re-initiate themselves at the end of execution of their code.

First stage:
The first stage of the CLK() procedures represents the actual execution of the behavior (algorithms) of the processes: the process clock in the extended modeling framework. By using the delay(WindowCycleTime) statement, all process objects synchronize their execution of the CLK() procedure. It models also the delay of process behavior execution.

In the first stage of the process, requests and grants are exchanged between Arbiter (fig. 6.3a) and the other modules. This stage represents the synchronous exchange of messages regarding requests and grants as it is performed at triggering of the process clock in the extended modeling framework. Due to the nature of the DataFlow system, requests and grants must be exchanged before Receiver (fig. 6.3b) and EPC (fig. 6.3c) can take action.
The Arbiter asks the Receiver for its requests through the Rx1!GetRequests statement. The Receiver receives this message through the Arb1?GetRequests statement. Next, the Receiver determines its requests by consulting its PrepArea Data object. It then sends in this stage firstly its requests Arb1!Requests (RxRequests). The Arbiter expects these requests and uses its data object ArbiterAlg to determine the grants and sends them with Arb1!Grant (RxGrant) to the Receiver (which in turn accepts these grants through Arb1?Grant (RxGrant)).

The Receiver uses the grants to determine the number of FCBs that need to be sent to the EPC: FCBs := PrepArea ProcessGrants (RxGrants). Simultaneously, the EPC uses received FCBs from the previous
window cycle for process: \(\text{ResultFCB := EPCAlg.ProcessFCB(ReceivedFCB)}\) (using its data object EPCAlgorithm). Note that data that needs to be available in the next window cycle is not transferred in this phase (in the second phase).

**Second stage:**
The third phase involves the exchange of messages involving results produced in the second phase. The phase starts with a `delay(0)` statement to synchronize the beginning of execution of POOSL code. This phase is essentially similar to the propagation phase of the extended modeling framework. Data for next window cycle is here exchanged. The Receiver sends FCBs that are created in the first phase, in this phase to the EPC: \(\text{EPC!FCB(aFCB)}\). The EPC expects these FCBs: \(\text{Rx?FCB(ReceivedFCB)}\).

After the second phase finishes, the CLK()() procedures reinitiates itself (calls itself) and through tail recursion the CLK()() procedures are restarted (next window cycle).

POOSL is not used in its most effective way in the discussed example, as will be shown now. POOSL offers through its synchronous blocking communication the possibility to synchronize process procedure execution of asynchronously operating process objects. Instead of using the delay statement in each process object, the model of 6.2 can be reduced to a model in which merely one process object (the Arbiter) has knowledge of the elementary window clock cycle regarding the accesses to the DataStore memory (see fig. 6.4). The other process objects (Receiver and EPC) operate asynchronously and only synchronize with the Arbiter to exchange requests and grants. Fig. 6.4 shows a part of the asynchronous model of Arbiter and Receiver. Only the Arbiter has knowledge of the WindowCycle clock and has the statement: `delay(WindowCycleTime)`. Due to blocking communication, the Arbiter waits for the requests to be sent at the beginning of the execution of its CLK()() procedure. After transferring the requests, the Arbiter determines the appropriate grants and sends them to the Receiver (which is blocked until reception of the grants). After that, the Receiver handles the grants and can send an FCB to the EPC, after which the CLK()() procedure is reinitiated. The delay statement in the Arbiter models the delay of the cycle. Note that the Receiver procedure CLK()() has, by the blocking communication the same delay as the Arbiter CLK()() procedure.

### Receiver
```plaintext
CLK()()
|Data1: Integer; RxRequests: Requests;
RxGrant: Grant; aFCB: FCB; |
Arbl?GetRequests;
RxRequests := PrepArea.GetRequests();
Arbl?Requests(RxRequests);
Arbl?Grant(RxGrant);
/* Algorithm/behavior */
aFCB := PrepArea.ProcessGrants(RxGrants);
EPC!FCB(aFCB);
CLK()()
```

### Arbiter
```plaintext
CLK()()
|Data1: Integer; RxRequests: Requests; RxGrant:
Grant; aFCB: FCB |
Rx1?GetRequests;
Rx1?Requests(RxRequests);
RxGrant := ArbiterAlg.ProcessRequests(RxRequests);
Rx1!Grant(RxGrant);
delay(WindowCycleTime);
CLK()() .
```

Fig. 6.4: Rebuilt asynchronous Arbiter and Receiver model

What can be learnt from this is that a synchronous model, as made in extended modeling framework, can also be made with POOSL. Moreover, an asynchronous model of the same design can be made with POOSL.

With the extended modeling framework, scheduling must be provided through the use of two clock triggers, whereas with POOSL the scheduling is performed according the semantics. (Asynchronous) concurrency is part of the POOSL language, whereas with the extended modeling framework, concurrency and parallelism must be created. The sequential nature of the used language (C++) is the cause of this.
6.3 A comparison between the extended modeling framework and POOSL

The following comparison the extended modeling framework and POOSL can be made:

Modeling paradigm:
The extended modeling framework uses a sequential language to model concurrency and parallelism. The execution schedule has to be specified by the designer. POOSL combines sequential composition and parallel composition in one language (execution scheme of POOSL models is implicitly defined by the POOSL language). Both the extended modeling framework as POOSL can be used to model hardware systems like DataFlow. The models match the designs made of this type of hardware systems. However, in case of multiple, asynchronous concurrently operating processors (e.g. DataFlow and EPC), POOSL can model the behavior and interaction between the processors exactly, whereas with the extended modeling framework special measures as random latency FIFOs are needed (see chapter 9 for explanation on random latency FIFOs).

Abstraction:
In the extended modeling framework abstractions are made of time and data. Regarding time abstractions are made at level of the complete model within the same clock domain. All modules that operate within the same clock domain are slaved to the same clock. Abstractions made of time in a POOSL model are made at process object level. Example: at every arriving quad word of data to the Receiver, the behavior regarding the handling of the input frame data is executed. The (modeled) execution time of the algorithm matches that of the arrival time of one quad word. POOSL process objects can operate asynchronously. As shown in this chapter, in POOSL both asynchronous as synchronous behavior of hardware modules can be specified.

Non-determinism:
A model created with the extended modeling framework is deterministic. A fixed schedule is used to schedule the execution of concurrent activities. Non-determinism in a design is difficult to discover with a model that is deterministic. The sharing of resources can cause non-determinism. The order in which modules are allowed to access a shared resource can have a major influence on system performance. With a model created with POOSL non-determinism can be discovered in a design solution.

Model building:
Executable POOSL models are created with SHESim and Rotalumis [POOSL]. POOSL models can be created, validated, verified and debugged in the graphical environment of SHESim. Models can be created according the extended modeling framework in RoseRT. Both RoseRT as SHESim offer rich debugging capabilities.
PART II: Performance simulation output analysis

The second part of this thesis focuses on the output analysis of performance simulation. Many aspects have to be taken into account in stochastic performance simulation. This involves both the selection of statistically correct pseudo-random number generators as the correct statistical analysis of output data collected during simulation. Both aspects are discussed in this section.

7 Simulation output analysis: confidence intervals

7.1 Introduction

Stochastic simulation of telecommunication systems is a technique that is in widely used for performance modeling and evaluation. At Alcatel Bell simulations of the performance of networks and network components are e.g. performed using the Opnet Modeler event driven simulation tool. Opnet Modeler is a so-called discrete event driven simulator (DES). Discrete event simulation models created with a DES are dynamic models that evolve in time by the occurrence of events at possibly irregular time intervals. The simulations performed on hardware systems as discussed in the first part of this document are part of a newly adopted view on system level design [NIO0a] for the development of hardware systems. The simulations are so-called discrete time driven simulations. Basically, these simulations are a special case of discrete event simulation in a control-based model. Events and data are treated as if they arrive at a time-instance \( t \) at which the model handles the events and the data. The operations caused by events and data processing are executed instantaneously in the model. In this type of model the assumption is made that the modeled real-life system’s operations are finished before the time event \( t + \) cycle time, which is the next cycle of the model. Under this assumption, the system timing can be modeled exactly (see chapter 5). In these simulations (and also in a DES) the generated events and data are functions of (pseudo)random numbers, generated by random number generators. Any simulation in which simulated events are functions of (pseudo)random numbers is a statistical experiment [LK00]. Adequate output analysis of these statistical experiments is of great importance. In a statistical experiment the precision of output results must be considered before any constructive conclusions about the investigated system can be made.

In this chapter, some methods will be presented that can be used to validate the output results of a simulation. The presented methods are furthermore useful in determining the required run length of a simulation. The work done on this subject and the contents of this chapter is structured as follows:

- A survey of the basic aspects of stochastic performance simulation is given.
- Output analysis confidence methods are discussed and two promising methods are worked out and evaluated.
- A test method for evaluating the output confidence methods is worked out and used on the chosen confidence methods.
- The structure of algorithms and practical implementations of the confidence methods are discussed.

The goal is to evaluate and work out some methods that can be easily used in practice for performance simulations involving hardware systems.

7.2 Stochastic simulation and performance analysis

The main reason to create an executable model of a design and to acquire performance simulations is to investigate system characteristics that cannot easily be calculated. An example of this are the statistical accesses to the DataStore memory performed by the Receiver, EPCController and Transmitter. Stochastic simulation is the tool to determine the influence of these accesses on system performance. System characteristics as these memory accesses and for instance the interaction of the system with its environment is modeled on a statistical basis. Arrival processes of data to the DataFlow system are modeled using a random number generator. Furthermore, the delay of behavior of hardware blocks can be modeled as randomly distributed. The purpose of performance simulation is to gain insight in the long-run behavior of the system. The aim of the simulations
involving systems as DataFlow (and with many other telecommunication systems and networks) is at estimating cumulative measures of performance as mean values, max values and probabilities of events. These measures are for instance: the mean/max delay experienced by packets, mean/max occupation of buffers or buffer overflow probabilities. For the averages the focus is in this chapter on sample-averages.

7.2.1 Non-terminating simulation

The simulations performed involving hardware systems as DataFlow are non-terminating simulations. Non-terminating simulations have no predefined event in the model that terminates the simulation naturally. With most metrics in the models of hardware systems one is interested in the long-term averages of system metrics. A steady-state simulation is a simulation for which the measures of performance are defined as limits as the length of the simulation goes to infinity. The length of a simulation is made large enough to get good estimates of the quantities of interest.

This chapter is aimed to obtain insight into the behavior of hardware systems regarding their queuing processes after a long period of time. Typically, queuing systems are at start of the simulation in a non-stationary phase. Usually queues in a system are initially empty at the start of a simulation. The non-stationary phase is the initial transient or warm-up period of the simulation. Then, if the process is stable, its performance metrics (measures of performance) move asymptotically towards their long-term steady state average.

7.2.2 Initial transient and steady state

Using discussion in [LK00], the following example can be given to explain steady state: for a M/M/1 queue the delay $D_i$ of customers in the queue is evaluated. Let $L(0) = l$ be the initial customers in the queue at time $i = 0$ (usually $l = 0$). And let:

$$F_{i, l}(x) = P[D_i \leq x | L(0) = l]$$

With $F_{i, l}(x)$ the transient distribution of delay $D$ at time $i$ given $L(0) = l$. Note that 'transient' means here that there is a different distribution for each time $i$. For any $x \geq 0$ it can be shown that:

$$F(x) = \lim_{i \to \infty} F_{i, l}(x) \quad \text{for} \quad L(0) = \text{any } l$$

Where $F(x)$ is called the steady-state distribution of delay $D$. From eqn. 7.2 follows that there exists a time $i'$ such that for all $i \geq i'$, $F_{i, l}(x) \approx F(x)$ for all $x \geq 0$. At the point in time $i'$ when $F_{i, l}(x)$ is essentially no longer changing with $i$, the process $\{D_i, i \geq 1\}$ is said to be in steady state. Steady state means essentially that the distribution of the delay $D_i$ becomes invariant. Note that the delay $D_i$ itself does change during steady state of a simulation. The process $\{D_i, i \geq 1 + k\}$ is approximately covariance stationary [LK00] if the first $k$ observations $(D_1, ..., D_k)$ of the process $\{D_i, i \geq 1\}$ are deleted and $k$ is sufficiently large. This means that if the $k$ observations of $D_i$ for $i < i'$ are deleted the process of $D_i$ is covariance stationary and the process can be considered to be in steady state. Theoretically the steady state of a simulation is reached when $i$ goes to infinity. However, a process can be considered to be in steady state when it is approximately covariance stationary after deletion of $k$ initial observations.

The phase in which $i < i'$ is the initial transient phase of the simulation. The phase of $i \geq i'$ is the steady-state phase of the simulation. An algorithm to find a possible $i'$ and hence the number of samples $k$ that need to be deleted before an output process is covariance stationary is given in section 7.11.

7.2.3 Steady state simulation

The main questions involving this type of simulations are:

- How can the system be simulated for a desired length of simulation time within a practical length of real-time?
At what stage during the simulation should data collection begin? When should the simulation be stopped? What duration of simulated time should the system be simulated before stopping the simulation?

There are two ways of determining the run length of a steady-state stochastic simulation for non-terminating simulations [LK00]:

- Fixed length simulation or the fixed-sample-size scenario
- Sequential precision simulation

With fixed length simulation the simulation length is determined before simulation start. The simulation procedure is stopped unconditionally when a fixed simulation time has passed or a fixed number of samples is obtained. Fixed length simulation is simple to incorporate in a simulation model. A fixed length simulation merely allows estimating the accuracy of a point estimator based on the available samples. It does not allow the producing of an estimator at a desired level of accuracy. With sequential precision simulation this is indeed possible.

7.2.4 Random generators and random generator seeds

Next to the simulation run-length, the used random number generator can have major influence on the accuracy of the simulation results (see appendix A). The simulation run-length can be determined by the methods presented in this chapter. The influence of the random number generator on the output of a simulation is implicit to a simulation model. The use of a ‘good’ random number generator is therefore of vital importance.

As stated in appendix A, the choice of seed for the random number generator influences the simulation output. Therefore, the following guidelines are given:

- A simulation must be performed multiple times with different random number generator seeds to compare output results. Seeds can best be chosen in such a way that non-overlapping streams of random numbers are used. Non-overlapping streams of random numbers can be obtained by selecting seeds through seed entry tables (see appendix A).
- For random numbers used at different places in a simulation model it is best to use multiple non-overlapping streams of random numbers provided by a single random number generator in order to prevent unintended correlation between simulation data caused by correlated streams of random numbers [PA99].

7.2.5 Probability and non-determinism

Probability in a model is created by using a random number generator. Events and data are generated with a certain probability. The models created of DataFlow with the extended modeling framework are completely deterministic since the scheduling mechanism used to schedule behavior execution is predetermined. Model operation is a function of the used random number generator(s) and therefore probabilistic. The scheduling mechanism used, has direct influence on obtained performance measures obtained through simulation. As stated in [TH00b]: the scheduling of concurrent activities entails the resolving of non-determinism in the model. In case such non-determinism is not resolved properly, different performance measures can be obtained for different scheduling solutions. The extended modeling framework can through its predetermined scheduling mechanism leave out of consideration other scheduling implementations and therefore miss design alternatives or misjudge the performance metrics of a system since non-determinism is not recognized. This is a potential danger and needs to be taken into account (if possible) during model building and simulation.
7.3 Simulation output analysis

7.3.1 Output analysis: introduction to confidence intervals

Estimates and confidence intervals

Obtaining statistically valid results by simulation is difficult due to the fact that samples of metrics in a model collected during simulations are typically correlated (see [LP99]). Furthermore, a simulated process initially moves along a non-stationary trajectory, the initial transient phase.

A sequence of observations (e.g. metric samples) \( x_1, x_2, \ldots, x_n \) collected during a simulation run is a sequence that is the output of the statistical experiment driven by the used random number generator. The observations can be used to estimate an unknown average \( \mu_x \) by calculating the average of the sequence:

\[
X(n) = \frac{1}{n} \sum_{i=1}^{n} x_i
\]  

(7.3)

Note that this average assumes a sequence of random observations \( x_1, x_2, \ldots, x_n \). Using standard statistics the accuracy with which this average estimates the unknown parameter \( \mu_x \) can be evaluated through the probability:

\[
P\left( X(n) - z_{1-\alpha/2} \frac{\sigma[X(n)]}{\sqrt{n}} \leq \mu_x \leq X(n) + z_{1-\alpha/2} \frac{\sigma[X(n)]}{\sqrt{n}} \right) = 1 - \alpha
\]  

(7.4)

Where \( z_{1-\alpha/2} \) is the \((1-\alpha/2)\) quantile of the normal distribution, and \( \sigma[X(n)] \) is the estimator of the variance of \( X(n) \). The accuracy of \( X(n) \) is determined by \( \Delta_x = z_{1-\alpha/2} \frac{\sigma[X(n)]}{\sqrt{n}} \), which is the half-width of the confidence interval, at an assumed confidence level \((1-\alpha)\) \((0<\alpha<1)\). \( \Delta_x \) is called precision here. Eqn. 7.4 says that with probability \( 1-\alpha \) the confidence interval \((X(n) - \Delta_x, X(n) + \Delta_x)\) contains the unknown (true) average \( \mu_x \). Note that eqn. 7.4 also says that with probability \( \alpha \) the confidence interval does not contain \( \mu_x \), or that repeating the simulation a number of times the unknown value \( \mu_x \) would be outside the interval in \( 100\alpha\% \) of the cases.

Note: \( X(n) \) is sometimes called the point estimate of \( \mu_x \) and \( \Delta_x \), the interval estimate.

For observations \( x_1, x_2, \ldots, x_n \) that are realizations of independent and normal distributed random variables \( X_1, X_2, \ldots, X_n \), the unbiased estimator of the variance of \( X(n) \) is (see e.g. [JO00]):

\[
\sigma^2[X(n)] = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{X}(n))^2
\]  

(7.5)

The formula 7.5 can also be applied when the observations \( x_1, x_2, \ldots, x_n \) are not drawn from a normal distribution, but if they represent independent and identically distributed (IID) [WA98] random variables \( X_1, X_2, \ldots, X_n \). In such case the number of observations has to be large \((n > 100)\) or the batch means method can be used (see section 7.3.2) in order to obtain an acceptable approximation.

Unfortunately, observations collected during practical simulations are not statistically independent, but they appear to be strongly autocorrelated. A general formula for the variance of the mean \( \bar{X}(n) \) of observations \( x_1, x_2, \ldots, x_n \) collected from a covariance stationary process is [WA98]:

\[
\sigma^2[\bar{X}(n)] = \frac{1}{n} \left[ \hat{R}(0) + 2 \sum_{k=1}^{n-1} \left( 1 - \frac{k}{n} \right) \hat{R}(k) \right]
\]  

(7.6)
With help of these formulas, it can be explained what the influence is of ignoring the autocorrelation of the observations. Eqn. 7.7 gives the autocovariance $\hat{R}(k)$ of lag k. Eqn. 7.7 calculates for a certain lag k (the distance k between samples in a sequence) the covariance of a certain sequence of samples $x_1, x_2, ..., x_n$. Note that since the covariance involves one variable, it is called autocovariance. Comparing eqn. 7.6 with eqn. 7.5, it can be seen that neglecting the existing statistical autocorrelations and hence using eqn. 7.5 rather than 7.6 would be equivalent to ignoring all $R(.)$ terms in eqn. 7.6 except for $R(0)$. (leaving out the n-1 factor instead of n used in eqn. 7.5 to create a less biased estimator of $[\sigma^2(x(n))]$). Neglecting the autocorrelation would then lead to wrong interval estimates and wrong confidence intervals.

It would lead to too large (pessimistic, with too large $[\sigma^2(x(n))]$) confidence intervals for negatively correlated ($R(k)<0$ for $0<k<n-1$) or to too small confidence intervals (optimistic, with too small $[\sigma^2(x(n))]$) for $\mu_x$ for positively correlated observations (since terms in the actual autocorrelation are neglected). It is known (see also section 7.91) that a positive correlation between observations is typical in simple queueing systems without feedback connections, and it is usually stronger for higher system loads. The models of the DataFlow hardware system are of this kind. It can be expected that in general with simulations with these models the confidence intervals will be estimated to small when using eqn. 7.5 (the correct confidence intervals will be larger).

An additional problem, specific for steady state simulations is caused by the fact that simulated stochastic processes typically have an initial non-stationary phase (changing their statistical properties in time). As shown in 7.2.2, the observations collected during the initial transient phase do not belong to a covariance stationary process. Neglecting the existence of initial transient periods can lead to an unknown bias in steady-state estimates.

**Precision control using confidence intervals**

With sequential precision simulation the confidence intervals of measures (metrics) can be used to determine the length of a simulation run. By assessing the confidence intervals and the precision of estimates during a simulation run, the precision can be used to stop a simulation when the required precision is reached. The stopping criterion is based on comparing the current value of the relative precision of the estimate with its maximum acceptable value. The relative precision of an estimate can be defined as the relative half-width of its confidence interval:

$$\varepsilon = \frac{\Delta_x}{X(n)} \text{ with } \Delta_x = \varepsilon_{1-\alpha/2} \cdot \frac{\sigma([X(n)])}{\sqrt{n}} \quad (0<\varepsilon<1)$$

(7.8)

or with the more conservative estimator of the relative precision [THOOb]:

$$\varepsilon = \frac{\Delta_x}{X(n)-\Delta_x} \quad (0<\varepsilon<1)$$

(7.9)

In this chapter and for the simulations performed eqn. 7.8 is used. The simulation is stopped (stopping rule) when $\varepsilon \leq \varepsilon_{\text{max}} \quad (0<\varepsilon_{\text{max}}<1)$, where $\varepsilon_{\text{max}}$ is the required maximum relative precision (relative error).

**7.3.2 Short survey of existing methods of confidence analysis and the methods choosen**

Several methods have been found in literature that can be used to determine estimates and variances of output data of stochastic simulation and that are meant to deal with the autocorrelation between observations. The methods and a brief discussion on their operation is given here:

- Method of independent replications
- Method of batch means: non-overlapping batch means, overlapping batch means, spaced batch means
Independent replications

The method of Independent Replications [LK00] tries to overcome the problem of estimation of estimates and variances created through the auto-correlated nature of the output data of a simulation in the following way: a simulation is repeated a number of times, each using a different, independent sequence of random numbers (non-overlapping streams of random numbers). Observations are gathered during a replications \( i \) and are used to calculate the estimate \( \bar{X}_i(l) \) (of a certain metric, where \( l \) is the number of observations gathered during a replication) with eqn. 7.3. The set of estimates \( \bar{X}_1(l), \bar{X}_2(l), \ldots \) are treated as independent identically distributed observations and are used to calculate \( \bar{X}(n) \) and \( \sigma^2 \hat{\bar{X}}(n) \) (with \( n \) as the number of replications) with:

\[
\bar{X}(n) = \frac{\sum_{l=1}^{n} \bar{X}_i(l)}{n} \quad (7.10) \quad \text{and} \quad \sigma^2 \hat{\bar{X}}(n) = \frac{1}{(n-1)} \sum_{l=1}^{n} \left( \bar{X}_i(l) - \bar{X}(n) \right)^2 \quad (7.11)
\]

\[
(\bar{X}(n) - t_{n-1,\alpha/2} \cdot \sigma \hat{\bar{X}}(n), \bar{X}(n) + t_{n-1,\alpha/2} \cdot \sigma \hat{\bar{X}}(n)) \quad (7.12)
\]

and hence constructing the appropriate confidence intervals (eqn. 7.12), where \( t_{n-1,\alpha/2} \) is the \((1-\alpha/2)\) quantile in the Student t-distribution with \( v=n-1\) degrees of freedom.

With the method of independent replications the length of a replication (for instance the number of observations gathered) is determined in advance. The method requires multiple simulation runs, therefore the simulation run-length cannot be determined with this method during the simulation. Regarding the method of independent replications, two approaches can be identified:

1. The first approach is to estimate the length of the initial transient phase and discard the first \( k \) observations that are obtained during this phase. Observations \( i \) for \( i>k \) are then used to calculate the estimates per replication \( \bar{X}_1(l), \bar{X}_2(l), \ldots \) and with these estimates \( \bar{X}(n) \) and \( \sigma^2 \hat{\bar{X}}(n) \) are calculated, using eqn. 7.3 and 7.5 (with \( l \) the number of observations gathered per replication without the first \( k \) samples). Confidence intervals are created from these calculations.

2. The second approach ignores the presence of the initial transient phase and assumes that data is collected during the steady-state phase of a simulation. Danger with this approach is that the estimates and variances can be biased which could lead to unreliable confidence intervals.

Disadvantages of the method of independent replications are that the length of a simulation must be known in advance (fixed sample size simulation) and that simulations have to be repeated multiple times, which can lead to long total simulation times. Although the method has these disadvantages, independent replications is used often, for instance with network simulations performed using the Opnet modeler tool (example of such simulations is e.g. given in [NI00a]). With these simulations the (worst case) approach 2 is mostly used. The method is not further discussed in this chapter and is considered to be not a good candidate to use since it is difficult to use for sequential simulation run-length determination.

Method of batch means

The methods of batch means are mainly of three types: non-overlapping batch means, overlapping batch means and space batch means (see e.g. [LK00, PA90, WA98]). The method of non-overlapping batch means is further investigated here and will be referred to as simply batch means.
The batch means method requires that the sequences of analyzed data are covariance stationary; therefore, initial observations, collected during the initial transient phase need to be discarded.

With the batch means observations (metric samples) are collected during a single simulation run. To reduce the correlation between the sequence of observations collected from the simulation, the original sequence of observations is divided into non-overlapping batches (see fig. 7.1).

![Figure 7.1: Sequence of observation divided into \( k_b \) batches of size \( m \)](image)

The batches are of size \( m \) and are made sufficiently large as such the mean values \( \bar{X}_1(m), \bar{X}_2(m), \ldots \) of the batches can be considered to be statistically independent. This independence makes it possible to use the mean values \( \bar{X}_1(m), \bar{X}_2(m), \ldots \) as input to standard statistical formulas eqn. 7.3, 7.4 and eqn. 7.5 to construct the confidence interval.

The batch means is based on the assumption that observations more separate in time are less correlated. The mean value and variance are calculated with (note: \( n = m \), with \( k_b \) as the number of collected batches):

\[
\bar{X}(kb,m) = \frac{\sum_{i=1}^{k_b} \bar{X}_i(m)}{k_b} \quad (7.12)
\]

\[
\sigma^2 \left[ \bar{X}(kb,m) \right] = \frac{1}{(k_b - 1)} \sum_{i=1}^{k_b} (\bar{X}_i(m) - \bar{X}(kb,n))^2 \quad (7.13)
\]

The 100(1-\( \alpha \))% confidence interval can then be obtained accordingly:

\[
\left[ \bar{X}(kb,m) - \frac{z_{1-\alpha/2}}{\sqrt{kb}} \sigma \left[ \bar{X}(kb,m) \right], \bar{X}(kb,m) + \frac{z_{1-\alpha/2}}{\sqrt{kb}} \sigma \left[ \bar{X}(kb,m) \right] \right] \quad (7.14)
\]

The major issue with the batch means is the selection of an appropriate batch length \( m \). In section 7.9 an algorithm will be discussed that can be used to determine an appropriate batch length at the beginning of a simulation run (batch length determined and used during simulation run). Note furthermore that the number of batches \( k_b \) must sufficiently large (rule of thumb: \( k_b > 30 \)) in order to be able to treat the batch means as normally distributed. The batch means method is further investigated and used for simulations.

**Method of regenerative cycles**

With the method of regenerative cycle or recurrent points observations are, just as with the batch means method, grouped into batches, but in this case the batches are of random length. The batch length is determined by successive instants (regeneration points) in time at which the simulated process starts afresh. The regeneration points are typically certain states of the simulation process. When such a state is reached a cycle of the regeneration cycles method starts and the end of the cycle is reached when the process reaches the same state again. When reaching such a state, no event from the past influences the future process of the system. Batches of observation gathered during regeneration cycles are statistically independent and identically distributed and so are their means. For example, the regeneration points in the behavior of a \( M/M/1 \) queuing system can for instance be the empty state of the queue or the time instant at which a packet arriving in the queue finds the queue empty.

In practice a (hardware) system can exist of many queues (e.g. the DataFlow system).
**Ideal regenerative cycles**: a chosen set of states of the queues as recurrent point. A recurrent point is then reached when the same set of states is reached again. Typical recurrent point is the empty state of all queues (typical simulation start). When all queues reach the empty state again in the simulation, the recurrent point is reached. This state of the system is however not likely to be reached often during a simulation, which leads to long regenerative cycles and hence long simulation times (rule of thumb is that a minimum of 30 batches is needed for normality of the batch means).

**Less optimal solution**: a less perfect approach can be to choose a recurrent point per queue or per performance metric (per queue a certain queue filling level can be chosen). This is the approach which is used with the simulations in this chapter. With this approach the quality of confidence intervals is likely to be less, since dependency exists between the queues in the system. Fig. 7.2 shows how batches are created of observations.

![Fig. 7.2: The creation of batches through the use of recurrent points](image)

The \( r \)'s in fig. 7.2 represent the recurrent points, each \( Y_i \) represents the sum of the observations in a batch \( i \), and \( L_i \) represents the length of the batch \( i \). The estimate of the metric is calculated with eqn. 7.15 and the variance for constructing the confidence interval with eqn. 7.16. Appendix F gives the derivation of eqn. 7.15 and eqn. 7.16.

Since the output data collected in batches created by using regenerative cycles have identical distributions, the problem of the initial transient does not exist. The gathering of observations starts with the entry of the first recurrent state, deleting all gathered observations in batch \( Y_0 \). For instance in the case of a simulation starting with initial empty queues and \( r=0 \) (queue empty) as the recurrent point, the collection of observation in the first batch starts immediately at the start of the simulation without discarding observations. In general can be said that initial observations until the first recurrent point us reached need to be discarded. The major problem of the regenerative cycles method is that it requires a priori knowledge of the simulated process.

**Method of uncorrelated sampling**

Uncorrelated sampling can be regarded as a special case of the batch means method, where the size of the batches is randomly chosen. The method is discussed in [LK00], where it is stated that uncorrelated sampling is merely producing reliable confidence intervals if the analyzed process is a normal process. This cannot be guaranteed for the simulation of hardware systems. Furthermore little is known on how to implement this method. Therefore the method is not further investigated.

**Method of spectral analysis**

Methods of estimation based on spectral analysis use the serial correlation (correlation between observations in the sequence \( X_1, X_2, ..., X_n \)) between observations collected during one long simulation run. It is required that analyzed observations represent a stationary sequence [LP99], and therefore observations gathered during the initial transient have to be discarded. The analysis of variance with this method is shifted into the frequency
domain by applying a Fourier transformation to the autocorrelation function $R(k)$. The method produces, according to [LP99], very good simulation results. The method is not used here since it is expected that its implementation (e.g., using the algorithm described in [LK00]) will be slow since many calculations are needed for the Fourier transforms and the method will consume a considerable amount of storage space. The method needs in fact a lot of sequentially obtained observation at once for its analysis calculations. Therefore this method is found less suitable to use with sequential simulations.

**Method based on standardized time series**

The method of standardized time series uses the fact that a standardized random process converges to a Wiener random process. The method is described in [LP99, LK00]. It is not used for the same reasons discussed involving the method of spectral analysis.

**Selected methods**

With the method of non-overlapping batch means and the method of regenerative cycles it was expected that implementation of the algorithms can be less complex than with the other methods. Therefore these methods were selected for further investigation. The method of regenerative cycles has the advantage that it needs no detection of the initial transient, whereas with the method of batch means observations gathered during the initial transient have to be discarded.

### 7.4 Benchmarking the methods: coverage analysis

In order to compare the confidence methods investigated, a test method needs to be defined. This test method involves the definition of the performance metrics to be evaluated of confidence methods and a test bench to test the performance experimentally.

**7.4.1 Performance metrics**

The performance of the confidence methods analyzed is compared in the sense of:

- The coverage (experimental confidence level).
- The required simulation run length, measured by the number of observations needed for stopping the simulation with the required level of precision.
- The final precision of the results at the end of a simulation.

The coverage is determined with the aid of the test of section 7.4.2. Some remarks have to be made concerning the required simulation run length. Since the simulations are driven by random number generators used, it can be expected that the length of a simulation is dependent on the sequence of random numbers used. With the tests performed on the confidence methods this is covered by using disjoint streams of random numbers in the experiments.

**7.4.2 Coverage analysis**

The quality of the approximations made by interval estimation methods has been analyzed experimentally. It is important to note that in an ideal case the final confidence interval would contain the true value of the estimated parameter with the probability $1-\alpha$, with $1-\alpha$ the required level of confidence, with $0<\alpha<1$. For example, a 95% confidence interval ($\alpha=0.05$) would mean that in an ideal case the final confidence interval would contain the true estimator of a parameter with a probability of 95%.

The following is also true: If an experiment is repeated many times, then the interval estimators produced by an ideal confidence method would contain the true value in $(1-\alpha)100\%$ of the cases. A suited empirical measure of the robustness and quality of interval estimators is the coverage of confidence intervals (see [JO00]). The coverage of confidence intervals will be denoted here with $C$. $C$ is here defined as the frequency with which the confidence interval $(\bar{x}(n) - \Delta_x, \bar{x}(n) + \Delta_x)$ contains the true $\mu$, at a given confidence level $(1-\alpha)$. For $0<\alpha<1$. 

80 Alcatel Bell N.V.
In order to be able to do a performance test of confidence methods a test experiment has to be defined. It seems to be a good idea to use a system of which true means $\mu$ of metrics can be calculated mathematically and compared with the confidence intervals determined by confidence methods. The choice is made to use a simple M/M/1 queue as a test system.

**Test system**

The test that is used is based on a M/M/1/$\infty$ queuing system of fig 7.3. Since the coverage analysis requires that the exact characteristics of the simulated processes are known, the M/M/1/$\infty$ queue was found suitable. An M/M/1/$\infty$ queue has a Poisson stream of arriving customers, an exponentially distributed service time, one server and infinite buffer capacity. Customers will be denoted here as packets. From a given arrival rate $\lambda$ and service rate $\mu$, the expected queue occupation and delay can be calculated. Note that from the viewpoint of system dynamics, the M/M/1/$\infty$ queue can be considered as a worst case system. The M/M/1/$\infty$ queue behaves very dynamically, especially at high loads ($p$). It can therefore be considered as a good example system in relation to the simulations performed on hardware systems, which can be expected to have less dynamic characteristics. The delay of the customers in the M/M/1 queue can be estimated by calculating the average of the samples taken at dequeue of the customers. The delay metric is a sample average. The delay of customers in the queue is therefore used as metric to test the confidence methods.

![Queueing System Diagram](image)

**Fig. 7.3: M/M/1/$\infty$ queuing system used for coverage analysis**

**Coverage test**

The test for coverage is based on the standard statistical coverage analysis as discussed in [JO00]. To test the coverage of a confidence method, the method is applied to a sufficient number of independently repeated simulation experiments. For the simulations the minimum standard random number generator is used (see appendix A for an explanation on random number generators). The independently repeated simulations were performed using non-overlapping streams of random numbers for every simulation. Therefore independent sequences of pseudo-random numbers are used. As a result of these experiments the coverage estimate $\hat{C}$ is given by:

$$\hat{C} = \frac{\text{number of experiments in which the confidence interval estimate contains the true mean}}{\text{total number of independent experiments}}$$

(7.17)

Fig. 7.4 shows how the confidence intervals determined with a certain method are checked for coverage. For each generated interval there is checked whether the interval contains the true mean $\mu$. In the shown example the coverage is 5/6 or 83.33%.
If the number of experiments is sufficiently large then \( \hat{p} \) can be regarded as approximately normally distributed and a confidence interval of \( C \) can be determined by applying the standard formula for confidence intervals for proportion (see formula 7.18). This formula can only be used for a large set of samples \( n \) (\( n > 100 \)).

\[
\hat{p} - z_{1-\alpha/2} \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} < p < \hat{p} + z_{1-\alpha/2} \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}
\]  

(7.18)

Where \( \hat{p} \) is the number of percentage of successes in a binomial experiment, \( z_{1-\alpha/2} \) is the \((1-\alpha/2)\) quantile of the standard normal distribution at a confidence level of (1-\( \alpha \)). And \( n \) is the sufficiently large number of experiments. For \( C \) the confidence interval is given in eqn. (7.18).

\[
\left( \hat{C} - z_{1-\alpha/2} \sqrt{\frac{\hat{C}(1-\hat{C})}{n_{\hat{C}}}}, \hat{C} + z_{1-\alpha/2} \sqrt{\frac{\hat{C}(1-\hat{C})}{n_{\hat{C}}}} \right)
\]  

(7.19)

\( n_{\hat{C}} \) must be sufficiently large. A total of 1000 experiments can be considered large enough. The following test conditions were used in the experiments:

- Each experiment is performed with different random seeds selected with a seed entry table, thus using non-overlapping streams of random numbers. (see Appendix A for justification). The linear congruential random number generator with multiplier \( 7^5=16807 \) and modulus \( 2^{31}-1 \) is used.
- Simulations are run sequentially, meaning that a simulation is stopped when the used confidence method has evaluated the required precision. Simulations are stopped as soon as the results reached a relative precision of at least \( \varepsilon_{\text{max}} = 0.01 \) at the 95% (\( \alpha = 0.05 \)) confidence level.
- Simulations are performed at least 1000 times at each chosen working point.
- If required by the method, initial data in each simulation is discarded in order to eliminate the effects of the initial transient.
- The mean-delay of packets in the M/M/1/\( \infty \) queue is used as metric to determine the confidence intervals in each experiment. Reason for choosing the mean delay as test metric is that it can be estimated with a sample-average.
- The mean-delay is measured for the following M/M/1/\( \infty \) queue loads:
  \[
p = \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}
\]

Remark: Instead of using a fixed number of experiments (1000), coverage experiments could be performed until a required precision (relative error) is reached regarding the confidence intervals for \( C \). However, in practice it is experienced that for queue loads \( p > 0.5 \) and a significance level of 1% far more than 1000 experiments are needed (>5000). For 5% significance >2000 experiments are needed. The needed experiments for this test take for these loads an excessive amount of simulation time on a normal PC and is therefore not used.
Coverage analysis test outcome interpretation:
Of a confidence method, used for determining the 100(1-\(\alpha\))% confidence intervals of a \(\mu_x\), the confidence interval of its coverage \(C\) is e.g. determined by \((c_1, c_2)\). The confidence method is considered valid if:

\[
\text{The upper limit } c_2 \text{ of the confidence interval of its coverage } C \text{ equals at least } (1-\alpha).
\]

It essentially means that it is required that the confidence interval of the coverage \(C\) for the evaluated test system parameters, contains the required coverage \((1-\alpha)\). If not, then the method is considered inadequate for the applied test system parameters.

Discarded samples in initial transient:
Samples in the initial transient are discarded in order to assure that coverage experiments are done in the steady state phase of the simulation. For all of the methods tested, with exception of the classical method (7.4.3) and the regenerative cycles method (7.8), the number of initially deleted samples are selected with the Schreuben algorithm, which is discussed in section 8.5.1.

Coverage of the classical method of confidence intervals
To show the importance of the use of confidence methods that deal with the autocorrelation of observations in a simulation, the coverage experiment is performed on the method that assumes that observations are not correlated. This method is called here the classical method of confidence intervals. The following algorithm is used in the independent method:

<table>
<thead>
<tr>
<th>Algorithm:</th>
<th>Simulation input:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical method</td>
<td>95% confidence interval: (z_{1-\alpha/2} = 1.96), relative precision (\varepsilon_{\text{max}} = 0.01)</td>
</tr>
</tbody>
</table>

Per new metric sample the following calculations are performed:

\[
\begin{align*}
\overline{X}_n &= \frac{\overline{X}_{n-1} \cdot (n-1) + X_n}{n} \quad \text{with } X_n \text{ the } n^{\text{th}} \text{ (current) metric sample} \\
\overline{X}_n^2 &= \frac{X_{n-1}^2 \cdot (n-1) + X_n^2}{n}
\end{align*}
\]

Every time the confidence interval of a observed metric is constructed, the following calculations made:

Variance: \(\sigma^2[\overline{X}_n] = \overline{X}_n^2 - (\overline{X}_n)^2\)

Precision: \(\Delta_x = \frac{z_{1-\alpha/2} \cdot \sigma[\overline{X}(n)]}{\sqrt{n}}\)

Relative precision: \(\varepsilon = \frac{\Delta_x}{\overline{X}(n)}\)

Check: if \(\varepsilon \leq \varepsilon_{\text{max}}\) then Confident := true; \(\Rightarrow\) Stop simulation

Confidence interval:

\[
\left(\overline{X}(n) - z_{1-\alpha/2} \cdot \frac{\sigma[\overline{X}(n)]}{\sqrt{n}}, \overline{X}(n) + z_{1-\alpha/2} \cdot \frac{\sigma[\overline{X}(n)]}{\sqrt{n}}\right)
\]

Fig. 7.5: Algorithm for creating confidence intervals with classical method

The confidence intervals produced by the classical method are used to determine the required simulation run length. Simulations are stopped when the relative precision is reached. With the classical method, initial transient detection is not used and therefore initial transient observations are not discarded. The coverage
experiment of the classical method is repeated 1000 times per load \( p \in \{0.1; 0.2, \ldots, 0.9\} \) for a \( M/M/1 \) queue. The results of the coverage analysis is shown in fig. 7.6.

![Coverage Classical method of confidence intervals](image)

**Fig. 7.6: Coverage results of the classical method of confidence intervals**

The graph of fig. 7.6 shown on the X-axis the load at which the experiments are performed, whereas on the Y-axis the obtained coverage \( C \) is shown. The required (ideal) coverage is shown (0.95 or 95%) as a horizontal line. The graph of the coverage is shown with its confidence intervals regarding the coverage \( C \). Applying Rule 1, in none of the simulated work points (loads \( p \)) the confidence interval of the coverage covers the required coverage of 0.95. The classical method is not a valid one for the simulated loads. For small loads (e.g. \( p < 0.2 \)) the method has a coverage, which is still within an acceptable range. However, most simulations are expected to be performed with systems with higher utilizations. For moderate loads (let's say \( p = 0.5 - 0.7 \)) the method produces in too many cases unreliable confidence intervals. The conclusion that can be made from this is that with the classical method of confidence intervals reliable confidence intervals can hardly be produced when using the precision for determining simulation length. In general it was noticed that the determined simulation run length by the classical method is too short in most cases.

Note that the confidence intervals created by a method have a chance of \( 1 - \alpha \) to contain the true average \( \mu_x \): 

\[
P(\mu_x \in [\bar{X}(n) - \Delta_x, \bar{X}(n) + \Delta_x]) = 1 - \alpha
\]

Produced confidence intervals also have a chance of \( \alpha \) not to contain the true average \( \mu_x \): 

\[
P(\mu_x \notin [\bar{X}(n) - \Delta_x, \bar{X}(n) + \Delta_x]) = \alpha
\]

In any experiment performed using a confidence method, the true average \( \mu_x \) is either within the confidence interval determined, or not. A perfect confidence method would contain the true average \( \mu_x \) in \((1 - \alpha)\) 100% of the cases over an infinite number of experiments. A way of testing a method is to perform a large number of experiments and evaluate whether produced confidence intervals contain the true average \( \mu_x \) (coverage experiment). The relative number of times the produced confidence intervals contain the true \( \mu_x \), is the experimental coverage. A confidence interval of the coverage experiment can be constructed. This confidence interval of the coverage has a chance of 0.95 (95% confidence intervals) to contain the required coverage \( C \) (here \( C = 0.95 \), since \( 1 - \alpha = 0.95 \)). If the coverage confidence intervals do not contain the required coverage \( C \), then the method is considered inadequate.
7.5 Fixed sample length simulation

With fixed sample length simulation the length of a simulation must be known in advance. Simulations are performed with a fixed length, where the length of the simulation is determined by a certain amount of simulation time or the number of gathered samples. The method of Independent replications repeats these fixed length simulations $m$ times and calculates estimates and confidence intervals using eqn. 7.10 and 7.11. The method is discussed here since it is widely used with discrete event simulations. To demonstrate the effect of making more replications of a simulation run, the method is evaluated for several values of $m$.

The simulation experiments can be characterized as follows:

Experiment 1:
{Independent Replications on M/M/1 queue; $m=5$; $T_a=1.0$; $T_s=\{0.1, ..., 0.9\}$; $T_{\text{end}}=10^{6}$}

Experiment 2:
{Independent Replications on M/M/1 queue; $m=10$; $T_a=1.0$; $T_s=\{0.1, ..., 0.9\}$; $T_{\text{end}}=10^{7}$}

Note 1: $T_a$ = the average time between arrivals of customers in the queue; $T_s$ = the average waiting time between service of customers; $T_{\text{end}}$ = simulation length in seconds of model time.

Note 2: the chosen simulation run length ($T_{\text{end}}$) is an arbitrary, but reasonable choice.

Note 3: for each replication different random number generator seeds are used.

Fig. 7.7 and fig. 7.8 show the results of a coverage analysis experiment performed on the method of independent replications for $m=5$ (5 simulation runs) and $m=10$ (10 simulation runs) respectively. The graphs show that for $m=5$ the method performs well for $p<0.4$. For higher loads the method has unreliable behavior. For $m=10$ the coverage shows some improvement. However, the method in the experiment 1 and 2 uses a too small simulation run-length for higher loads ($p$) and therefore provides unreliable confidence intervals for these loads. For $m=10$ the method improves, producing acceptable coverage for loads $p<0.5$. What can be learnt from this is that making more replications (using different random number generator seeds) can produce more reliable confidence intervals (better coverage), but coverage is not likely to be improved by more replications if a simulation is too short.

![Coverage Independent Replications method (m = 5)
(1000 x MM1 queue simulation)](image)

Fig. 7.7: Coverage results of the classical method of confidence intervals: experiment1
Fig. 7.8: Coverage results of the classical method of confidence intervals: experiment 2

The major drawbacks regarding the method are:

- Since the number of replications is usually rather small, the variances of the estimates can be large and in that case the produced confidence intervals are large.
- When an appropriate simulation run length is not known the method can produce seemingly reliable confidence intervals, but these intervals can be too large and can have poor coverage. Reliable decisions made, based on these intervals, are difficult to make.
- The total run length needed for the method is $m$ times the fixed simulation run length. Usually one is not willing to make that many replications.

The method is attractive due to its simple operation, however its reliability (coverage) can hardly be guaranteed and the method asks a deeper a priori knowledge of the system from the user than the methods presented in chapter 8. The required simulation run-length must be known.

A method often used by model designers to determine simulation run-length is a visual inspection of the distribution plot of an observed metric. The simulation run-length is with this ‘method’ considered sufficiently long enough when the distribution plot becomes stable. Stable means in that case that the shape and values of the distribution remain ‘relatively’ constant when additional metric samples are gathered. The distribution of a performance metrics is however likely to change continuously during simulation, even when a simulation is performed long enough for a certain confidence level. Gathered new observations change estimate and variance (note that the variance is proportional to $1/n$).

The determination of simulation run-length based on visual inspection of the distribution of metrics must be considered to be an unreliable method and it is therefore not advised to use this method.
8 Simulation output analysis: confidence intervals

This chapter presents in detail the two methods that are worked out and implemented in a developed performance analysis library (PA library) that can be used for general purpose with discrete event simulations.

8.1 Sequential procedures for steady-state simulation

The main problem with fixed sample size simulation nothing is known of the required simulation run length. Sequential precision simulation tries to deal with this problem. The two methods selected, can be used for sequential precision simulation:

- Regeneration points
- Batch means

Fig. 8.1 shows the basic operation of a simulation that works with the PA library. Sequential simulation exists basically of two stages. The first stage deals with the initial transient, whereas the second stage is the actual steady state simulation. The second stage deals with the problem of autocorrelation between gathered samples. With the method of regenerative cycles the initial transient detection involves the discarding the initial observations until the first recurrent point is reached. With the method of batch means, all observations in the initial transient need to be discarded. After start of a simulation from its initial state the initial transient is detected. If the initial transient involves to many observations the simulation must be stopped. Data involving the initial transient is outputted for analysis purposes (i.e. the number of discarded observations).

![Diagram](https://example.com/diagram.png)

**Fig. 8.1:** Sequential method for data collection and analysis existing of two stages

After detection of the initial transient (the steady state reached) the actual estimation of performance metrics begins. The simulation is run until all estimated performance metrics have acceptable confidence intervals (required precision reached). A simulation can be stopped by two events: 1. All confidence intervals of the estimates have the required precision or 2. The passing of a specific amount of events or simulation time that is set as a maximum.
In order to be able to use sequential precision simulation both phases of fig. 8.9 need to be supported. An algorithm that deals with the first phase (detection of the initial transient) is given in section 8.10. The second phase is dealt with by the batch means method and the method of regenerative cycles.

8.2 Batch means

As explained in section 7.3.2.2, the method of batch means uses the grouping of observations to overcome the fact that sequential observations are autocorrelated. The major question with the use of the batch means is the length \( m \) of the batches. In order to obtain insight in the influence of the batch length on the coverage of the method, a number of coverage experiments are performed for various batch lengths. With these experiments the observations in the initial transient are discarded. The algorithm to determine the number of initial transient observations is provided in section 8.10. Fig. 8.2 shows the coverage experiment done for a fixed batch size of \( m=10 \), \( m=100 \) and \( m=1000 \) observations.
Fig. 8.2: Coverage analysis for batch means method with batch size (a)10, (b)100, (c)1000

The graphs show that the coverage of the method improves as the batches become larger (note that the graphs all have different scales). For a batch size of m=10 the method has reliable coverage for loads up to p=0.3. For m=100 the batch means method produces reliable confidence intervals for loads of p<0.6, whereas for m=1000 the method produces valid intervals for all loads p<0.8). Fig. 8.3 shows the coverage for m=2500. For all tested loads the method performed well with this batch length.

Fig. 8.3: Coverage batch means for m=2500

It is shown here that a larger value for m leads to better coverage. It shows that the means of larger batches, containing more spread observations, are less correlated and therefore more statistically independent. It is important to note that with the batch means for every simulation at least 30 batches have to be collected in order to be able to treat their averages as normally distributed. It seems desirable to just use the batch length of m=2500. However for this batch length an considerable amount of observations is needed (>75·10³). Furthermore, the coverage of the batch means using a certain batch length is system dependent. Therefore it is better to develop an algorithm that selects the appropriate batch length dependent on the simulation. The autocorrelation between batch means can be used for this as is shown in the new algorithm for batch means of section 8.9.

Note: for higher loads larger batch sizes are needed and for low loads small batch sizes can be used.

Alcatel Bell N.V.
8.3 Method of regenerative cycles

The method of regenerative cycles is discussed in section 7.3.2.3. The algorithm used with this method is shown in fig. 8.4.

<table>
<thead>
<tr>
<th>Algorithm:</th>
<th>Simulation input:</th>
</tr>
</thead>
<tbody>
<tr>
<td>regenerative cycles</td>
<td>Recurrent point ( r ); 95% confidence interval: ( z_{1-\alpha/2} = 1.96 ), relative precision ( \varepsilon_{\text{max}} = 0.01 )</td>
</tr>
</tbody>
</table>

Batches of data are collected between recurrent points. For each batch:

\[
\bar{Y}_{nr} = \frac{1}{k_r - k_{r-1} - 1} \sum_{s=k_{r-1}+1}^{k_r} Y_s, \quad \text{The sum} \quad \bar{Y}_{s} \quad \text{of samples} \quad Y_s \quad \text{in one batch of length} \quad L_{nr} \quad \text{between recurrent points} \quad r_r \quad \text{and} \quad r_{r-1}, \quad \text{with} \quad L_{nr} = k_r - k_{r-1} \quad \text{and with} \quad k_r \quad \text{the first observation of the next cycle},
\]

Per new found recurrent point \( r \) the following calculations are performed:

\[
\bar{Y}_{nr} = \frac{n_r}{n_r - 1} \bar{Y}_{nr} - \frac{1}{n_r} \bar{Y}_{nr}^2, \quad \bar{L}_{nr} = \frac{n_r}{n_r - 1} \bar{L}_{nr} - \frac{1}{n_r} \bar{L}_{nr}^2.
\]

\[
\bar{Y}_{nr-1} \cdot (n_r - 1) + \bar{Y}_{nr} \quad \text{Per check for confidence (construction of the confidence interval) calculations made: Estimation:} \quad E = \frac{\bar{Y}_{nr}}{\bar{L}_{nr}}
\]

\[
\text{Variance:} \quad \bar{\delta}[E] = \left( \frac{n_r}{n_r - 1} \right) \left[ \bar{Y}_{nr}^2 - 2E \cdot \bar{Y}_{nr} + E^2 \bar{L}_{nr}^2 \right], \quad \text{term} \quad \left( \frac{n_r}{n_r - 1} \right) \quad \text{is used to obtain an unbiased} \quad \bar{\delta}[E]
\]

\[
\Delta_x = z_{1-\alpha/2} \cdot \bar{\delta}[E] \quad \text{and relative precision:} \quad \varepsilon = \frac{\Delta_x}{X(n)}
\]

\[
\text{Check: if} \quad (\varepsilon \leq \varepsilon_{\text{max}}) \quad \text{then} \quad \text{Confident} = \text{true} ; \quad \Rightarrow \text{Stop simulation}
\]

\[
\text{Confidence interval:} \quad (E - \Delta_x, E + \Delta_x)
\]

Fig. 8.4: Algorithm for the regenerative cycles method using (classical estimators)

The method of regenerative cycles is applied using the classical estimators of eqn. 7.15 and 7.16. The main issue with the method is the choice of recurrent point. In order to obtain as many recurrent points as possible (to obtain as much batch means as possible), a recurrent point must be chosen with high probability of occurrence during a simulation. Fig. 8.5 shows the relative frequency of occurrence of several queue occupation levels at arrival of a new customer in the M/M/1 queue in a single simulation. From fig. 8.5 (simulation time is here 1EB sec) can be seen that an occupation level of 0 is the most frequent and is a good choice as recurrent point. The question arises whether this level is the best choice. Fig. 8.6 shows for the recurrent points of fig 7.5, the number of used samples and used batches. The simulation results are produced using the algorithm of fig. 8.4 and averaged for 100 simulations for a load of \( p = 0.5 \) and 0.7.
The graph shows that for $p=0.7$ and for the recurrent point occupation 0 a large number of batches (approx. $5\times10^9$) is used that are relatively short ($m=3.5$) compared to other recurrent points. For the recurrent point of 22 with a low relative frequency of occurrence (queue occupation at arrival of customer) the batch length is long ($1.2\times10^4$), whereas the number of used batches is $=200$. Note that fig. 8.6 has log-scale on the vertical axis. The total number of samples needed per recurrent point is also shown in the graph. The influence of the choice of recurrent point regarding the simulated time for an M/M/1 queue simulation with load $p=0.7$ is given in fig. 8.7. What can be learnt from this graph and the total samples amount of samples needed per recurrent point (fig. 8.6) is that for the simulation the choice of recurrent point has little influence on the simulated time for recurrent points with relative frequent occurrence. Batch length is traded in for the number of required batches. For recurrent points with a very small relative frequency of occurrence, the simulated time is larger and becomes higher as the minimum of 30 batches is reached.
It is interesting to know the coverage of the regenerative cycles method with respect to the chosen recurrent point. Fig. 8.8 shows the coverage for a set of recurrent points for a M/M/1 queue with load $p=0.7$ coverage experiment. The number of simulations performed here is 1000. The graph shows that choice of recurrent point can have influence of the coverage, but that is for the investigated recurrent points the coverage is good. For coverage analysis, the delay of packets in the M/M/1 queue is observed. The regeneration points for the process $\{D_i, i \geq 1\}$ are the indices of the packets that arrive at the queue with occupation $r$. In other words the packets arriving in a queue filled with $r$ customers is the recurrent point and their delay is used as first sample of the next batch. The minimum run length is set to 30 batches.

The following can be concluded from performed experiments and fig. 8.5 through 8.8:

- The choice of recurrent point has influence on the batch size and the number of batches needed in order to produce a confidence interval with required precision.
- The length of a simulation regarding the number of used metric samples is determined by the chosen recurrent point. However, for recurrent points that have frequent occurrence this influence is very small. The simulation length is constant for recurrent points $0\sim20$ for the M/M/1 simulation. Recurrent points that have very low chance of occurring can lead to long simulation times. For every simulation at least 30 batch means are needed (for normality) to determine the confidence intervals.
- Fig. 8.14 shows that for a recurrent point \( r > 30 \) (queue occupation) and load \( p = 0.7 \), the method is influenced by the boundary of 30 batches. The simulation run-length (in samples) increases for these recurrent points that have less probability of occurrence (lower frequency of occurrence as given in fig. 8.5). What can be concluded from this is that it is better to choose a recurrent point with frequent occurrence.

Note that results of fig. 8.8 show that the coverage for the recurrent points \( r = 0 \) and 1 the coverage is good despite of the short batches.

### 8.3.1 Jackknife estimator versus classical estimator

An estimator that is claimed to be less biased than the classical estimator for the ratios of expectations is the jackknife estimator [LK82, LK90, CL99]. This estimator is used with the method of recurrent points. Eqn. 8.1 gives the estimator \( \hat{\mu}_j \), whereas eqn. 8.2 gives the variance of \( \hat{\mu}_j \). Note that \( \mu_c \) is the classical estimator determined according eqn. 7.15.

\[
\hat{\mu}_j = n\mu_c - \frac{(n-1)}{n} \sum_{i=1}^{n} \hat{\mu}_{-i} \quad \text{where} \quad \hat{\mu}_{-i} = \frac{\sum_{i \neq j} Y_j}{L_j} \tag{8.1}
\]

or \( \mu_j \) and the variance of \( \mu_j \) no recursive formulas could be derived, therefore using the jackknife estimator has the disadvantage that all values \( Y_i \) and \( L_i \) need to be stored (in a linked list of values) and evaluated with eqn. 8.2 and 8.3 each time the confidence interval needs to be constructed. In the simulations performed, the delay of customers in a queue is evaluated (e.g. the coverage experiments). The recurrent point used, is the filling level of the queue. \( Y_i \) is the experienced latency by customers in a queue. \( L_i \) is the number of customers in a batch, where a batch is determined by a sequence of customers. A batch starts with a customer arriving at a particular queue filling level (recurrent point \( r \)) and ending with the last customer before a next customer finds the same particular queue filling level. The \( n \) in eqn. 8.1 is the number of batches gathered during the simulation at the instance eqn. 8.1 is evaluated.

To compare the coverage of the cases in which classical and jackknife are used fig. 8.9 gives the coverage of using the classical method for various loads of the \( M/M/1 \) queue and fig. 8.10 the coverage for the case the jackknife estimator is used. For the coverage a recurrent point \( r = 0 \) is used.

Both the classical as the jackknife estimator show acceptable performance. The classical approach has a coverage that is relatively constant in the performed experiments. The jackknife estimator performs very well for higher loads, but less with the lower loads (<0.3). Note that with all the experiments the confidence intervals are probabilistic experiments and statements need to be made carefully.

Comparing the two methods, the preferred estimator is the jackknife estimator (since it is known that is less biased). However the major disadvantage is that no recursive algorithm is available for the estimator, making its implementation in the performance analysis library not very efficient.
8.4 Method of uncorrelated batches

8.4.1 Investigation of the problem

Section 8.2 discussed the problem of selection of the batch size m. As shown the coverage improves as the batches become larger. It is interesting to know if the correlation between batch means becomes smaller for larger batches and if the correlation can be used to determine an appropriate batch size. The idea behind this is that, starting from an initial batch size \( m_0 \), the batch size can be increased until the correlation has reached an acceptable level.
Correlation between the batch means can be calculated with (using the estimators of the autocorrelation coefficients in [LK82]):

\[
\hat{r}(k,m) = \frac{\hat{R}(k,m)}{R(0,m)}
\]  

(8.3) \hspace{2cm} \hat{R}(k,m) = \frac{1}{k_b - k} \sum_{i=k+1}^{k_b} \left[ \bar{X}_i(m) - \bar{X}(m) \right] \left[ \bar{X}_{i-k}(m) - \bar{X}(m) \right]

(8.4)

Where \( \hat{r}(k,m) \) is the autocorrelation of lag \( k \) for batch size \( m \); \( \hat{R}(k,m) \) the autocovariance; \( \bar{X}(m) \) the overall means of the evaluated batches. Furthermore \( k_b \) is the number of collected batches, \( m \) is the used batch size and \( k \) the lag of the autocorrelation. The lag \( k \) is defined as the distance between two samples \( X_i \) and \( X_{i+k} \).

It is generally accepted (see [WA98]) that the lag \( k \) should be smaller than \( 1/10 \) of the maximum number of evaluated samples \( N \), thus: \( k \leq N/10 \). The choice is made to use a fixed number of batch means \( k_b = 100 \) to evaluate the autocorrelation between batches of maximum lag \( k=10 \) (see fig. 8.11). A sliding window is used to calculate the autocorrelations of lag 1,2,\ldots,\( k \). These autocorrelations are averaged over one simulation.

**Sliding window of batch means of observations**

\[ \frac{X_L(m)}{X_1(m)} \]

**Fig. 8.11: Autocorrelations calculated of a range of \( k_b=100 \) batch means**

![Autocorrelation of batch means with classical method](image)

**Fig. 8.12: Autocorrelation of batch means as function of the batch size determined with classical estimator**

Fig. 8.11 shows the autocorrelations for lag 1,2 and 10 as function of the batch length averaged over a simulation of a M/M/1 queue using the following simulation parameters:

\{M/M/1 queue; load \( p=0.7 \); \( T_a=1.0 \); \( T_s=0.7 \); \( T_{end}=1E7 \)\}.

The autocorrelation between batched is clearly positive for all evaluated lags (as expected).

The variance of autocorrelation coefficients can be calculated with eqn. 8.5 [WA98] and is used to construct the confidence intervals of the lags \( k=1, 2, 10 \):

\[
\sigma^2[\hat{r}(k,m)] = \begin{cases} 
\frac{1}{k_b} & \text{for } k=1 \\
\frac{1}{k_b} \left[ 1 + 2 \sum_{i=1}^{k-1} \hat{r}^2(k,m) \right] & \text{for } k > 1 
\end{cases}
\]

(8.5)
From fig. 8.11 can be seen that the correlation for the evaluated lags drops as the batches become larger. Furthermore it is clear that the autocorrelation of lag $k=1$ is larger in all cases than the other lags. In [LK82] the less biased jackknife estimator for estimating autocorrelation is given:

$$ r(k, m) = 2 \cdot \hat{r}(k, m) - \hat{r'}(k, m) - \hat{r''}(k, m) $$

In this formula, $k$ is again the lag and $m$ the size of the batch. $\hat{r}(k, m)$ is the classical estimator of lag $k$ calculated over all $kb$ available batch means, $\hat{r'}(k, m)$ the classical estimator for the first $kb/2$ batch means, $\hat{r''}(k, m)$ the estimator calculated over the last $kb/2$ batch means and $\hat{r}(k, m)$ the jackknife estimator. Note that $kb$ must be even (here chosen 100).

The experiment given with fig. 8.12 is performed using the jackknife estimator. Results are shown in Fig. 8.12.

![Autocorrelation of batch means with Jackknife estimator](image)

**Fig. 8.12: Autocorrelation of batch means as function of the batch size determined with jackknife estimator**

The jackknife estimator produces more sensitive results regarding the batch size and the lag. Since it is also less biased, the jackknife estimator is chosen to be used in an algorithm to determine the batch size.

### 8.4.2 The uncorrelated batches algorithm

What can be learnt from fig. 8.11 and 8.12 is that the relatively small batch size of $m=10$ has high autocorrelation for all lags and that for batch size $m=10000$ the autocorrelation is low. What also can be learnt from the diagrams is that after a batch size $m=1000$ the changes in autocorrelation are hard to detect.

Based on this an algorithm is developed. The method concerning this algorithm will be called uncorrelated batches.

In order to create the algorithm a number of parameters had to be selected.

- It seems sensible to choose an initial batch size $m_0 = 50$ (based on fig. 8.12).
- Lags $k=1,2,\ldots,10$ are evaluated for 100 batch means ($kb=100$)
- Batched are found un-autocorrelated at significance $\alpha$ if all lags $k$ give statistical negligible values. Defined is $\beta$ as the significance level per lag. $\beta$ is determined using Bonferroni inequality [WA98]: $\beta \geq \alpha/\text{LAG}$, where LAG is the evaluated number of lags (here $\text{LAG}=10$). Significance $\alpha$ is chosen $0.1$ here, meaning that $\beta=0.01$. 

96

Alcatel Bell N.V.
Fig. 8.13 gives the algorithm for the uncorrelated batches method. The algorithm assumes that observations gathered in the initial transient are discarded. The algorithm has two phases. In the first phase, the batch size determination phase, an appropriate batch size is determined and selected. In the second phase the selected batch size is used to determine the point estimators and the confidence interval of the measured performance metric.

**Batch size determination phase**

During this phase samples are collected and a batch means is calculated over the initial batch size $m_0$. These batch means are stored in a buffer (linked list) of batch means which will be called BatchSequence. The BatchSequence contains the values $\bar{X}_1(m_0), \ldots$, see fig. 8.14. When a sequence of $kb=100$ batch means is gathered, the batch means are stored in a buffer of size $kb$ that is used to analyze the autocorrelation within the sequence. The algorithm to test for autocorrelation is given in fig. 8.23.

If the batches in CorrelationSequence are found uncorrelated, the batch size is selected $s\cdot m_0$, where $s$ initially equals 1. The batch size determination phase is then over.
If the sequence of batch means fails the test for autocorrelation; s is increased and an additional \( kb \) batch means of size \( m_0 \) have to be collected. After collection, the batch means in the BatchSequence are grouped to form again \( kb \) batch means that are stored the CorrelationSequence. After that the sequence is again tested for autocorrelation, until the sequence in CorrelationSequence is found un-autocorrelated. Notice that CorrelationSequence remains a fixed sized buffer, whereas BatchSequence becomes larger. After the batches are found uncorrelated, the sequence stored in CorrelationSequence is used to make a first estimation of the performance metric. The BatchSequence is deleted.

(Remark: in the implementation of the algorithm CorrelationSequence is an array and BatchSequence a linked list).

**Algorithm:**

**Input:**

- \( \alpha = 0.1 \), \( \text{LAG}=10 \), \( \beta = \alpha / \text{LAG} = 0.01 \), \( z_{1-\beta/2} = 2.57 \)

(Tests of autocorrelations between batch means for a given batch size, with \( \beta \) significance level of \( k \) tests of autocorrelation)

UnCorrelated := true;

\[ k = 1 \]

**while** \( k < \text{LAG} \) and Uncorrelated **do**

\( \{ \text{test whether all LAG autocorrelation coefficients are statistically negligible each at the } \beta \text{ significance level.} \}

Calculate the jackknife estimator \( \hat{r}(k, sm_0, kb) \) of the autocorrelation coefficient of lag \( k \) for the sequence of batch means \( X_1(sm_0), X_2(sm_0), \ldots, X_{kb}(sm_0) \) stored in the CorrelationSequence. Here \( \hat{r}(k, sm_0, kb/2) \) is the autocorrelation coefficient calculated over the first \( kb/2 \) batches and \( \hat{r}(k, sm_0, kb/2) \) over the last \( kb/2 \) batches in CorrelationSequence.

\[
\hat{r}(k, sm_0, kb) := 2 \cdot \hat{r}(k, sm_0, kb) - \frac{\hat{r}(k, sm_0, kb/2) + \hat{r}(k, sm_0, kb/2)}{2}
\]  

(8.7)
(Calculate the variance of the autocorrelation coefficient of lag \( k \))

\[
\sigma^2[\hat{r}(k, sm_0, kb)] = \begin{cases} 
\frac{1}{k_b} & \text{for } k = 1 \\
\frac{1}{k_b} \left[ 1 + 2 \sum_{i=1}^{k_b} \hat{r}^2(k, sm_0, kb) \right] & \text{for } k > 1 
\end{cases}
\] (8.8)

\{ Hypothesis test : \\
\( H_0 \): batches of lag \( k \leq (1-\beta)100\% \) uncorrelated \\
\( H_1 \): batches of lag \( k > (1-\beta)100\% \) uncorrelated \\
\}

Test statistic : 
\[
Z = \frac{\hat{r}(k, sm_0, kb) - \mu_n}{\sigma[\hat{r}(k, sm_0, kb)]} \quad \text{With:} \ E[\hat{r}(k, sm_0, kb)] = \mu_n = 0 
\] (8.9)

\( H_0 \) rejected if 
\[
|Z| < z_{1-\beta/2} \implies \left| \frac{\hat{r}(k, sm_0, kb)}{\sigma[\hat{r}(k, sm_0, kb)]} \right| < z_{1-\beta/2} 
\] (8.10)

With \( z_{1-\beta/2} \) is the upper \( 1 - \beta/2 \) critical point of the standard normal distribution.

\}

if \( \left| \frac{\hat{r}(k, sm_0, kb)}{\sigma[\hat{r}(k, sm_0, kb)]} \right| < z_{1-\beta/2} \cdot \sigma[\hat{r}(k, sm_0, kb)] \) then
\{ \( H_0 \) is rejected. The lag \( k \) autocorrelation is statistically negligible at the confidence level \( 1 - \beta \) \}
UnCorrelated := true;
else UnCorrelated := false;
endif

\( k := k + 1; \)
enddo

if (UnCorrelated = true) then
\{ Accept the current batch size \}
\( m := sm_0; \)
endif

\textbf{Fig. 8.15: Algorithm for test for autocorrelation}

The shown algorithm basically checks for every lag whether the sequence of batch means in CorrelationSequence are un-autocorrelated. Input to the test is the required significance level \( \alpha = 0.1 \), and the evaluated maximum lag \( \text{LAG} \). The algorithm calculates for every lag \( k \) the jackknife estimator of the autocorrelation coefficient \( \hat{r}(k, sm_0, kb) \) for the \( kb \) batches of size \( sm_0 \) stored in CorrelationSequence, eqn 8.22. With this \( \hat{r}(k, m, kb) \) is autocorrelation coefficient of lag \( k \) between the \( kb \) means of batches of size \( m \). Some additional explanation:

\[
\hat{r}(k, m, kb) = \frac{\hat{R}(k, m, kb)}{\hat{R}(0, m, kb)} \quad (8.11) \quad \text{with} \quad \hat{R}(k, m, kb) = \frac{1}{k_b - k} \sum_{i=k+1}^{k_b} \left[ \bar{X}_i(m) - \bar{X}(k_b, m) \right] \left[ \bar{X}_{i-k}(m) - \bar{X}(k_b, m) \right] 
\] (8.12)

With \( \hat{R}(k, m, kb) \) the autocovariance of lag \( k \) \((k = 0, 1, 3, \ldots)\) in the sequence of batch means \( \bar{X}_i(m), \bar{X}_2(m), \bar{X}_3(m) \)

Eqn. 8.7 is used to calculate the variance of the jackknife estimator of the autocorrelation coefficient (see also eqn. 8.5. With estimator and variance the test statistic \( Z \) is calculated. Note that the expectation of \( \hat{r}(k, sm_0, kb) \) is \( 0 \) \((\mu_n = 0)\). The hypothesis test \( H_0: >(1-\beta)100\% \text{ correlated} \) fails if \( |Z| < z_{1-\beta/2} \). If the test fails for one lag, the CorrelatedSequence is considered correlated, else uncorrelated.

It is noticed that in most cases the influence of the first lag \( k=1 \) is the largest on the selection of a batch size.
Estimation phase

At the start of the estimation, \( kb \) batches of size \( m \) are available that can be used to determine a confidence interval:

\[
\overline{X}_n = \frac{1}{kb} \sum_{i=1}^{kb} X_i, \quad n=kb, \quad \overline{X}_n^2 = \frac{1}{kb} \sum_{i=1}^{kb} \overline{X}_i^2
\]

Additional batches of size \( m \) are gathered in order to construct the confidence intervals and reach the required precision.

<table>
<thead>
<tr>
<th>Algorithm:</th>
<th>Simulation input:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimation phase</td>
<td>95% confidence interval: ( z_{1-\alpha/2} = 1.96 ), relative precision ( \varepsilon_{\text{max}} = 0.01 )</td>
</tr>
</tbody>
</table>

Every new batch of samples the following calculations are performed:

\[
\overline{X}_n = \frac{1}{m} \sum_{i=1}^{m} X_i \quad \text{(in fact } \overline{X}_i = \frac{\overline{X}_{i-1} \cdot (i-1) + \overline{X}_i}{i} \text{ is repeated for } i=0 \text{ to } m) \]

Every new batch means the following calculations are made: \( n:=n+1 \)

\[
\overline{X}_n = \frac{\overline{X}_{n-1} \cdot (n-1) + \overline{X}_n}{n} \quad \text{with } \overline{X}_n \text{ the } n^{th} \text{ (current) batch means, } \overline{X}_n^2 = \frac{\overline{X}_{n-1}^2 \cdot (n-1) + \overline{X}_n^2}{n}
\]

Per check for confidence (construction of the confidence interval at checkpoint) calculations made:

Variance:

\[
\sigma^2[\overline{X}_n] = \overline{X}_n^2 - \left(\overline{X}_n\right)^2
\]

Precision:

\[
\Delta_x = z_{1-\alpha/2} \cdot \frac{\sigma[\overline{X}(n)]}{\sqrt{n}} \quad \text{and relative precision: } \varepsilon = \frac{\Delta_x}{\overline{X}(n)}
\]

Check: \( \text{if}(\varepsilon \leq \varepsilon_{\text{max}}) \text{ then } \text{Confident := true; } \Rightarrow \text{Stop simulation} \)

Confidence interval:

\[
\left( \overline{X}(n) - z_{1-\alpha/2} \cdot \frac{\sigma[\overline{X}(n)]}{\sqrt{n}}, \overline{X}(n) + z_{1-\alpha/2} \cdot \frac{\sigma[\overline{X}(n)]}{\sqrt{n}} \right)
\]

8.4.3 Coverage analysis

Fig. 8.16 shows the results of a coverage experiment for the uncorrelated batches method for a initial batch sizes \( m_0 = \{50, 100\} \) and a significance level \( \alpha = 0.1 \). The coverage experiment is based on 1000 replications.

What can be seen from fig. 8.16 is that the algorithm performs acceptable for an initial batch size of 50, but is better for the batch size \( m_0 = 100 \) (as expected).

Table shows an indication of the performance of the algorithm. With a larger initial batch length, the eventually used batch size becomes larger, which can be seen as a disadvantage since a simulation will last longer (notice that the batch size determination stage needs \( kb \times sm_0 \) samples).
For the method of uncorrelated batches, the control parameters \( k_b \) and \( \alpha \) need further investigation.

### 8.5 Detection of the initial transient

After starting a simulation, the simulation model is initially in a non-stationary phase. Steady state simulation is however aimed at the behavior of a system (e.g. DataFlow) in its steady state. Observations gathered during the transient phase are results of random variables whose distributions are different from those in steady-state. Or as stated in [LK82], an observation \( x_i \) in the transient phase can be modeled as: 

\[
x_i = \mu_x + b_i + \epsilon_i,
\]

with \( b_i + \epsilon_i \) the difference between the sample and the true mean \( \mu_x \), and \( b_i \) the bias in this difference. Using this sample and calculating a means can lead to an unknown bias \( b(n) \) in the steady state mean: 

\[
E[\overline{X}(n)] = \mu_x + b.
\]
Since observations gathered during the initial transient period do not characterize the steady state, an appropriate way of dealing with this is to discard all observations during this period. This requires an estimation of the length of the initial transient period. Disadvantage is that the removal of observations can increase the variance of the estimate. A larger variance gives larger and worse confidence intervals. Since bias in a sample mean is difficult to detect and variance can be decreased by performing longer simulations, the choice is made to discard observations in the initial transient period.

8.5.1 Algorithm

Basic problems related to the existence of the initial transient and the detection of the end of it, is given in [PA90]. In this reference an algorithm based on Schreuben test statistics is given for automatic detection of the initial transient is described. The algorithm is presented in fig. 8.17. In appendix G gives a more detailed description of the algorithm and its implementation. The algorithm follows the following steps:

A rough first approximation of the number \( n_0 \) of initial observations is that should be discarded is obtained by applying a heuristic rule (heuristic phase). The rule states:

\[
\text{The initial transient period is over after } n_0 \text{ observations if the time series } x_1, x_2, \ldots, x_{n_0} \text{ crosses the mean } \overline{X(n_0)} \text{ } k \text{ times.}
\]

This rule is sensitive to the value of \( k \). A too large value for \( k \) will lead to an overestimated value of \( n_0 \) regardless of the system load. A too small value for \( k \) can result in an underestimated value of \( n_0 \) in more heavily loaded systems. It is known that \( k \) must be 25 for highly dynamic systems as the M/M/1 queue.

Following the first rough selection of the end of the initial transient \( (n_0^*) \), the length of the initial transient is more precisely determined by applying a Schreuben test statistic for testing stationarity (Schreuben phase). The \( n_0^* \) initial samples are discarded and an additional sequence of observations of length \( n \) is gathered. For an explanation of variables see appendix G. See fig. 8.18 for explanation of algorithm operation.

Of the newly collected \( n \) observations the last \( n_0 \) are used to determine \( \delta[\overline{X}(n_0)] \), which is needed for the Schreuben test statistic \( T \). The \( n_0 \) observations are used to calculate eqn. 8.12.

\[
T = \frac{\sqrt{45}}{n_0^{1.5} \cdot \frac{25}{\delta(\overline{X}(n_0))}} \sum_{k=1}^{n_0} k \left( 1 - \frac{k}{n_0} \right) \left| \frac{\overline{X}(n_0) - \overline{X}(k)}{\overline{X}(n_0)} \right| \quad \text{with} \quad \overline{X}(i) = \sum_{j=n_0+1}^{n_0+i} \frac{x_j}{i}
\]  

(8.13)

If the \( T \) test accepts the hypothesis that the initial transient is detected correctly \( (|T| \leq z_{1-\alpha/2}) \) and the process has already entered the stationary phase, the gathering of steady state samples can begin. If the test fails an \( \Delta_n = \gamma \cdot n_0^* \) samples are discarded from the beginning of the \( n \) observations are discarded and \( \Delta_n \) new observations are gathered to form a new sequence of length \( n_0 \) used for analysis. Fig. 8.26 shows the process for \( N \) cycles in which the stationarity test fails.
Fig. 8.17: Schreuben algorithm for detection of initial transient

The procedure is repeated until a sequence $n_0$ is found stationary or a certain maximum is reached (Transient too long). After detecting stationarity, the collection of observations for the steady state simulation can begin.

**8.5.2 Benchmarking the Schreuben algorithm**

In order to test the quality of Schreuben algorithm performance the following test was used: fig 8.19 shows a filtering procedure using a moving average filter with length $2w+1$ (window length $w$) that is used to filter simulation data and to determine the end of the initial transient visually.
The procedure operates as follows: 1. m separate simulations are made each obtaining I observations (and using m different random number generator seeds). 2. Corresponding observations with the same order number in the m simulations are averaged to form the stream $\bar{Y}_1, \bar{Y}_2, \ldots, \bar{Y}_I$. 3. This stream is used as input to a causal moving average filter with window length w (note that this filter has a delay w). For the first nsw samples applied to the filter the multiplication factor $A$ is dependent of n (where n is the number of the current sample applied to the filter). The effect of the filter is shown in fig. 8.19 through 8.21. Fig. 8.19 shows the delay of a packet experienced with a uniform arrival and service process (GI/G/1 queue) for a load $p=0.7$ obtained in a single simulation run. The arrival process is uniformly distributed between 0 and 1.0, whereas the service process is distributed between 0 and 0.7.

The graph shows the detection of the end of the initial transient by the heuristic rule and the detection phase of the Schreuben algorithm for stationarity (also shown the long term average of the simulation metric). Fig. 8.20 shows the same simulated process, but in this case the filter procedure is applied, with $m=20$ replications and a window length for the moving average filter $w=100$). It shows that the heuristic phase determines the end of the initial transient correctly (notice that the correction for the delay w of the moving average filter is taken into account) and the Schreuben phase is at its shortest length (= 200 samples).
Fig. 8.21: Visual comparison Schreuben algorithm and filter procedure output

What can be learned from this is that the Schreuben algorithm is capable of determining the start of the stationarity phase of simulation output. Fig. 8.22 shows the operation of the heuristic rule on the same process, showing the average XAV calculated at moment that it is crossed $k=25$ times. Remark: in the implementation samples in the heuristic phase need to be stored and re-evaluated every time the crossings need to be counted. Fig. 8.23 gives the results of the application of the Schreuben algorithm to 200x M/M/1 queue simulations, showing also the output of the applied filter procedure. Due to the dynamic nature of the queue, the output is more dynamic than in fig. 8.21, hence the visual determination of the end of the initial transient is more difficult. However, the Schreuben algorithm shows proper performance.

Fig. 8.22: Crossings counted in the heuristic phase
Fig. 8.23: Schreuben algorithm and filter procedure applied to M/M/1 queue simulation

A comparison between the performance of the Schreuben algorithm and the estimation made with the filter procedure is made in fig. 8.24 for M/M/1 simulations at various loads. Each simulation is performed 200 times. Visual estimation is generally made conservatively. For high loads \( p > 0.6 \) it is difficult to determine the end of the initial transient visually. Too less filtering makes the graph too variable, too much filtering over aggregates the simulation output.

Fig. 8.24: The mean length of the initial transient period measured in the number of observations generated during the initial transient by the Schreuben algorithm, compared to the visual estimation with filter procedure

8.6 Discussion explored solutions

The methods of recurrent points and uncorrelated batch means are both selected as candidates and implemented in the PA library, presented in chapter 9. Regarding both methods the coverage as a function of the normalized load \( p \) is determined (both methods controlling simulation run length). As seen the correlation between observations is an important aspect (as clearly demonstrated in fig. 8.6), especially for the higher traffic levels. Despite the fact that all other investigated methods take into account existing correlations, the
uncorrelated batch means method and the method of regenerative cycles were found most suitable (despite that candidates as spectral analysis most likely perform better, but need much storage space and calculation times).

- The method of regenerative cycles appears to be the most attractive one to use, since it requires no initial transient detection algorithm. Furthermore it is shown that the simulation run length is under influence of the relative frequency of the chosen recurrent point, but that the effect is small for regeneration points that have relatively high frequencies of occurrence. The coverage is good when using the jackknife estimator. However, no recursive formula could be derived and therefore application of this estimator requires much storage space. The classical estimator shows acceptable coverage for lower loads and reasonable coverage for the higher traffic levels. The major problem with the regenerative cycles method is the selection of the recurrent point \( r \). With a single queue system this is in most cases easy, but with multiple queues a recurrent point with high frequency of occurrence is mostly difficult to discover. A typical recurrent point is the system state in which all queues are empty (simultaneously), however with the simulation models used in the experiments (the hardware models of DataFlow) this recurrent state has a low frequency of occurrence. The influence of the choice of recurrent point on the coverage is investigated for some recurrent points, however additional research is needed on this issue to gain more insight.

- The method of uncorrelated batches is a very good candidate for practical implementation. Acceptable coverage is obtained with this method for lower traffic loads. For higher loads the initial batch size needs to be increased (default value \( m_0 = 50 \)). With this initial batch length, the method needs at least 5000 samples (\( kb = 100 \)) which is acceptable. The Schreuben algorithm is used together with the method of uncorrelated batches to determine the end of the initial transient.

The selected methods are compared in fig. 8.25 regarding the average needed samples in 200x M/M/1 queue simulations at relevant traffic loads. For the method of regenerative cycles the empty queue found at arrival of a customer is chosen as recurrent point. The method of uncorrelated batches uses the Schreuben algorithm (discarded samples are accounted for), \( \alpha = 0.1 \) and \( m_0 = 100 \). Fig 8.25 shows (notice the log-scale of the Y-axis) that the method of regenerative cycles is only cheaper regarding the needed samples for lower loads. For \( p > 0.3 \), the method of uncorrelated batch means needs less samples.

![Fig. 8.25: Samples needed per method at various loads](image)
9 Application of confidence analysis in hardware simulation

9.1 PA library

The practical implementation involves a PA library that can be used in all C++ based DES. Appendix E gives the structure of the PA library in form of a UML model. The library is a separate module with well-defined interfaces. All statistical analysis is incorporated in the library and shielded of from the simulation model that uses the library through clear interfacing. Therefore the user of the library is freed from statistical analysis. Fig. 9.1 shows the general flow of a simulation and the interfacing between a performance simulation model and the PA library.

![Flowchart of PA library and performance simulation](image)

**Fig. 9.1: Interfacing PA library and performance simulation**

Basically the library exists of an active metrics-manager object (separate thread) that controls the simulation. The metrics-manager uses a confidence analyser to check the confidence of the observed metrics. Multiple confidence methods can be used (selectable per metric).

Samples are sent by the simulation model to the performance library module through `ProcessMetricSample(metricNum, Value)`, whereas the PA library provides feedback to the simulation model regarding the relative precision of the performance metrics (`ConfidenceSimFinished()`). Furthermore random numbers are provided in disjoint streams (≤1000 streams) through `GetRandomNumber(stream)`.

The PA library uses checkpoints in order to maintain simulation speed (let the performance library not slowdown the simulation). Fig. 9.2 shows how the streams of observations are handled internally of the performance library. The PA library uses an analyser to check the precision of the confidence intervals of metrics at consecutive checkpoints, until the required precision is reached. The metrics manager ports metric samples provided by the simulation to an instance of a confidence method class (type of class is predefined). For each metric observed a confidence method object exists. The metrics are handled according the algorithms discussed.

The metric manager incorporated in the library controls the simulation progress. The metrics manager provides the simulation process with information regarding the confidence of the observed metrics. Furthermore, the metrics manager provides the random numbers to the simulation. The metric manager orders the analyser to check the confidence precision of the metrics at consecutive times (confidence is not monitored constantly to speed up the simulation). When all metrics are found confident, the simulation is stopped. Checkpoints at which confidence is not yet reached are shown grey, black checkpoints mark a confident metric. Remark: notice that the collection of observations can start at different moments in time (followed firstly by the initial transient).
From [LK00] it is known that the evaluation of multiple performance metrics and hence construction of confidence intervals, is subject to the problem that the chance of \( n \) confidence intervals containing the true average \( \mu_x \) (for \( \alpha = 0.95 \) confidence intervals) is not \( 1 - \alpha \), but \( 1 - n \cdot \alpha \). In order to create \( 1 - \alpha \) confidence intervals for \( n \) metrics evaluated in a simulation, the confidence significance per metric must be \( 1 - \alpha \), with \( 1 - \alpha \leq 1 - \sum_{i=1}^{n} i \cdot \alpha \), (note that \( \alpha \) does not have to be equal for all metrics). This means that for \( n = 5 \) (5 evaluated metrics) and required confidence level of 0.95 per metric, confidence intervals for \( 1 - \alpha \) = 0.99 must be constructed per metric. To cope with this problem, the number of evaluated metrics must be kept small or confidence intervals with lower confidence must be accepted. In the performance analysis library estimates and confidence intervals are produced until all metrics are confident. Although confidence intervals tend to become smaller as the simulation continues (gathering of new observations continues even when metrics are found confident) confidence intervals are still of the \( 1 - \alpha \) confidence level. Notice that by checking the confidence not continuously, but only at certain points spaced with a predefined number of samples, simulation speed is maintained by keeping the frequency of construction of confidence in intervals as low as possible.

### 9.2 Hardware performance simulations

The confidence method discussed and implemented in the created PA library can be applied in performance simulations regarding hardware systems. A corner case example of a simulation involving the DataFlow system is given here. Regarding the DataFlow system, it is interesting to know the EPC throughput and the occupation and latency of the GQueue, MCBRead and MCBWrite. Fig. 9.3 shows the relevant modules and queues within the simulation model involved. The GQueue, MCBRead and MCBWrite are part of the EPController that is subject of interest, whereas the EPC is a separate processor. The extended modeling framework is used to model the system (see clock trigger provided to all modules). In the simulation case studied the focus is on Tolly-test conditions where a traffic level of 1 Erlang is applied to the model at various IP frame lengths. The case evaluated is when a stream of frames of 42 bytes (40 bytes IP payload plus 2 bytes overhead) is applied to the system. Although the EPC is a separate chip, the EPC is in agreement with the extended modeling framework applied with the same clock as the other modules and queues in the system. The model has the following parameters:

- Frames 40 bytes IP payload applied to single port receiver.
- GQueue elasticity queue size is large (>1000 FCBAs)
The average rate of the FCBAs applied by the Receiver to the EPC GQueue elasticity queue is 1.6 for 40 bytes IP payload input frames at 1 Erlang load due to read access grants provided by the memory arbiter.

The influence of EPC performance on model operation is investigated here. With L=28 and ΔL=3. The EPC can process 48 FCB at a time. A FCB in the EPC has a latency that cannot be smaller than its timestamp, but it can be larger due to head of line blocking (HOL) [TA96] caused by conservation of FCB order (a FCB with a ready timestamp not the head of the queue cannot leave the queue) or by a full MCBWrite FIFO. Notice that MCBWrite and MCBRead need to access the DataStore memory (not shown here). Using the following calculation one can predict the effect of this and the used frame length of 40 bytes on performance of the queues.

\[
\bar{F} \cdot \bar{L} \leq 48
\]  \hspace{1cm} (9.1)

With \( \bar{L} \) average latency. Thus: \( \bar{L} \leq \frac{48}{1.61} \Rightarrow \bar{L} \leq 29.77 \). If the experienced latency (by the FCBAs) exceeds 29.77 Windows, the expectation is that MCBRead FIFO will build up in time and that eventually FCBAs will be dropped if GQueue becomes full. A frame is enqueued in MCBWrite by the EPC if the frame has spent

Fig. 9.3: Relevant queues in a simulation model regarding EPCController

Input frame rate to the EPC (the elasticity queue GQueue has in proper operation no influence on this rate): average frame-rate \( \bar{F} = 1.61 \text{ frames/Window} = 1.61 \times 40\text{B}/66\text{ns} = 7.8\text{Gbs} \) (elementary window cycle period is 66ns). The maximum frame rate \( F_{\text{max}} = 2 \text{ frames/Window} \). Since the number of threads in the EPC is limited to 48:

\[
\bar{F} \cdot \bar{L} \leq 48
\]  \hspace{1cm} (9.1)

With \( \bar{L} \) average latency. Thus: \( \bar{L} \leq \frac{48}{1.61} \Rightarrow \bar{L} \leq 29.77 \). If the experienced latency (by the FCBAs) exceeds 29.77 Windows, the expectation is that MCBRead FIFO will build up in time and that eventually FCBAs will be dropped if GQueue becomes full. A frame is enqueued in MCBWrite by the EPC if the frame has spent
at least a number of windows equal to its timestamp in the EPC, is at the head of the queue and there are less than 48 frames in MCBWrite. The latency $L$ of the EPC is distributed in the interval $[25, 31]$, with average $L=28$. However, the simulated average latency of FCBs in the EPC is 29.49 Windows. Under the given circumstances, where the simulated EPC latency is close to allowable maximum of 29.77 windows, it expected that queue buildup will occur in MCBRead, MCBWrite and GQueue, which can not be recuperated. This queue buildup is caused by:

- EPC is ready to accept a FCB from MCBRead, but no FCB in MCBRead is ready for dispatch to the EPC (due to read accesses to DataStore memory by the MCBRead queue granted by the Arbiter).
- The latency of the FCBs in the EPC is frequently larger than the allowed maximum (29.77 windows).

Fig. 9.4 shows that latency experienced by frames in EPC is frequently larger than allowed.

Fig. 9.5 shows the effect on the occupation of the queues. In the shown diagram, also the detection of the end of the initial transient determined by the Schreuben algorithm is given, indicated by the small vertical lined crossing the graphs. The MCBRead queue starts building up FCBAs since the EPC is frequently not eligible to accept a FCBA. Notice that the EPC in the model provides the GQueue with credits to write new FCBAs to the MCBRead when it has idle threads. The moment less credits are given to the GQueue, the GQueue starts building up FCBs. This relieves momentarily MCBRead. During a small period the Schreuben algorithm detects stationarity and recognizes the end of the initial transient (wrong). Shortly after this MCBRead starts building up again until it reaches its maximum. From that moment on all new FCBs written by the Receiver are stacked up in the elasticity GQueue and eventually frames will be lost if GQueue runs over. What can be seen here is that the Schreuben algorithm detects the end of the initial transient in MCBRead incorrectly. Also with the GQueue the initial transient is determined incorrectly. For MCBWrite and EPC the end of initial transient is determined correctly. Latency is shown in fig. 9.6. The graph shows that the latency of the EPC is on average about 29.5 (as expected). The latency of messages in MCBRead grows until a steady situation is reached (queue full) in which on average the same number of FCBAs enter as leaves the queue.

Confidence results obtained from the PA library are given in table 9.1 determined with the uncorrelated batch means and regenerative cycles method at same simulation run length. The recurrent point was chosen a queue occupation level at close to expected average (level at FCBA arrival for latency, at dequeue for occupation).
Fig. 9.5: Occupation of queues as function of simulated time

Fig. 9.6: Latency experienced by frames in EPC, MCBRead and MCBWrite

Table 9.1a: Confidence output information PA library: uncorrelated batch means (after (checkpoint) 50000 Window cycles)
Table 9.1b: Confidence output information PA library: regenerative cycles (after checkpoint) 50000 Window cycles

Tables 9.1a and b show that the two methods give different results. Since the recurrent point has to be chosen, a first simulation run is needed to determine an adequate recurrent point. On the other hand, incorrect interpretation of the end of initial transient can be avoided.

![Confidence evaluated at checkpoints](image1)

![Relative precision at check points](image2)

Fig. 9.7: (a) Confidence checked at checkpoints; (b) relative precision

Demonstrated in fig. 9.7a is how confidence is frequently checked (at checkpoints) by the Analyzer (method of uncorrelated batches used). Relative precision evaluated at these checkpoints (fig. 9.7b). The sudden drop of relative precision is caused by the fact that the used uncorrelated batches method a set of batch means is required to test for autocorrelation. During batch size estimation phase relative precision is not checked. In the cases shown, immediately after batch size is determined, the analyzed sequence to determine the appropriate batch size holds enough batch means to construct reliable confidence intervals. In these cases too much samples are gathered for the required precision. This is a side effect of the uncorrelated batch means.

Fig. 9.8 demonstrates how the PA library is used in UML-RT simulation.
9.3 Conclusions regarding confidence analysis

Stochastic simulation is a statistical experiment. The type of simulations that are of interest regarding the performance of hardware are steady-state simulations. Or in other words, long term cumulative measure of performance as mean/max values and probabilities of events. Estimates obtained from simulation metrics (observations collected during simulation runs) have to be properly, statistically analyzed.

Unfortunately, determination of steady state estimators is difficult because the output data of a simulation is usually correlated. Moreover, a simulation model typically has a transient phase before reaching its steady state. The main aspects of stochastic discrete event simulation are discussed, such as steady state simulation, random number generators and probability and non-determinism. The problem of the initial transient and output analysis methods based on confidence intervals are discussed. Several methods to determine confidence intervals are discussed. Of these methods, the method of batch means and regenerative cycles are found most suitable to use with the simulations of hardware systems, but also for application with network simulators such as Opnet modeler. A coverage test to benchmark and compare confidence methods is introduced based on a system with known (and analytically estimable) characteristics, the M/M/1 queue is used. Simulations with this coverage are performed a large number of times, to estimate the coverage or the percentage of confidence intervals produced by the method that covers the true, known mean of the parameter investigated. The experiments are performed for relevant loads of the M/M/1 queue. Confidence intervals of the coverage are constructed.

With the coverage test, existing methods, as the classical method for constructing confidence intervals, the independent replications method and the method of batch means are benchmarked. The results are conclusive: the classical method for determination of confidence intervals is a very poor performing method and coverage is unacceptable for every load. The method of independent replications is used with many discrete event simulators, e.g. Opnet and is a fixed length simulation method. Simulation run length must be known in advance. The coverage analysis of this method shows that coverage will improve if more replications are used. A major danger with this method is that it can produce confidence intervals that appear to be valid but that are too large (too large variance) or result in poor coverage. Simulations are then clearly too short. With the
method of independent replications the required simulation run length to obtain reliable results is difficult, if not impossible, to determine. Making more replications can improve the coverage if simulations are performed long enough. To create reliable confidence intervals at least five replications are needed. This makes the method unattractive to use, since simulation run length will be five times as long as with a sequential method.

An appropriate way of determining simulation run length and hence reliable confidence intervals is sequential simulation. With this approach the precision of a confidence interval is used as a stopping rule to determine if the end of a simulation is reached (meaning that the confidence intervals of all evaluated performance metrics are sufficiently small).

The method of regenerative cycles needs a recurrent point can be used as sequential procedure for simulation termination. Since the method uses recurrent points at which a system starts a-fresh, the method is theoretically perfect. At entering the recurrent point the collection of data for a new batch begins. The empty state of a queue is a possible recurrent point. Simulations show that the choice of recurrent point has little influence on the simulation run length for frequently occurring recurrent points. For less frequent recurrent points the simulation time will increase (at least 30 batches are needed). The method shows good coverage for the investigated recurrent points. Using the less biased jackknife estimator gives better coverage, but this estimator has the disadvantage that no recursive formula is available (means from past batches need to be stored). Advantage of the method is that no initial transient detection is needed.

An algorithm is created that uses the autocorrelation between batch mean to determine an appropriate batch size. The jackknife estimator for autocorrelations is chosen above the classical estimator, since it is more sensitive to batch length change in the beginning of a batch length determination run. The algorithm uses efficient data structures in the batch size determination phase. The coverage of the method is reasonable for an initial batch size of 50, but better for larger initial batch sizes. The method needs at least \( m_0 \) samples in the batch size determination phase (a trade off between used samples and coverage). The method needs initial transient detection.

The Schreuben algorithm detects initial transient. First phase involves a heuristic phase, the second phase tests for stationarity. Samples in the initial transient are discarded. The end of initial transient can also be determined visually, by using a moving average filter. Comparing the two methods, the proper operation of the Schreuben algorithm is shown.

The developed library for performance analysis can be used in corporation with any discrete event simulation (in C++). The significant advantage is that any existing simulator written in C++ or capable of importing C++ routines can make use of the PA library. Furthermore, it is suitable for simulation of hardware system modeled with the extended modeling framework and can also be used for network simulation as are for instance performed with the DES Opnet modeler. The random number generator of the library provides 1000 disjoint sub-sequences of pseudo-random numbers.
10 Conclusions and recommendations

PART I
In the first part of this thesis it is investigated how UML can be used in the design flow of hardware systems. The choice is made to use UML for performance modeling, since performance modeling and simulation are important mediators between design of functionality and selected hardware resources and are important input to the further stages in the design flow. Discussion and documentation play an important role. Starting point of the research done is the currently used performance modeling framework which uses the C++ language.

The modeling of hardware systems requires methods that can address the characteristics of hardware, as timing, concurrency, behavior and system architecture. Requirements involving these characteristic and general performance modeling aspects are specified. Based on these requirements performance modeling methods are evaluated. No UML based methods were found that support the modeling of hardware. Exception to this is SHE, which incorporates UML compliant diagrams.

From the work done involving the modeling of the DataFlow network processor with UML can be concluded that UML can be used to model hardware, but there are a few drawbacks:

- UML is a modeling language that needs heuristics in order to enable the use of this language for modeling of hardware. Based on performance models created of the DataFlow hardware system a set of six UML diagrams is selected and a modeling framework is introduced for performance modeling. The focus of this framework is on the modeling of requirements that involve performance modeling, on formulation of design solutions and on formalization of conceptual models into an executable performance model. With the developed modeling framework, performance models of DataFlow are created. SHE provides a more consistent set of diagrams than provided by UML. The UML diagrams selected for the performance modeling framework are similar to the diagrams provided in SHE. This confirms the choice of diagrams for the performance modeling framework. Unfortunately, currently no tool is available to create SHE diagrams. The evaluated UML tools offer possibilities for UML syntax and consistency checking. The tools are however not mature yet.

- Performance modeling requires possibilities to create executable models. UML has not the required semantics to let it be used as a visual programming language. In order to create executable models object-oriented code is needed in addition to UML models. It is shown that some UML tools offer possibilities for the creation of executable models through use of State Charts. Although State Charts are in theory an attractive way of modeling behavior, the offered implementations in the evaluated tools are found unsuitable for the modeling of hardware systems. The evaluated tools lack the required support of timing and concurrency needed to create executable hardware performance models. Furthermore, modeling behavior with State Charts as provided with the evaluated tools does not support currently used performance modeling methods.

The currently used performance modeling framework can be used to create synchronous models of hardware systems, using two separate model clocks. Hardware components are modeled as synchronous concurrent modules. Since this approach has proven to work well and matches the designs made of hardware systems like DataFlow, this approach is also used in the UML modeling of DataFlow. However, the currently used modeling framework had to be extended to make it clearer where and when which of the clock triggers has to be used. Furthermore, through the created extensions, the model execution schedule has become clearer and hierarchical models can be created. The extensions are evaluated in C++ and in UML-RT using the RoseRT tool. Both applications of the extended modeling framework have proven to work well. The advantage of using UML-RT for performance modeling is that the UML models created with the performance modeling framework can be made executable in RoseRT. Disadvantage is that models have to be provided with a clocking module that
provides the necessary scheduling. This violates the requirement that model and scheduling mechanism must be separated. Moreover, RPCs are allowed and can be difficult to schedule. Furthermore, the requirement of encapsulation is violated, which is mainly caused by the fact that channel splitting is not supported in the RoseRT tool. The extended modeling framework can be used to model systems that have synchronous concurrent modules correctly. In order to model asynchronous concurrent modules, special measures have to be taken as random latency FIFO’s.

In POOSL hardware specific aspects like timing and concurrency are covered. POOSL covers all requirements specified regarding modeling methods. It is demonstrated that with POOSL synchronous concurrent hardware models can be modeled. A comparison between the extended modeling framework and POOSL is made. In POOSL the execution scheme is implicitly defined by the POOSL language and therefore no scheduling mechanism has to be provided separately in a model. Both the extended modeling framework as the POOSL can be used to model hardware systems like DataFlow, however POOSL has the advantage that asynchronous concurrency is supported.

PART II
Performance simulations of hardware systems are steady-state simulations. Confidence intervals can be used to determine the required simulation run-length in order to obtain reliable simulation results (sequential simulation). With the coverage test, existing methods for constructing confidence intervals, as the classical method, the independent replications method and the method of batch means are benchmarked. The results are conclusive: the classical method for determination of confidence intervals is a very poor performing method and coverage is unacceptable for every load. The method of independent replications is used with many discrete event simulators, e.g. Opnet and is a fixed length simulation method. Simulation run length must be known in advance. With the method of independent replications the required simulation run length to obtain reliable results is difficult, if not impossible, to determine.

The method of regenerative cycles needs a recurrent points at which a system starts a-fresh. The method is theoretically perfect. Simulations show that the choice of recurrent point has little influence on the simulation run length for frequently occurring recurrent points. For less frequent recurrent points the simulation time will increase. The method shows good coverage for the investigated recurrent points. Using the less biased jackknife estimator gives better coverage, but this estimator needs in an implementation much storage space. Advantage of the method is that no initial transient detection is needed.

An algorithm is created that uses the autocorrelation between batch mean to determine an appropriate batch size. The jackknife estimator for autocorrelations is chosen above the classical estimator, since it is more sensitive to batch length change in the beginning of a batch length determination run. The algorithm uses efficient data structures in the batch size determination phase and needs initial transient detection. Coverage experiments show satisfying results for this method. The Schreuben algorithm detects initial transient. Evaluation of the algorithm has shown the proper operation. The developed library for performance analysis can be used in corporation with any discrete event simulation (in C++).

Recommendations for further research regarding performance modeling with UML are:
- Investigate the possibilities of using UML in other phases of the design flow.
- Combine the UML models used for performance modeling with models of other phases in the design flow. Tuning is needed between UML models created for software design and the extended modeling framework. The expectation is that this will not be a complex problem.
- The clocking module has to be incorporated in all performance models. Only when using POOSL this can be avoided. It needs to be investigated whether the scheduling mechanism provided by the clocking module can be built into the used tools (like RoseRT). Other performance modeling projects need to be conducted to test the validity of the extended modeling framework.
Regarding confidence analysis, some additional issues for further research are:

- More research is needed on how to perform sequentially performed coverage experiments, instead of fixed length simulation.
- More research on the influence of the recurrent state on the coverage and the simulation run length is needed. It is shown that the used recurrent point influences the coverage of the method. For complex systems the choice of recurrent point can be difficult. It is currently not clear how to chose a recurrent point with respect to frequency of occurrence of the recurrent point, the simulation run-length and the coverage.
- Evaluated methods deal with sample averages. Other metrics than sample averages need to be investigated.
- Recursive formula for the Jackknife estimator is needed.
Bibliography

Books

[AL90] Allen, S.
PROBABILITY, STATISTICS AND QUEUEING THEORY WITH COMPUTER SCIENCE APPLICATIONS

[BR99] Booch, G. and J. Rumbaugh, I. Jacobson
THE UNIFIED MODELING LANGUAGE USER GUIDE
Harlow: Addison-Wesley, 1999

[DO99] Douglass, B.
DOING HARD TIME: DEVELOPING REAL-TIME SYSTEMS WITH UML, OBJECTS, FRAMEWORKS, AND PATTERNS

[JO00] Johnson, R.
MILLER & FREUND'S PROBABILITY AND STATISTICS FOR ENGINEERS, sixth edition
Upper Saddle River: Prentice Hall, 2000

[LK82] Law, A. and W. Kelton
SIMULATION MODELING AND ANALYSIS, first edition

[LK00] Law, A. and W. Kelton
SIMULATION MODELING AND ANALYSIS, third edition

[PV97] Putten, P. van der and J. Voeten
SPECIFICATION OF REACTIVE HARDWARE/SOFTWARE SYSTEMS: THE METHOD SOFTWARE/HARDWARE ENGINEERING (SHE)

REAL-TIME OBJECT-ORIENTED MODELING

[TA98] Tanenbaum, A.
Computer networks, 3rd edition

[WAS98] Wadsworth, H.
HANDBOOK OF STATISTICAL METHODS FOR ENGINEERS AND SCIENTISTS

[WS96] Ziegler, J. And M. Sawad
OCTOPUS -OBJECT-ORIENTED TECHNOLOGY IN REAL-TIME SYSTEMS
Harlow: Addison-Wesley, 1996.

Reports

[NIO0a] Niemegeers, A.
PERFORMANCE MODELLING AND ARCHITECTURE EXPLORATION - METHODOLOGY AND TECHNIQUES
Alcatel Bell, Antwerp, internal report 3EZ XXXX BEZZA01

[NIO0b] Niemegeers, A.
DATAFLOW ARBITER FUNCTION MODEL
Alcatel Bell, Antwerp, internal report 3EZ 01504 AAAA BE
[RA00] Kruchten, P.
RATIONAL UNIFIED PROCESS
URL: http://www.rational.com/uml

[RS00] Rumbaugh, J. and B. Selic
USING UML FOR MODELING COMPLEX REAL TIME SYSTEMS
URL: http://www.rational.com/uml

[TH00a] Theelen, B and J. Voeten
GUIDELINES FOR SYSTEM-LEVEL METHODS AND TOOLS-PERFORMANCE ANALYSIS FOR
SYSTEM-LEVEL DESIGN
Alcatel-TUE Frame Project – Project: Performance Analysis

[VR00a] Vrana, M.
DATAFLOW CHIP FUNCTIONAL DESCRIPTION
Alcatel Bell, Antwerp, internal report 3EZ 01478 AAAA EBZZA

[VR00b] Vrana, M.
TRS-DataFlow chip.
Alcatel Bell, Antwerp, internal report 3EZ 01478 AAAA DTZZA

Articles

[CA98] Calvez, J.
A FOUR-STEP PARTITIONING PROCESS FOR SYSTEM DESIGN
In: Proc. International Conference and Workshop on Engineering of Computer Based Systems,

BOOTSTRAP CONFIDENCE INTERVALS FOR RATIOS OF EXPECTATIONS
In: ACM Transactions on modelling and computer simulation, vol. 9(1999), no.4, p.326-348

[DO98] Douglass, B.
DESIGNING REAL-TIME SYSTEMS WITH UML II.

[HE99] Herzberg, D.
UML-RT AS A CANDIDATE FOR MODELING EMBEDDED REAL-TIME SYSTEMS IN THE
TELECOMMUNICATION

MODELLING OF REAL-TIME EMBEDDED SYSTEMS IN AN OBJECT-ORIENTED DESIGN
ENVIRONMENT WITH UML

[LE00] L'Ecuyer, P.
GOOD PARAMETERS AND IMPLEMENTATIONS FOR COMBINED MULTIPLE RECURSIVE
RANDOM NUMBER GENERATORS

120
Alcatel Bell N.V.
Manuscript, Montreal, Canada, 2000.
URL: www.iro.umontreal.ca
Recent date: 20 June 2000.

[LP99] Lee, J. and K. Pawlikowski, D. McNickle
SEQUENTIAL STEADY STATE SIMULATION: RULES OF THUMB FOR IMPROVING ACCURACY
OF THE FINAL RESULTS

[ME00] Mendelbaum, D.
JAVA-PROTOTYPING OF HARDWARE/SOFTWARE CBS USING A BEHAVIOURAL OBJECT
ORIENTED MODEL
In: Proc. 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed

[NJ00] Niemegeers, A. and G. de Jong
AN INCREMENTAL SPECIFICATION FLOW FOR REAL TIME EMBEDDED SYSTEMS [VDSL MODEMS]

[NJ99] Niemegeers, A. and G. de Jong
EARLY SYSTEM (CO-)VALIDATION BASED ON A HIERARCHICAL SPECIFICATION FLOW.
In: Proc. IEEE International High Level Design Validation and Test Workshop, 1999, San Diego Bay

[PA99] Pawlikowski, K.
SIMULATION STUDIES OF TELECOMMUNICATION NETWORKS AND THEIR CREDIBILITY

[PA90] Pawlikowski, K.
STEADY STATE SIMULATION OF QUEUEING PROCESSES: A SURVEY OF PROBLEMS AND
SOLUTIONS
In: ACM Computing Surveys, vol. 22, no. 2 (June)

[PM99] Pasquier, O. and F. Muller, J. Calvez, D. Heller, E. Chenard
THE MCSE APPROACH FOR SYSTEM LEVEL DESIGN.

[PV98a] Putten, P. van der and J. Voeten, M. Geilen, M. Stevens
OBJECT-ORIENTED ANALYSIS AND SYSTEM LEVEL ARCHITECTURE.
In: Proc. 9th Annual ProRISC/IEEE Workshop on Circuits, Systems and Signal Processing,

[PV98b] Putten, P. van der and J. Voeten, M. Geilen, M. Stevens
SYSTEM LEVEL DESIGN METHODOLOGY
In: Proc. IEEE Computer Society Workshop on VLSI'98 System Level Design, Orlando

[PM88] Park, S. and K. Miller
RANDOM NUMBER GENERATORS: GOOD ONES ARE HARD TO FIND

[SE99] Selic, B.
PROTOCOLS AND PORTS: REUSABLE INTER-OBJECT BEHAVIOR PATTERNS
In: Proc. 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing,


[TH00b] Theelen, B. and J. Voeten, L. van Bokhoven, P. van der Putten, A. Niemegeers, G. de Jong
PERFORMANCE MODELING IN THE LARGE: A CASE STUDY

[VP98] Voeten, J. and Putten, P van der, M. Geilen, M. Stevens
SYSTEM LEVEL MODELLING FOR HARDWARE/SOFTWARE SYSTEMS

Webpages

[JAVA] www.java.sun.com
[POOSL] www.ics.ele.tue.nl/~lvbokhov/poosl

Software

[Artisan] www.artisan.com
[Rose] www.rational.com/uml
[RoseRT] www.rational.com/umlrt
[Rhapsody] www.i-logix.com
[Telelogic] www.telelogic.com/products/tau
[Visio] www.microsoft.com/office/visio

Standards

[UML1.3] OMG Unified Modeling Language Specification, UML standard 1.3,
www.uml.com/forum
[UML2.0] Preliminary documents: www.uml.com/forum
Abbreviations

ADP Architecture design phase
BCB Buffer control block
BOOM Behavioral object oriented modeling
DR Design request
EPC Embedded processor complex
FCB Frame context block
FRA Feasibility report available
FOC Focus of control
HdS Hardware dependent software
MSCE Embedded Systems CoDesign Methodology
MPSR Multi-path self-routing
NGN Next generation networks
OCL Object constraint language
OOA Object-oriented analysis
OOD Object-oriented design
OMG Object management group
POO SL Parallel object-oriented specification language
RPC Remote procedure call
ROOM Real-time object-oriented modeling
ROPES Rapid object-oriented process for embedded systems
RTC Run to completion
RUP Rational Unified Process
SDR System design review
SHE Software hardware engineering
TLDC Top-level design completed
TLK Termination link
TRS Technical requirements specification
UML Unified modeling language
UML-RT UML for real-time
Acknowledgements

Through this page I would like to express some personal thanks to the people that gave me the opportunity and great support in performing my graduation assignment and in writing this thesis.

My special thanks goes out to Alex Niemegeers, my supervisor at Alcatel Bell, for his endless support, advice and patience in explaining and discussing about the subjects of my thesis and much more. I would also like to thank Roel Marichal of Alcatel Bell for the countless discussions on design flows and UML.

From the Section Information and Communications Systems of the Faculty of Electrical Engineering at the Eindhoven University of Technology, I would like to thank Bart Theelen my supervisor, Jeroen Voeten for the discussions on simulation and output analysis and Prof.- M. Stevens for supporting and commissioning my graduation project.

Thanks to all who supported me and gave me a great time at Alcatel Bell in Antwerp.

Arjan van Ewijk
Antwerp, August 2001
Appendix A: Deterministic Random Number Generators

A1 Need for good random number generators

The use of good random number generators (RNGs) is of great importance in simulations done on systems like DataFlow as will be shown in this chapter.

The Sanford system exhibits characteristics that are probabilistic and in some cases non-deterministic. These characteristics are sometimes wanted and created by the designer of the system, but in other cases they are caused by the nature of the system-architecture or by the properties of the used implementation technology (non-deterministic behavior).

In the case of the DataFlow system two types of cases can be distinguished:
- The Embedded Processor Complex (EPC) behaves as N concurrent threads. To process the frames dispatched to the EPC takes a certain amount of processing time. The processing time is not fixed, but variable. To be able to model the EPC latency, the average processing time of the EPC is modeled as an average plus a randomly distributed value.

This type of randomization is deliberately applied by the designer.

To be able to model the random characteristics described above as trustworthily as possible, good quality random numbers generators are needed. Unfortunately, all computer (or logic) based random number generators used in computer simulation are deterministic, which means that their generated values (e.g. $U_i$, with $i = 1, 2, 3, ...$) are not independent and identically distributed (i.i.d.) i.e. truly random. Most random number generators tend to generate values in the domain $[0, 1)$, but other domains are also possible. The elementary distribution of true random variables is the continuous uniform distribution in the interval $[0, 1]$, known as the i.i.d. $U(0, 1)$ (see [LK00]).

Any variate from another distribution $F(x)$ can be obtained by applying the continuous uniformly distributed values $U(0, 1)$ to a function that describes the other distribution $F(x)$.

The quality of the random number generator can be defined as how well $U_i$ do or can resemble values of true i.i.d. $U(0, 1)$ variates.

Choosing a very poorly performing random number generator could unintentionally cause a simulation to produce unreliable results, making a performed simulation useless. Therefore it is very important to choose a good quality RNG. Unfortunately, there are no perfect RNGs available, which forces us to make a choice among the imperfect ones, based on their characteristics and the implementation we need them for.

In this chapter some well-known RNGs are identified as potential candidate for use in simulation studies performed on systems like Sanford. Important selection-criteria were the availability of a correct and simple implementation of the RNG. Also, an impression of the quality of these RNGs is given.
A2 Linear Congruential Generators

A2.1 Basics of Linear Congruential Generators

The first types of RNGs that among others were investigated are known as Linear Congruential Generators or LCGs. These types of RNGs can be defined as follows (for a sequence of \(x_1, x_2, \ldots\) of integer values):

\[
x_n = (ax_{n-1} + c) \mod m \quad \text{with} \quad 0 \leq x_n \leq m-1 \quad (11.1)
\]

With \(m\) = modulus, \(a\) = multiplier, \(c\) = increment and \(x_0\) the seed (starting value). To obtain nonnegativity and prevent implementation problems, \(a, c, m,\) and \(x_0\) should satisfy [LK00]:

\[
m > 0, \quad a < m, \quad c < m \quad \text{and} \quad x_0 < m.
\]

In [LK00] it is proven (and this can also intuitively be seen) that the \(x_i\)'s generated by (11.1) are not truly random, but that every \(x_n\) (for \(n = 0, 1, 2, \ldots\)) is completely determined by \(a, c, m,\) and \(x_0\). Notice that the output of this type of RNG is integer in the range \([0, m)\). To obtain the in paragraph 11.1 described \(U_i\)'s (in the \([0, 1)\) domain) the following has to be done:

\[
U_n = \frac{x_n}{m} \quad \text{for} \quad n = 1, 2, 3, 4, \ldots \quad (11.2)
\]

Random number generators producing other distributions than uniform that were used during simulations are explained in Appendix A1.

LCG are known for their ease of implementation, but there are large pitfalls, see [PM88].

A2.2 LCGs in practice

From several articles random number generators were obtained, given below (table 11.1). The search for good RNGs was lead by two main constraints: the availability of an implementation and the proof that the implementation is correct. These two constraints are very important since we know from [PM88] that although a RNG may be good in theory, in practice an implementation can be very wrong. This is mainly caused by multiplication or addition overflow.

The judgment whether these generators are good or not, is mainly left to the authors of the articles. Reason for this is that performing tests on RNGs is not considered as the main scope of this thesis. Some test results found of the presented RNGs are given in table 11.3.

<table>
<thead>
<tr>
<th>Name generator</th>
<th>Algorithm</th>
<th>Source implementation</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum Standard</td>
<td>(x_n = 16807 x_{n-1} \mod (2^{31}-1)) Full period LCG, period length: (2^{31}-1) Seed: (1 \leq x_0 \leq 2^{31}-2)</td>
<td>[PM88]</td>
<td>Works with 32 bit integers Generates: ((0, 2^{31}-1))</td>
</tr>
<tr>
<td>Mars &amp; Roberts</td>
<td>(x_n = 630360016 x_{n-1} \mod (2^{31}-1)) Full period LCG, period length: (2^{31}-1) Seed: (1 \leq x_0 \leq 2^{31}-2)</td>
<td>[LK00]</td>
<td>Works with 32 bit integers Generates: ((0, 2^{31}-1))</td>
</tr>
</tbody>
</table>

The found generators are often called in literature Multiplicative generators, also know as Lehmer generators (named after its inventor).

From these found generators the latter (Mars & Roberts) is said [LK00] to be more desirable to use than the first one, because of its better statistical properties (see section 11.4). On the other hand, the implementation of the
Minimum Standard is faster, since its implementation needs less calculations. Of the given RNGs is said that all possible seeds are equally valid.

Other implementations that were found use 64bit arithmetic, which implies that the RNG implementation need more calculations to let it work on a 32bit machine.

A3 Other RNGs

One generator that has exceptionally good statistical properties [LE00], is a so-called Combined Multiple Recursive Random Number Generator (MRG), created by L’Ecuyer. MRGs are composite generators, combining two or more LCGs (see table 11.2).

The presented RNG has a longer period \(2^{191} = 3.10^{57}\) and much better statistical properties than the RNGs mentioned in section 11.2. For the Sanford simulations the number of random values needed is about \(2.10^7\) (so, perhaps this period length is a bit overdone). Furthermore, this generator needs six seeds and its implementation is based on 32bit IEEE floating-point arithmetic rather than integer operations (floating point operations are slower and the 32bit IEEE standard must be supported by the compiler used).

<table>
<thead>
<tr>
<th>Name RNG</th>
<th>Algorithm</th>
<th>Source</th>
<th>Implem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRG32k3a</td>
<td>(X_{1,n} = (1403580 \times x_{1,n-2} - 810728 \times x_{1,n-3}) \mod (2^{32}-209))</td>
<td>[LE00]</td>
<td>32 bit IEEE floating point. Generates: ((0, 1)).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name RNG</th>
<th>Algorithm</th>
<th>Source</th>
<th>Implem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mars&amp;Roberts</td>
<td>(x_1 = (527612 \times x_{1,n-1} - 1370589 \times x_{1,n-3}) \mod (2^{32}-22853))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Y_n = (x_1 - x_2) \mod (2^{32}-209))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(U_n = Y_n / (2^{32}-209))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Full period LCG, period length: (2^{191})</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Seed: 6 needed, are generated by the implementation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A4 Empirical tests on random number generators

There exist a wide variety of empirical tests of randomness for RNGs. These tests use generated U;’s (values) and examine them statistically to see how closely they resemble IID U(0,1). To be able to say something about the quality of the presented RNGs, one test is presented (see [LK00]). This test uses the chi-square distribution to determine how well values generated by a RNG are uniformly distributed between 0 and 1 (generated values must be scaled to the [0,1] interval). With this test nonoverlapping tuples are created of subsequently generated values (e.g. tuples: \(U_1 = (U_1, U_2, \ldots, U_d)\) and \(U_2 = (U_{d+1}, U_{d+2}, \ldots, U_{2d})\) etc.) for dimensions from 1 to d, and with \(U_1,\ldots,U_{nd}\) as the output of the RNG. Where d is the currently used dimension in the test and \(n\) the total amount of generated values. \(U_1\) and \(U_2\) should be IID random vectors uniformly distributed on a d-dimensional unit hypercube. In other words, vectors are created of the output values of a RNG for a set of dimensions. The interval [0,1] is then divided into k subintervals with \(n/k^d \geq 5\), giving \(k^d-1\) as the degree of freedom of the chi-square distribution. With the tuples and the degree of freedom known, the chi-square distribution \(\chi^2(d)\) for the dimension d is calculated. This calculated \(\chi^2(d)\) is then compared with the \(\chi^2_{k,1,\alpha}\) distribution. If this \(\chi^2(d)\) fails the chi-square distribution (\(\chi^2(d) > \chi^2_{k,1,\alpha}\), with e.g. \(\alpha = 0.01\)), the RNG has problems with uniformity for that dimension, which implies that the tested U;’s behave in a way that is significantly different from truly IID U(0,1) random. Table 11.3 gives some results of tests performed on the presented RNGs. The shown must be examined in a comparative way. The given results are not absolute. The Minimum standard performs well up to dimension d = 4. The Minimum Standard is known as a moderate (but reliable) RNG. The Mars&Roberts RNG performs (with d = 8) better, up to 8 dimensions. This should not be interpreted as if the second RNG is twice as good as the first, it is just an indication based on the given test (it is known that this RNG performs better.
than the Minimum Standard). The MRG32k3a RNG has good uniformity properties up to \( d = 32 \). This RNG is known to be of good quality.

Table 11.3. Performance of RNGs

<table>
<thead>
<tr>
<th>Name generator</th>
<th>Chi-square test</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum Standard</td>
<td>Performs well up to ( d = 4 )</td>
<td>Well tested generator.</td>
</tr>
<tr>
<td>Mars &amp; Roberts</td>
<td>Performs well up to ( d = 8 )</td>
<td>Known to perform better than the minimum standard</td>
</tr>
<tr>
<td>MRG32k3a</td>
<td>Performs well up to ( d = 32 )</td>
<td>Long period, very good statistical properties</td>
</tr>
</tbody>
</table>

**A5 Influence of the seed**

According to the designers of the given RNGs, the choice of seed (within the given limits) is of no influence on the randomness of the RNG (implementation). The advice is given [LE00] that when using more than one RNG of the same type simultaneously it is best to use non-overlapping streams of random numbers. Seeds can be selected through the use of seed entry tables. A seed entry table provides seeds that divide the cycle of random numbers in parts. Fig. A.1 explains this.

![Seed entry table](image)

**Fig. A.1 Seed entry tables used to create non-overlapping streams of random numbers**

The seeds in the seed entry table \((S_1, S_2, \ldots)\) are selected through an index (e.g. between 0..N) to select a seed that is the first random number in a non-overlapping stream. The length of the available streams depends on the length of the cycle of a random number generator and the number of seeds in the entry table: Non-overlapping streams = Length cycle RNG/ N.

For example:
- LCG: 1000 streams, LCG cycle of \( 2^{31} - 1 \) \( \Rightarrow 2.1 \cdot 10^9 / 1000 \Rightarrow 1000 \) streams of \( 2.1 \cdot 10^6 \) random numbers.
- MRG: 100000 streams, MRG cycle of \( 3.10^{57} \) \( \Rightarrow 3.10^{57}/100000 \Rightarrow 100000 \) streams of \( 3.10^{53} \) random numbers!

Of course, the seed entry table must be available or created in order to be able to use them.

**A6 Conclusions and recommendations**

From the study done on RNGs is learnt that RNGs with a reliable implementation are not so easily obtained. The found RNGs are well-tested types that can be used without doubt about their quality. Although the given test cannot give a complete view on the quality of the RNGs, it can be used to say something about their relative quality, even though the results have to handled with care.
The question is: where do we use which RNG. Some recommendations are given here:
For simulations that need less than \(2^{31}\) subequent random numbers, use either the Minimum Standard or the Mars & Roberts RNG, where because of its better statistical performance, the Mars & Roberts generator is preferred when calculation time is not an issue.

For simulations that need long ranges of random numbers, use the MRG32k3a MRG when the compiler used, supports IEEE 32 floating-point operations.

With seed entry tables non-overlapping streams of random numbers can be obtained.

Some methods to create other distributions

```c
//Generate Uniform random distribution in [a, b)
double Uniform(double a, double b)
{
    return a + (b - a) * Random();
}

//Generate Uniform random distribution in [a, b) (integer)
int UniformInt(int m, int n)
{
    return m + int((n - m + 1.0) * Random());
}

//Generate Exponentia1 random distribution in [a, b)
double Exponential(double x)
{
    return - x * log(Random());
}
```
Appendix B: UML modeling of DataFlow subsystem

Fig. B.1: Structural requirements model: Deployment diagram showing system in its hardware environment
Fig. B.2: Functional model: Use Case diagram depicting functionality in ingress mode in the Functional model

**DataFlow Ingress OC192**

- **Framer Ingress**: Sends IP frames to dataflow
  - Send IP frames for streams up to 10Gbps
  - Frame size is dispatched to EPC

- **EPC Ingress**: Frames are dispatched to EPC
  - Read Frame pointer and Retrieve Frame Context
  - Issue requests for read/write parts of frame to DataStore
  - Pass frame context to DataFlow
  - Receive data and control information (enqueue)
  - Wrap interface (unidirectional) used for receive data from Egress to Ingress (10Gbps)

- **Switch Ingress**: Receives PRIZMA cells at line rate up to 14.2 Gbps
  - Control Flow of PRIZMA cells
  - Obtain metering (queue filling level) information
  - Select frame from context in appropriate queue for transmission

- **Scheduler Ingress**: Receives frame context and enqueue it
Fig. 8.3: Functional model: Use Case diagram depicting functionality in egress mode in the Functional model

Data Flow Egress OC192

Framer Egress: Receives IP frames to Egress

EPC Egress: Frames are dispatched to EPC

Wrap Interface (unidirectional) used for transmit data from Egress to Ingress (1 Gbps).

Control Flow of PRIZMA cells

Send PRIZMA cells at line rate up to Switch Egress

Scheduler Egress

Receive frame context and enqueue it

Select frame from context in appropriate queue for transmission

Obtain metering (queue filling level) information

Issue requests for read/write parts of frame to DataStore

Pass frame context to DataFlow

Receive frame context and enqueue it

Send data and control information (enqueue)

Wrap Interface Egress

Read Frame pointer and Retrieve Frame Context

EPCEgress (from EPC Egress)

<<communicate>>

<<realize>>

Framer Egress: Receive IP frames for streams up to 10 Gbps

<<communicate>>

<<realize>>

Move to functional model
Fig. B.4: Functional model: Context diagram in the Structural model showing interaction with external actors (ingress)
Fig. 8.5: Functional model: Context diagram in the Structural model showing interaction with external actors (egress)
Fig. B.6: Dynamic model: Message Sequence Chart depicting relevant scenario of system operation

Arriving Frame

Event ID

Timing Mark

Descriptive Annotation

Frame is sent to switch in PRIZMA cells

(d-c = latency EPC)

(b-a = latency DataFlow)
Fig. B.7: Formulation phase 1: Object diagram

Fig. B.8: Formulation phase 1: Deployment diagram
Fig. B.9a: Formulation phase 1: Collaboration diagram
Fig. B.9b: Formulation phase 1: Collaboration diagram
Fig. B.10: Formulation phase 1: Class diagram

DataFlowinggress

Transmitter
Buffer Acceptance
CellQueue

PreparationArea(PREPSIZE): PrepAreaArray
PreparationArea

Receiver
PreparationArea

DataStoreMemory

PreparationArea

EPController

Arbiter

EPController

PreparationArea

MCBRead

MCBWrite

RequestQueuingLogic

Gqueue

DataStoreMemory

...
Fig. B.11: Formalization phase 1: Structure diagram of DataFlow abstract level model (System level)

Fig. B.12: Formalization phase 1: Structure diagram of DataFlow abstract level model (Intermediate level)
Fig. B.12b: Environment structure of DataFlow

- Clock Module
  - Clock Module (from Model Execution)
  - CLKCount: int = 0
  - PortEnable: int[2] = 1
  - + / CLK_66ns
  - # / log
  - # / TSB
  - # / TSC~
  - + / CLK_36ns
  - + / CTRL~

- Traffic Generator
  - Traffic Generator (from DataFlowIngress)
  - + / Output
  - + / CLK~

- DataFlowIngress
  - DataFlowIngress

- Sink
  - Sink

- DataFlow in its environment
  - DataFlowIngress
  - DataStoreMemory (from DataFlowIngress)
  - + / Framer~
  - + / CLK_66ns~
  - + / Switch

- Cell Interface
  - Cell Interface (from DataFlowIngress)
  - + / Tx
  - + / CLK~
  - + / Switch

- Receiver
  - Receiver (from DataFlowIngress)
  - # / log
  - + / CLK~
  - + / DataStore
  - + / EPCController
  - + / In~
  - + / Arbiter~

- EPC Controller
  - EPC Controller (from DataFlowIngress)
  - + / Tx
  - + / CLK~

- Arbiter
  - Arbiter

- Transmitter
  - Transmitter (from DataFlowIngress)
  - + / Arbiter~
  - + / EPCController~
  - + / DataStore
  - + / Cell Interface~
  - + / CLK~

Alcatel Bell N.V.
Fig. B.13: Formalization phase 1: Collaboration diagram depicting messages with stereotypes and parameters
Fig. B.13b: Formalization phase 1: Collaboration diagram depicting messages with stereotypes and parameters
Fig. B.14: Simulation phase1: Message sequence diagram depicting scenario based on 42 IP payload simulation

Scenario1: Frame length: 42 Bytes IP payload
Fig. B.15: Formulation phase 2: Object diagram (block diagram)
Fig. B.16: Formulation phase2: Deployment diagram

- DDR DRAM 32bits data @166 MHz
- Flexbus64 (OC192c)
- DataStore_Controller
- BufferAcceptance
- Memory_Arbitrator
- SchedulerController
- Transmitter
- Switch interface (PRIZMA)
- EPCController
- FCBMemoryArbitrator
- FCB_QDR_SRAM
- BCB_QDR_SRAM
Fig. B.17a: Formulation phase2: Class diagram: structure of the models
Fig. B.17b: Formulation phase2: Class diagram: class diagram depicting classes and the associations
Fig. B.17c: Formulation phase 2: Class diagram: class diagram depicting classes and the associations
Fig. B.18: Formulation phase 2: Collaboration diagram: <<simple>> messages
Fig. B.19a: Formalization phase 2: Structure diagram system level: two clock domains, performance analysis library used
Fig. 8.19b: Formalization phase 2: Structure diagram intermediate level: two clock domains, performance analysis library used
Fig. B.20a: Formalization phase 2: Class diagram intermediate level
Fig. B.20b: Formalization phase 2: Class diagram intermediate level
Fig. B.20c: Formalization phase 2: Class diagram intermediate level
Fig. B.20d: Formalization phase 2: Class diagram intermediate level

<table>
<thead>
<tr>
<th>FCBCache</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadFCBPointer : int</td>
</tr>
<tr>
<td>WriteFCBPointer : int</td>
</tr>
<tr>
<td>NumberOfFCBPresent : int</td>
</tr>
<tr>
<td>NumberOfFCBRequested : int</td>
</tr>
<tr>
<td>RemoveFCBPointer : int</td>
</tr>
<tr>
<td>NextRequestBCBPPointer : int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Init() : void</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCBCache() : FCBCache</td>
</tr>
<tr>
<td>ReadBCB(bcba : int) : BufferControlBlock</td>
</tr>
<tr>
<td>ReadFCBCache() : FCBCacheEntry</td>
</tr>
<tr>
<td>FillFCBCache(fcba : int) : void</td>
</tr>
<tr>
<td>NumberOfFCBRequests(target_port : int) : int</td>
</tr>
<tr>
<td>AckFCBRequest(target_port : int) : void</td>
</tr>
<tr>
<td>AckBCBRequest() : void</td>
</tr>
<tr>
<td>BCBRequests() : BCBRequestsFromFCB</td>
</tr>
<tr>
<td>RemoveFromFCBCache() : void</td>
</tr>
<tr>
<td>ReadNumberOfFramesPresentInTPQ(target_port : int) : int</td>
</tr>
<tr>
<td>IssueNewFCBRequest(fcba : int, target_port : int) : void</td>
</tr>
<tr>
<td>ReadFCBPageBCBCount(fcba : int) : int</td>
</tr>
<tr>
<td>ReadFCBPageSourcePort(fcba : int) : int</td>
</tr>
<tr>
<td>ReadFirstBCBAC(fcba : int) : int</td>
</tr>
<tr>
<td>FCBRelease(fcba : int) : void</td>
</tr>
<tr>
<td>TPDequeue(target_port : int) : int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FCBAccessTokenFIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO : FCBAccessToken [FCB_LATENCY] [FCB_REQUESTS]</td>
</tr>
<tr>
<td>InputFIFOPointer : int</td>
</tr>
<tr>
<td>OutputFIFOPointer : int</td>
</tr>
<tr>
<td>RequestNumberForThisCycle : int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FCBAccessTokenFIFO : FCBAccessTokenFIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewRequest(fcba : int, target_port : int) : void</td>
</tr>
<tr>
<td>CLK() : void</td>
</tr>
<tr>
<td>FillFCBCache(fcba : int, target_port : int) : void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BCBAccessTokenFIFO : BCBAccessTokenFIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO : BCBAccessToken [BCB_LATENCY] [BCB_REQUESTS]</td>
</tr>
<tr>
<td>InputFIFOPointer : int</td>
</tr>
<tr>
<td>OutputFIFOPointer : int</td>
</tr>
<tr>
<td>RequestNumberForThisCycle : int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BCBRequests() : BCBRequestsFromFCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCBAccessTokenFIFO() : FCBAccessTokenFIFO</td>
</tr>
<tr>
<td>NewRequest(target_port : int, bcba_request_type : BCBRequestType, bcba : int) : void</td>
</tr>
<tr>
<td>CLK() : void</td>
</tr>
<tr>
<td>FillFCBCache(target_port : int, bcba_request_type : BCBRequestType, bcba : int) : void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EPCIngress</th>
</tr>
</thead>
<tbody>
<tr>
<td>AverageLatency : int</td>
</tr>
<tr>
<td>DeltaLatency : int</td>
</tr>
<tr>
<td>Timer : int [NUMBER_OF_EPC_THREADS]</td>
</tr>
<tr>
<td>TimeStamp : int [NUMBER_OF_EPC_THREADS]</td>
</tr>
<tr>
<td>FreeEPCThreads : int</td>
</tr>
<tr>
<td>EPCQueue : Queue</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Init() : void</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPCIngress(average_latency : int, delta_latency : int) : EPCIngress</td>
</tr>
<tr>
<td>Enqueue(fcba : int) : void</td>
</tr>
<tr>
<td>Dequeue() : fcbac</td>
</tr>
<tr>
<td>UpdateNumberOfWaitingMessages() : void</td>
</tr>
<tr>
<td>WriteStatistics(filehandle : FILE*) : void</td>
</tr>
<tr>
<td>EnqueueMCBWriteMessageToMCBWrite(fcba : int) : void</td>
</tr>
<tr>
<td>CheckMCBReadStatus() : bool</td>
</tr>
<tr>
<td>GetMessageFromMCBRead() : DFMessage</td>
</tr>
<tr>
<td>IncrementGQueueCredit(credit : int) : void</td>
</tr>
<tr>
<td>EPCProcess() : void</td>
</tr>
<tr>
<td>CLK() : void</td>
</tr>
</tbody>
</table>
Fig. B.20e: Formalization phase2: Class diagram intermediate level

```
ElasticityQueue

ReceiveFishPointer : int
CurrentFishFill : int
SendCellPointer : int
SendFishPointer : int
ReserveFishPointerTemp : int
ReserveFishPointerCorrect : int
SendBytePointer : int
ReservedFish : int
TestEmpty : int
TestReserved : int
NumberOfFramesPresent : int

UpdateState() : void
Init() : void
ElasticityQueue() : ElasticityQueue
ReserveBytes(reserve_bytes : int, end_of_frame : bool) : void
PortRequest() : int
RetrieveNextCell() : Cell
SendStatus() : bool
SendCellToReceiverEgress(cell : Cell) : void
```
Fig. B.20f: Formalization phase 2: Class diagram intermediate level

<table>
<thead>
<tr>
<th>PCBEntry</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ FM : bool [MAX_PCB_PRIORITY] [N_FISH]</td>
</tr>
<tr>
<td>✓ Slice : int [MAX_PCB_PRIORITY]</td>
</tr>
<tr>
<td>✓ TargetPortID : int</td>
</tr>
<tr>
<td>✓ NumberOfBCBPresent : int</td>
</tr>
<tr>
<td>✓ NumberOfRequestedBCB : int</td>
</tr>
<tr>
<td>✓ FirstBCBPointer : int</td>
</tr>
<tr>
<td>✓ NextBCBPointer : int</td>
</tr>
<tr>
<td>✓ CurrentBCBRequestsFromFCB : BCBRequestsFromFCB</td>
</tr>
<tr>
<td>✓ NumberOfNewBCBRequests : int</td>
</tr>
<tr>
<td>✓ NumberOfGrants : int</td>
</tr>
<tr>
<td>✓ NextBufferAddress : int</td>
</tr>
<tr>
<td>✓ CurrentSourcePort : int</td>
</tr>
</tbody>
</table>

### Methods
- `Init(target_port_ID : int) : void`
- `PCBEntry(target_ID : PCBEntry)`
- `NumberOfBCBRequests(target_port : int) : int`
- `AckBCBRequestByBCBArbiter(target_port : int) : void`
- `FillBCBFIFO(target_port : int, bcb_request_type : BCBRequestType, bcba : int) : void`
- `Confirm(priority : int) : void`
- `UpdateRequestState() : void`
- `UpdateRequestArrayInRQL(priority : int, TargetPortID : int) : void`
- `ElasticityQueuePriority(target_port_ID : int) : int`
- `NotifyElasticityQueue(reserve_bytes : int, end_of_frame : bool, target_port : int) : void`
- `ReadBCB(bcba : int) : BufferControlBlock`
- `BCBRelease(bcba : int) : void`
- `IssueBCBRequest(target_port : int, bcb_request_type : BCBRequestType, bcba : int) : void`
- `GetRequestsFromFCB(target_port : int) : BCBRequestsFromFCB`
- `NotifyFCBOIGrantedBCBRequest(target_port : int) : void`
- `ReadFCBCache(target_port : int) : FCBCacheEntry`
- `RemoveFrameFromFCB(target_port : int) : void`
Fig. B.21: Formalization phase2: Collaboration diagram intermediate level: messages defined with type and parameters

WriteDS(B : , BCBA : , bank_mask : )
ReadDS(bcba : , bank_mask : )

ArbitReadRequests ( )
ArbiterAcknowledgement(RxAck : )

ReadRequests ( )
Acknowledgement(epc_ack : )

WriteBCB(bcba : , next_address : )

EPController

FreeQDechain() WriteFCB() LeaseFCB()

FCBMemory Arbiter

FreeQChain() FrameEnqueue()

BCBMemory Arbiter

FreeQChain() LeaseBCB()

Cell Interface

Receiver

Arbit

Transmitter

TX_ReadTransmitterRequests ( )

TXRetrieveCell():Cell

DataStoreMemory

LeasBCB()
Fig. B.22: Formalization phase2: Structure diagram sub capsule intermediate level
Fig. B.23: Formalization phase2: Class diagram sub capsule intermediate level

Gqueue
- FCBA : int [FIFO_ADDRESS_SPACE+1]
- Head : int
- Tail : int
- NumberOfSamples : double
- DequeueCredit : int
- DroppedFrames : int
- AverageOccupation : double
- MaximumOccupation : int
- OccupationHistogram : int [FIFO_ADDRESS_SPACE+2]
- gqueue : Gqueue
- Enqueue(fcba : int) : void
- Dequeue() : fcba
- IncrementCredit(Credit : int) : void
- CLK()

EPCController
- EPCcontroller() : EPCController
- EPCcontrollerProcess() : int
- FCBEnqueue(FCB : FCBAddress) : int
- Acknowledgement(EPACak : EPCRequests) : void
- CheckFrameReadyForDispatch() : void
- ReadOutFrameContext() : FCB
- ReadFrameData(FCB : FCB) : void
- WriteFrameContext(FCB : FCB)
- ReadRequests() : EPCRequests
- CLK()

EPCingress
- AverageLatency : int
- DeltaLatency : int
- Timer : int [NUMBER_OF_EPC_THREADS]
- TimeStamp : int [NUMBER_OF_EPC_THREADS]
- FreeEPCThreads : int
- EPCQueue : Queue
- Init() : void
- EPCingress(Average_latency : int, delta_latency : int) : EPCingress
- Enqueue(cba : int) : void
- Dequeue() : fcba
- UpdateNumberOfWaitingMessages() : void
- WriteStatistics(filehandle : FILE*) : void
- Enqueue(MCBWriteMessageToMCBWrite(cba : int) : void
- CheckMBWriteStatus() : bool
- GetMessageFromMCBRead() : FMessage
- IncrementGQueueCredit(credit : int) : void
- EPCProcess() : void
- CLK() : void

RequestQueueingLogic
- RQL[SLICES][BANKS][RQL_ADDRESS_SPACE+1] : RQI'm
- Head[SLICES][BANKS] : int
- Tail[SLICES][BANKS] : int
- QueueEmpty[SLICES][BANKS] : bool
- read_weight[N_FISH+1] : int
- write_weight[N_FISH+1] : int
- weight_translation : int
- RR_ReadWrite : bool
- RR_LastReadSlice : int
- RR_LastWriteSlice : int
- EPCGrants : int

MCBRead
- ReadMessage(MCBADDRESS_SPACE+1) : MCBReadMessage
- Status(MCB_ADDRESS_SPACE+1) : bool
- Head : int
- Tail : int
- QueueEmpty : bool
- MCBRead() : MCBRead
- Enqueue(m : MCBReadMessage) : void
- Dequeue() : MCBReadMessage
- CheckStatus() : bool
- Confirm(fishnumber : int) : bool
- CLK()

MCBWrite
- TargetPortID : int
- WriteMessage(MCB_ADDRESS_SPACE+1) : MCBWriteMessage
- Status(MCB_ADDRESS_SPACE+1) : bool
- Head : int
- Tail : int
- QueueEmpty : bool
- MCBWrite() : MCBWrite
- Confirm(fishnumber : int) : void
- ReadBCB() : BufferControlBlock
- ReadFirstBCBAddress() : int
- Enqueue(Message : MCBWriteMessage) : void
- Dequeue() : MCBWriteMessage
- EnqueueFrameToPCB() : void
- CLK()
Fig. B.23b: Formalization phase2: Class diagram sub capsule intermediate level
Fig. B.24: Simulation phase 2: Message Sequence diagram

Execution scheme. Scenario: up and running (48 byte frames)
Fig. B.24b: Simulation phase2: Message Sequence diagram
Fig. B.24c: Simulation phase2: Message Sequence diagram
Appendix C: Extended modeling framework in C++

C1 Ports in the C++ modeling framework

FIG. C.1 shows the static position of the port objects in a performance model. Ports classes are grouped in a UML package. The simulation model is also grouped in a separate package and uses the package containing the Ports. The <<utility>> package contains classes that have general purpose in the executable model. Contained in this package is:

- Classes that have general purpose in the model, like: data classes (e.g., particular hardware types: BCB/FCB class, Buffer, fish etc.), queues classes, classes defining hardware objects (e.g., memory)
- Utility classes, like general types

The <<utility>> package is used (see uses relation depicted with the dotted arrow) by all three other packages.

The MessageSequenceRecording package groups classes that are used with the Ports to generate message sequence diagrams from the messages sent between ports. This package uses both the utility package as the ports package. The contents of the Ports package is shown in fig. C.2. The port class defined in this package is used by the Performance simulation model as defined in chapter 5. The port class is a parameterized class, meaning that the appearance of the class at runtime is determined at compile-time and dependent on the parameters (polymorphism). A port is declared by the line:

```
PORTN<ASignal1, ASignal2, ..., ASignalN> PORT_Name;
```

Where ASignal1, ASignal2, ..., ASignalN represent UML signals (here C++ objects) that can be transferred through this port. The first signal (ASignal1) represents the return signal (return data) of a RPC. The parameterization of the class is implemented with C++ templates. The simplest port is the one defined with PORT1<ASignal1, ASignal2> PORT_simple; which can send one data object through the port and return an object of a RPC. As can be seen in fig. C.2 port classes are available which can handle various sets (lengths of sets) of parameters (1..N). A port class is defined by its port type (port class uses PORTType class) base, conjugated or relay. The port classes furthermore use a multi-queue class. The multi-queue class defines a queue class that can queue sets of objects parallel, where the number of parallel queued objects is dependent on the parameters of the class. The multi-queue is used in the conjugated port which is used for asynchronous communication. The multi-queue class has an aggregate relation with the queue class. The
number of queue objects that are used in the multi-queue class is dependent on the number of parameters the multi-queue class has. This queue class is a specialization of an abstract queue class.

The connection between ports of different modules is created through a pointer. The PORT1 class has an aggregate relation with itself (with multiplicity of one). Each PORTx (with x = 1, 2, .., N) class has a privately declared pointer pPORT (see e.g. class PORT1) that is used to point to an object of the same class with the same parameters (this makes sure that connections between different port types is detected at compile-time since it will not be accepted by the compiler). The pPORT pointer is used to connect ports for communication (the UML-RT connector).

Fig. C.2: Static structure of the Ports classes in the Ports package
Note that the actual implementation of a port is dependent on the number of class parameters defined (the signals Asignal1, ... Asignal2). In the implementation only the Port class with the correct number of parameters is included in a model.
The Port class offers all the functionality needed to perform that base, conjugated or relay role in a communication between modules. Fig. C.3 shows the PORT1 class. The class has two parameters ASignal1 (the return object of an RPC) and ASignal2 the data object to be transferred through the port.

```
PORT1<void, Buffer> PORT_EPC;
```

PORT_EPC is of type PORT1<void, Buffer> and is used for asynchronous communication. Notice that ASignal1 is of type void here.

Table C.1. Operations for use of ports

<table>
<thead>
<tr>
<th>Operation</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORT1(...)</td>
<td>Constructors are used to specify the port.</td>
</tr>
<tr>
<td>ConnectTo(Aport: PORT1&amp;)</td>
<td>Operation is used to connect ports and is called in the constructor of a top-level module. <strong>Operation is only called with a base port.</strong></td>
</tr>
<tr>
<td>Send(X1: ASignal2&amp;): void</td>
<td>Sends asynchronously the object X1 (of type ASignal2) to a connected port. Return value is void. <strong>Operation is only called with a base port.</strong></td>
</tr>
<tr>
<td>Invite(X1: ASignal2&amp;): ASignal1</td>
<td>Calls synchronously through RPC the object X1 (of type ASignal2) to a connected port. Return value is of type ASignal1. <strong>Operation is only called with a base port.</strong></td>
</tr>
<tr>
<td>AttachRPC(theRPC: ASignal1(*)(class ASignal2)): void</td>
<td>Attaches a procedure of the module that owns the port to the RPC pointer of that port. A RPC to the port directly calls the attached operation of the module. Parameters and return values should match and are checked at compile-time. <strong>Operation is only called with a conjugated port.</strong></td>
</tr>
<tr>
<td>DataXchange(...)</td>
<td>Internal private operation.</td>
</tr>
<tr>
<td>GetPortData(X1: ASignal2&amp;): bool</td>
<td>Operation retrieves data from conjugated port FIFO. Each call de-queues one position in the queue. <strong>Operation is only called with a conjugated port.</strong></td>
</tr>
</tbody>
</table>
C2 Model created with extended modeling framework in C++
The example given in fig. 5.7 (chapter 5) is here worked out.

Top level module:

```cpp
class TopLevelModule : AlObject
{
public:
    TopLevelModule()
    {
        pArbiter = NULL, pReceiver = NULL, pEpc = NULL;
        //dynamic creation of sub modules
        pArbiter = new Arb("Arbiter");
        pReceiver = new Rx("Receiver");
        pEpc = new EPC("EPC");
        if(pArbiter == NULL || pReceiver == NULL || pEpc == NULL) exit(0);
        //Connect the ports
        Arbiter.PORT_Rx1.ConnectTo(Receiver.PORT_Arb1);
        Arbiter.PORT_RX2.ConnectTo(Receiver.PORT_Arb2);
        Receiver.PORT_EPC.ConnectTo(Epc.PORT_RX);
    }
    //Declaration of sub modules’ context
    Arb pArbiter;
    Rx pReceiver;
    EPC pEpc;
    ~TopLevelModule() {}
    //Process clocks of sub modules
    void Process()
    {
        Receiver.Process();
        Epc.Process();
        Arbiter.Process();
    }
    //Propagation clocks of sub modules
    void Propagation()
    {
        Receiver.Propagation();
        Epc.Propagation();
        Arbiter.Propagation();
    }
};
```

Sub modules:

**Receiver module:**

```cpp
#define N 1
class Rx : AlObject
{
public:
    Rx(char* Name) :
    //Init ports
    PORT_EPC(Base, "PORT_EPC", this),
    PORT_Arb1(Conjugated, "PORT_Arb1", N, this, true),
    PORT_Arb2(Conjugated, "PORT_Arb2", N, this, true)
    {
        SName = new char[strlen(Name)];
        strcpy(SName, Name);
        Init();
        PORT_Arb1.AttachRPC(RPCReg);
    }
};
```

Class definition of active object/module (notice: class derived from a general object 'AlObject')

Constructor of the active object. Used to define the PORTs (their type: Base, Conjugate or Relay, their queue depth (N) and whether their actions are recorded in a message sequence chart)

Attach the procedures of the receiver module to the appropriate ports
PORT_Arb2.AttachRPC(RPCGrant);

//Declaration of ports
PORTl<void, FCB> PORT_EPC;
PORTl<int, void> PORT_Arb1;
PORTl<void, int> PORT_Arb2;

//RPCs
int RPCReq();
void RPCGrant(int Grant);
void Process();
void Propagation();

Private: //operations
void Init();
void FillPrepArea();

Private: //attributes
PREPAREA PrepArea;
int Requests;
int WG;
char* SName;

void Rx::Propagation()
{
  FillPrepArea();
}
void Rx::Process()
{
  //Algorithm + behavior
  PORT_EPC.Send(WG);
}

int Rx::RPCReq()
{
  return PrepArea.Req();
}
void Rx::RPCGrant(int Grant)
{
  WG = Grant;
}

//Operations for RPC
//Operations attached to ports owned by the receiver module and called through RPCs (in this case by the Arbiter module).

EPC module:

#define N2 3
class EPC : AlObject
{
  public:
    EPC(char* Name) :
      //Init ports
      PORT_Rx(Conjugated, "PORT_Rx", N2, this, true)
    {
      SName = new char[strlen(Name)];
      strcpy(SName, Name);
      Init();
    }

    ~EPC() {}

    //Declaration of ports
    PORTl<void, FCB> PORT_Rx;

Declarations of the PORTs of the active object (notice the declaration of the signals <signal1, signal2, ....> used to communicate through the public PORTs.

Operations used for RPCs

Process() and Propagation() the two procedures called by the top-level object (or simulation scheme).

DataObjects (non active objects) and procedures are 'private' members of active objects.

Process and Propagation operations

EPC module:
void Process();
void Propagation();

Private: //operations
    void Init();

Private: //attributes
    Queue<FCB> EPCFIFO;
    char* SName;
    int Datal;
);

//Process and propagation operations
void Rx::Propagation()
{
    while(PORT_Rx.GetPortData(Datal))
        EPCFIFO.Enqueue(Datal);
}

void Rx::Process()
{
    //Process
}

Arbiter module:

class Arb : A1Object
{
public:
    Arb(char* Name) :

    //Init ports
    PORT_Rx1(Base, "PORT_Rx1", this),
    PORT_Rx2(Base, "PORT_Rx2", this)
    {
        SName = new char[strlen(Name)];
        strcpy(SName, Name);
        Init();
    }

    ~Arb();

    //Declaration of ports
    PORT1<int, void> PORT_Arbl;
    PORT1<void, int> PORT_Arb2;

    void Process();
    void Propagation();

Private: //operation
    void Init();

Private: //attributes
    int RxReq;
    char* SName;
};

//Process and propagation operations
void Arb::Propagation()
{
    //Propagation
}

void Arb::Process()
{
    RxReq = PORT_Rx1.Invoke();
    PORT_Rx2.Invoke(RxReq);
}

At propagation empty the PORT_Rx FIFO and enqueue data in the EPCFIFO

At process clock trigger a RPC (Invoke()) to the Receiver to determine the requests. Requests are immediately completely granted through another RPC.

Declaration of ports. Notice that the types of the parameters of the class match those of the complementary port defined in the Receiver.
Main simulation schedule:
Main simulation schedule: one autonomous thread.

#define SIMEND le5

void main ()
{
    //Creation of top-level module
    TopLevelModule TopLevel("TopLevelModule");

    int SimulationTime = 0;
    int SimulationEnd = SIMEND;

    while(SimulationTime < SimulationEnd) // loop until end of simulation.
    {
        TopLevel.ProcessClock();
        TopLevel.PropagationClock();
        Increment(SimulationTime);
    }

    //The actual simulation 'run'. Execution performed in two phases: 'process' and 'propagation'. The process
    //and propagation triggers of the encapsulated modules are called when the process and
    //propagation trigger of the top-level module are
}
Appendix D: Extended modeling framework in RoseRT

Code executed at state entry:

```c
const int CLKPort66nsSize = CLK1.size(); //Cardinality = 9
const int CLKPort36nsSize = CLK2.size(); //Cardinality = 1
RTMessage replyCLK_66ns[CLKPort66nsSize];
RTMessage replyCLK_36ns[CLKPort36nsSize];

const int CLK_66nsPeriod = 11; //Relative clock period (66ns)
const int CLK_36nsPeriod = 6; //Relative clock period (36ns)

int i,

///////////CLK RPC Invocation///////////////////////////

/*** CLK_66ns Process ****
if(CLKCount % CLK_66nsPeriod == 0)
{
    for(i = 0; i < CLKPort66nsSize; i++)
        CLK_66ns.CLK().invokeAt(i, &replyCLK_66ns[i]);
}

/*** CLK_36ns Process ****
if(CLKCount % CLK_36nsPeriod == 0)
{
    for(i = 0; i < CLKPort36nsSize; i++)
        CLK_36ns.CLK().invokeAt(i, &replyCLK_36ns[i]);
}

///////////CLK RPC Reply check///////////////////////////

/*** CLK_66ns Propagation check ****
if(CLKCount % CLK_66nsPeriod == 0)
{
    for(i = 0; i < CLKPort66nsSize; i++)
        if(!replyCLK_66ns[i].isValid())
            log.log("ERROR: reply CLK_66ns NOT RECEIVED !!!");
}

/*** CLK_36ns Propagation check ****
if(CLKCount % CLK_36nsPeriod == 0)
{
    for(i = 0; i < CLKPort36nsSize; i++)
        if(!replyCLK_36ns[i].isValid())
            log.log("ERROR: reply CLK_36ns NOT RECEIVED !!!");
}

CLKCount++; //Increment clock counter

int tmp = 1;
TJB.Self_Message(tmp).send();
```

Synchronous messages (RPC) are sent at the CLK_36ns port.

Ports of capsules connected to the CLK_66ns port of the ClockModule receive a synchronous message (RPC) in the order of their number in the cardinality of the CLK_66ns port and need to reply to this invocation (after RPC code execution is finished).

‘Relay’ merely relays the self messages and generates through this messages in the priority queue of the virtual machine.

PA Port (uses the CTRL protocol) used by performance analysis (PA) library to control the simulation.

//Simple relay at state transition
int theData = *rtdata;
out.StateIn(theData).send();
Fig. D.1: Clock module class diagram

Features:
- Clock module has two ports for two clock domains with ratio 11:6.
- Clock triggers are provided through port with cardinality and triggers to these ports are scheduled according the number of the port in the cardinality.
- Reply to the clock triggers (CLK()) messages are checked.
- A Clock module uses an internal relay capsule to re-trigger itself.
Appendix E: Performance analysis library in UML

Fig. E1: Performance analysis library Class Diagram
batch means  spectral  Recurrent points  independent

Confidence Analysis  Include  <<utility>>

Performance simulation model  random number generators
ConfidenceMethodProcess: ConfidenceMethodProcess

GetDouble(name)
GetString(name)

RacessCheckPoint(MetricNum, cp)
SimulationOver()
WriteReport()

WriteReport(MetricNum, Confidence)

Begin(num Params)
End()

Begin()
End()

RandomNumberSetSeed(seed, stream)
GetRandomNumber(stream)
GetEnum(name, value)

ProcessMetricSample(value)
GetCheckPoint(cp)
ReachedCheckPoint()

ConfidenceSampleMetric(MetricNum, value)
ConfidenceSimFinished()
GetRandomNumber(stream)
RandomNumberSetSeed(seed, stream)

SimulationOver()
WriteReport()

ProcessMetricSample(MetricNum, value)
WriteReport()

GetDouble(name)
GetString(name)

GetEnum(name, value)
SetSeed(Seed, Stream)
GetRandom(Stream)

Fig. E2: Performance analysis library Collaboration Diagram
Appendix F: Derivation of the classical estimator for ratios of expectations

The following is based on [CL99]. The aim is obtain an expression for the estimate and the variance to construct a confidence interval for the method of recurrent points.

Let: state $r$ is the recurrent state
$Y_i := \text{"Sum of the observed metric rewards in cycle } i, \text{ where one cycle is from } r \text{ to } r\"$
$L_i := \text{"length of cycle } i\"$

Note: for the batch means (recurrent points), $Y_i$ is the sum of all observation in a batch and $L_i$ is the length of a batch.

Then:

$E[Y] = \frac{\mu}{\pi_r}$ and $E[L] = \frac{1}{\pi_r}$, with $\pi_r$ the equilibrium change of $r$

then $\mu = \frac{E[Y]}{E[L]}$

Let: $Z_i := Y_i - \mu L_i$

$E[Z] = 0$ since $E[Z] = E[Y] - \mu E[L] = \frac{\mu}{\pi_r} - \frac{\mu}{\pi_r} = 0$

Then:

$\text{Var}(Z) = \sigma^2$ and $Z_i$s are IID (independent and identically distributed) random variables, therefore

$\sum_{i=1}^{n} Z_i = N(0, n\sigma^2)$

$\frac{\sum_{i=1}^{n} Z_i}{\sqrt{n} \cdot \sigma} = N(0, 1)$

Determine $C$ as such that $P(-C \leq N(0,1) \leq C) = \gamma$

$P\left(-C \leq \frac{\sum_{i=1}^{n} Z_i}{\sqrt{n} \cdot \sigma} \leq C\right) = \gamma \Rightarrow \{\text{Rearranging}\}$

$P\left(-C \cdot \sqrt{n} \cdot \sigma \leq \sum_{i=1}^{n} Z_i \leq C \cdot \sqrt{n} \cdot \sigma\right) = \gamma \Rightarrow \{\text{Filling in } Z_i := Y_i - \mu L_i\}$

$P\left(\sum_{i=1}^{n} \frac{Y_i}{L_i} - C \cdot \sqrt{n} \cdot \sigma \leq \frac{\sum_{i=1}^{n} Y_i}{\sum_{i=1}^{n} L_i} \leq \frac{\sum_{i=1}^{n} Y_i}{\sum_{i=1}^{n} L_i} + C \cdot \sqrt{n} \cdot \sigma\right) = \gamma$
\[ \frac{E[Y]}{E[L]} = \frac{\mu}{\frac{1}{\pi_r}} = \mu \]

Point estimator:

\[ \frac{1}{n} \sum_{i=1}^{n} Y_i = \hat{\mu} \]

\[ \frac{1}{n} \sum_{i=1}^{n} L_i \]

\[ \sigma_z^2 = \frac{1}{n} \sum_{i=1}^{n} Z_i^2 = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \mu L_i)^2 \]

\[ \sigma_z^2 = \frac{1}{n} \sum_{i=1}^{n} Y_i^2 - 2\mu \frac{1}{n} \sum_{i=1}^{n} Y_i L_i + \mu^2 \frac{1}{n} \sum_{i=1}^{n} L_i^2 \xrightarrow{a.s.} \sigma_\mu^2 \]

\[ \mu \text{ can be replaced by its point estimator } \hat{\mu} \text{, thus:} \]

\[ \sigma_{\hat{\mu}}^2 = \frac{1}{n} \sum_{i=1}^{n} Y_i^2 - 2\frac{1}{n} \hat{\mu} \sum_{i=1}^{n} Y_i L_i + \frac{1}{n} \hat{\mu}^2 \sum_{i=1}^{n} L_i^2 \]
Appendix G: Algorithm for detection of the initial transient (Schreuben algorithm)

The algorithm described here is presented in [PA90]. The algorithm is used to test the hypothesis that a sufficient number of initial transient data has been discarded. The tested sequence of observations is compared with the corresponding value from a standard sequence, and the decision about rejection or acceptance of the hypothesis is taken at an assumed level of confidence \( \alpha \). The \( \alpha \) can be regarded as the probability that the hypothesis that the tested sequence of observations is stationary is rejected (H0 rejected).

The algorithm has the following parameters:

- \( n_0^* \): the number of samples at which the initial transient is over according to the rule presented in section 8.10.1.
- \( n_{\text{max}} \): the maximum allowed length of the simulation run measured in the number of recorded observations (this has to be decided in advance of the simulation). A maximum number of events of \( 10^7 \) or \( 10^8 \) seems reasonable.
- \( n_{0,\text{max}} \): the maximum allowed length of the initial transient period. Here used \( n_{0,\text{max}}=0.1 \cdot n_{\text{max}} \).
- \( n_0 \): the length of the sequence used for estimating the steady state variance \( \sigma^2[\overline{X}(n)] \). \( n_0 = 100 \).
- \( n_t \): the length of the sequence tested for stationarity: \( n_t = \max(\gamma n_0, \gamma \cdot n_0^*) \).
- \( \alpha \): the significance level of the stationarity test. Here is used \( \alpha=0.05 \).
- \( \gamma \): safety coefficient for the estimator of variance \( \sigma^2[\overline{X}(n)] \) to represent the steady state. Default \( \gamma = 2 \).
- \( \gamma \): exchange coefficient, determining the number of new observations included in each sequential test for stationarity. The default value is: \( \gamma=0.5 \).

Algorithm:

```
Procedure DetectInitialTransient;

Step1
Start the simulation run from the empty-and-idle state;
Apply the heuristic rule to determine \( n_0^* \)
if (the initial transient \( n_0^* \) is longer than \( n_{0,\text{max}} \) ) then goto step6
else discard the first \( n_0 \) observations
endif;

\( n_t = \max(\gamma n_0, \gamma \cdot n_0^*) \)
\( \Delta_n = \gamma \cdot n_0^* \) (\( n_t \) observations will be tested for stationarity, \( \Delta_n \) old observations will be replaced by new ones)

Step2
if \( n_0 + n_t \leq n_{0,\text{max}} \) then
append \( \Delta_n \) observations to the tested sequence.
goto Step3
else if \( n_0 + n_t > n_{0,\text{max}} \) then
goto Step6
endif;

Step3
Determine the variance \( \delta_{\text{ap}}[\overline{X}(n_t)] \) using the last \( n_t \), collected observations starting from the observation \( (n_0^* + n_t - n_0 + 1) \). Call the procedure (assuming \( x_s=x_{n_0+n_0^*-n_0+1} \) for \( s=1,2,...,n_t \)):
SpectralVarianceAnalysis;

Step4
(The test of stationarity: Schreuben statistical test)
```
Take all $n_0$ observations, starting from the observation $(n_0+1)$, and calculate the test statistic:

$$T = \frac{\sqrt{45}}{n_0^{0.5} n_v^{0.5} \delta[\bar{X}(n_v)]} \sum_{i=1}^{n_0} k \left(1 - \frac{k}{n_i}\right) [\bar{X}(n_i) - \bar{X}(k)],$$

where $ar{X}(i) = \sum_{j=n_0+1}^{n_0+i} x_j / i$

if $|T| \leq z_{1-\alpha/2}$, then

{ $z_{1-\alpha/2}$ is the $(1-\alpha/2)$ critical point of the standard normal distribution.}

The initial transient is not longer than the $n_0$ observations; start collecting observations for the estimation phase (steady state phase).

else (if $|T| > z_{1-\alpha/2}$)

discard first $\gamma \cdot n_0^*$ observations from the tested sequence;

$\hat{n}_0 = n_0 + \gamma \cdot n_0^*; \Delta_n = \gamma \cdot n_0^*$;

goto Step2

endif

Step6
Stop the simulation run

End DetectInitialTransient.

Algorithm:

Procedure SpectralVarianceAnalysis;
{
$X_1, X_2, \ldots, X_n$. A sequence of $n$ observations taken from a stationary process. $n_v = 100$.
$nap$: the number of points of the averaged periodogram used to fit it to a polynomial by applying the least squares procedure ($nap=25$).
$\delta$: the degree of the polynomial fitted to the logarithm of the averaged periodogram. $\delta = 2$.
$C_0$: a normalizing constant. $C_0 = 0.882$.

Step1
Calculate $2nap$ values of the periodogram of the sequence $X_1, X_2, \ldots, X_n$:

$$\Pi(j) = \frac{\sum_{j=1}^{n_v} x_j \exp \left[ - \frac{2m(s-1)j}{n_v} \right]^2 / n_v}{n_v}$$

for $j=1, 2, \ldots, 2nap$ and $i = \sqrt{-1}$

step2
Calculate $nap$ values of the function $\{L(f_j)\}$ for $j=1, 2, \ldots, nap$, where $f_j = (4j-1)/2n_v$

$L(f_j) = \log \left\{ \prod (2j-1)/n_v + \prod (2j/n_v) / 2 \right\}$

step3
Apply the least squares extrapolation procedure for determining the coefficient $a_0$ in the polynomial:

$g(f) = a_0 + a_1 f + a_2 f^2$

fitted to the function $\{L(f_j) + 0.270\}$, $1, 2, \ldots, nap$

(The value $a_0$ is the unbiased estimate of $\log(p_x(0))$)

step4
Calculate

$$\hat{p}_x(0) = C_0 \cdot e^{a_0} = 0.882 \cdot e^{a_0}$$

then the variance is: $\hat{\sigma}^2_x[\bar{X}(n_v)] = \hat{p}(0) / n_v$
Implementation of the algorithm:

Variance estimation
Methods used here are based on those described in \[PA90\]

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include ".../Bool.h"
#include "Variance.h"
#include "LeastSquares.c"

const double pi = 3.1415926536;

void LookUp_K_d(int K, int d, double &Cl, int &C2);
void CalculatePeriodogram(double X[], int n, double P[], int nP);
void LogAveragePairsAndOffset(double P[], double Lfj[], int K, double offset);

void SpectralVarianceAnalysis(double X[], int N, double &sigma_sq, int &kappa, Interface *pi)
{
    int K = 25; //PeriodogramPoints
    int d = 2;  //PolynomialDegree
    double Cl = 0.882; //Normalizing constant
    double P[2*K]; // P[j] = X(j/n) (periodogram values)
    double f[K];  // f[j-1] = (4j-1)/(2n)
    kappa = 7;

    CalculatePeriodogram(X, N, P, 2*K);

    for (int j = 1; j <= K; j++)
    f[j-1] = (4*j - 1) / (2.0 * N);

    LogAveragePairsAndOffset(P, L, K, 0.270);

    double a0 = LeastSquaresPolyAt0(f, L, K, d);
    double px0 = Cl * exp(a0);
    sigma_sq = px0 / N;
}

void CalculatePeriodogram(double X[], int n, double P[], int nP)
{
    for (int n = 1; n <= nP; n++)
    {
        double rsum = 0, isum = 0;  // Real and imaginary parts of sum
        for (int j = 0; j < N; j++)
        {
            double theta = -(2 * pi * j * n) / N;
            rsum += X[j] * cos(theta);
            isum += X[j] * sin(theta);
        }

        P[n-1] = (rsum * rsum + isum * isum) / N;
    }
}

void LogAveragePairsAndOffset(double P[], double Lfj[], int K, double offset)
{
    for (int j = 0; j < K; j++)
    
Alcatel Bell N.V.
\begin{verbatim}
double x = (P[2 * j] + P[2 * j + 1])/2;
// It is possible to get zero here, e.g. if X[] are all equal.
// Substitute a very small number to avoid singularities.
if (x == 0)
    x = 1e-38;
Lfj[j] = log(x) + offset;
\end{verbatim}