Software-Enabled Modular Instrumentation Systems

Software-Enabled Modular Instrumentation Systems

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op woensdag 26 november 2003 om 10:30 uur
door

Marco Willem SOIJER
ingenieur luchtvaart en ruimtevaart

geboren te Nijmegen

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. J.A. Mulder

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter
Prof.dr.ir. J.A. Mulder, Technische Universiteit Delft, promotor
Prof.ir. B.A.C. Ambrosius, Technische Universiteit Delft
Prof.dr. A.J. van der Wal, Universiteit Twente, Enschede
Prof.dr. M.H. van Emden, University of Victoria
Prof.Dr.-Ing. W. Alles, Rheinisch-Westfälische Technische Hochschule, Aachen
Prof.Dr.-Ing. P. Vörsmann, Technische Universität Carolo-Wilhelmina, Braunschweig
Dr.ir. M.M. van Paassen, Technische Universiteit Delft

ISBN 90-9017445-1

Published 2003 by Marco W. Soijer
Printed in the Netherlands by Ponsen & Looijen, Wageningen

AAN MIJN MOEDER

Software-gefaciliteerde Modulaire Instrumentatiesystemen

Soijer, M.W. (2003). *Software-Enabled Modular Instrumentation Systems* (Dissertation, Delft University of Technology), ISBN 90-9017445-1.

Samenvatting

De snelle ontwikkeling van computersystemen in de jaren 1980 en 1990 leidde tot de mogelijkheid om vliegproefdata-acquisitie en -verwerkingstechniek te integreren. Vliegende simulatoren, het genereren van adaptieve signalen in ware tijd voor experimenten in een gesloten lus en het analyseren van gegevens tijdens de vlucht zijn nieuwe toepassingen in het bereik van vliegproeven die in de afgelopen decennia voor het eerst werden gedemonstreerd. De bijbehorende introductie in vliegende toepassingen van apparatuur die traditioneel op de grond wordt ingezet, heeft tot een stijging van het aantal verschillende computersystemen geleid waarop de vliegproefgegevens worden verwerkt. De betrouwbaarheid en mogelijkheid tot verandering en uitbreiding van instrumentatiesystemen en verwerkingsgereedschappen hebben geleden onder deze paradox van computersysteemdifferentiatie. Om de situatie te verbeteren, is een algemeen toepasbare methodologie voor het ontwikkelen van vliegproefinstrumentatiesystemen voor signaalverwerking in ware tijd ontwikkeld, die zowel simulatie vóór, data-acquisitie tijdens, als gegevensverwerking na de vlucht omvat.

De methodologie gebruikt een object-georiënteerde benadering om het concept van intelligente instrumentatie – waarbij sensoren, actuatoren en signaalvoorbereiders worden geïntegreerd in componenten die via een standaardinterface communiceren – te implementeren in een op software gebaseerde omgeving die met conventionele sensoren en transducers samenwerkt. De omgeving maakt het mogelijk instrumentatiesystemen met een gesloten lus en real-timegedrag modulair te ontwikkelen. De omgeving draagt via de middleware zorg voor het activeren van sensoren, transducers en verwerkingscomponenten op de juiste tijd. Daarnaast stelt de middleware een gesynchroniseerde schatting van de ware tijd ter beschikking op alle locaties in het systeem.

De maatgesneden levenscyclus voor het ontwikkelen van dergelijke vliegproefinstrumentatiesystemen is een combinatie van het evolutionaire-prototypemodel en elementen uit throwaway prototyping en hiërarchisch ontwerpen zoals gebruikelijk in software engineering. De methode legt sterk de nadruk op het ontwikkelen van standaardcomponenten – zowel voor hardware als voor software – die in meer toepas-

singen zijn te gebruiken. De ontwikkeling van de componenten is daarom in een eigen levenscyclus geplaatst. Standaardcomponenten worden gespecialiseerd voor de specifieke toepassing; de levenscyclus van de toepassing bestaat voornamelijk uit het vastleggen van eisen en het maken van een algemeen ontwerp voordat de componenten worden ontwikkeld en de integratie van het systeem nadat de componentenontwikkeling is afgerond.

Afhankelijk van de wijze waarop verwerkte gegevens worden teruggekoppeld naar het testsignaal van het systeem, wordt een vliegproefsysteem geclassificeerd als een open-lussysteem, een adaptief systeem, een ergonomiesysteem, of een vliegende simulator. Als uitbreiding op de verzameling diagrammen die de Unified Modeling Language omvat, is een nieuw type noodzakelijk voor de analyse van een signaalverwerkingsssysteem. Het nieuwe diagram, genaamd signaaldiagram, modelleert de volledige gegevensstroom door het systeem, waarbij de nadruk op de gegevens zelf ligt en niet op de componenten die de gegevens produceren of verwerken. Samen met een contextmodel dat de typische structuur van de vier toepassingstypes reflecteert, vormt het signaaldiagram de basis voor het systeemontwerp.

De componenten van een meetsysteem worden onderverdeeld in drie groepen: platform-, acquisitie- en verwerkingscomponenten. Elke groep omvat zowel hardware- als software-elementen. Voor componenten die gegevens in- of uitvoeren, wordt maximale herbruikbaarheid bereikt door het ontwerp te verdelen in hiërarchische lagen. Een vierlagenmodel scheidt de in- en uitvoerhardware van de signaalverwerker, de bijbehorende besturingssoftware en bekabeling, de sensor of actuator en de software die met de andere componenten samenwerkt in gestandaardiseerde abstractieniveau's. Elk element is hierdoor uitwisselbaar zonder dat de andere delen hoeven te worden aangepast.

Middleware voor een digitaal signaalverwerkingssysteem dient een aantal extra functionaliteiten te bezitten in vergelijking met algemene middleware-standaards. Naast het verbinden van leveranciers en gebruikers van signalen, moet er een scheduler zijn voor het actieve beheer van de toestanden van de modules en moeten klokken op gedistribueerde locaties nauwkeurig worden gesynchroniseerd. Het concept van unconfined threads modelleert half-actieve modules in een gedistribueerd real-timesysteem. Elke module specificeert een interval dat het venster bepaalt waarin activering moet plaatsvinden. Na afloop van de berekeningen kan de module vrij een nieuw schedule-interval kiezen. Met unconfined threads vervalt de noodzaak voor toegevoegde activeringsmechanismen en worden elegante toestandsovergangen mogelijk gemaakt. De middleware omvat verder een nieuw kloksynchronisatiealgoritme, dat de stochastische informatie in het systeem van klokken optimaal benut om tot een synchronisatie te komen die zowel preciezer als nauwkeuriger is.

De nieuwe methodologie voor het ontwikkelen van modulaire instrumentatiesystemen, maar de geïntegreerde ontwikkeling van data-acquisitie- en data-verwerkingshardware en -software mogelijk voor alle fases van een testprogramma. Dataverwerkingscomponenten die zijn ontwikkeld voor bureausimulaties kunnen zonder aanpassing worden hergebruikt in gesloten-lussimulaties met hardware of een piloot in de lus,

voor vliegproeven en voor gegevensanalyse na de vlucht. Bovendien is optimaal hergebruik van de componenten van één project naar het volgende gegarandeerd. De methodologie resulteert op deze manier in afgenomen ontwikkelingstijd en -kosten en in toegenomen betrouwbaarheid van het systeem.

De nieuwe methodologie is ontstaan tijdens de vernieuwing van de instrumentatie in het laboratoriumvliegtuig van de Technische Universiteit Delft. In de zomer van 2001 is aan de Faculteit Luchtvaart- en Ruimtevaarttechniek onderzoek gedaan naar de cybernetica van een tunnel-in-de-sky display tijdens een daadwerkelijke vlucht. In voorbereiding op het vliegproefprogramma is de volledige ontwikkelingsmethodologie voor het eerst toegepast. Gezien de omvang van de nieuwe ontwikkelingen in het systeem en in vergelijking met het tijdschema van eerdere projecten, werd met een looptijd van vier maanden tussen projectbegin en de eerste vliegproeven een significante afname van de ontwikkelingstijd gerealiseerd. Bovendien hielp de test- en systeemintegratiefilosofie van de methodologie bij het voorkomen en verwijderen van implementatiefouten. Als gevolg hiervan heeft de daadwerkelijke looptijd van het project de geplande niet overschreden.

Software-Enabled
Modular Instrumentation Systems

Soijer, M.W. (2003). *Software-Enabled Modular Instrumentation Systems* (Dissertation, Delft University of Technology), ISBN 90-9017445-1.

Summary

The rapid development of computer systems during the 1980s and 1990s opened the possibility to integrate flight test data acquisition and data processing techniques. Real-time adaptive signal generation for closed-loop experiments, on-line data analysis, and in-flight simulation are new applications in flight test that have been demonstrated in the past decades. The corresponding introduction of traditionally ground-based equipment to airborne applications has increased the number of different computer systems on which data processing tasks are performed. As a consequence of what is referred to as the paradox of computer platform differentiation, reliability and maintainability of the instrumentation systems and processing tools have suffered. In order to improve this situation, a generic methodology is presented for the development of real-time signal processing systems in flight test, covering all of pre-flight simulation, in-flight data acquisition and processing, and post-flight analysis.

The methodology uses an object-oriented approach to implement the concept of intelligent instrumentation – which combines sensors, actuators, and preprocessors into components that communicate through interfaces that are standardized for all – in a software-based environment that is compatible with conventional sensors and transducers. The environment enables modular development of both distributed and real-time applications as they are encountered in closed-loop flight testing. By means of the middleware, it facilitates the activation of sensors, transducers, and processing components at the correct time and provides a synchronized notion of time throughout the nodes in a distributed system.

The life cycle model that is tailored to the development of such flight test instrumentation systems is a combination of the evolutionary prototyping model and elements from throwaway prototyping and hierarchical design as they are known from software engineering. The method strongly encourages the development of standardized components – both for hardware and software – that can be used in multiple applications. Component development is therefore placed in a separate life cycle within that of the application. Standardized components are specialized for use in the present

application; the application life cycle merely consists of requirement specification and architectural design before, and system integration after component development.

Depending on the way information is fed back from data processing to system excitation, each flight test application is classified as an open-loop testing system, an adaptive testing system, a human-factors testing system, or an in-flight simulation system. A new type of diagram is introduced as an extension to the set of diagrams that are offered by the Unified Modeling Language. The new diagram is referred to as the signal diagram; it models the complete flow of information through the application, while focussing on the information itself rather than on the components by which it is created or processed. Combined with a context model that reflects the characteristic structure of the four application classes, the signal diagram forms the basis for application design. The components in an application are categorized in three groups: platform, data acquisition, and data processing components. Each group covers both hardware and software elements. For data acquisition or publication components, maximum reusability of the hardware and software elements is achieved by applying hierarchical layers to the design. A four-layer model separates the port hardware of the digital signal processor, the corresponding driver software and wiring, the sensor or actuator, and the software that interacts with the other components at standardized levels of abstraction. Each of the elements can therefore be exchanged without affecting the other parts of the component.

Middleware for a digital signal processing system must provide additional functionality with respect to generic middleware standards. Apart from providing registration services that connect signal producers and consumers, it must include a scheduler that actively manages the states of the application modules, and it must accurately synchronize the real-time clocks of the distributed nodes. The concept of unconfined threads is used to model semi-active modules in a concurrent system. Each module can set an interval that defines the window in which it demands activation. Upon completion of its computation, the module can freely choose the new schedule interval. Unconfined threads remove the need for additional activation mechanisms in the middleware and facilitate graceful mode changes by the modules. The middleware includes a novel clock synchronization algorithm, referred to as the probabilistic peer-to-peer algorithm, which exploits the stochastic information in the system of clocks to yield more accurate and more precise synchronization.

The new methodology for developing modular instrumentation system enables the integrated development of data acquisition and data processing hardware and software for all phases of a test and evaluation program. Data processing components that are developed for desktop simulation can be reused without adaptation in hardware- or pilot-in-the-loop simulation, in flight test, and in post-flight data analysis. Moreover, optimum reusability of the components from one test program to the next is ensured. The methodology thus results in reduced system development time and cost, and improved system reliability.

The new methodology for instrumentation system development originated during an upgrade project for the instrumentation of the Delft University laboratory aircraft. Summer 2001, the Faculty of Aerospace Engineering investigated the cybernetics of a tunnel-in-the-sky display in actual flight. In preparation to the flight test program, the fully matured development methodology was applied for the first time. Considering the new developments that were incorporated in the system and comparing the four-month time schedule between project initiation and flight test to that of previous projects, a significant reduction in development time was achieved. Moreover, the methodology's testing and system integration philosophy helped to avoid and eliminate implementation flaws. As a result, the project's original time plan was kept.

Contents

Nomenclature xvii

Introduction 1

1. Development Philosophy 17
   1 Life cycle models 18
   2 Concepts and strategies of object orientation 26
   3 Concepts and strategies of concurrency 29
   4 UML notation for object-oriented modeling 38
   5 The flight test instrumentation development life cycle 39

2. Application Development 45
   1 Requirements analysis 47
   2 Context analysis 53
   3 Design 59
   4 Synthesis 61
   5 Documentation and maintenance 65

3. Component Development 69
   1 Analysis and prototyping 72
   2 Hierarchical layers 75
   3 Interfaces 77
   4 Modules in data acquisition and processing 83
   5 Design and implementation 86

4. A Middleware Pattern 91
   1 Requirements 93
   2 Time 98
   3 Scheduling policies and performance 105
   4 Architecture 111
   5 Application topology 116
   6 Activities 121

5. Clock Synchronization 131
1 Probabilistic peer-to-peer synchronization 132
2 Practical aspects 143
3 Simulations 146
4 Activities in the middleware 153

6. Case Study in Human-Factors Testing 159
1 Application modeling 161
2 Component development 176
3 Application synthesis 182

Discussion 187

Acknowledgments 193
References 195

Appendix
A Unified Modeling Language 205
B Activities in Instrumentation Development 217
C Bayesian Estimation 223
D Case Study Implementation Details 227

Glossary 229
Index 241

Nomenclature

Sets

R	all real numbers	
R^n	all n-tuples of real numbers	$\{[x_1 \ldots x_n]^\mathsf{T}\},\ x_i \in \mathrm{R}$

Symbols

B	confidence matrix	(C.12)
b	parameter vector	
$\mathrm{b}\|y$	parameter vector posterior to observation y	
C_{xy}	covariance matrix of x and y	$E\{(\mathrm{x}-E\{\mathrm{x}\})(\mathrm{y}-E\{\mathrm{y}\})^\mathsf{T}\}$
$C\langle x\rangle$	global clock value for event x	
$C_p\langle x\rangle$	local clock value for event x in process p	
$C_i(t)$	logical clock value for time t on node i	
D	communication delay	
D_{xy}	average delay between nodes x and y	
d	zero-mean, random delay	
d_i	deadline of job i	
$d_{\mathrm{boc},\,i}$	beginning-of-computation deadline of job i	
$d_{\mathrm{coc},\,i}$	completion-of-computation deadline of job i	
$F_x(a)$	probability distribution function of x	$P_x[-\infty < x \le a]$
G_i	granularity of clock on node i	
$\hat{G}_i$	normalized granularity of clock on node i	
L_i	logical clock offset on node i	
$\hat{L}_i$	normalized logical clock offset on node i	
l_i	laxity of job i	$d_{\mathrm{coc},\,i} - s_i - t$
m	measurement noise	
N	counter range	
$P_x[a]$	probability of occurrence of event $a(x)$	
p_x	probability density function of x	$\frac{dF_x}{dx}$
$p_{x,\,y}$	joint probability density function of x and y	
$p_{x\|y}$	conditional probability density function of x given y	

r_i	release time of job i	
T	counter base	
T_e	external reference time	
t	time	
t_i	arrival time of job i	
$\mathbf{u}$	input vector	
$\mathbf{y}$	output vector	
s_i	computation time of job i	
s_x	standard deviation of scalar x	$\sqrt{C_{xx}}$
$t_i(t)$	counter value on node i at time t	
t'_i	artificially rolled counter value on node i	
$\hat{t}_i$	normalized counter value on node i	
u	output observation	

Delimiters

$\mathrm{x}_{[i]}$	element i of vector $\mathbf{x}$	
$A_{[ij]}$	element i,j of matrix A	
(a_{ij})	matrix $A \in \mathrm{R}^{n\times m}$	$\begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix}$
$\lvert A\rvert$	determinant of square matrix A	
$\{x\}$	set of all elements x	
$[f]_a^b$	change of function f over interval $[a, b]$	$f(b) - f(a)$

Operators

$E\{x\}$	expected value of x	$\int_{-\infty}^{\infty} p_x(x)\, x dx$
A^{T}	transpose of matrix A	
$\frac{\partial}{\partial \mathbf{x}} f$	Jacobi matrix for function $f(\mathbf{x})$: $\mathrm{R}^m \to \mathrm{R}^n$	$\left(\frac{\partial f_i}{\partial x_j}\right) \in \mathrm{R}^{n\times m}$

Relations

$\in$	belongs to	
á	happened before	Lamport (1978)
$\rightarrow$	is mapped to	
$\Leftrightarrow$	entails and is entailed by	

Abbreviations and acronyms

ADC	analog-digital converter
AGARD	Advisory Group for Aerospace Research and Development
AIAA	American Institute of Aeronautics and Astronautics
COMET	Concurrent Object Modeling and Architectural Design Method
CORBA	Common Object Request Broker Architecture
CPU	central processing unit
DSP	digital signal processor
DUECA	Delft University Environment for Communication and Activation
EDD	earliest due date
EDF	earliest deadline first
ESA	European Space Agency
GMT	Greenwich mean time
GPS	Global Positioning System
HOOD	Hierarchical Object-Oriented Design
HUD	head-up display
ICAO	International Civil Aviation Organization
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
LLF	least laxity first
NTP	network time protocol
OMT	Object Modeling Technique
OOM	object-oriented modeling
PCM	pulse code modulation
RMA	rate-monotonic analysis
SDC	synchro-digital converter
SI	International System of Units
SJF	shortest job first
TAI	international atomic time
UML	Unified Modeling Language
USN	United States Navy
UT	universal time
UTC	coordinated universal time
VHF	very high frequency

Introduction

THE history of flight testing started on Thursday December 17th, 1903, when Orville and Wilbur Wright achieved the very first powered, sustained, and controlled flight. Not only was the flight of the Flyer 1 preceded by several years of careful research and development, including a series of wind tunnel tests, the results of the trial itself were recorded in a way that is typical for the Wright brothers' engineering approach. Flight time and distance were meant to support their claim that they had actually succeeded in conducting a flight with a heavier-than-air machine, but the registration of wind speeds, both from a nearby meteorological station and from personal measurements, was unmistakably meant to provide data for further development of the aircraft (Gibbs-Smith, 1960, pp. 226-227).

Wright Flyer 1, flown by Orville Wright on December 17, 1903.
This photograph by John T. Daniels is the only one of what is to be regarded as the first flight test in history. The missing lower left corner is a chip from the glass negative.

There are many aspects in which the achievements of the Wright brothers differ from those of other aeronautical pioneers. Setting the stage for an engineering practice that is now known as flight testing, is certainly one of them. Unlike famous predecessors, among who Lilienthal and Chanute deserve the credit for documenting their contributions to aviation in a scientific manner (Gibbs-Smith, 1960, p. 32), the Wrights were not satisfied by the mere success of performing the world's first flight in

the presence of witnesses. In addition, they introduced the practice of measuring physical quantities that are related to the flight, and which are of great importance to assessment of the trials.

Ever since, flight test instrumentation systems and testing techniques have kept pace with the development of aviation technology itself. The continuous increase in airspeed, flight altitude, range, and duration, as well as the growing number of parameters that needed to be monitored, required improved and more powerful data acquisition and data recording systems with every new generation of aircraft. Because of the required capability to withstand the harsh environment in which airborne equipment is to operate, the development of flight test instrumentation systems is an engineering discipline in itself. Yet, instrumentation development has not only been motivated by the continuous progress in aviation and space technology, but has also been among the first to apply new technology arising from it. For example, the development of digital solid-state computers during the Apollo program in the 1960s, provided the world with the hardware to develop the information technology that has found application in both flight test instrumentation and all of society.

The rapid development of computer systems during the 1980s and 1990s opened the possibility to integrate data acquisition practices with flight data processing techniques. On-line parameter estimation, full accuracy quick-look data assessment, and in-flight simulation are a few of the applications of flight test instrumentation and data processing that have become reality. Nevertheless, these applications have so far been cases of pioneering work that remained isolated. Pushing the limits of computer performance, both hardware systems and software environments have been dedicated developments that lacked usability to other applications. A more generic approach to the development of real-time computer systems for flight test data acquisition and processing is desirable, if not necessary.

Flight test instrumentation

The term *flight test instrumentation* refers to all equipment that is installed on board an aircraft with the purpose to excite, measure, condition, record, process, or display flight data or aircraft parameters, and that is not a part of the aircraft's standard systems. A flight test instrumentation system comprises multiple subsystems, which consist of chains of components. In general, the *excitation chain* contains the equipment that excites the aircraft or system that is to be tested by injecting meaningful test signals; it is connected to the *measurement chain* by means of the system under test itself. The measurement chain covers all the components that convert the physical phenomena that are observed into the numeric data that can be used for processing. Data recording is the final stage in the measurement chain. When data processing is performed on-board, the *processing chain* can be attached directly to the measurement chain. Otherwise, data replay is a separate task that is covered by the *reproduction chain*. Data replay includes both off-

line data reproduction from a storage medium that was recorded on board, and on-line reproduction through a telemetry downlink. Because the reproduction chain is ground based and not part of the airborne instrumentation, it is traditionally not a part of flight test instrumentation. Likewise, the processing chain is only considered to be a part of flight test instrumentation when the processing is performed on-line. Treatment of instrumentation system design in literature, for example by Borek and Pool (1994), is often limited to the development of the measurement chain.

The measurement chain begins at the physical phenomena that are to be observed, and includes – but is not necessarily limited to – sensors and transducers, signal conditioners, filters, samplers and multiplexers, digitizers, and recorders. When telemetry is used, the data link transmitters and receivers can be inserted anywhere in the measurement chain, although immediately before the recorders is the most common solution. The excitation chain is the smallest of the subsystems in flight test instrumentation and is often not present at all. The excitation chain covers all equipment that is used to excite the aircraft or system under test, in order to obtain meaningful test data. For many performance and handling qualities test flights, system excitation is performed by the test pilot and does not require any additional equipment. For aircraft systems testing, for flight control system testing outside the dynamic range of the pilot, or in cases where optimal input signals need to be reproduced with a level of accuracy that is unobtainable for the human pilot, signal generators and actuators may be used. These excitation chain components are connected to the rest of the flight test instrumentation system via the system under test only, although in some cases a signal generator might be connected to the data recorder. The processing chain begins at the end of the measurement chain, but cannot be defined in terms of equipment as easily as the measurement chain. Data replayers are the starting point for processing, but the rest is highly dependent on the type of application.

Real-time data processing is defined as the computations or actions for which success or failure not only depends on the correctness of the results, but also on the time at which these results become available (Stankovic and Ramamritham, 1993, p. 1). It is noteworthy that this definition does not specify the size of the time margin that is available for processing, neither in absolute terms nor relative to the operating speed of the system. Systems that exhibit a significant time lag are therefore still regarded as real time, as long as the delay is bounded to a maximum. All components of the excitation and measurement chains typically operate in real time. The recordings that result from the data acquisition processes in the measurement chain, are time traces of the parameters at hand. Data processing can be conducted in real time, but non-real-time processing is more common. When data from time traces is not processed in real time, the computations proceed with the next time point after the completion of the current step. The process clock is thus continuously increased with the step size of the time trace and not linked to the real, wall clock time. To refer to this kind of processes, the term *step-time processing* is introduced.

A traditional flight test instrumentation system consists of the excitation and measurement chains only. The data recording equipment in these systems has gradually been replaced by digital devices, often using solid-state recorders. The rapid development of digital computers in the last few decades has resulted in increased computing power at low cost and with high reliability. Apart from using digital computers for data acquisition and recording tasks, this development led to the introduction of on-line data analysis and visualization. In the old situation, the measurement chain and processing chain would be completely separated. Software components would therefore be developed for either airborne application in the real-time measurement chain, or for step-time application in the processing chain. Powerful digital equipment was introduced to flight test with the purpose the simplify the complete process of data acquisition and analysis. The use of airborne computer equipment – which was initially meant to make data acquisition easier and more robust – for on-line analysis and visualization, transferred part of the data processing software to the measurement chain hardware. However, the same software components are still used for step-time analysis. Not only will raw data recordings have to undergo identical or similar processing for the final analysis as during quick-look analysis, simulated data will be applied to all algorithms before embarking on the flight test program, in order to validate the correctness of both the designed algorithms and their software implementations. The existence of different computer systems, operating systems and software environments for airborne data acquisition, post-flight processing and pre-flight simulation thus requires porting of analysis software from one computer system to another and from one software environment to another. The result is the paradox of computer platform differentiation: the introduction of powerful digital equipment to airborne applications, has increased rather than reduced the number of different computer systems on which certain tasks in the data processing chain are performed. Separate sets of software must be developed, from which maintainability and re-usability of computer code have suffered.

Nevertheless, the possibility for real-time data processing as provided by the replacement of dedicated data recording equipment by generic digital signal processors (DSPs), greatly enhances the process of flight test by presenting data analysis results to the flight test engineer during the execution of the experiment. Present-day computer interfaces allow for interactivity in the data display system, allowing the operator to monitor all of the raw data, generated results and related processes. Malfunctioning sensors or interfaces, inadequate test maneuvers or weather conditions, and other obstacles for meeting the trial's goals can be recognized and possibly obviated during the flight. This can improve both effectiveness and efficiency of a flight test campaign and largely reduce the related costs. An example was presented by Laban (1994), who developed a real-time flight test instrumentation system for on-line estimation of aerodynamic model parameters. For safety critical flight tests, for example envelope expansion for a new type of aircraft, the possibility to identify malfunctions or unexpected system behavior in real time, extends the benefits of on-line data pro-

cessing from economical issues to a matter of improved safety. Yet, without mitigation of the economical and safety-related benefits of real-time data analysis, the single largest change associated with the introduction of digital signal processors is the possibility to attach output interfaces to the system. The system no longer only records and processes the information coming from its environment, but it can also influence that environment by providing new inputs that are based on the outcome of previous measurements and the operator's decisions.

A DSP-based flight test instrumentation system therefore no longer needs to be at the end of a one-way information chain, but can be part of a closed-loop system. For traditional open-loop systems as shown in figure 1, the system being tested is enclosed between the input generators – either an automatic signal generator, autopilot, or the test pilot or flight test engineer who controls the aircraft or system – and the measurement chain. Even when automatic signal generators are used that are considered to be part of the flight test instrumentation system's excitation chain, the only link between the input generators and the data acquisition hardware is the test subject. This is different for the *closed-loop flight test instrumentation systems* as depicted in figure 2. The signal generators are not independent, but connected to the output interfaces of the digital signal processor. The additional link provides a way to feed back the results of the measurements to the system under test.

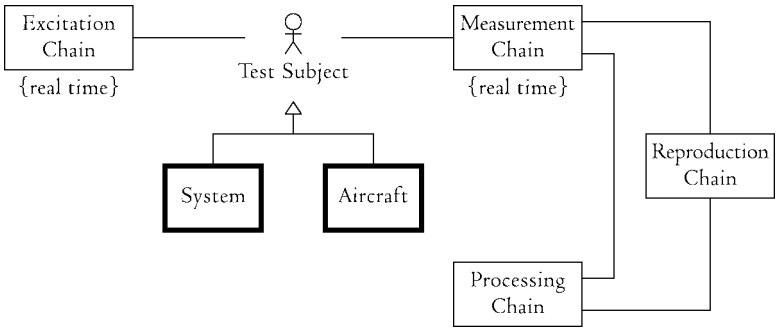


Figure 1: Open-loop flight test instrumentation system.
The diagram shows a static model in conceptual perspective according to the Unified Modeling Language (UML). The system under test, either the aircraft or any (sub)system, is external to the instrumentation system and therefore indicated as an actor. The excitation and the measurement chain are associated directly with the test subject; both must therefore operate in real time. As data processing for open-loop systems is not subjected to a time constraint, the processing chain does not necessarily provide real-time delivery of the analysis results.
The UML is discussed in section 1.4; an overview of the UML notations that are used in this thesis is presented in appendix A.

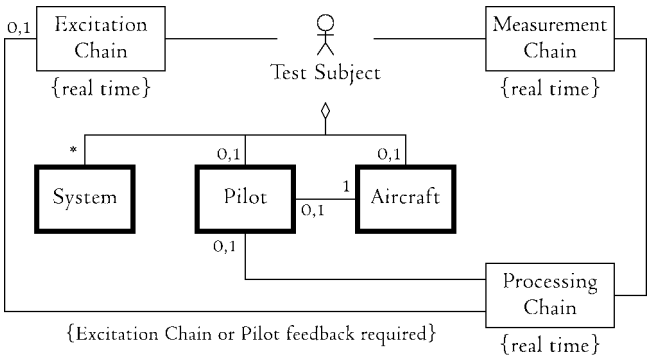


Figure 2: Closed-loop flight test instrumentation system.
Like figure 1, the diagram shows a static model in conceptual perspective according to the Unified Modeling Language (UML). The test subject is an arbitrary combination of the aircraft, a pilot, and any number of systems. Excitation of the test subject is controlled from the processing chain. Loop closure is obtained through the excitation chain, addressing any component of the system under test, or through the human pilot directly. An example of the latter is the feedback of analysis results to the pilot by means of a display system. Pilot and excitation chain loop closure are not mutually exclusive.

A closed-loop flight test instrumentation system is a prerequisite for experimental handling qualities flight testing, in which the developmental components are implemented as part of the instrumentation system. For this type of flight tests, the system under test is the combination of a pilot, the aircraft, its flight control system and possibly experimental display or inceptor systems. A digital signal processor can simulate the behavior of the components that need to be evaluated, while recording all the relevant data, including those that are part of the experimental subsystem and that may not be available when the test would be performed in a traditional fashion. Two characteristic examples of experimental handling qualities testing are the assessment of a flight guidance display system, and the evaluation of a fly-by-wire control law. In the first case, the DSP gathers the information that is required for composing the display, creates the image, and presents it to the pilot. The pilot controls the aircraft using the existing control system. The loop closing thus contains the display, the pilot, and the control system respectively. In the case of fly-by-wire control law evaluation, the DSP gathers the pilot's commands from the inceptors and the flight data that serve as input for the flight control laws; it then performs the necessary computations and uses the results to adjust the aircraft's control actuators. In this case, the control laws and the actuators form the closed loop, while the pilot acts as an open-loop input to the flight test instrumentation system. A combination of both typical closed-loop flight tests is

possible as well; the full system then contains two feedback loops, one through the pilot and one through the flight control system.

Because of the replacement of conventional data recorders by digital signal processors, the role of instrumentation components like signal conditioners, filters, analog-digital converters and multiplexers, is taken over by data acquisition boards that can be connected to the digital signal processing system. The integration of flight test instrumentation system components into off-the-shelf boards drastically reduces system design and preparation times. As a result, the traditional roles of the flight test engineer and the instrumentation engineer in the system development process are easily obscured. These traditional tasks and responsibilities are outlined by Adolph (1994), Knight and Dove (1994), and Crounse (1995). The instrumentation system design is centered around the composition of a *measurand list*†. During the composition of the measurand list, it is the responsibility of the flight test or system engineer to ensure that all parameters that are needed for analyzing the flight test, are recorded with the required accuracy, range, resolution, and data rate. The instrumentation engineer should counterbalance the desires of the flight test or system engineer by proposing adjustments to the measurand list, in order to match the capabilities of commonly supplied or stock sensors and transducers. Thus, the instrumentation engineer prevents the system under design from becoming excessively expensive or from using special case solutions for standard applications.

For traditional flight test instrumentation systems, the development process described above results in the measurand list as the only record of instrumentation characteristics, including the set of measurands and the corresponding ranges, resolutions and accuracies. From an information management point of view, this is a desirable situation. The absence of multiple, local copies of the same information ensures consistency of the data that are being used by the flight test engineer, the instrumentation engineer and all other parties involved. Data redundancy poses a significant threat to the maintenance of the instrumentation system documentation, since data anomalies are easily introduced. An important integrity constraint for any record of information is therefore the uniqueness of data. In information technology theory, this uniqueness is part of the concept of database *normalization* (O'Neil and O'Neil, 2001, pp. 353-358). However, when a flight test instrumentation system is constructed using off-the-shelf components, extensive sets of documentation for these components are maintained by the instrumentation engineer. These documents contain information on possible measurands, together with the available measurement characteristics. For the preparation of a flight test program, flight test and system engineers still have the responsibility to compose the list of measurands and required characteristics. The different origin of the application and the stock sensor measurand list is easily

† Although the phrases *measurement list* and *parameter list* are used in the same sense, the term measurand list is preferred, because it refers to the quantities that are to be measured rather than the numbers that result from the measurements.

overlooked, especially when during the negotiation process between the flight test or system engineer and the instrumentation engineer, the specifications of the available instruments are transferred to the required measurand list. Additionally, data redundancies between the instrumentation background documentation and the current flight test measurand list are likely to be introduced. The concept of the measurand list therefore exhibits some inherent disadvantages for instrumentation systems that are composed from off-the-shelf components.

Object-oriented modeling and intelligent instrumentation

Although state-of-the-art computer hardware is well represented in flight test instrumentation systems, modern developments on the more theoretical side of information technology have so far hardly found application in instrumentation design. The most important of these developments, which is also the most promising for contributing to a solution to the problems encountered in flight test instrumentation design, is *object-oriented modeling* (OOM). In the 1970s and 1980s, object-oriented programming arose as a software development paradigm (Parnas, 1972; Parnas, 1979; Meyer, 1987). Afterwards, during the 1990s, the concept of object-oriented design was extended to system design in general (Rumbaugh et al., 1991; Coad and Yourdon, 1991). Recently, object-oriented modeling has been extended with techniques to design *concurrent applications* (Douglass, 1998; Gomaa, 2000).

Concurrent applications are characterized by the simultaneous occurrence of multiple events, which are related to multiple activities that take place in parallel. Concurrency is a typical characteristic of real-time systems and distributed systems. In a real-time system, incoming events occur in an unpredictable manner. The order in which they arrive is arbitrary and one incoming event might overlap the handling of another. Yet, all events must be responded to in a timely manner. A distributed system is a system that consists of multiple nodes, each of which is assigned a part of the activities that make up the full application. These activities run in independent threads of control on the multiple nodes. Flight test instrumentation systems are inevitably at least partly real-time. As will be demonstrated below, they are a natural candidate for implementation as a distributed system as well.

The object-oriented development paradigm is based on the breakdown of a system in a set of cascading subsystems and on separation of interface and implementation. The partitioning of a system into subsystems, referred to as objects, is carried out with a focus on the function of each subsystem, rather than the task. The function of an object is its contribution to the overall system or the environment, and thus the reason for its existence. The task of an object is the collection of actual activities that need to be completed to fulfill its function. The function of an object is therefore the goal of the tasks that are carried out. By focussing on the function of subsystems instead of the tasks, the possibility to separate interface and implementation is cre-

ated. The interface is the boundary between the object and the surrounding systems, through which all communication takes place. As a result, the interface defines the function of the object. To the contrary, the implementation is closely related to the object tasks. It defines how the object perform its task and with that, how it fulfills its functions. The separation of interface and implementation is the rationale for the term object-oriented design; the alternative is procedural design in which the focus is on the tasks and implementation.

Object-oriented design provides a solution to the problems that arise when modern computer equipment is used in a closed-loop flight test instrumentation system. As explained in the previous section, a trade-off exists between flexibility of a system in terms of adjustability, and flexibility in terms of ease-of-use. The assembly of an instrumentation system from integrated off-the-shelf components will greatly reduce costs with respect to a custom-made system, but the flexibility to apply various combinations of sensors, transducers, filters, and converters is lost. With the object-oriented approach, the instrumentation is considered as a system of which the signal processing components are subsystems. By standardizing the interface between each pair of subsystems, any object can be replaced by an equivalent unit without modification of the rest of the system. Separate data paths are modeled by systems at a common level; sensors, actuators, filters, converters, or amplifiers for a single data chain are modeled as cascading subsystems to a single supersystem.

Decentralization of data acquisition components is a natural outcome of the object-oriented separation of interface and implementation for flight test transducers. In a traditional system, only sensors and actuators are located in the vicinity of the physical quantities that are to be measured, or the aircraft systems that are to be excited. All other components are grouped around the core of the instrumentation system. In an object-oriented layout, components which are related to the sensors and actuators, like filters and converters, are relocated with the sensors and actuators. Communication with the core of the instrumentation system takes place through standardized channels, implemented as databuses that interact with the interface of the sensors. This way, remote parts of the system are replaceable as long as the communication protocol remains unchanged. The actual implementation, the way the physical quantity is converted into the standardized data that is handed over through the interface, is hidden from the overall system and thus of no importance to its design. Although the concept of instrumentation decentralization in itself does not require a physically decentralized implementation, it is most easily realized by developing integrated instrumentation components that physically combine sensor or actuator hardware with their corresponding processing components.

Integrated transducers in which the original sensor or actuator, additional signal processing components, and a digital interface are combined, are referred to as *intelligent instruments* (Kopetz, 1997), also known as *smart sensors*. The most important advantage of intelligent instrumentation, is the provision of *agreed data* to the rest of the system. Agreed data is defined by Kopetz (1997) as a data element that has been checked

for plausibility. Thus, agreed data can reach a high level of reliability by identifying and either correcting or disregarding faulty data. As a result, agreed data can be used reliably without additional effort on the receiver side and without relying on implementation-specific characteristics. Because agreed data is expressed in a standardized format and has been checked for plausibility, the use of intelligent instrumentation moves the tasks that are often referred to as *data preprocessing*, for example the removal of spikes or drop-outs, from the processing chain to the measurement chain. Other benefits of intelligent instrumentation are a reduction in instrumentation cabling, because the digital interface for the transducer allows for the use of a single network bus for multiple instruments, and a potential increase of signal fidelity because analog to digital conversion is carried out close to the sensor itself and no additional noise can disturb the signal during transport.

An industrial standard for intelligent instrumentation is currently being developed. The IEEE has so far issued two parts of a standard on smart sensors (Institute of Electrical and Electronics Engineers, 1997; Institute of Electrical and Electronics Engineers, 1999). Application of the standard is not limited to aeronautical or astronautical engineering; the intended audience are measurement and control engineers in general. An additional two parts of the standard are under preparation. Notwithstanding the broad scope of the document being prepared, the working group for part three is dominated by the aerospace industry. It is aimed at the development of a network structure that can be used to provide remote units with both power and a communication bus over a single pair of wires (Eccles, 2000). The result will be a maximum reduction in instrumentation wiring, accompanied by a maximum increase in flexibility. The latter is achieved because additional transducers can simply be directly connected to the network. Eccles (2000) compares the current practice of building a flight test data acquisition system around a pulse code modulation (PCM) encoder, with the proposed solution using intelligent instrumentation. PCM allows for multiplexing of many data channels into a single stream; timing is controlled by the PCM unit using a built-in crystal oscillator. Data skew is therefore unavoidable, but well controllable. Using smart sensors, simultaneous sampling and analog to digital conversion is made possible, but common timing between all nodes in the network is not straightforward. Part three to the IEEE standard on smart sensors aims to develop a network protocol that provides the remote sensors with a common time base.

Although the new IEEE standard for intelligent instrumentation is promising, the lack of a common time base for all components can be a precluding problem for application in flight test systems that require high data fidelity. In addition, the new concept is incompatible with existing instrumentation equipment; many sensors may not be and might never become available as smart instruments. Finally, changes in transducer settings like sample rates or digitization resolutions might necessitate interventions at the transducer's physical location, which is generally more difficult to reach than the location of the core instrumentation system components. Consequently, a solution that allows for object-oriented modeling of a flight test instrumentation sys-

tem, while preserving the centralized layout of existing technology, is desirable. Such a solution can be found by designing the instrumentation system as an object-oriented distributed system. This way, the advantages of intelligent instrumentation can be utilized by implementing the data acquisition components as objects on a series of nodes in the distributed system. The nodes are selected in a way that optimizes the trade-off between flexibility and system simplification of the decentralized approach, and the compatibility and physical accessibility of the centralized approach.

Closed-loop flight testing

The increased possibilities for closed-loop instrumentation systems that have emerged with the availability of digital signal processors, are not only a chance for increased safety and efficiency in flight test, but rather a necessity for the flight testing demands of the future. Modern avionics and flight control systems are dominated by digital computers. Their importance, including their influence on operating procedures and handling qualities of the aircraft, is likely to increase further. Design of higher-order flight control systems using digital computers is still subject to investigation. In fact, the potential for designing aerodynamically unstable aircraft, combined with the desire to design for *carefree handling* or *flight envelope protection* – the respective military and civil terms for a flight control system that protects the pilot from leaving the flight envelope – has put back the existing knowledge on design for handling qualities with respect to the possibilities for implementing the resulting control laws as offered by the airframe and the flight control system. To be able to flight test experimental higher-order control systems in a safe and cost-effective way, a closed-loop instrumentation system is required. By means of *in-flight simulation*, control laws can be evaluated in real flight before the aircraft for which they are designed is available, and with the increased safety of being able to revert to the original control system of the simulating aircraft. As described by Gibson (1999), in-flight simulation is an excellent way to gain knowledge on new fly-by-wire control laws and its appropriate use might prevent accidents during flight test. The equipment used for in-flight simulation, is in fact a closed-loop instrumentation system as previously shown in figure 2.

An overview of various possibilities to assess aircraft handling qualities is given in figure 3, which was adapted from Höhne (2001). In-flight simulation is an element in a continuous range of testing methods that starts with off-line simulation and ends with flight test of the flight control system that is being evaluated. The existence of four different experimental concepts, completed with off-line simulation in which the test pilot is replaced by a pilot model, underlines the resolving distinction between tools and methods for data acquisition, simulation, and processing that was noted earlier as part of the paradox of computer platform differentiation. Another sign for this is the separate classification of flight test and in-flight simulation, where the latter is a type of flight test in itself. This distinction is correct when regarded from the view-

point of categorizing possibilities for handling qualities assessment, but from an instrumentation point of view, in-flight simulation is more closely related to flight test than to fixed-base or motion base simulation.

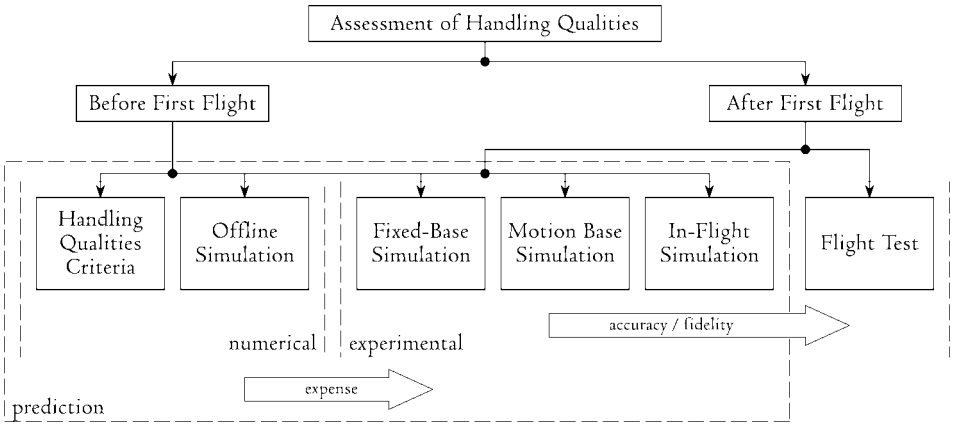


Figure 3: Possibilities of handling qualities assessment (adapted from Höhne, 2001, p. 6).
Apart from the application of design criteria, flight test and four classes of simulation form a continuous range of testing resources to analyze handling qualities with increasing fidelity at the cost of increasing expenses.

Further research on higher-order flight control systems is not the only reason for the growing demand for closed-loop flight testing in the area of handling qualities. Aircraft navigation systems have developed from stand-alone devices to sources of navigation information that are connected to a central processor. In the traditional situation, a typical navigation aid consists of an antenna, a receiver with a processing unit, and an indicator. All civil aeronautical radio navigation systems that are currently in use adhere to this principle, including automatic direction finders (ADFs), VHF omnidirectional radio range receivers (VORs), distance measuring equipment (DMEs), and instrument and microwave landing systems (ILSs and MLSs). The International Civil Aviation Organization (ICAO) regulates the use of radio navigation aids on board of civil aircraft, by demanding certain systems to be installed and by specifying the navigation accuracy that must be achieved with each of these (International Civil Aviation Organization, 1996). However, these requirements do not suit the recent development towards integrated avionics and hybrid navigation systems. In these systems, the navigation information from at least two types of equipment is fused to yield an optimal estimate of the aircraft's position. The navigation equipment that can be used in a hybrid system, is not limited to ground-based radio navigation aids, but can also include inertial or satellite navigation systems. A central processor, for example a flight

management system or flight control computer, now delivers the combined navigation solution for use by the autopilot, or for presentation on a single display. Alternatively, many satellite navigation systems accept supporting information from an inertial or radio navigation system, in order to provide a hybrid navigation solution directly. For all of these systems, the combined navigation accuracy and reliability are more important than the characteristics of the individual sensors.

To address the problem of inappropriate requirements for aeronautical navigation aids, Kelly and Davis (1994) introduced the concept of *required navigation performance* (RNP). It relates the accuracy of a navigation system to the required performance that ensures safe aircraft guidance during all-weather operations, without referring to the specific characteristics of the navigation system at hand. The requirements, characterizing the navigation system performance in terms of accuracy, integrity, continuity and availability, are defined for each type of airspace in which the aircraft is operated. To satisfy required navigation performance, an aircraft's *total system error* (TSE) must exceed certain deviations from a nominal flight path with sufficiently small probability only. Total system error is the sum of *navigation sensor error* and *flight technical error*. The first is the discrepancy between the aircraft's actual position, and the position indicated by the navigation system; the latter is the error between the position that is indicated by the navigation system, and the desired position of the aircraft. Because flight technical error is a part of total system error, the accuracy at which an aircraft's position as it is known can be controlled to equal the desired position, plays an important role in achieving required navigation performance. Thus, the aircraft's handling qualities affect the overall navigation performance. Their influence may become significant during manually controlled phases of flight with tight navigation requirements. It is noteworthy in this respect that manual control and the most stringent navigational requirements for civil air traffic coincide during final approach and landing.

ICAO is in the process of accepting required navigation performance as a replacement for the existing radio aids specifications, in which only the performance of the navigation equipment is regulated. Once RNP has been accepted as an ICAO requirement, the concept of total system error effectively forces all future certification flight tests for navigation and flight control systems to be a combination of an aircraft systems flight test, in which the navigation sensor error is assessed, and a human-factors or automatic flight controls test, in which the flight technical error is assessed. During the development phase that precedes the certification, a closed-loop instrumentation system can be used to assess adjusted flight control system or pilot display characteristics. If the instrumentation system is equipped with its own high-accuracy navigation sensors, it can be used to insert known errors in the navigation solution, in order to investigate the effect of navigation sensor error. Additionally, a navigation solution can be obtained that can be used as a reference for validating navigation sensor error during both development and certification.

Motivation and project scope

Since the inception of flight testing in 1903, data acquisition and data processing equipment have matured with both aircraft and signal processing technology. In the last three decades of the 20th century, developments in digital computer hardware found their way to flight test instrumentation. Nowadays, all of data processing and most data acquisition systems are completely digital. Measurement chains are controlled by microprocessors throughout and data storage is carried out using solid-state recorders. Yet, the practice of instrumentation system design and maintenance has hardly changed since the beginning of flight test.

Consequently, flight test data acquisition systems and data processing equipment do not form an integrated environment for system analysis, although both are now largely based on the same equipment. Software for data processing thus has to be redeveloped for simulation purposes, on-line application purposes, and post-flight analysis purposes. Digital signal processors allow for rapid assembly of an instrumentation system with standard components, but pose a threat to instrumentation suitability for a specific application, and may endanger documentation reliability by obscuring the importance of a measurand list. Recent developments in intelligent instrumentation may offer a solution to the latter problems, but are far from being available as stock products and are not easily combined with existing instrumentation. A gradual transition to smart sensors, or application of smart sensors in combination with custom-made instruments, is therefore difficult to accomplish. Finally, intelligent instrumentation as currently being developed will not solve the problem of computer platform differentiation between airborne and post-flight applications, nor does it provide a solution for other problems in real-time data processing, like clock drift in distributed applications.

In this thesis, a new methodology for instrumentation system development is presented with the objective to address the issues that are mentioned above. A system that has been developed according to this methodology, provides a data acquisition and processing environment in which software can be reused throughout all stages of a development program and in which hardware components are both reusable and replaceable. The software for data processing can be applied to both off-line data analysis and stringent real-time applications, without the need for programmatic changes. Application of the methodology – which in itself is not limited to flight test – lifts the traditional restriction that flight test instrumentation consists of only airborne equipment. Instead, the previously separate developments of simulators, airborne instrumentation hardware, and data processing tools are fully integrated. In order to yield an instrumentation system maximum flexibility while ensuring reliability of the system and its documentation, the methodology follows the object-oriented analysis and design paradigm. The concept of intelligent instrumentation is implemented in a distributed real-time environment that is compatible with conventional sensors and transducers. The software-based environment enables the development of the modular instrumentation

system by providing mechanisms for scheduling and activation of acquisition and processing objects, by managing data exchange between the nodes of a distributed system and between individual objects, and by synchronizing the clocks in a distributed system. Finally, the environment supports different computer platforms, while maintaining computational performance with respect to dedicated software developments on each of these.

The methodology is applied in a case study in human-factors flight test. An instrumentation system was analyzed, designed, and implemented in accordance with the new approach and has been used in a series of flight tests. During these experiments, the use of a perspective flight path display for aircraft guidance was investigated in flight. The experiment is a typical example of the closed-loop handling qualities flight tests that have been described in the previous sections. The presence of the pilot in the loop and the desire to evaluate handling qualities with the perspective display, pose the most stringent real-time requirements for the whole of the instrumentation system.

This thesis comprises six chapters, each of which starts with an abstract that summarizes the original contributions of the work that is presented. Chapter 1 enumerates the philosophies and concepts that are behind the methodology and introduces the development life cycle for an instrumentation system. Chapter 2 elaborates the analysis, design, and implementation of a flight test instrumentation application. It identifies the primary subsystems and explores their role in various typical open-loop and closed-loop applications. The synthesis and maintenance of an instrumentation system from existing and customized components is discussed, without going into the details of component development. Chapter 3 then describes the development method for the components that make up the instrumentation system, including platforms, sensors and signal conditioners, actuators, and processing software. Chapter 4 presents the exemplary design for the software environment that is required for implementing an instrumentation system according to the methodology. Special emphasis is placed on the features that support distribution of the application over multiple computer platforms, and the activation and interaction of the object-oriented instrumentation components. Chapter 5 introduces a novel clock synchronization algorithm. The algorithm allows local clocks in a distributed system to be synchronized mutually and to an external reference time, thus forming the enabling technology for real-time applications in a distributed system. Finally, chapter 6 describes the case study in a representative closed-loop flight test program. It shows the application of the new design methodology during each of the analysis, design, and implementation activities.

1
Development Philosophy

A development methodology is shaped by a collection of concepts. The instrumentation system development methodology that is presented in this thesis primarily relies on a life cycle model and the concepts of object orientation and concurrency. Object orientation, concurrency, and development life cycles stem for software engineering. Their applicability is based on the strong similarities from a development point of view between flight test instrumentation systems and software packages. In both cases, maintenance does not consist of examination and repair of the finished product, but to extension and adaptation to continuously changing requirements over a comparatively long period of time. This requires improved flexibility of the development life cycle in comparison with more traditional methods.

This chapter presents a new life cycle model that is tailored to the development of flight test instrumentation systems. It is a combination of the evolutionary prototyping model with elements from throwaway prototyping and hierarchical design. The new method strongly encourages the development of standardized components – both for hardware and software – that can be reused in future applications. Component development is therefore placed in a separate life cycle within that of the application. Standardized components are specialized for use in each application; the application life cycle merely consists of requirement specification and architectural design before component development, and system integration afterwards.

A development method prescribes the designer a series of steps that lead from the requirements for a product to its design and implementation. A method covers a description of each step, and the sequence in which they should be undertaken. As such, the development method helps the designer in making the right design decisions by offering both the criteria and the appropriate context, whenever the development process arrives at a point where a decision must be made.

Development methods rely on design concepts, which are the fundamental ideas on which a design can be based. A successful – or even correct – design in terms of the development method, is a design that incorporates the design concepts. As such, design concepts can be seen as requirements for the final design; the development method is a procedure that ensures satisfaction of these requirements. The key activities that make up the analysis and design procedures, are referred to as modeling strategies. A modeling strategy provides a specific approach to complete a single step in the development method.

The complete development method is based on a collection of design concepts, applies a collection of modeling strategies and documents the results using a design notation. Although the latter is no part of the development method, the notation must be closely related to the method's underlying design concepts. The design notation must allow for a graphical or textual description of the design in a way, that all characteristics of the design are documented. A design notation that has not been developed with the appropriate design concepts in mind, will generally not meet this requirement. The development methodology that is presented in this thesis, uses the Unified Modeling Language (UML) as its design notation. The Unified Modeling Language is introduced in section 1.4; the UML diagrams that are used in this thesis and their graphical appearances are summarized in appendix A.

1.1 Life cycle models

While design concepts and modeling strategies are the building blocks of a development method, the method itself is embedded in the development life cycle. The term

life cycle as it used in this thesis stems from software engineering; it indicates a phased approach to the development of an application (Gomaa, 2000, p. 92). Software engineering was developed in the late 1960s as a solution to the problems of ineffectiveness and inefficiency that haunted large-scale software projects. Life cycle models are an important product of the software engineering discipline. They designate the various phases of software development and define how and when each phase goes over in the next, and which iterations can be made. This way, they help the designers to plan the project with respect to time and to avoid confusion of design activities from different phases.

For the development of hardware, the term life cycle refers to the lifetime of the physical machinery. Although the life cycle includes its production, the cycle as a concept focusses on the product's maintainability. The ease at which correct functioning of the hardware can be supported and the ease at which repairs can be carried out, determine the success of the design in terms of the product's maintainability. Although the term maintenance is used in software engineering as well, it relates to a concept completely different from the one in hardware development. Software does not wear down; it requires no attention or repairs to preserve its functionality. For software systems, the term maintenance actually refers to redesign and reimplementation. Software maintenance covers the extension or improvement of the system, in order to make it more suitable for its existing application or to extend its applicability. In conjunction with software development, the term maintenance therefore indicates a new iteration through the design and implementation process. In the same context, it is exactly this repeated sequence of development phases for which the term life cycle is used.

Flight test instrumentation design is inherently a combination of hardware and software design. Hardware such as sensors, interfaces, and processors, is equally important to the overall system as software for data acquisition and data processing. For a successful system design in which flight test instrumentation hardware and software optimally cooperate, it is essential for the hardware and software components to be developed in a single life cycle. The application of object-oriented concepts is a necessity to keep such a large-scale design manageable. By packaging software and hardware components that belong together into delimited objects, the design of the flight test instrumentation system is broken down into the development of individual components and the subsequent assembly of the final application.

A flight test instrumentation system is a living product. Its development is normally not a one-off event, but a continuing process of extending, changing, and reorganizing the components and their synthesis. Although the hardware components of the system will require maintenance in the traditional sense, the continuous process of flight test instrumentation system development means that maintenance is primarily applied in the same sense as maintenance of pure software systems: a repeated iteration of the life cycle to carry out redesign and reimplementation. Consequently, it is advantageous to apply software engineering life cycle models to the development of a flight test instrumentation system.

Development phases

Although it is possible to make various further subdivisions, the software life cycle comprises four basic phases: analysis, design, implementation, and system testing (Booch, 1994; Rumbaugh et al., 1991; Wirfs-Brock, Wilkerson, and Wiener, 1990). During the analysis phase, the scope of the problem to be solved is investigated. The functional requirements for the system under development are specified and analysis models of the systems are created. These two activities are sometimes separated into two different phases: requirements modeling and analysis modeling (Gomaa, 2000, p. 110). Analysis models show the structure of the system in terms of its functionality; analysis modeling thus takes place in the problem domain. The design phase moves the development to the solution domain. In the design phase, the overall structure for the system is created by mapping the analysis models to the operational environment for the system. The system is then divided into subsystems and each subsystem is designed in more detail. As such, the design is typically performed top-down. The implementation phase covers the actual construction of the system. Implementation is usually performed bottom-up. Starting with the creation of the most specific components as identified in the design phase, the system is integrated step-by-step by assembling the components. In some cases, implementation and assembly are identified as separate phases. Additional activities in the implementation phase are unit testing and integration testing. Unit testing deals with a verification of the individual components that have been implemented; integration testing verifies the correct interaction between the components. Unit and integration testing ensure that the implementation satisfies the design – they do not guarantee that the full system meets its functional requirements. Such testing for functionality is done during the last development phase, the system testing phase. In this final phase before the system is released for operation, the correct functioning of the complete system is validated against the original requirements.

The waterfall model

The waterfall model is the simplest life cycle model. It was first proposed by Royce (1970) and further developed and advocated by Boehm (1976). Figure 1.1 shows an interpretation that stays close to the one presented by Sommerville (2000). It shows the straightforward sequence of consecutive development activities that is typical for the waterfall model. Each activity is only started when the previous one has been completed. The five development activities differ slightly from the development phases that were presented in the previous subsection. Requirements analysis and architectural design cover the analysis phase; detailed design in the waterfall model matches the usual design phase. Coding and unit testing, together with integration from the next step, covers the implementation phase. Finally, the system testing part of the penultimate state in the waterfall model equals the general system testing phase.

The major drawback of the waterfall model is the lack of feedback in early stages of the life cycle. The model does not provide for a flexible mechanism to incorporate

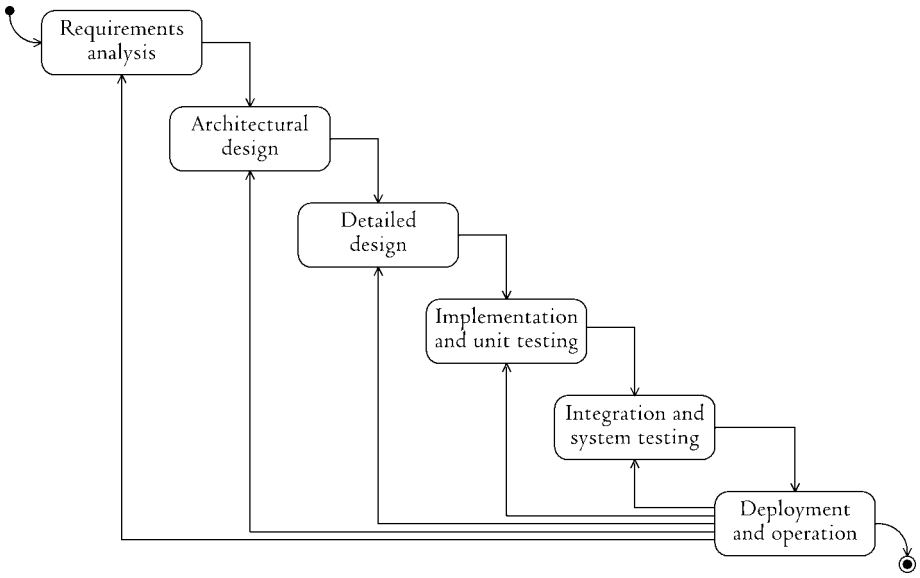


Figure 1.1: Waterfall model.
The development phases are depicted as activities in a UML activity diagram, which is a special form of the statechart diagram where state transitions coincide with the completion of an activity. System development takes place in the first five phases; there is only one direction of flow through these phases. After integration and system testing, the system is deployed and operated. Iteration only occurs after system deployment. To incorporate system extensions, the life cycle is repeated from the requirements analysis phase; corrections to the system can be made by reverting to a design or implementation phase and proceeding from there.

changes to the requirements, analysis, or design in response to errors or weaknesses that are found downstream in the development process. As a result, there is a tendency to correct such problems with a local fix that does not affect the previous development stages. This habit results in poor design and inefficient, unreliable implementation.

In addition to the discouraging nature of the waterfall model towards upwards feedback in case of flaws, the first activities of the life cycle are complicated by a lack of insight in the definitive functional requirements of the system. The waterfall model aims to complete the requirements analysis before architectural design is started, and to complete the overall structure design before the detailed design is started. For large and complicated systems, however, it is often difficult to fully survey the requirements in advance of any development. This makes it more difficult to start the development of a new system and thus reduces the development efficiency.

Prototyping

As a first solution to the disadvantages of using the waterfall model, the system life cycle can be extended with *throwaway prototyping* (Agresti, 1986). A throwaway prototype is a rapid implementation of the system that is constructed directly from the functional requirements or the system's analysis model. The prototype should reproduce the critical behavior of the full system, so that it can be tested in its operational environment. It is used to evaluate the requirements and the initial analysis; the implementation itself is of no value and should be abandoned as soon as practical. Hardware throwaway prototypes can be constructed using less durable or reliable components or materials; software throwaway prototypes are often created in dedicated prototyping languages or environments which do not provide the optimized performance of a definitive implementation. Generally, throwaway prototypes will exhibit limitations with respect to performance, capacity, robustness, and integration with the environment. Nevertheless, throwaway prototypes provide a functioning instance of the system that allows for a direct feedback from the operational environment of the system to the analysis phase at an early stage in the development process. After a throwaway prototype has been used to improve the requirements analysis and the architectural design of the system, further development proceeds along the traditional life cycle model. Throwaway prototyping is depicted in figure 1.2.

Throwaway prototyping is an effective technique to accelerate the analysis phase of large projects and to improve the quality of the requirements specification. Consequently, it will reduce the chance of design errors in the following development phases. Nevertheless, the major handicap of the waterfall life cycle model is unaffected: Requirements, analysis or design flaws that are uncovered during later stages in the development cannot be corrected easily. This problem can be solved by making the system life cycle *incremental*. Incremental development is characterized by repeated cycles through the development phases. Two prototyping models, *incremental prototyping* and *evolutionary prototyping*, have been developed as a way to apply an incremental approach to the system life cycle (McCracken and Jackson, 1982; Gomaa, 1986).

When incremental prototyping is used, the life cycle is intentionally repeated without changing the requirements analysis. As depicted in figure 1.3, the development process starts in the traditional way. When appropriate, the analysis phase can be supported by throwaway prototyping. After finalizing the requirements specification and architectural design, a subsystem of the full product is designed in detail and implemented. Successive increments are built and evaluated in the operational environment until the full system has been completed. From each increment, operational experience is fed back into the detailed design and implementation phases. Finally, the full system is tested against the original requirements and deployed. Similar to the waterfall model, changes to the requirements specifications can only be incorporated by restarting the whole design process.

Evolutionary prototyping constitutes an additional step towards the rapid evaluation of a prototype in the operational environment. Named "iterative enhancement",

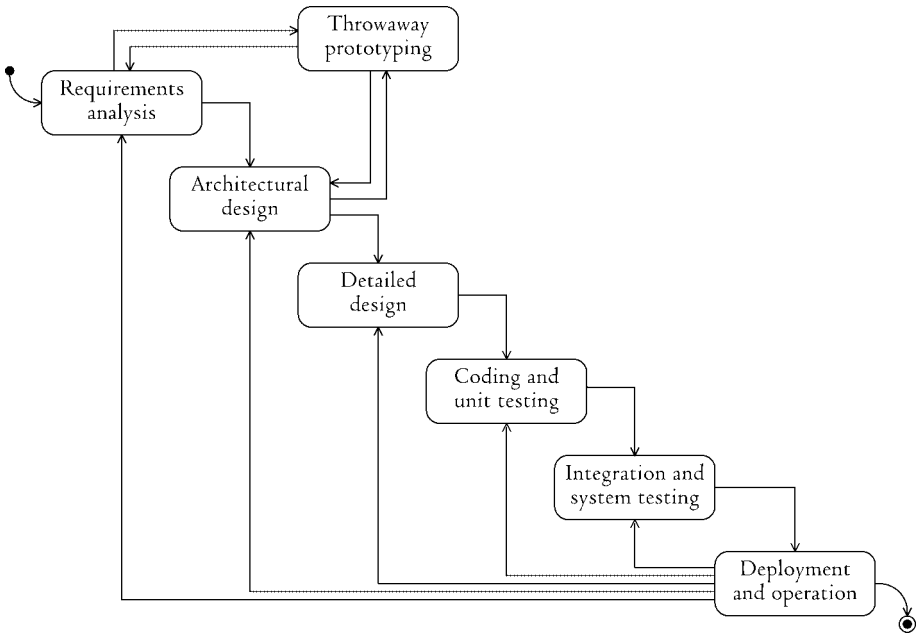


Figure 1.2: Throwaway prototyping.
The construction and assessment of the throwaway prototype interacts with the analysis phases of the life cycle, but not with any phase further downstream the development. Requirements prototypes can lead to the identification of flaws in the requirements; both requirements flaws and architectural design failures may surface from an architectural design prototype.

the concept was introduced by Basili and Turner (1975). In order to overcome the problem of fully having to survey the functional requirements of the system in an early analysis stage, the evolutionary prototyping life cycle is aimed at analyzing, designing, building, and testing a subsystem with only minimal functionality at the first iteration. Using the evolutionary prototype, the system is adapted and extended until the final system has emerged. Evolutionary prototyping is therefore characterized by iterations through the development phases including the requirements specification and architectural design. The evolutionary prototyping life cycle is shown in figure 1.4.

Although evolutionary prototyping is a continuation of the trend towards obtaining a functional system for operational assessment that was started by throwaway prototyping and incremental prototyping, it can also be regarded as a return to the original waterfall life cycle model. In fact, the activity diagrams for the waterfall model (figure 1.1) and the evolutionary prototyping model (figure 1.4) are highly similar. The difference between the two life cycle models lies in what is intended to be accom-

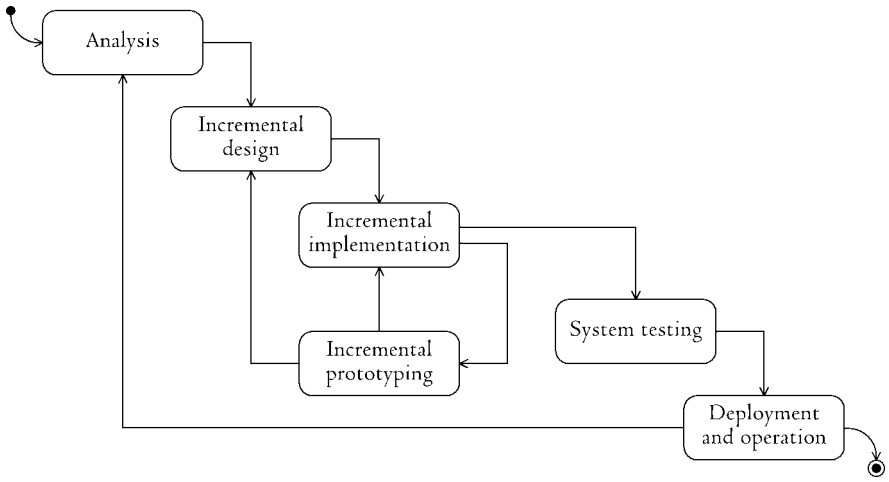


Figure 1.3: Incremental prototyping.
Unlike throwaway prototyping, the incremental prototyping activity does not include the construction of a model system. Instead, the incremental build from the implementation phase is used for evaluation in the operational environment. System testing can only take place after completion of the iteration of incremental prototypes, because the individual increments will not meet the specified requirements of the full system.

plished during the first iteration through the development phases. Using the waterfall model, developers aim to produce a complete analysis of the system on the first, and preferably only, iteration. When the system is deployed, it should immediately fulfill its function. With evolutionary prototyping, an incomplete system is deliberately developed through the whole life cycle. During the process, less time is spent on verifying the correctness and completeness of the analysis and the design, because both are meant to evolve during the following iterations.

Finding the right model

The lack of feedback during the waterfall life cycle endangers a successful and timely completion of the development process. Despite the clear advantages that the evolutionary prototyping model offers in this respect, the progression to detailed design and incremental implementation without having finalized the analysis, poses a threat of a different nature to the development process. Evolutionary prototypes are not throwaway prototypes: They are not intended to be disposed of, but remain the basis for the final system. An effective and reliable structure may be difficult to design on the first iterations when no comprehensive analysis of the system is available yet. This can result in a poor architectural design that is maintained throughout the system's opera-

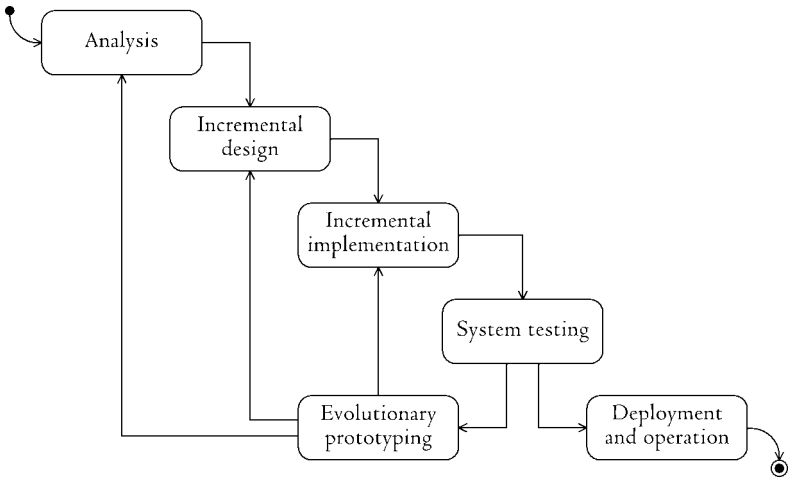


Figure 1.4: Evolutionary prototyping.
Each incremental build, although known to lack functionality with respect to the final system, is tested against the evolving requirements in the system testing activity and subsequently assessed in operational use during the evolutionary prototyping activity. The results of the evaluation are used to correct and extend both the incremental design and implementation, and the requirements specification and architectural design.

tional life. Moreover, evolutionary prototyping inherently contains an aspect of 'trial and error'. Neither analysis nor design are goals of their own – they are means to an end. A reliable design is a prerequisite for a successful implementation. Changing the analysis or the design during an evolutionary process may be harmless to the analysis and the design itself, it may have a strong impact on the previous incremental implementations. The necessarily following iterations in the implementation phase not only lead to a decrease in efficiency, but also bear the risk of overlooking a design change and thus creating a faulty system.

Generally, the evolutionary prototyping life cycle model has clear advantages over the traditional waterfall model – with or without throwaway prototyping – and the incremental prototyping model, when the functional requirements of the ultimate system cannot be overseen at the beginning of the analysis phase. Nevertheless, it is important to create analyses and designs as generic as possible and to include provisions for future extensions, in order to minimize the risk of excessive corrections in later iterations. Because it allows for simpler and better quality control, the waterfall model is preferred when functional requirements can be specified in advance with comparable ease. Additionally, external conditions can influence the choice for a particular life cycle model. If prototypes are difficult or costly to build, which will typically be

the case for hybrid or hardware systems when compared to pure software systems, an incremental life cycle model is unsuitable. In those cases, the throwaway prototyping model may offer a good compromise of early feedback and a single development iteration.

1.2 Concepts and strategies of object orientation

According to Gomaa (2000, p. 27), Goldberg and Robson (1983) were the first to use the term *object oriented*. It was introduced in relation to the Smalltalk programming language, which marked the breakthrough of object-oriented programming. Nevertheless, the principle concepts of object-oriented analysis and design were developed earlier. The two most important of these concepts are *information hiding* and *inheritance*.

Information hiding was first described by Parnas (1972). His paper on the criteria for deciding on module boundaries when decomposing a large system is now considered a classic in the area of object-oriented development. Parnas proposes to assign the tasks of a full system to the individual modules in a way that as many details – and therefore design decisions – on the implementation are kept within a single module. As a result, changes to the implementation require redevelopment of only a single component of the system. This approach greatly improved the maintainability of software with respect to the traditional approach for modular programming, in which the system was decomposed according to the major chronological activities that the application performs. In the latter case, design decision generally affect many of the modules that the system consists of. Any change to these decisions thus has an impact on all of these modules.

Inheritance, in combination with the concept of *classes*, was introduced with the Simula programming language (Dahl, Dijkstra, and Hoare, 1972), which was developed by Dahl and Nygaard (1966). Simula is regarded as the first language that applied object-oriented concepts. Whereas information hiding greatly improved the maintainability of software, the major contribution of inheritance lies in the development phase. Based on information hiding, the concept of a class combines the behavior and implementation of a subsystem in a single entity. Inheritance allows for the reuse and sharing of code between classes by letting new classes that are derived from existing ones, automatically obtain the behavior and the characteristics of the existing class. By subsequently extending or changing the implementation of the derived class, collections of classes with similar behavior can easily and consistently be obtained.

In spite of the information-technological background of object orientation, its concepts and strategies can be applied to other fields of analysis and design as well. The primary goals of object orientation – maintainability, reusability and extensibility – are equally important to any other modular system design. Because flight test instrumentation is assembled from existing data acquisition and processing components, it is

inherently modular. Hence, the object-oriented approach is particularly applicable to flight test instrumentation system design.

Information hiding, encapsulation, and abstraction

Reverting to the original paper on information hiding by Parnas (1972) – which was written before the concepts of object-oriented modeling had been recognized and formulated in detail – information hiding can be regarded as the most important concept behind object-oriented analysis and design. Information hiding aims to separate an entity's characteristics that are required by its environment, from those that are not; the latter are subsequently hidden from the surrounding systems. The terms *encapsulation* and *abstraction* are closely related to information hiding. Encapsulation refers to the generic grouping of items in a single container. Such a container, or capsule, is required to facilitate information hiding. The public interfaces for the items are visible for the environment, while the other parts of the combined entity are hidden inside the capsule. Abstraction† is related to the various levels of detail that are required when viewing an item from different perspectives. Abstraction refers to the inclusion of only those aspects that are necessary for the chosen view, and to the suppression of all others. Thus, abstraction provides the criteria for the decision which information can be hidden and which must be available to the environment.

The close relation between information hiding and encapsulation led to an ongoing debate on whether these two concepts are actually equal. Rumbaugh et al. (1991, p. 7) do not distinguish between information hiding and encapsulation; their vision is close to that of Booch (1994, p. 49), who considers encapsulation as a means to achieve information hiding. On the other side, Berard (1993, pp. 68-69) emphasizes that encapsulation differs from information hiding on the visibility of the information. According to Berard, considering encapsulation and information hiding to be interchangeable leads to the false conclusion that anything that is encapsulated, is also hidden. Berard substantiates his position with the formal definition of encapsulation and its interpretation by Wirfs-Brock, Wilkerson, and Wiener (1990), which states that encapsulation is the act of building a capsule, a conceptual barrier, around a collection of things. Berards argues that the definition does not restrict the boundaries of the capsule to be opaque for some or more of the enclosed items. Any process that is used

† Although information hiding is generally attributed to Parnas (1972), Dijkstra (1968b) already uses abstraction in his description of a hierarchically structured operating system. Its importance may be obscured by the fact that Dijkstra's use of the term abstraction, which basically refers to information hiding between cooperating classes, has a meaning that is slightly different from the one in recent literature, where abstraction is mostly used in conjunction with inheritance (see next subsection). In his paper, Parnas discusses the structured system that was presented by Dijkstra and concludes that abstraction into a hierarchical structure and information hiding are two desirable, but independent properties of a system structure. Though this is undoubtedly true, Dijkstra applied information hiding to the design of his operating system several years before Parnas identified and named it as a design concept.

to group multiple items, would thus satisfy the definition of encapsulation. Since this includes structures that are completely transparent, encapsulation could not be equalled to information hiding.

Nevertheless, even Berard (1993) defines encapsulation as the act of "packaging information in such a way as to hide what should be hidden and to make visible what is intended to be visible". Using this definition, structuring mechanisms that only group information and that do not allow for hiding that information, do not qualify as an encapsulation mechanism. It is noteworthy that information hiding is clearly a design concept, whereas the definitions of encapsulation describe the process that leads to the goal of information hiding. Thus, encapsulation is a modeling strategy rather than a design concept. It was mentioned already that an encapsulation mechanism is a prerequisite for information hiding. Now, this dependency can be extended with the duality that a structuring mechanism can only be regarded as an encapsulation when the concept of information hiding is applied. Consequently, it is desirable to distinguish between information hiding and encapsulation as a design concept and a modeling strategy respectively, but with a one-to-one relation.

Classes and objects, inheritance, and polymorphism

Objects are the physical or conceptual entities that make up an object-oriented application. The objects interact more or less autonomously, resulting in the desired behavior for the whole of the application. By packaging data and procedures that operate on the data in a single entity, objects are the vehicle for constructing an application that adheres to the concept of information hiding. A *class* is a description of a collection of objects with similar characteristics, properties and behavior (Coad and Yourdon, 1991; Rumbaugh et al., 1991). Classes are used to construct objects. The objects, which are also referred to as the instances of the class, are created with all the properties and behavior as defined in the class.

Classes and objects provide a convenient mechanism to implement a system in accordance with the concept of information hiding. Nevertheless, information hiding as a criterion for system design can also be applied without the availability of classes, as was shown upon its introduction by Parnas (1972). The true power of classes lies in the concept of *inheritance*, rather than in information hiding. Inheritance is a mechanism to share code between related classes and to be able to address a specialized class through an interface that was defined for a more general class. When a new class is created, it can be based on an existing class by *deriving* the new class from it. The derived class, also referred to as the child class, automatically obtains all of the data and behavior of the base or parent class. After a child class has inherited its parent's properties, it can be extended with additional data or behavior, or parts of the inherited data and behavior can be changed. The child class becomes a specialized version of the parent class and vice versa, the parent class is a generalized version of the child class. Generalization/specialization is closely related to abstraction. Generalized classes are at a higher level of abstraction than specialized classes. By zooming out, details of the indi-

vidual subclasses are blended out and only the properties that are common to all subclasses are maintained.

Substitutability is an important requirement for specialized classes. The substitutability requirement states that a class can always be replaced by any of its subclasses, without losing any of its functionality. As mentioned before, specialized classes can override the data or behavior of the generalized class with an alternative implementation, or extend its functionality with new data or behavior. However, a specialized class cannot delete any inherited data or behavior. The substitutability requirement thus protects the universality of the generalized class. Snyder (1986) and Cook, Hill, and Canning (1990) among others do not acknowledge the substitutability requirement for derived classes. Instead, they distinguish between inheritance and *subtyping*. The former term is only used for deriving new classes from existing ones; the latter refers to specialized classes of any origin that satisfy the substitutability requirement. In this approach, derived classes are allowed to delete inherited behavior, subtypes are not. In this thesis however, substitutability is regarded as a prime concept to design for reusability. It is therefore desirable to require derived classes to implement at least all of the behavior of the parent class. In this respect, the need to delete data or behavior from a derived class is a sign of poor modeling. Either the parent-child relation should be reversed, in which the original parent extends the behavior of the original child, or both classes are in fact children of a third class.

Polymorphism is an object-oriented concept that relies on the substitutability requirement. Polymorphism means that different classes may implement a conceptually identical operation – with an identical function in its environment – with different behavior. The environment of an object of these classes can request such an operation with a reference to a generalized class. Only upon the execution of the operation, the actual type of object is determined and the appropriate specialized operation is performed. This way, the environment can interact with an object as if it were an instance of the generalized class, where in fact the specialized implementation of the behavior is executed.

1.3 Concepts and strategies of concurrency

The concepts of concurrency deal with the parallelisms that are inevitably present in any real-time or distributed application. The several threads of control that run in parallel, whether as outside stimuli that must be responded to in a real-time system or as the distributed activities on a collection of nodes, lead to the simultaneous occurrence of multiple events. In the design of a system that can successfully handle these parallelisms, special measures must be taken that allow the threads to be executed in time, to communicate, and to synchronize their operations. These issues are addressed by the concepts of concurrency.

In a *static model*, all of a system's objects appear concurrent. The model shows all components in parallel, without disclosing a mutual exclusiveness of any two objects with time. Hardware components are necessarily concurrent in the implementation as well, but software components need not be. A *dynamic model* of the system reveals which objects are interdependent and are always activated sequentially. These objects are not inherently concurrent; there is a single thread of control along which only one of the objects is active at a time. Identifying and distinguishing between mutual exclusiveness and inherent concurrency is the responsibility of dynamic modeling (Rumbaugh et al., 1991, p. 202).

Processes, threads, and fibers

To address the various possibilities in concurrency and mutual exclusiveness of software objects, the terms *process*, *thread*, and *task* are used in various meanings. Generally, a process is an activity that executes in its own memory space. When a single computer or a node in a distributed system supports multiple processes, each process runs in its own environment, invisible to the other processes. Communication and data exchange between processes on a single computer therefore requires similar techniques as for processes that run on the various nodes of a distributed system. Concurrency is not limited to multiple processes in a single application, but can also occur within a process. Concurrent activities in a single process are named threads. A thread shares memory with all the other threads in the same process. A thread is sometimes referred to as a lightweight process; a process with multiple threads of control is then referred to as a heavyweight process. Because they share memory, the multiple threads of a heavyweight process can easily communicate and exchange data. However, access to shared data must be synchronized to guarantee the integrity and currentness of the data.

Bacon (1992) uses the term *process* for a single activity with its own thread of control, including both single-threaded heavyweight processes and individual threads within a heavyweight process. A more common term for this concept is *task*. Rumbaugh et al. (1991, p. 202) define a task as the implementation of a thread of control on a computer system. Nevertheless, they generally interchange the concepts of task and thread of control, even though their definition suggests a minor distinction between the two. Gomaa (2000, p. 40) considers a task to be a generalization of a thread, embracing both threads within a process and single-threaded processes. To avoid any confusion on the various levels of concurrency when the term *task* can refer to both a thread and a process, the word task is not used in this thesis with respect to concurrency. Instead, *task* refers only to the activities that an object has to complete in order to fulfill its function. All processes, including single-threaded and heavyweight processes, are simply referred to as a process. Each process has at least one thread of control. A lightweight process has exactly one; heavyweight processes have more. A task as defined by Gomaa (2000) is referred to as a thread of control, or simply a thread.

In addition to processes and threads, a third level of concurrency can occur when a single thread is used to implement multiple activities. Concurrent activities within a

thread are referred to as *fibers*. The difference between a thread and a fiber lies in the way the concurrency is achieved. Concurrent processes can share a computer system on which they are implemented, but they can also be distributed over various nodes in a distributed system. Concurrent threads share the same memory, so they must be implemented in a single process on a single node. However, they might be executed by multiple processing units (CPUs). When concurrent threads are executed by a single CPU, alternating activation of the various threads is taken care of by the software environment in which the application runs. Fibers share not only memory, but also the processing time of the thread to which they belong. As a result, the fibers of a single thread all execute on the same processor. Their activation is controlled by the thread itself: One fiber cannot interrupt the execution of another.

Environments for concurrency

As indicated above, the implementation of a concurrent application does not require a hardware environment with multiple processors. When only a single processor is available, a scheduling mechanism can sequentially allocate processing time to the various threads. Likewise, a single pool of memory can be divided into virtual memory spaces – referred to as address spaces – that each are assigned to a separate process. Whenever a thread is activated, the appropriate memory block for the corresponding process is made visible to the processor. The technique of changing memory environments for multiple processes in a single memory space, is referred to as *context switching*.

A hardware environment with only one processor is named a *multiprogramming environment*. Consequently, all threads in a multiprogramming environment share the same processor and the same memory. Scheduling and context switching are provided by the operating system, so that the lack of actual concurrency is completely transparent to the application. In a *symmetric multiprocessing environment*, there are two or more processors, but all processors share the same memory. True concurrency can be achieved with a symmetric multiprocessing environment, because the multiple processors can execute various threads simultaneously. The need to synchronize access to the common memory space, means that an operating system that supports concurrent applications is still required. In both the multiprogramming environment and the symmetric multiprocessing environment, communication between threads of the same process can take place through the shared memory. A *distributed processing environment* consists of a collection of nodes with local memory. Each node is a system that can be either a multiprogramming or a symmetric multiprocessing environment. Between the nodes in a distributed system, the only means of communication is a network†. Data exchange between processes on distributed nodes is achieved by sending messages across the network. For

† The nodes in a distributed system communicate only through a network; by definition, the nodes do not share any memory. In this respect, the term shared memory refers to the normal address space for each of the nodes in the system. The network that connects the nodes can be of arbitrary nature, including so-called reflective memory networks, or even shared memory networks between physically colocated nodes.

multiple processes on a single node, a similar facility of messaging is available to maintain transparency with respect to any form of inter-process communication. This transparency is provided by a special software layer that resides between the operating system and the concurrent application, referred to as *middleware*. Middleware offers a common interface to all processes that make up the concurrent application (Bacon, 1992; Gomaa, 2000, p. 78). Processes can communicate with other processes through the middleware, without having to know the actual location of the remote process. Middleware can thus be regarded as an application of the object-oriented concept of information hiding to the area of concurrency.

Scheduling

Although concurrent applications do not require a hardware environment that supports concurrency, an appropriate software environment is a necessity. In multiprogramming and symmetric multiprocessing systems, scheduling and context switching must be taken care of by the operating system. In a distributed system, middleware must hide the actual location of the remote processes from the local process and should provide platform independence, in order to be able to construct a distributed system from different computer systems.

When there are fewer processors for a concurrent application than the number of threads, the available computation time must be divided over the threads that are ready to use it. The act of subsequently allocating processing time to individual threads is referred to as *scheduling*. Scheduling techniques are governed by various concepts on concurrent thread control, leading to different mechanisms for handing over the processing time from one thread to another. The term *control* itself refers to the authority over the transitions between the thread's states. The various concepts of control differ in the number and type of states that each thread can be in, and in the way the threads are aware of and cooperate with the scheduling process.

As a contrast to concurrent systems, Rumbaugh et al. (1991, pp. 208-209) discuss two types of systems with more traditional control concepts: *procedure-driven systems* and *event-driven systems*. In a procedure-driven system, control resides with the program code. A procedure-driven thread has only one state. Once it has been created, it executes until its task has been completed. If a procedure-driven thread needs external input, it calls a procedure that executes as part of the calling thread of control. When the procedure finishes, the original program flow is resumed at the point where it was interrupted by the procedure call. Although the thread is often said to wait for the procedure to return (Rumbaugh et al., 1991, p. 208), this is actually incorrect. The calling thread does not enter a waiting state, which would introduce a concurrency of the original thread and the procedure being executed. Instead, there is only a single thread of control that is temporarily handed over to the program code of the procedure. Procedure-driven system have no built-in facilities that support concurrency. Any communication with an external thread can only be realized through a procedure call that handles the communication. This might block the thread for an unknown period of time,

because the thread cannot force the external thread to respond or to deliver data in time. If a procedure-driven thread is to be used in a multiprogramming environment, the operating system must have a mechanism to *preempt* the thread. By preemption, the operating system freezes the execution of each thread after a fixed unit of time, called the *time slice*, to hand over control to another thread. Because a thread that is allocated processing time at the cost of other threads might actually be blocked in a procedure call that waits for external inputs, scheduling procedure-driven threads in a multiprogramming environment with mere time slicing is inefficient. Procedure-driven threads are therefore unsuitable for real-time applications.

In an event-driven system, control is continuously exchanged between the program code and a dispatcher. Event-driven threads comprise a collection of callback functions that are activated by the dispatcher when a corresponding event has occurred. When the thread has completed its task in response to the event, control is returned to the dispatcher. Thus, event-driven threads never block to wait for input. Instead, they are characterized by two different states as shown in figure 1.5. A transition from the thread's idle state to the executing state occurs when the dispatcher receives an event to which the thread must respond. When the thread finishes its task, a transition back into the idle state takes place. If an event-driven thread requires external input, it returns control to the dispatcher and resumes execution in a callback function that is activated when the data has arrived. This way, event-driven systems provide a basic mechanism for concurrency. Multiple threads can run in a multiprogramming environment without the need for preemption. Because threads are only allocated processing time when they actually have a task to perform, event-driven systems use the available processing time more efficiently than procedure-driven systems. Nevertheless, the absence of a preemption mechanism poses a threat to the applicability of event-driven systems for real-time applications. The occurrence of a high-priority event does not lead to the scheduling of the appropriate thread until the current thread returns control to the dispatcher. This may endanger a timely response to all critical events.

A concurrent system is therefore essential to the successful implementation of a real-time application. In a concurrent system, control resides with several threads simultaneously. Each thread has a multitude of states as shown in figure 1.6. After its creation, the thread awaits scheduling by the operating system. When it is allocated processing time, the thread starts executing until it blocks or until it is preempted by the operating system. When a thread is preempted, it reenters the state of awaiting scheduling because its operations have not finished or blocked yet. A thread can block for several different reasons. For a real-time system, these can be generalized into three groups. First, a thread can complete its task for the current time point. When it has to wait for the next time point before processing can continue, it enters a waiting state from which a transition to the state of awaiting scheduling takes place when the real time reaches the next time point. Second, a thread can depend on a certain event before it can continue. It then enters the state of waiting for the event, from which it is released into the state of awaiting scheduling when the event has occurred. Third, a

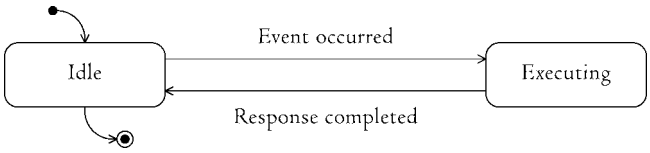


Figure 1.5: States of an event-driven thread.
Event-driven threads are either idle or executing. A transition from idle to executing is initiated by the dispatcher, which activates the thread through a callback upon the occurrence of an event. The transition back to the idle state is initiated by the thread; there is no preemption. Because control resides with the dispatcher, the thread cannot destroy itself. Therefore, the thread not only starts in the idle state when it is created, but must also return to the idle state before it is destroyed.

thread can require input or output operations with an external thread to be completed. A corresponding waiting state is left when the communication has been finished.

It is possible to consider the completion of input/output communication and the progression of time as a special type of event, and thus to generalize the three waiting states into a single case of "waiting for event". For example, Gomaa (2000, p. 64) identifies a single transition from the execution state to the state of waiting for an event as "block for timer or internal event". Alternatively, additional waiting states can be added, such as "waiting for message" (Gomaa, 2000). However, the distinction into three groups as presented here is based on decisive differences between the occurrences that release the thread from its waiting state. A thread that blocks to wait for the next time point, is not dependent on any concurrent activity, only on the progress of time. Internal events are stimuli from other threads in the same process. A thread that waits for an internal event, thus depends on the processing of other threads in the same process. Communication with threads outside the same process, whether in the same multiprogramming or symmetric multiprocessing system, or on a remote node in a distributed system, always takes place through the middleware. A thread that waits for the completion of inter-process communication thus depends on the middleware and on the activity in the remote process. These three levels of dependence are a unique qualifier for the various waiting states that are indicated in figure 1.6. Using the terminology of Paassen, Stroosma, and Delatour (2000), the generic internal or external event that releases a thread from a waiting state is referred to as a *trigger*. A trigger that releases a thread from waiting for the next time point is named a *ticker*.

The presence of a scheduler is an important property of a concurrent system. Similar to the operating system for a procedure-driven thread, the scheduler preempts threads that are executing to allocate processing time to others. The crucial difference with a procedure-driven system is the existence of the waiting states of the concurrent

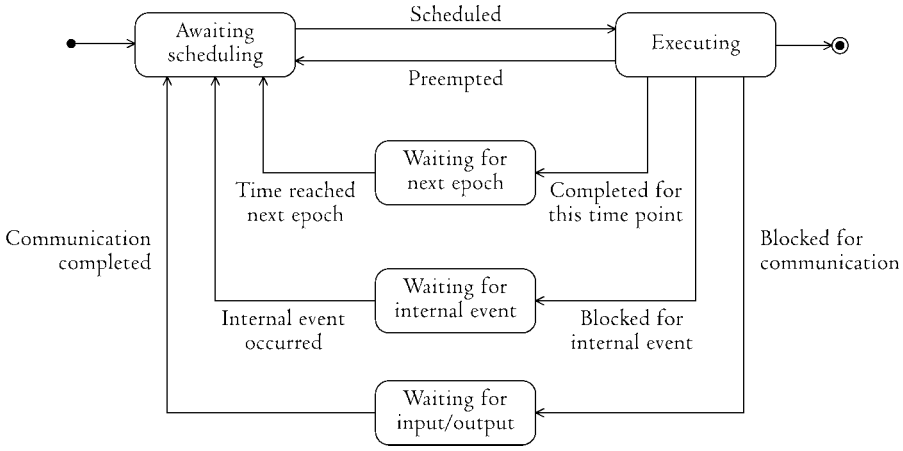


Figure 1.6: States of a concurrent thread.
Concurrent threads are characterized by three major states: ready for processing, executing, and blocked. When the thread is ready for processing, it awaits scheduling by the operating system and transits to the executing state. A return to the ready state can occur directly when the thread is preempted, or through one of several blocked states in which the thread waits for certain prerequisites for a continuation of its activities. Because control resides with the thread, its termination is initiated from the executing state.

threads. When a thread is blocked, it is not on the list of threads that await scheduling. As a result, the scheduler does not allocate the thread any processing time. For this reason, the transition out of one of the waiting states does not lead back to the execution state, but rather to the state where the thread is waiting to be scheduled. It is the responsibility of the scheduler to decide when the pending thread will actually be allocated processing time again.

In a multiprocessing system or a distributed system, it is possible that a separate CPU is available for each thread in the application. In this case, all threads can actually execute concurrently and no preemption will take place. Full true concurrency however is highly improbable, even in large distributed systems. Because a blocked thread does not consume any processing time, a system that allows full true concurrency does not make efficient use of its hardware resources. Concurrency without preemption is therefore not only unlikely, but also undesirable.

Various scheduling algorithms have been developed with the aim to ensure an optimal distribution of CPU time over the threads that are ready for processing. Algorithms that support preemption fall into two categories: round-robin scheduling and priority scheduling. In the concept of round-robin scheduling, all threads have equal

priority. Similar to the rotation of thread activity in a procedure-driven system, the allocation of the CPU to a thread is limited to a time slice. When a thread is preempted, it is put at the end of the queue of the threads that await scheduling. This way, all non-blocked threads receive a comparable amount of processing time. The difference between a concurrent system with round-robin scheduling and a procedure-driven system lies in the recognition of waiting states. Nevertheless, the first-in-first-out way of scheduling with time slicing does not protect important threads from being postponed by less important ones. This problem is addressed by the concept priority scheduling. In priority scheduling, each thread is assigned a priority. The scheduler allocates processing time to the highest-priority thread that is not blocked. There is no time-slicing with priority scheduling. Thus, a thread will execute until it blocks or until it is preempted by a thread with a higher priority that becomes unblocked.

The remaining issue with priority scheduling is how to assign priorities to the individual threads. Liu and Layland (1973) have shown that the most efficient processing and therefore the best overall performance of a concurrent system with fixed priorities for all threads is achieved when the thread with the shortest task is assigned the highest priority. This concept, named shortest job first (SJF), is the basis for rate-monotonic analysis (RMA). RMA is a strategy for assigning fixed priorities to periodic activities; each thread is assigned a priority that is inversely proportional to the duration of the thread's periodic task. Priority scheduling using RMA was proven to meet all deadlines of the system, if it is possible for a fixed-priority scheduling to meet all deadlines in the first place. However, with an increasing number of threads, the worst-case schedule bound of RMA approaches $\ln 2$ or approximately 69% (Serlin, 1972; Liu and Layland, 1973). This means that only when the sum of the fractional processing time – the ratio of the run time of a thread to its request period – of all threads is less than this bound, all deadlines are guaranteed to be met. To fully exploit the capacity of a concurrent system, scheduling with variable priorities is necessary. The most intuitive strategy for assigning such priorities is the earliest-deadline-first algorithm (EDF). If a system that uses earliest-deadline-first priority scheduling does not meet its deadlines, it cannot meet its deadlines with any other scheduling algorithm. The EDF strategy can only be used when not the computation time, but rather the next deadline for each thread is known, which may not always be the case. As opposed to rate monotonic analysis, EDF is equally applicable to periodic and aperiodic threads.

Cooperation between threads

In a non-concurrent system, the existence of only a single thread of control prevents cooperation problems between various objects. When an external procedure is called, data and the thread of control are handed over simultaneously. Cooperation between concurrent threads is complicated by a number of issues, such as the mutual exclusion problem and the producer/consumer problem. If these problems are not dealt with appropriately, erroneous system behavior in the form of *deadlocks* or *race conditions* can occur. A deadlock is a failure of the system in which multiple threads are infinitely

blocked due to a mutual dependency. Each of the threads involved requires one of the other threads to finish its operations first, in order to provide data or to release a shared resource. A race condition is a system failure in which the outcome of activities unintendedly depends on the relative timing of events. Race conditions occur when the correct functioning of an operation depends on the previous completion of an activity in another thread, whereas the sequence of activities between threads is not properly synchronized.

The mutual exclusion problem is the most probable cause for a race condition failure. When two threads simultaneously access a shared resource, for example a data object in shared memory, the data may become corrupted or outdated information may be used. This is avoided by granting access to the shared resource to only one thread at a time. Before a thread is allowed to use the resource, it must first acquire it. When the thread has completed its operation, it releases the resource. A thread that tries to acquire a resource while another thread has locked it, is blocked in a waiting state until the resource is released. As a mechanism for ensuring this type of mutually exclusive access to a resource, Dijkstra (1968a) introduced the *binary semaphore*. A binary semaphore is a two-state variable that represents the shared resource and indicates whether the resource has been acquired or not. Implementation of a binary semaphore is not as straightforward as it may seem, because access to the semaphore itself is subject to the same mutual exclusion problem as the resource it represents. Access to a binary semaphore is therefore limited to two *atomic* operations: one to acquire the semaphore and the corresponding resource, and one to release the semaphore. Atomic operations, or atoms for short, are indivisible operations. They cannot be interrupted, preempted, or run on two processors at the same time. Although binary semaphores provide a solution to the mutual exclusion problem, they can easily be the cause for a deadlock.

The producer/consumer problem (Gomaa, 2000, pp. 45-46) addresses synchronization issues between a thread that produces information and another thread that consumes it. The information must be communicated from the producer to the consumer. When the consumer is ready to receive the information, but the producer has not yet produced it, the consumer has to wait for the producer thread to provide the information. Vice versa, if the producer has produced the data and the consuming thread is not ready to receive yet, the producing thread has to wait. If information is buffered in a queue between the two threads, the producer/consumer problem is divided into two similar issues. The producer can only write information to the buffer when it is not full; the consumer can only read from the buffer when it is not empty. Otherwise, either the producer or the consumer must wait for the thread on the other end of the queue.

The producer/consumer problem is solved by using messages for communication between threads. Message communication handles both the transfer of information and the synchronization between the producer and the consumer thread. Without synchronization by packaging information into messages, producer and consumer threads would not be able to recognize a full or empty buffer and information might be lost or

erroneously duplicated. Synchronous message communication, also referred to as tightly coupled message communication, consists of two-way messaging. The producer sends a message and waits for a reply from the consumer. Only after reception of the reply, both threads can continue. This way, tightly coupled communication achieves full synchronization between producer and consumer. Alternatively, asynchronous message communication, also referred to as loosely coupled communication, consists of a single message only. The producer thread sends a message and continues to execute without waiting for the consumer.

1.4 UML notation for object-oriented modeling

A design notation for object-oriented modeling must allow to record an analysis or a design in a way that the prime characteristics of an object-oriented system can be expressed in the model. This requirement cannot be satisfied by a single graphical or textual notation that describes the model in terms of the design concepts of object orientation, but also requires various views of the model. A *static model* provides an overview of all the components that the system comprises and all the relations between these components. As such, a static model is a description of all the elements that must be implemented during the construction and integration phases of the development cycle. However, it does not provide any information on the way the relations between the elements are used, and how both the system's components and their relations vary with time when the system is in operation. *Dynamic modeling* provides an overview of such dynamic behavior. A dynamic model usually contains only a subset of the components of the full system. For such a subset, it describes an operational case in a sequential manner. It shows how the components of the system interact, and what effect these interactions have on the state of the individual components.

The Unified Modeling Language was defined in 1997 by Jacobson, Rumbaugh and Booch in response to the need for a standard notation in which an object-oriented system can be described graphically. Until then, Jacobson, Rumbaugh and Booch, among others, had pursued separate developments in object-oriented modeling languages. Booch (1994) focussed on static modeling. An important contribution from the Object Modeling Technique (OMT) by Rumbaugh et al. (1991) was the notion that dynamic modeling is equally important as static modeling. Jacobson (1992) introduced the concept of *use case modeling*. Use case modeling is neither a part of static modeling nor of dynamic modeling. It is used to define the functional requirements of a system by describing the typical applications of the system in its environment. Use case modeling is part of the analysis phase of the development cycle. It is the basis for dynamic modeling, in which use cases are elaborated into *scenarios*. A scenario is a single, specific development of a use case.

The Unified Modeling Language was created with the aim to generate a notation with generic applicability. UML therefore covers nine different types of diagram, each

for a different view of the model. Apart from a use case diagram, there are four types of static and four types of dynamic modeling diagrams. Although each of these diagrams has a different objective and is optimized for a different phase of the development process, some types of diagram clearly overlap. It is therefore probable that the number of UML diagrams will be reduced as the notation develops further (Fowler and Scott, 1999). The development methodology that is presented in this thesis, utilizes three of the most essential UML diagrams: the class diagram, the statechart diagram, and the sequence diagram. The presentation of these diagrams in appendix A is limited to those aspects that are actually used in the methodology. Rumbaugh, Jacobson, and Booch (1999) present the full theory and use of UML; Booch, Rumbaugh, and Jacobson (1999) give an introduction to the application of UML with a focus on software engineering.

1.5 The flight test instrumentation development life cycle

The similarities between software development and flight test instrumentation development suggest the use of a software life cycle model to support the analysis, design and implementation of an instrumentation system. The key similarity between the two is found in the character of software and flight test instrumentation maintenance. In both cases, new versions of the system are typically developed well before the previous version has expired. New releases are not demanded because the existing implementation does not function anymore; they are merely developed in response to a change in the requirements to the system.

This makes the evolutionary prototyping life cycle the natural model for the development of a flight test instrumentation system. However, as noted in section 1.1, the development of a system without knowledge on the requirements and architectural design of the final or even the immediately following version introduces a considerable risk. As an ultimate consequence, new requirements for the system might demand such changes to the existing incremental implementation, that in fact a completely new system needs to be developed. This risk is inherent to the evolutionary prototyping approach to system development and can never be eliminated completely. However, its consequences can be mitigated considerably by strict application of object-oriented design concepts. By constructing the application from logical objects that match the physical components of the system and the environment, the architectural design can be made almost immune to requirement changes. Since the function and mutual relationships between the existing components will hardly change with an extension or adjustment of the functional requirements to the full system, the structure of logical objects in an object-oriented system model will not be affected either. Thus, evolutionary prototyping will only lead to the extension of the system with additional objects and to changes in local requirements for the existing objects. When the concept of information hiding is applied to the detailed design of the object-oriented system, any

change in the requirements to a single component will be transparent to the surrounding objects. This reduces the impact of evolutionary requirement changes on the existing implementation significantly. As a result, the preferred life cycle model for flight test instrumentation development is based on evolutionary prototyping, in combination with an object-oriented development method.

Figure 1.7 shows the development life cycle that is used in this thesis. Its key characteristic and a consequence of the object-oriented approach, is the separation of component development and application synthesis. Rather than designing and implementing a single application, the flight test instrumentation system is constructed from standardized components that are customized for the application. The components that form the application have their own life cycle, which is enclosed in that of the full system. Flight test instrumentation development starts with analysis of the system's requirements, followed by analysis of the context in which the system operates. This separation of analysis activities corresponds to the distinction between requirements modeling and analysis modeling that is made in the *concurrent object modeling and architectural design method* (COMET) for concurrent system design (Gomaa, 2000). Because the term modeling is often used to describe the combination of analysis and design, the two activities are renamed *requirements analysis* and *context analysis*. The requirements analysis activity focuses on the role of the system in its environment; the context analysis activity subsequently models the environment and the way the system interfaces with the environment. Being part of the evolutionary life cycle, both analysis activities do not need to be comprehensive from the first cycle through the development process. Corresponding to the analysis models, an incremental architectural design is made. The application design lays out the components that the system is composed of. The component life cycle then starts with a new analysis activity, in which the requirements for each component are analyzed in more detail. The second activity of this phase is the identification of standard components that are available from other applications and that match the required components for the new application. For components that must be developed from scratch, a throwaway prototype can be constructed. The next activity is incremental component design. During this activity, the components are designed in detail, using the object-oriented concepts of information hiding and inheritance. As a result, each component is preferably modeled as a specialized version of a generalized component, adapted to this specific application. When such a generalized component is already available, the detailed design only has to deal with the specifics of the derived class. For components that cannot be based on an existing generalized class, the standardized component is designed and implemented first. Implementation then proceeds with the adaptation of the specialized versions of the components and the corresponding unit testing. This concludes the embedded life cycle of component development. In the following development phase, the application is integrated from the new components. Incremental prototyping can be used to develop the system components iteratively. When the system is believed to meet the application requirements, system testing is performed and the flight test instrumenta-

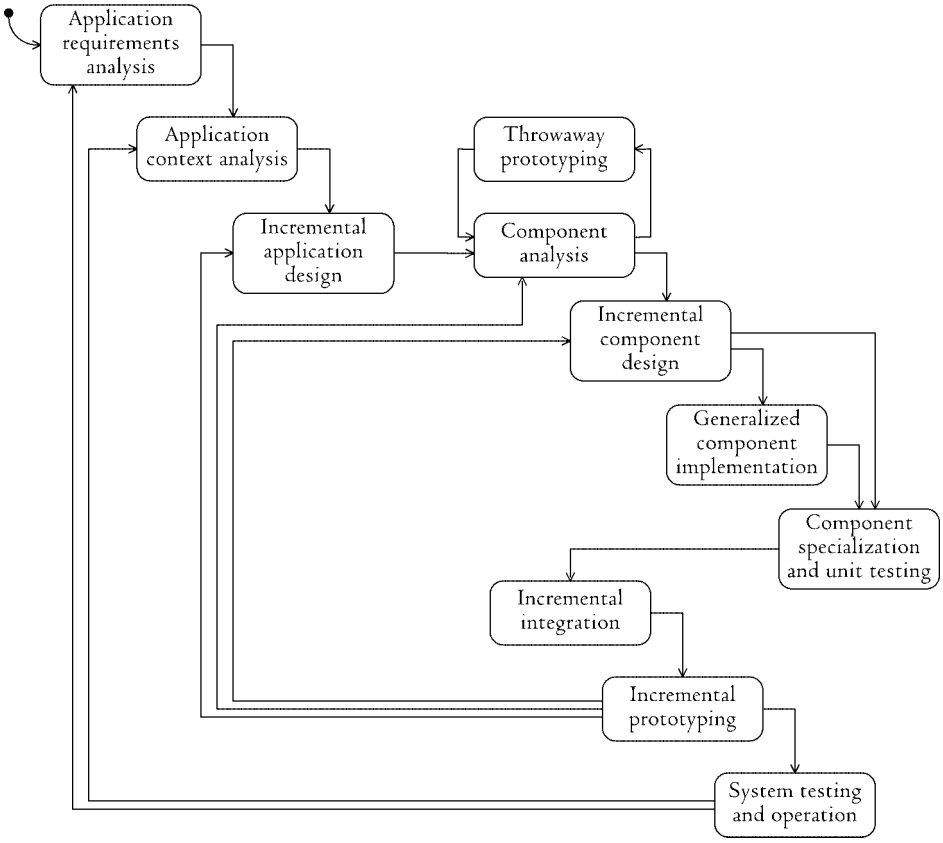


Figure 1.7: Flight test instrumentation development life cycle used in this thesis. The UML activity diagram groups the development activities in three phases: application modeling, that starts with requirements analysis and ends with incremental design, component development, which forms a development subcycle from component analysis through specialization and unit testing, and application synthesis, that starts with incremental integration and ends with testing and operation.
Feedback from the implementation and operation activities to the analysis and design activities is a combination of throwaway, incremental and evolutionary prototyping. Application analysis and design are followed by an embedded life cycle for component development, starting with component analysis and ending with customization of standard components and unit testing. Application integration is followed by incremental prototyping and operational use. Since flight test instrumentation development is typically an in-house activity with a long-term and everlasting life cycle, evolutionary prototyping is replaced by system testing and operation as a single activity with feedback to the application analysis. For the same reason, the development process does not show a termination state.

tion system is set to operation. Changing circumstances or new functional requirements then feed back to the application analysis, making the operational system a kind of evolutionary prototype.

The components that form a flight test instrumentation system are divided into three groups: platform components, data acquisition components, and data processing components. The platform components are the computer systems on which the software parts of the data acquisition and data processing components are implemented. Apart from the hardware of the computer system and all associated peripherals, a platform component consists of the middleware that hides the computer system specifics from the software components. Interfaces such as digitization boards are closely related to the data acquisition components, but belong to the platform component nonetheless because they are common to all the data acquisition modules that connect to them, but unique to the computer system to which they interface. Middleware performs a similar function as interfaces. While interfaces make it possible to connect standardized hardware to the platform, middleware allows standardized software to be implemented on the platform. During the development of a platform component, a generic version of the middleware is adapted to the platform specifics. The generalized middleware has its own life cycle. The software environment for object-oriented flight test instrumentation systems that is presented in this thesis, was developed using an evolutionary prototyping life cycle for which the model is shown in figure 1.8. Both the intermediate prototypes and the final version of the middleware can be used in the development of platform components. This way, the complete middleware life cycle takes place within the generalized component implementation activity that is indicated in figure 1.7; the component specialization activity covers the adaptation of the generic middleware to a specific platform.

The second group of components, that of data acquisition modules, also combines hardware and software. A typical data acquisition component consists of a sensor, interfacing or signal conditioning hardware, and software that activates and reads out the sensor and that provides agreed data to the other components in the system. Because interfaces are part of the platform components and because the middleware protects software modules from any platform specifics, data acquisition components should be platform independent. Finally, data processing components are often limited to software, but hardware is not strictly excluded. Processing modules that feed back information to a human operator will include display hardware or actuators in the case of active flight controls. Additionally, the components in the excitation chain of a closed-loop flight test instrumentation system will comprise actuator hardware. Whenever a data processing component includes hardware, its organization into a hardware and a software part is exactly the same as that for a data acquisition component.

Returning to the life cycle for flight test instrumentation system development that is shown in figure 1.7, the prime objective of the incremental application design activity is the identification of the platform, data acquisition and data processing components that are needed to fulfill the requirements as specified in the preceding analysis

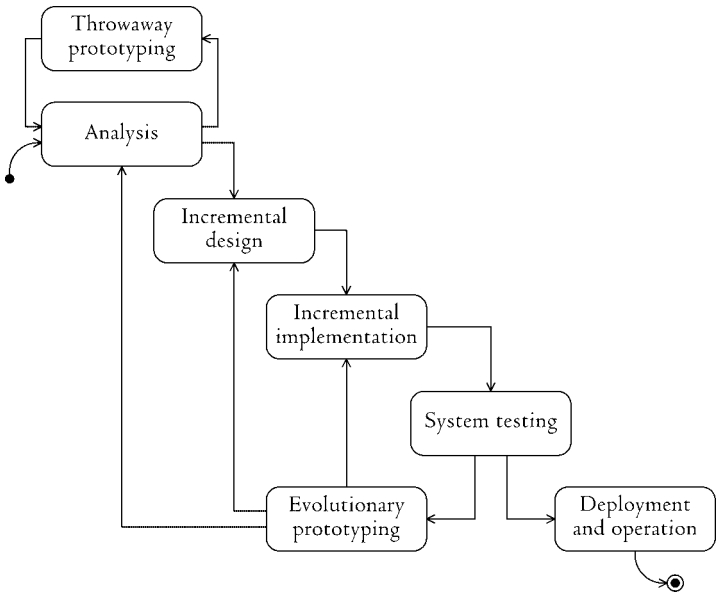


Figure 1.8: Middleware life cycle used in this thesis.
The evolutionary prototyping life cycle is typically extended with throwaway prototyping for any new feature that is added to the environment. In contradiction to application development, the middleware life cycle has a clear end. After a period of evolutionary prototyping, the environment is deployed and operationally used in the development of platform components.

activity. The independence of data acquisition and processing components from a certain platform component guarantees the flexibility to change the architectural design as the application design matures. The component life cycle covers the development of standard and customized components, including platforms, data acquisition modules, and data processing modules. As discussed above, a generic version of middleware serves as the generalized platform component; platform hardware only exists in the specialized component. The same applies to data acquisition and processing components that are a combination of hardware and software. The generalized components usually do not contain any hardware. Instead, a standardized software representation of the input or output device is created. During the customization activity, the front-end of the generic component is left unchanged. At the back-end, the actual hardware is added and the connection to the standard software is established.

Using the customized components, instrumentation system synthesis is a comparatively small task. For this reason, there is no direct feedback from the incremental prototyping activity to the incremental integration activity. Any change to the application

that arises from the prototype must be incorporated through a change in the application analysis or the component development; application integration is a mere combination of the previously implemented components.

The following chapters explore the details of the evolutionary flight test instrumentation development life cycle. The activities in the development method are discussed in detail. A summarized overview of the method is given in appendix B.

2
Application Development

To streamline the analysis and design of a flight test instrumentation system, four archetypal applications are identified. Open-loop testing systems, adaptive testing systems, human-factors testing systems, and in-flight simulation systems are four classes of applications that differ in the way information is fed back from data processing to system excitation. Classification of a new instrumentation system into one of these categories is the first step in application development. It supports system analysis and design by providing a framework for its common components.

The traditional techniques to model application requirements as part of an evolutionary development life cycle are not applicable to signal processing systems in general and flight test instrumentation systems in particular. For this reason, a new type of diagram is introduced that can be regarded as an extension to the set of diagrams that are offered by the Unified Modeling Language. The new diagram is referred to as the signal diagram; it models the complete flow of information through the application, while focussing on the information itself rather than on the components that are used to create or process it. Combined with a context model that reflects the characteristic structure of the four application classes, the signal diagram forms the basis for application design.

With the development methodology that is presented in this thesis, the measurand list has become obsolete. It entangles application requirements and performance specification of the components that are used to construct the system. Instead of the measurand list, the signal diagram and context model describe the application requirements; the application design and references to the component specifications document the actual achievements of the system without any loss or redundancy of information.

THE use of the development life cycle for flight test instrumentation systems that was introduced in the previous chapter changes the process of system analysis and design. Facilitated by the concept of object orientation, system development separates into the development of individual components and the synthesis of the complete system. Both stages have their own analysis and design activities. The modeling activities are followed by the implementation of the components and the integration of the full system. Application modeling is the first phase in this evolutionary life cycle. It comprises three activities: application requirements analysis, application context analysis, and incremental application design. These application development activities are followed by the development of the individual system components, which is discussed in the next chapter. Integration of the components, incremental prototyping, and system testing are activities of application development again, jointly referred to as application synthesis.

During application requirements analysis, the functional requirements for the instrumentation system are defined by specifying its role in the environment. The analysis during this activity starts at the external interfaces of the system. From this black box view, the prime requirements for the internal structure of the system are defined. During application context analysis, a static model of the system in its environment is created. It defines the interaction of the environment and the instrumentation system. Both the requirements analysis activity and the context analysis activity focus on the problem; during the incremental application design, the modeling is moved to the solution domain. An architectural design is constructed that specifies the components that are necessary for the system to meet its requirements. Additionally, the requirements for each of these components are specified. After completion of application modeling, detailed design and implementation of the various components are carried out in the incremental components modeling subcycle.

There is a distinct difference between a flight test instrumentation system and most other real-time or distributed systems. Flight test data acquisition and data processing applications are not dominated by contingent user input. Although an operator or a pilot in a closed-loop system might influence the way data are processed, the core of the system will generally run continuously and process data in a consistent manner.

Requirements analysis for a flight test instrumentation system therefore cannot be covered by the standard way to model a concurrent system, in which the system reacts to discrete user inputs and reverts to an idle state when the event has been handled.

2.1 Requirements analysis

Most design methodologies for object-oriented system development start with use case modeling to analyze the system requirements (Rumbaugh et al., 1991; Jacobson, 1992; Jacobson, Booch, and Rumbaugh, 1999; Gomaa, 2000). In use case modeling, the system is treated as a black box. Its functional requirements are represented in terms of the tasks that the system will perform for an outside user. Such outside users are referred to as actors. Each use case describes a typical application of the system and the major actions that are performed in the process. During the use case, the system may interact with additional actors. All use cases are nevertheless initiated by a single actor, which is referred to as the primary actor. Thus, a user-initiated, event-driven character of the system's application is essential to use case modeling. As a second step in traditional use case modeling, each case description is detailed in terms of one or more scenarios that textually describe a particular development of the use case. For example, the use case for a client who wants to withdraw money using an automated teller machine, will have different scenarios in case the user enters the correct identification number for the bank card that is used, and in case an incorrect number is entered.

Clear-cut use cases and scenarios are rare in flight test application modeling. In the event-based processes for which use cases are typical, the consecutive steps of a scenario are synchronized by interaction with the actor. In a real-time system that processes streaming data, an actor can influence the way the data is processed, but neither does the actor initiate the use case, nor are the system's activities synchronized with the user. Requirements modeling for flight test instrumentation systems therefore has to follow an approach that differs from traditional use case modeling. Instead of discrete use cases that are triggered by an actor, flight test instrumentation systems are characterized by continuous application as a signal processor. In this thesis, *signal modeling* is therefore proposed as a counterpart of use case modeling, adapted to the specific requirements of real-time signal processing systems. A new type of diagram describes the various signal streams in the system. It is named the *signal diagram* and combines elements from the UML's class, use case, and sequence diagrams with elements from traditional signal flow diagrams as used in electrical and electronics engineering. Just as a use case model is meant to serve as a basis for static modeling of the system's components, the annotated diagrams from signal modeling should be such that the signal descriptions allow for a detailed task analysis of the system components that are responsible for transforming one signal into another.

Types of applications

Before the analysis of the requirements of a flight test instrumentation system starts with the construction of a signal diagram, it is advisable to classify the application as one of four types of instrumentation systems. These four archetypal systems are the *open-loop testing system*, the *adaptive testing system*, the *human-factors testing system*, and the *in-flight simulation system*. The latter three systems are examples of closed-loop instrumentation systems. All four systems vary in the way feedback is used; the systems are listed in the order of increasing complexity.

- Open-loop testing systems are used when there is no feedback of the data processing results to the excitation of the system that is being tested. An open-loop testing system is the only system that can be implemented on a traditional open-loop instrumentation system as presented in figure 1 on page 5. For the excitation of the test subject, either there is no special test signal, or a predetermined signal is injected.
- Adaptive testing systems are the simplest implementations of a closed-loop instrumentation system as shown in figure 2 on page 6. An adaptive testing system provides basic feedback of data processing results to the excitation of the system, without involving the pilot or affecting the excitation itself. Although the last characteristic may seem contradictory to the mere feedback that is essential for an adaptive system, it distinguishes the adaptive testing system from the in-flight simulation system. In an adaptive system, the choice or duration of a test signal depends on the results of preceding data processing, but the shape of each test signal and the way it is applied are completely known before the experiment is started. This way, the safety-related preparations of an adaptive test can be the same as those of an open-loop test. A typical example of adaptive testing is data gathering on the performance of a system, where the required test duration depends on the information content of the measurements and where the information content that was actually achieved can only be determined when the data are processed. By processing the data in real-time, the test can be continued as long as the required information has not been gathered yet; when the minimum level of information has been achieved, the test can be stopped.
- Human-factors testing systems are characterized by the involvement of a pilot in the closed-loop instrumentation system. The system under test is not the aircraft or some of its systems only, but rather the combination of a human pilot, the flight guidance displays, and the aircraft's control systems and flight dynamics. Real-time processing is used to generate information for an experimental display or active flight controls. Through the displays or the active inceptors, results from the data processing are fed back to the system under test, in this case to the pilot. Since the instrumentation system performs the necessary computations to provide the

pilot feedback, it can alter the feedback signals with respect to the situation for normal operation. This may be used to optimize the assessment of the displays or the inceptors that are being evaluated.

- In-flight simulation systems are the most advanced closed-loop flight test instrumentation systems. In its simplest form, an in-flight simulation system can consist of only an experimental flight control system that excites the aircraft in response to control inputs from the pilot. Although it is arguable whether such a system is actually closing a flight test instrumentation loop, the pilot will normally adjust his inputs to the aircraft response. Hence, the system will at least operate as part of a closed loop; the requirements for the real-time performance of the data processing part will be similar to that for a human-factors testing system. An in-flight simulation system can be used to assess a prototype flight control system for the aircraft in which the instrumentation system is implemented, but it can also be used to mimic the handling qualities of another aircraft. It is this latter type of application that is traditionally referred to as in-flight simulation. It can be used to evaluate experimental flight control systems for aircraft in which a prototype control system cannot easily be implemented, or during the development of a new aircraft in order to assess some aspects of its handling qualities before the first flight. In its more extended form, an in-flight simulation system can be a combination of a human-factors testing system and an experimental fly-by-wire system. In such an application, the flight test instrumentation system occurs twice in the closed loop. At the first location, similar to the application in a human-factors testing system, the aircraft's flight data are used to feed back information to the test pilot. This feedback can occur directly from the data processing chain to the pilot by means of guidance displays, but in case the control system is equipped with an active-feel system, tactile responses can be returned to the pilot through the systems excitation chain as well. Based on the information from the pilot's inceptors, the instrumentation system excites the aircraft according to the flight control laws being assessed at the second location in the closed-loop. Both an in-flight simulation application that excites only the aircraft, and one that includes a human-factors system can be extended with an additional loop closure that directly feeds back the flight data to the control laws. Such a second loop closure will usually be present in any application where the flying qualities of another aircraft are simulated; it is required when an experimental flight control system for an aerodynamically unstable aircraft is being assessed.

A flight test instrumentation system that is used as a human-factors testing system or an in-flight simulation system needs to meet safety criteria that are absent in an open-loop or adaptive testing system. In both cases, the results of the real-time data processing are used in the closed-loop control of the aircraft. A human-factors testing system

does not interfere with the actual flight control system; instead, it influences the way the test pilot makes his control inputs. In this case, the presence of a safety pilot who can take over the control of the aircraft using non-experimental displays and controls will usually suffice to ensure experiment safety. An in-flight simulation system however interacts directly, and usually in a closed loop, with the aircraft dynamics. This introduces additional safety requirements that must prevent departure from the flight envelope. Characteristics like system reliability, functional redundancy, closed-loop stability margins, and failure mode behavior may all help in quantifying and analyzing the system's suitability for flight critical application in experimental test flights. As far as reliability is concerned, both the hardware and software side of the instrumentation system need to be considered. For analysis of software reliability, it is essential to distinguish between algorithm design, algorithm implementation, environment design and implementation, and real-time behavior. The former three aspects are related to the delivery of numerically correct results; the latter is related to the requirement for those results to be delivered in time. Each of these will have to be taken into account from the earliest analysis phases of instrumentation system development.

Signal modeling and signal diagrams

The signal diagram that is proposed in this thesis to support the requirements analysis of a flight test instrumentation system exhibits many similarities with conventional signal flow or block diagrams as used in electrical and electronics engineering. The key difference with the latter diagram types however is the purpose for which the model is used. Where the existing flow charts and block diagrams are mainly used during the design of an electrical or electronic system, the signal diagram is a product of the requirements analysis activity of a signal processing system. Consequently, the focus of the diagram is on the specification of requirements for the major signals in the system. The diagram does neither necessarily provide a comprehensive overview of all the signals in the system, nor must it depict all its anticipated components or show all cooperations or associations between components in the correct way. All these are objectives of the subsequent design activities. The signal diagram should display the primary signals and their flow through the system with a focus on the application of the full system. As such, the signal model should be more detailed and will represent the architectural design of the system more closely for signals that are closer to the prime output signals of the system. Because the intention of the signal diagram is to specify signal requirements, the diagram is extensively annotated with many characteristics of the signals that are depicted.

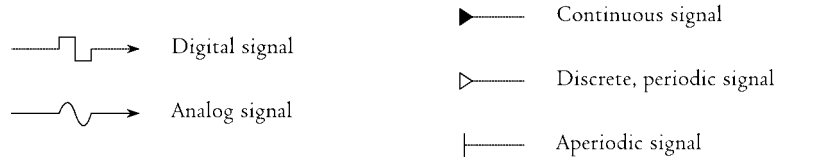
Figure 2.1 shows the elements that are used in the signal diagram. They are divided into three groups: the signals themselves, the annotations to the signals that specify their requirements, and the operators that process the signals. The signal symbology covers the primary signal characteristics. There are differently labeled arcs for analog and for digital signals; additional arrows at the start point of an arc indicate continuous or discrete signals. The actual signal type is described by its name. Similar to

mathematical convention, regular characters indicate a scalar signal and bold characters indicate a vector signal. Alternative types follow the UML convention for displaying objects: the underlined object name is followed by a colon and the class name. The class name can be used to refer to any user-defined type. If appropriate, it can be defined somewhere else in the signal diagram using the notation from the class diagram. Signal names and types can be followed by any combination of requirement specifiers. An at sign (@) and a colon precede the frequency for a periodic, discrete signal and the accuracy† of any type of signal respectively. Angle brackets are only used with digital signals; they include the resolution or precision, the range, or both the range and the resolution. The operators in a signal diagram are classes. They are depicted as usual in the class diagram. In addition to any user-defined operator, some common operators are predefined for the signal diagram. For an adder, an integrator, or a differentiator, the standard symbol as used in traditional flow chart diagrams can be used. From computer logic diagrams, the circle is copied as shorthand for a negator. It is used at the end of a signal, before connecting to another operator. In particular, it is often used in combination with an adder to indicate a subtraction.

Signal modeling typically starts at the back end of the system, by defining the signals that are produced as output. In this context, the term signal has a broader scope than it typically has in signal processing. It covers not only electrical signals that are used to transmit information, but also any other type of time-dependent physical quantity that conveys information or excites an adjacent system. For example, actuator positions or forces and moments that act on a body are considered signals as well, although they do not transmit information in the traditional sense. The output signals for the system follow directly from the application for which the system is developed. The requirements that are specified for the output signals in the signal model typically include the accuracy and any real-time or reliability requirements. The complete signal model is then constructed by analyzing the signal requirements from the system's back end towards the front. For each signal, the operator from which it originates is identified and indicated in the signal diagram. Each operator must have at least one outgoing signal; multiple output signals are possible. When an operator does not have an output signal, it is useless for the application and should be removed from the signal model. The operator and its output signals, in combination with the previously specified output requirements, determine the input signals to the operator and the corresponding input requirements.

This procedure ensures the completeness of the signal model with respect to the primary signals that contribute to the system's outputs. Additionally, it prevents the model from containing obsolete or redundant signal paths. This should not exclude

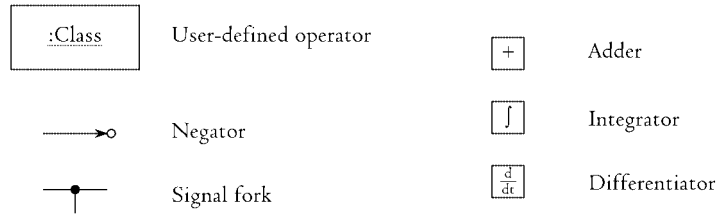
† It is suggested that accuracies are specified by the size of the single-sided interval about the true value that contains 95% of the signal's disturbed values, which corresponds to twice the standard deviation for an assumed Gaussian distribution. The use of different definitions – for example 65% or 99% intervals – should explicitly be mentioned in the diagram by means of a UML note.



(a) signal characteristics

name	Scalar	: acc	Accuracy
name	Vector	<res>	Resolution or precision
name:type	User-defined type	<min;max>	Range
@ freq	Update frequency	<min;res;max>	Range and resolution

(b) signal types and requirements



(c) operators

Figure 2.1: Elements of a signal diagram.
a. Signals are indicated by solid lines with an arrowhead in the direction of the signal flow. Signal qualifiers that distinguish between digital and analog signals, or between continuous and discrete signals are optional. Analog signals are continuous by default; digital signals are discrete and periodic by default.
b. Signal types are indicated by the signal name. Regular characters indicate a scalar signal; bold characters indicate a vector signal. Other types are associated with a user-defined class and indicated using the UML convention of underlining the object and separating the name and the type with a colon. All signal requirements like update frequency, accuracy, range, and resolution are optional.
c. Operators are indicated as a box. User-defined operators are objects and depicted by the corresponding rectangle from the UML class diagram. Predefined objects in the signal diagram include the adder, the integrator, and the differentiator. Subtraction is indicated by a negator at the end of a signal that arrives at an adder.

the possibility to leave out certain secondary signals in the signal diagram. Signals that do not participate in the major information flow, may not be essential for the architectural design activity that follows the signal modeling in the application development life cycle. Typically, these include signals that provide more or less static information that is needed by the primary signal flow. To avoid the signal model from becoming too complicated, such secondary signals can be simplified into a single reference at the location where the supporting signal is connected to the primary flow.

2.2 Context analysis

Following application requirements analysis, static modeling of a flight test instrumentation system starts with application context analysis. In general, analysis modeling focuses on understanding of the problem (Coad and Yourdon, 1990; Rumbaugh et al., 1991). The models that are created show the objects in the problem domain, which is the real world, and the way they interact. Gomaa (2000) proposes to construct a class model that highlights the information aspects of the system and its environment first. From this static model of the problem domain, a system context model is developed. Its emphasis lies with the interface between the external classes and the system. For flight test instrumentation systems, such an application context model provides an essential link between the requirements model and subsequent design models. However, the signal diagram that is proposed here for requirements modeling in flight test instrumentation development, is more detailed than traditional use case models. This makes the real-world model that precedes the system context model in Gomaa's method redundant.

Chains

Flight test instrumentation has been defined earlier as all equipment that is dedicated to performing or simulating flight test maneuvers, and to measuring, processing, and recording the results thereof. This definition marks the boundaries of the system. Interaction with the environment takes place at these boundaries, which are therefore the location at which the interfaces for the system are defined. A general idea of which interfaces require specification can be obtained by inspecting the definition. First, interfaces are found at those locations where the system initiates aircraft maneuvers, excites certain systems, or interacts with the pilot. Second, interfacing occurs where the effects of these excitations are measured by the instrumentation. Third, processing components can interact with the flight test engineer.

When developing an application context model with the intention to specify the flight test instrumentation system interfaces, it is desirable to recognize three of the various subsystems that make up the full system: the excitation chain, the measurement chain, and the processing chain. The excitation chain provides the interfaces to the aircraft that were mentioned first. The measurement chain provides the second set

of interfaces, by which information is imported from the aircraft. The processing chain provides the interfaces with the flight test engineer, mentioned third, and the interfaces that feed back information to the pilot other than by active controls.

It may be questioned whether all data visualization tools that interact with the flight test engineer are part of the processing chain. Typical on-line quick-look devices are attached to the rest of the flight test instrumentation system within the measurement chain, before the data recorders. To be able to fulfill their role in the instrumentation system, on-line quick-look devices must operate in real time, comparable to components in the measurement chain. Thus, pure quick-look devices are more closely related to measurement equipment than to processing equipment. The difference between visualization components and pure quick-look devices, is that the latter are limited to presenting parameters that are available within the measurement chain. All parameters that are calculated using multiple sources of information, or that are processed further with respect to the signal conditioning in the measurement chain, only become available in the processing chain. Visualization of those parameters is regarded as real-time processing and not as quick-look visualization. In a modern, DSP-based flight test instrumentation system, quick-look devices are typically implemented as displays that connect to the central processor. It is advisable to designate these as visualizers and to reserve the term quick-look device for equipment that is part of the measurement chain.

Archetypal system contexts

Application context models exhibit a clear pattern. For each type of flight test instrumentation application that was enumerated in section 2.1, context models will resemble archetypal class models that contain the external classes as actors, and all of the system's components as individual objects. The context model is more detailed than the class models in conceptual perspective that were shown earlier for the general open-loop and closed-loop instrumentation system. Context models are drawn from the specification perspective. As such, they provide a comprehensive and accurate overview of all the components of the system. The emphasis on the interfaces between the system and the environment allows the system requirements that were specified in the signal diagram to be addressed easily in the context model.

Typically, context models contain only instantiated classes. These are usually depicted as objects in an object diagram. This way, the functional requirements for each component can be specified alongside its dependencies on other objects, without having to identify all the similarities between objects by indicating parent classes. Abstraction of the instantiated classes in the context model into generalized classes is done during the design activity that follows the context analysis activity.

In an open-loop testing system, the only interaction between the system and the environment takes place at the end of the excitation chain and the beginning of the measurement chain. The context model, the archetype of which is shown in figure 2.2, contains the objects from the excitation chain and those from the measurement chain.

Since there is no feedback from the optional processing chain to the excitation chain, the latter typically consists of actuator objects and signal generators that excite the aircraft or system autonomously. For applications where a test pilot excites the aircraft without feedback from the system, the excitation chain is absent. For the measurement chain, all sensor, preprocessing, and recording components are shown.

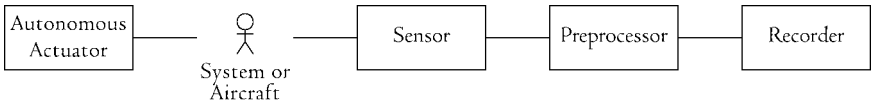


Figure 2.2: Archetypal system context for an open-loop testing system.
The test subject interacts with the autonomous excitation system on one side and the sensors on the other. The instrumentation system covers at least the sensors, preprocessors, and recorders. Data processing is not indicated here, but can be connected to the preprocessors for on-line analysis.

The archetypal context model for an adaptive testing system is shown in figure 2.3. The interactions with the environment are exactly the same as for the open-loop testing system; the objects in the excitation and measurement chains therefore are similar as well. The feedback of processing results to the selection and activation of autonomous excitation signals demands the presence of a processing chain. Processing components may interact with a flight test engineer by visualizing results or by accepting inputs that control the data processing. Interfaces between processing components and a flight test engineer are not shown in the figure.

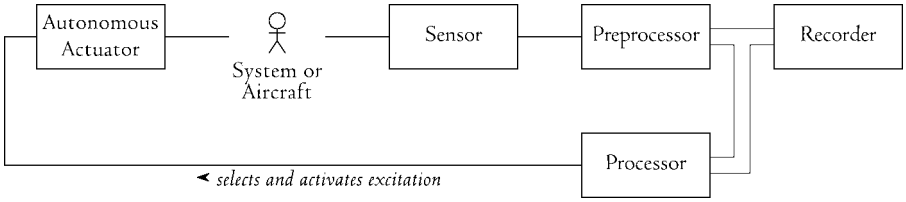


Figure 2.3: Archetypal system context for an adaptive testing system.
The test subject interacts with the instrumentation system in a closed loop, although its excitation is limited to predefined signals. Excitation, measurement and processing components are integrated further into a single instrumentation system than in the open-loop testing system.

Like the open-loop testing system, an adaptive testing system can be used for applications where the aircraft is excited by a test pilot instead of by autonomous actua-

tors. In an open-loop system, this only leads to a removal of the excitation chain; the test pilot is not depicted in the context model because he is completely hidden from the instrumentation system by the aircraft. For an adaptive testing system, the use of a pilot to excite the aircraft means that an additional interface is created: one between the processing chain and the pilot. Such an interface will be of the same nature as one between a processing component and a flight test engineer.

The context model for an adaptive testing system with pilot excitation shows strong similarities with that for a human-factors testing system as depicted in figure 2.4. The pilot directly controls the aircraft; the combination of both forms the environment to the instrumentation system. Interfaces are found at the aircraft side as part of the measurement chain – similar to the open-loop and adaptive testing systems – and at the pilot side as part of the processing chain. The difference between the adaptive testing system and the human-factors testing system lies in the type of interaction between the processing components and the pilot. For an adaptive testing system, the interface can only consist of a display that provides the pilot with information on when to excite the aircraft and what the results of his inputs are. It does not provide any information on how to excite the aircraft. Similar to the situation where no feedback is available, the pilot provides his inputs open loop. In the human-factors testing system, the pilot provides closed-loop inputs to the processing results. The actual shape of aircraft excitation therefore depends on the outputs from the processing components. Pilot interfaces in a human-factors system can consist of displays as well as active controls.

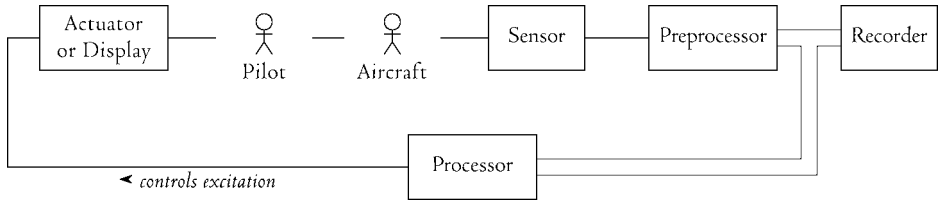
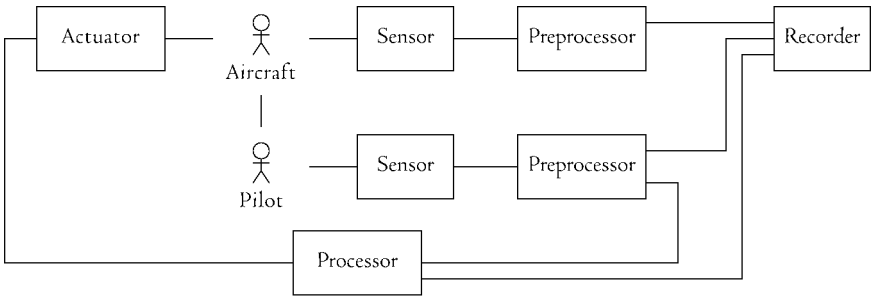
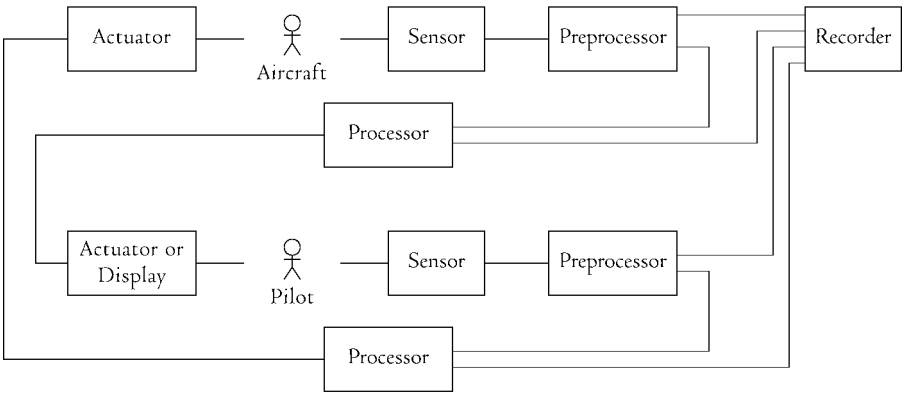


Figure 2.4: Archetypal system context for a human-factors testing system.
The pilot and the aircraft form a closed-loop with the instrumentation system. The on-line data processing components provide the pilot with feedback through display components or through actuator components when the system is to be equipped with active controls.

In-flight simulation systems are the most complex flight test instrumentation systems. The key characteristic of any in-flight simulation application is the interaction between the test pilot and the measurement chain. In all other application types where a test pilot is present, control inputs excite the aircraft without interference of the instrumentation system. Figure 2.5 shows various context patterns for instrumentation systems that share this key characteristic for in-flight simulation systems.



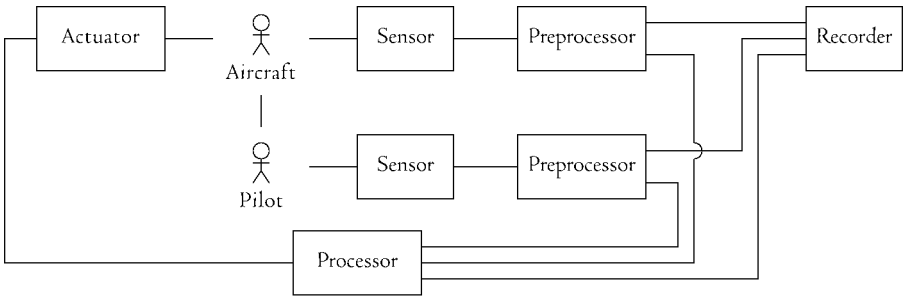
(**a**) single-loop fly-by-wire system



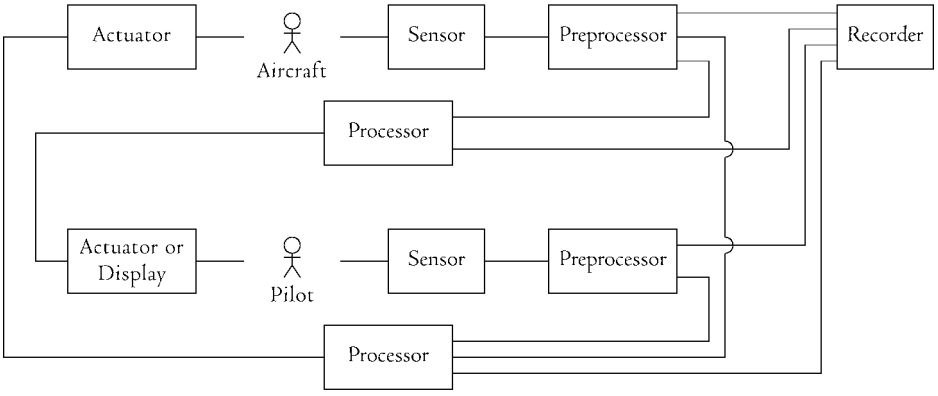
(**b**) single-loop man-machine system

Figure 2.5: Archetypal system contexts for an in-flight simulation system.
a. The structure of the single-loop fly-by-wire system shows strong similarities with the human-factors testing system. Processing components close the system loop by exciting the aircraft through actuator components.
b. The single-loop man-machine system combines the human-factors testing system with the single-loop fly-by-wire system. Processing components feed back the pilot inputs to the actuator components that excite the aircraft; another set of processing components feed back the aircraft response to the pilot's displays or active controls. (continued)

In its simplest form, the in-flight simulation system is a *single-loop fly-by-wire system*. It resembles the human-factor testing system where the location of the pilot and the aircraft have been interchanged. There are three boundaries where the system interfaces to the environment. Similar to any other system, aircraft parameters are mea-



(c) double-loop fly-by-wire system



(d) double-loop man-machine system

Figure 2.5, continued.
c. In the double-loop fly-by-wire system, the aircraft is excited by actuator components that interact with a single set of processing components. Control augmentation and stability augmentation are achieved through separate loops, comprising of sensors and preprocessor from the pilot inputs and the aircraft response respectively.
d. The double-loop man-machine system is characterized by sequential sensors, preprocessors, processors, and actuators or displays for aircraft excitation and pilot feedback in the control augmentation loop. The second loop, providing stability augmentation, is created by a direct feedback of aircraft response to the processing components that control aircraft excitation.

sured and recorded by the measurement chain. Additionally, the pilot's control inputs are measured in direct interaction with the instrumentation system. Unlike the measurement of control inputs in other applications than the in-flight simulation system,

the pilot commands do not excite the aircraft. This is the reason why control input measurements are no longer considered as the measurement of aircraft data. Aircraft excitation is performed directly and uniquely through interaction of actuator components with the airframe. Feedback of the aircraft response to the pilot is not covered by a single-loop fly-by-wire system. Instead, the pilot uses motion or visual cues from the aircraft or from the standard aircraft instruments as a reference. This marks the difference with respect to the *single-loop man-machine system*. The measurements on the aircraft are used to process information that is fed back to the pilot through a fourth interaction. Displays or active controls – similar to those found in the human-factors testing system – form a second instrumentation link between the pilot and the aircraft. The first link, connecting the pilot's control commands with the aircraft actuators, is referred to as *control augmentation* (Kelley, 1968). The second link, feeding back the aircraft response to the pilot, is best described as *display augmentation*. This term is used by Mulder (1999, p. 5) for the function of a flight director as a means for flight guidance automation. Although the pilot interface for an in-flight simulation system is not limited to a display, the role of this instrumentation link is the same as that of the flight director in normal aircraft operations.

Because no information is exchanged between the processors in the two aircraft-pilot links, this type of man-machine system operates in a single closed loop. Alternatively, information from the aircraft response can be included directly in the processing that precedes the aircraft actuation components. This type of link is referred to as *stability augmentation*; it directly connects the measurement and excitation chains on the side of the aircraft. Stability augmentation can be added to both a fly-by-wire system and a man-machine system. The result is a *double-loop fly-by-wire system* or a *double-loop man-machine system* respectively.

2.3 Design

Application design takes the application model from the problem domain to the solution domain. During application context analysis, a class diagram in specification perspective was constructed that contains all the system's objects. The context model is the starting point for the construction of a class diagram in implementation perspective. In addition to the individual objects in the system, it shows all generalizations of the classes to which the objects belong. Associations between classes are now specified at the correct abstraction level and aggregations and compositions are indicated. This way, the application design supports the analysis, design, and implementation of the system components and the subsequent integration and testing of the system.

The application design itself provides the information how all of the system's components are to be integrated to make up the full system. The complete representation of all class associations, generalizations, and aggregations provides a skeleton for the analysis of the system components. Especially the identification of standard compo-

nents that are used to derive specialized components from, is helped by the abstractions that are depicted in the static application design.

Because application design is incremental, it is not required to carry all of the classes from the context model to the design at once. Instead, a subset of the interactions can be implemented in a first prototype. The signal diagram that was created during application requirements analysis helps to identify subsets of the application's signal flow that are suitable for prototyping. The selection of such a subset is not arbitrary; an incremental prototype is not an intermediate system for testing purposes, but rather a complete system of its own that is to be used in the actual environment. The high costs that are related to flight testing, and thus to operation of a flight test instrumentation system, requires an incremental prototype to be a reliable, efficient, and productive asset.

Static modeling

Unlike class diagrams in the specification perspective, which focus on interfaces with the environment, static models that are drawn from the implementation perspective emphasize the relations, cooperations, and dependences of the classes within the system. In the signal model and the context model, only those classes are indicated that are directly involved with the signal flow through the application. The application design extends this view with generalized classes that group those classes that are associated because of similarities in their function. In addition, the static design model introduces classes and objects that are not directly related to the signal flow – for example platform components on which processing components are implemented.

The application design is therefore characterized by a more hardware-oriented view of the system than the requirements model or the context analysis. As such, typical designs do not fall into the four application types that were identified during requirements and context analysis. The distinction between open-loop testing, adaptive testing, human-factors testing, and in-flight simulation systems is based on the characteristic differences in the way these applications interface with the environment. The differences depend to a large extent on the logical structure of the application, not on its physical layout. With the emphasis on implementational aspects, it is unimportant for the application design whether feedback of processing results occurs in a single closed loop or in two closed loops, or whether processing results are fed back to the pilot, the aircraft, or both. As a result, the design of all closed-loop instrumentation systems can be based on a single pattern; a separate pattern is only applicable to open-loop instrumentation systems.

Figure 2.6 shows the archetypal design for an open-loop instrumentation system. Because the hardware structure does not support any feedback, the design layout covers merely the open-loop testing system. The reproduction chain, when present, will generally not operate in real time. Any ground processors that are attached to it are restricted to step-time operation as well. Figure 2.7 shows the archetypal design for a closed-loop instrumentation system. Feedback is achieved through any combination of

excitation chain interaction with the pilot or the aircraft, and direct visualization of processing results to the pilot. This way, the closed-loop instrumentation system design covers all adaptive testing, human-factors testing, and in-flight simulation applications. Because all components are part of the closed loop, the whole of the system operates in real time.

When an actual design is made, the archetype for the open-loop or closed-loop instrumentation system is extended with classes from the context model. In this process, the logical component classes from the context model are associated with the physical classes from the archetypal design. Classes like the ones labeled 'Processor' or 'Sensor' in the system context model indicate logical components that are responsible for a certain task in the chain of signal processing. In case of a processor, this will usually be a software component that performs certain computations. In the application design, the classes that are labeled 'Platform' refer to the actual hardware components on which the computations are performed. In a distributed system, multiple platforms are linked through a network that is indicated as an association class. Associations between the logical and the physical classes then indicate which software components are to be implemented on which platform. The complete application design will therefore be considerably more complicated than the archetypes shown in figure 2.6 and figure 2.7.

Dynamic modeling

The last step in application design is the specification of component requirements. Before that, a dynamic model of the application may be produced in aid of the static model that was described above. Sequence diagrams and statechart diagrams can be used to prescribe the dynamic behavior of certain system components. This is applicable for those operators in the signal model that depend on discrete inputs, and particularly for those that depend on aperiodic inputs. The desired response of the component to more or less random input sequences can be specified with a statechart diagram. Deterministic interaction between components, consisting of fixed signal sequences between them, can be modeled in a sequence diagram.

2.4 Synthesis

Application synthesis covers all of the activities in the evolutionary development life cycle that deal with the assembly and testing of the complete system. The availability of all system components, individually customized and unit tested, is a prerequisite for application synthesis. In figure 1.7, the activities that make up application synthesis are found at the end of the life cycle, following the subcycle for component development. The first activity in application synthesis is the integration of the incremental system. System integration actually consists of two steps: incremental application assembly and integration testing. The integration activity is followed by incremental prototyp-

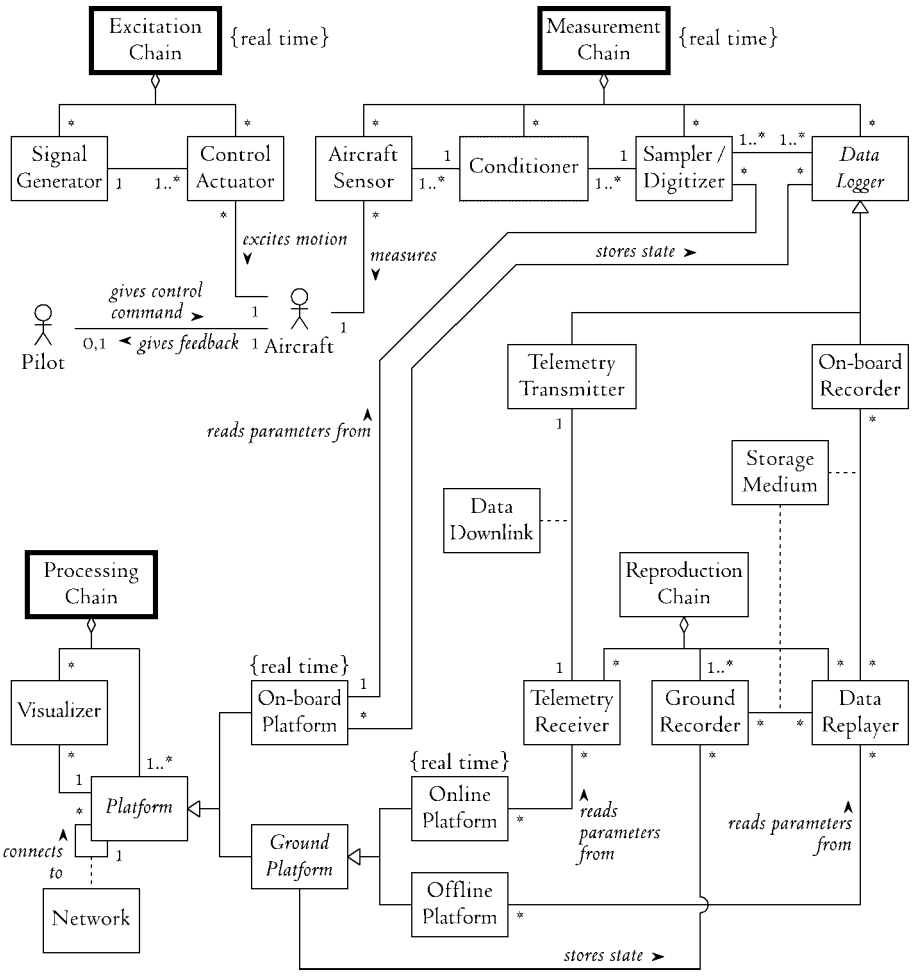


Figure 2.6: Archetypal design of an open-loop instrumentation system.
The class diagram from the implementation perspective shows the excitation, measurement, reproduction, and processing chains with corresponding components. The presence of excitation chain components and the pilot is optional. Processing can be performed on both on-board and on-ground platforms. Ground platforms either operate in real time, receiving flight data through a telemetry downlink, or in step time. The abstract data logger in the measurement chain is implemented as a telemetry transmitter or an on-board recorder accordingly. Both types of ground platforms store their results through a ground recorder. On-board platforms are connected directly to the measurement chain and always operate in real time. Because excitation, measurement, and data processing operate independently, each of the chains has its own control.

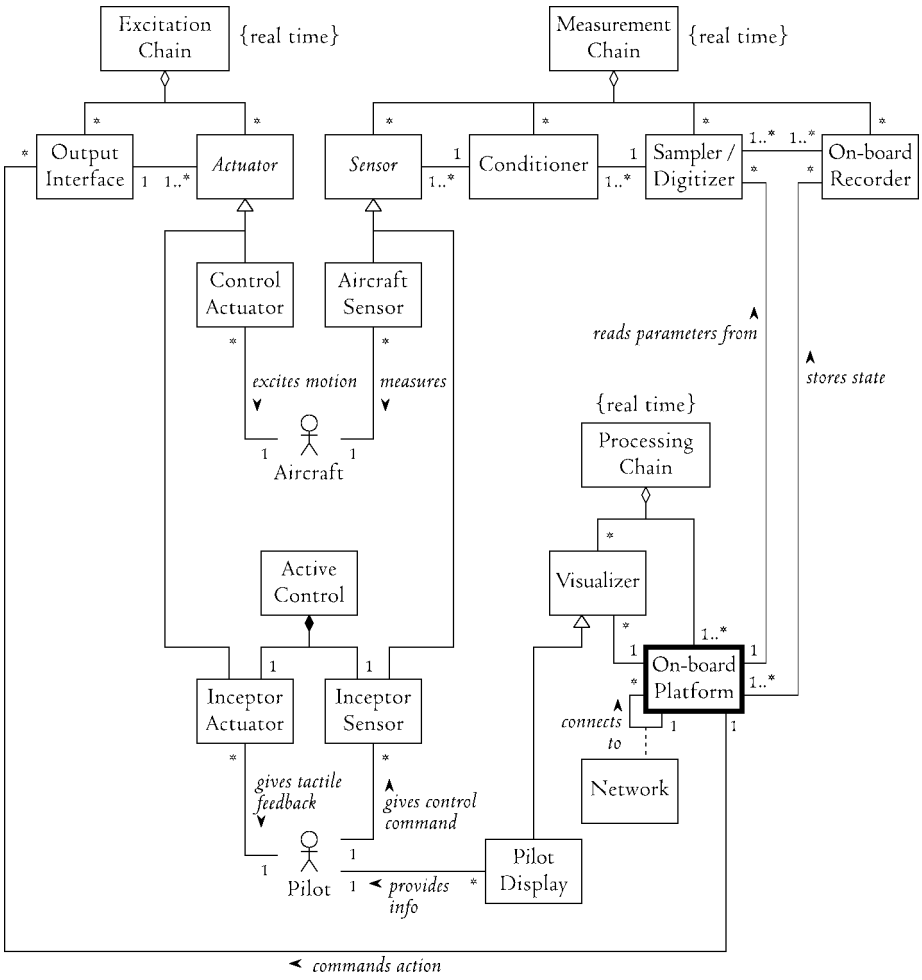


Figure 2.7: Archetypal design of a closed-loop instrumentation system.
The class diagram from the implementation perspective shows the excitation, measurement, and processing chains with corresponding components. The complete system operates in real time. No reproduction chain is part of the closed-loop system, but post-flight processing can be performed by a ground processor from an open-loop system as shown in figure 2.6.
Abstract actuators and sensors model the excitation and measurement role of aircraft actuators, sensors, and active controls. In their various instances, they interact with the aircraft or the pilot. Pilot feedback is not only achieved through active controls, but also through pilot displays. These displays are specialized versions of visualizers that can be part of the processing chain of any closed-loop system.
Because excitation and measurement components are controlled by the processing chain, the platforms are the only active components in the system.

ing. Incremental prototyping is the primary activity from which the development life cycle is iterated. Extended or amended application designs, or corrected component analyses and designs can be initiated directly from incremental prototyping. Alternatively, the development can move to system testing and operation.

It may be disputable whether application operation is to be considered as a part of application synthesis. Operation can be a final state. A return to application requirements or context analysis from the operation activity is not a necessity; as long as the environment and the intended use of the system do not change, the system can stay in use indefinitely. However, testing of a flight test instrumentation system and its operation are closely related. System development is usually done for a single implementation and system testing is impossible without operating the application in its true environment. System testing and operation are therefore considered as a single activity. Because system testing is doubtlessly a part of application synthesis, so is the joint activity of system testing and operation.

From the two steps that are part of the integration activity, incremental application assembly is a straightforward action. Given the incremental application design and all components that make up the system, application integration is nothing more than the combination of those components into the final system according to the design. During integration testing, the system is then tested against the application design itself. Integration testing is a case of white box testing (Gomaa, 2000, p. 100), in which the cooperation of the various components is verified. The result of the incremental integration activity is a *tested increment*: a prototype of the system that has been checked to exhibit the designed behavior.

Prototypes that are integrated during an early stage of the incremental design process cannot be expected to meet all of the system requirements. System testing, during which the system's performance is assessed against the functional requirements, is therefore not applicable to early incremental prototypes. Instead, the activity that follows the application integration will be incremental prototyping. The system is evaluated in operational use with the goal to identify design flaws as soon as possible. In case such a deficit is detected, the development process returns to the corresponding component analysis or design activity. When the prototype functions as expected, the development process continues with the next incremental application design activity, during which the functionality of the system is extended.

If a prototype is the product of the last cycle through the incremental development loop and incremental prototyping has not revealed design or implementation deficiencies, application integration is followed by system testing. System testing is a case of black box testing (Gomaa, 2000, p. 100), in which the functionality of the flight test instrumentation system is validated in its intended environment. As such, system testing can only be performed during a flight test program. Any laboratory testing is considered as a part of integration testing, including those tests in which a simulated environment – for example an iron bird – is used to mimic the operational environment. This type of testing can and should be applied to any prototype system, including both

the earliest increments and the final version. It is a natural preparation to incremental prototyping. Dedicated testing of the system in a flight test program is restricted to versions that are considered final. A successful completion of this validation promotes the tested increment to an operational system.

For some flight test applications, system testing is unpractical or economically undesirable. An instrumentation system that is to be used on the first flight of an aircraft type for which it was specifically developed, cannot be tested in its intended environment before being operated. New instrumentation systems for acquisition and analysis of flight data can only be tested by performing the appropriate experiments and assessing the results. If the system passes the test, there is no need to declare the flight test as only a successful instrumentation system test, and to repeat it during the following operation of the system. Thus, system testing of flight test instrumentation systems will be limited to those cases where the system directly affect safety – for example in an in-flight simulation system – or where a repeat of the flight tests in case of instrumentation failure is impossible or results in excessive additional costs. The latter are those trials for which specific experimental conditions must be awaited or created, or where complicated boundary conditions must be met. An example of such a situation is a flight test campaign that is conducted at a remote location. The cost of performing a flight test that merely validates the instrumentation system's correct functioning is then justified by the potential cost of a malfunctioning system that is only identified during the campaign.

2.5 Documentation and maintenance

The models that are created during application development – the signal diagram, the context model, and the application design – form the basis for the flight test application's documentation. Documentation must support the design and implementation of the system during its initial development, but it is just as important that future maintenance is supported by providing easy and reliable access to all information. For traditional development methodologies, in which development and maintenance are strictly separated consecutive phases, the documents that are created during system development are often unsuitable for maintenance. As a result, specific maintenance documents need to be created after system development. Such a practice is both inefficient and error prone. In contrast to traditional development methodologies, evolutionary system development is characterized by continuous redevelopment and extension of the system. All documentation that is created during the analysis and design activities is tailored to the repeated alternation of development and operation. Documentation development is therefore an inherent accomplishment of the evolutionary development methodology for flight test instrumentation.

The analyses of an instrumentation system are made in the problem domain; they are the responsibility of the flight test engineer. The requirements analysis, docu-

mented in the signal diagram, and the application context analysis, documented in the specification-perspective class diagram, fully describe the new instrumentation system as far as the end user of the system is concerned. Both the functional requirements and the general layout are the key characteristics that are important to the experimenter who applies the instrumentation system. However, these analyses are of little importance to the instrumentation engineer. The responsibility of the latter is the analysis, design, and implementation of all the hardware components that make up the system. The two responsibilities intersect when the flight test engineer and the instrumentation engineer jointly deliberate the application design. During the application design activity, the requirements for the system as represented by the flight test engineer, are translated into a set of component specifications that are the responsibility of the instrumentation engineer. The activities that follow this handover take place in the solution domain. In an evolutionary development process, the flight test engineer can thus be regarded as responsible for the problem domain, where the instrumentation engineer is responsible for the solution domain.

Although differently formulated, this separation of responsibilities is largely similar to the traditional roles of the flight test engineer and the instrumentation engineer (Adolph, 1994; Knight and Dove, 1994; Crounse, 1995). As already noted in the introduction, their distinct roles during instrumentation development are essential for a balanced, scientifically and economically sound system design. Nevertheless, redundancy and reliability problems for the documentation may arise when an instrumentation system is developed in a modular fashion with the use of as many stock or off-the-shelf components as possible. A conventional measurand list for an instrumentation system that is developed with an evolutionary life cycle, would be composed by the flight test or system engineer and the instrumentation engineer during application design. Such a measurand list would reflect the compromises that have been reached between the initial system requirements and the characteristics of existing or customized components that can be developed with comparative ease. As a result, information that is the result of component development influences the application design in advance and the difference between original system requirements and the actually achieved system specifications is obscured. Moreover, the duplication of component specifications into the measurand list breaches the requirement to maintain normalized databases; it introduces undesirable data redundancies and increases the risk for information anomalies.

While maintaining the traditional constellation of responsibilities between the flight test or system engineer and the instrumentation engineer, the documentation for the evolutionary development cycle provides a solution to these problems. The signal diagram contains all the information that is traditionally recorded in the measurand list, including the origin of the signals, their resolutions, ranges, accuracies, and update rates. Together with the context model, it fully specifies the functional requirements for the system. The application design provides a set of references to the components that make up the instrumentation system. The specifications for the components, and

hence the performance that the full system actually achieves, are recorded in the documentation that belongs to the individual components.

The measurand list has thus become obsolete. It is replaced by two sets of documents, which represent the different perspectives of the problem and the solution domain respectively. First, the signal diagram and the context model document the problem domain of the application, providing its functional requirements. These requirements are independent of the components that are used to realize them. Second, the application design documents the application's solution domain, providing the system specifications by reference to the documentation for the individual components. With this documentation concept, system maintenance is optimally supported. When application requirements change, it is easily verified which system components satisfy the new requirements already and which need to be updated, extended, or exchanged.

3
Component Development

The two stages of component development, development of a generic component followed by a specialization for the current application, have different effects for the various types of components. Software can truly be implemented in a generic form from which a specialized version can be derived easily. With hardware, it is at best possible to construct the design such that the major decisions are as generic as possible, in order to minimize the changes when the component is to be reused in a slightly different context. Where the development of software components therefore covers the full cycle of analysis, design, and implementation for the generic and the specialized components, hardware components are only implemented in their specialized form. On the other side, the inflexibility of a hardware implementation justifies the use of throwaway prototyping for a new component.

The components in a flight test instrumentation system are categorized in three groups: platform, data acquisition, and data processing components. Each group covers both hardware and software elements; the differences between hardware and software in the implementation of generalized components and in the use of throwaway prototyping therefore extend to all components. The characteristics of each component type are discussed comprehensively, emphasizing the way the components interact when they are integrated in an instrumentation system. For data acquisition or publication components, maximum reusability of the hardware and software elements is achieved by applying hierarchical layers to the design. A four-layer model is presented in which the port hardware of the digital signal processor, the corresponding driver software and wiring, the sensor or actuator, and the software that interacts with the other components are separated at standardized levels of abstraction. Each of the elements can therefore be exchanged without affecting the other parts of the component.

WITH the completion of the flight test application design, system development continues with the analysis, design, implementation, and unit testing of the individual system components. The object-oriented approach to flight test instrumentation system development ensures that each of these components can be reused in future applications, either as an evolved version of the current application or within a completely different context. Consequently, most components that make up an application are not proprietary to that application; they are mere specializations of standardized components that are used in various different systems.

This is the reason for the removal of component development as an integrated activity from the application life cycle. Instead, an independent component life cycle is embedded in the development process of the application. The proposed life cycle model for system components is that of throwaway prototyping. As previously shown in figure 1.7 on page 41, component development consists of the usual analysis, design, implementation, and testing phases. The additional throwaway prototyping – additional with respect to the waterfall model – takes place in interaction with component analysis. Especially in the case of hardware components, a rapid implementation of the component can be made and tested in combination with the rest of the system. Because of the comparatively large cost that is associated with hardware developments, throwaway prototyping both provides quick availability of a new component and reduces the risk of investing in a faulty or unsatisfactory design.

A difference between component development and the standard throwaway prototyping life cycle lies in the implementation activity. For component development, implementation is separated into two activities: generalized component implementation and component specialization. The main reason for this concept is the prevention of the development of single-use components. Every new development should be designed and implemented as a standard component as much as possible. Hence, a new component is modeled in its utmost abstraction before an implementation is made. The way the abstract design is used differs for the development of software and hardware components. Software components are not only designed, but also implemented and tested in their generalized form. The specialized version, tailored to the requirements of the current application, is subsequently derived from the generalized form.

This procedure is facilitated by the inheritance mechanisms in object-oriented programming languages. In case of hardware components, such an incremental implementation is often impossible or economically undesirable. However, the design of the component should still reflect the existence of a generalized component on which the specialization is based. When the new component is to be reused in a future system, this reduces the overhead that is required to adapt the component to its new environment, and improves the structure of the design, thus resulting in a more efficient and more reliable implementation.

The actual steps during component development depend to a large extent on the type of component. Differences that were already mentioned are the production of a throwaway prototype for new hardware components†, and the difference between hardware and software when creating a standardized component and deriving application-specific specializations from it. To all three primary types of components that were identified in chapter 1 – platform, data acquisition, and data processing components – the steps in component analysis are more or less common. Design and implementation however are completely different activities for the various component types.

The main reason for these differences is the combination of hardware and software subcomponents that is typical for each group. To model the components, the three groups are first separated into abstract and instantiated classes as shown in figure 3.1. Platform components form a straightforward concrete class. They are not derived from any other class. Both the data acquisition component class and the data processing component class are abstract. The latter covers output components and analysis components. Output components provide an interface between the instrumentation system and the environment by which processing results are published on-line; analysis components are used for internal processing or for computing results that are recorded for future analysis. Analysis components consist of only software; output components combine software with a hardware device. Input components provide an interface between the instrumentation system and the environment by which data is acquired. Because the modeling of an input device is similar to the modeling of an output device, both input and output components are derived from an abstract class that represents a generic device component.

Following the component analysis that is common to all component classes, the modeling of system components is driven by the specifics of the various abstract and concrete component classes. Platform components combine the computer hardware for

† Although throwaway prototyping originally stems from software engineering, its use in flight test instrumentation development is only adequate for hardware components. The incremental design and implementation of system components, which is part of the application life cycle, replaces any software throwaway prototype with a more efficient evolutionary prototype; unstructured implementations of a complete component are more laborious than fully structured incremental prototypes. For hardware components however, the first increment of a new component may be associated with high overhead costs. This can justify the implementation of a throwaway prototype with reduced investment of labor and material.

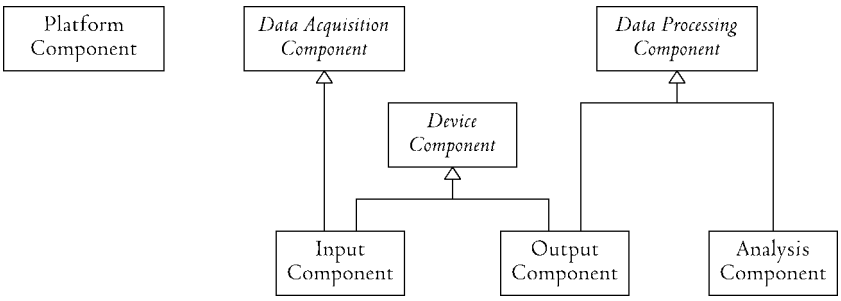


Figure 3.1: Classes of flight test instrumentation components.
Platform components form the computer nodes on which the distributed system is based. Data acquisition components always combine hardware and software to provide input data for processing. In data processing components, hardware is optional. The common hardware structure for data acquisition components and data processing components which communicate information to the environment, is classified as a device component. Input and output components are the instantiated classes that combine the characteristics of a device component with those of a data acquisition or data processing component. Processing components without output hardware are referred to as analysis components.

an instrumentation system node with the required middleware and certain interface hardware. Other parts of the interfaces between the system and the environment are modeled through the device components. The interfaces thus present the most complicated aspect of object-oriented flight test instrumentation design.

3.1 Analysis and prototyping

As described in the previous chapter, the final stage in application design is the creation of class diagrams from the implementation perspective, accompanied by optional dynamic models of the system and an overview of component requirements. By depicting the associations between platform components and data acquisition or data processing components, the class diagram shows on which platform each of the data acquisition and processing components is to be implemented. In addition, the application design presents abstractions of the system's components. Together, this information is the starting point for component analysis and for creating throwaway prototypes where they are deemed beneficial.

Component abstraction and task definition

Component abstractions that are indicated in the application design are usually limited to those that represent a generalization of two or more closely related component classes. To identify and model such abstractions is one of the activities during application design. However, this method to identify standardized components does not cover the rigorous abstraction of all components, including those that are not related to any other component classes in the present application. Therefore, the steps in component analysis start with abstraction.

Component abstraction aims to separate the characteristics of the component into groups that are associated with the different levels of application-dependent details. Only the component's functions, behavior, and associations that are truly unique for the current application should be modeled at the instance level. All other characteristics, which may also apply to a similar component in another application, must be modeled in abstract classes from which the component class is derived. Usually, abstraction is not limited to a single parent class, but extends to a series of levels that form a tree structure. For example, a data acquisition component that measures an angular velocity from a fiber optic sensor may be modeled over four levels: a generic component for inertial measurements, a rate sensor component, a generic fiber optic rate sensor component, and the application-specific fiber optic component. Each component is modeled as a child of the previous class. Alternative child classes can easily be added when accelerations are to be measured, or when a ring laser gyro is used instead of a fiber optic sensor.

When the process of abstracting component classes has completed, the functional requirements for the component can be assigned to the appropriate parent or child classes. Generally, associations with other components in the application should be modeled in one of the abstract classes; the standardized components thus define and provide the interface for the interaction of the component with its environment. The derived classes provide an application-specific *method* that fulfills the component's function. In this step of component analysis, the model is extended with information on the component task; until this point, only function has been modeled. Component analysis thus marks the step from modeling function, which defines what services the component must deliver, to modeling task, which defines what the component will do to achieve the fulfillment of its functional role. Nevertheless, the component analysis should be limited to specifying the principle of operation; the analysis must not contain all details that are required for implementation of the component. The latter is the responsibility of component design.

The reason for identifying the principle of operation for a component during the analysis rather than during the design phase, is the effect the overall task description may have on the associations between the new component and other classes. Especially with software components, the algorithms that are to be implemented may apply existing classes for data types or numerical procedures; hardware components may implement standardized support equipment like power supplies or data buses. All these

associations with external classes must be indicated in the final component analysis model. For this reason, an class diagram should be constructed that shows the component from the implementation perspective.

Throwaway prototyping

In software engineering, throwaway prototyping is used during the analysis phase of a new product. It aims to identify the functional requirements for the product from an evaluation in its operational environment. As such, this original type of throwaway prototyping is not applicable to the components of a flight test instrumentation system. The functional requirements for each component follow directly from the application design that specifies how the functionality of the entire system is distributed over the contributions from the individual components; there can be no unclarity on the requirements for a component that requires the use of a prototyping technique. For software components in flight test instrumentation systems, this means that throwaway prototyping is not applied at all.

For hardware components however, throwaway prototyping can offer clear benefits to the design rather than the analysis of a new component. In this respect, the prototype is not used to investigate the functional requirements for the new system element, but to evaluate the quality of a certain implementation. It is noted that such a prototype is closely related to the classical prototype that is usually constructed in the pre-production phases of any technical product. To assess its effectiveness or reliability, or – in isolated cases – to prove the principle of operation, a temporary implementation of the component is made and integrated in the system or in a special testing environment.

A prototype may incidentally take part in the application life cycle as a first implementation of the new component. In those cases, the throwaway prototype is applied like an evolutionary prototype with the difference that its implementation is not intended to be reused in a following iteration. This offers the opportunity to assess the prototype in the true operational environment through the incremental prototyping phase of the full instrumentation system. In deviation from the application life cycle that is shown in figure 1.7, this way to start the development of a new component jumps from the throwaway prototyping activity directly to incremental integration of the application; thus, formal design and implementation of the component are bypassed.

The result of throwaway prototyping for flight test instrumentation hardware is increased knowledge on the suitability of a certain conceptual implementation. This knowledge can be used to evolve the prototype and finally to improve the analysis and design of the durable version of the component. Because the principal design of the component is made as part of the analysis activity – in order to identify the external classes that are associated with the component – feedback from the throwaway prototype is used in the component analysis rather than the incremental design activity.

3.2 Hierarchical layers

The interface components of a flight test instrumentation system are the subsystems through which the system interacts with its physical environment. They cover everything from sensors and actuators, via signal conditioners and converters, to the interface hardware that belongs to the digital signal processors. The problem of object-oriented interface development is the dependence of the design on these many different properties that cannot be separated by information hiding within a single element. The solution, as introduced by Dijkstra (1968b) for a multiprogramming operating system, is the use of *hierarchical layers* in the object structure. In the digital computer system that is described by Dijkstra, all activities are distributed over a number of sequential processes, which are subsequently placed at various levels in the system structure. The activities at each level depend on the levels below, but not on any activity at a higher level. This way, a hierarchically layered structure is obtained that allows for thorough abstraction of the activities at each layer. Hierarchical layers thus facilitate the application of information hiding in a context with many interdependent elements.

Hierarchical layers in support of information hiding for complex systems have found wide acceptance (Shaw and Garlan, 1996) and are used in almost every ground-based computer system that has been developed since the 1980s. Amongst the most important applications are the layered structure of modern operating systems (Silberschatz and Galvin, 1998) and the ISO open systems interconnection (OSI) reference model for network communication (International Organization for Standardization, 1981), which is closely related to the layered network model that is used for the internet (Tanenbaum, 1996).

The A-7 case study

Application of hierarchical layers to the design of airborne systems is less common. Parnas, Clements, and Weiss (1985) present the results from a case study for applying information hiding and hierarchical structuring to an alternative design of the navigation and pilot display software that is used onboard the A-7E naval attack aircraft. It has the responsibility to provide information for the aircraft's head-up display, moving map, cockpit warning lights and dials at a rate of 25 Hz, calculated from air data measurements, forward-looking and Doppler radar, inertial sensors, cockpit switch settings and store configuration data. The software includes different altitude and navigation modes and supports the A-7's various target designation scenarios by providing ballistic computations and controlling automatic weapon release.

The project was conceived as a demonstrator for improved software design that exploits the advantages of information hiding. The case of the A-7E software was chosen because it allowed for the comparison of the new design with the existing software, which was characterized by poor maintainability and small margins with respect to violation of memory and real-time constraints. The modular structure of the new

Vought A-7E 'Corsair II' (photograph Vought Aircraft Industries, USA).
The A-7A light naval attack aircraft made its first flight in 1965. The E-version is characterized by increased thrust and greatly enhanced avionics; it stayed in operation with the United States Navy until 1991. Apart from 850 carrier-based USN A-7s, the A-7 was acquired by the Hellenic and Portuguese Air Forces and the Thai Navy.

design is presented by Britton and Parnas (1981). It contains a top-level decomposition into three layers†: the hardware-hiding layer, the behavior-hiding layer, and the software decision layer. Each layer is differentiated into two to five modules, which are all further divided into submodules. Although no hierarchical structure exists among the modules and submodules, the top-level layers exhibit clear client-server relationships. The hardware-hiding layer is at the lowest level; it provides *virtual devices* to the modules in the behavior-hiding layer. Because the virtual devices are independent from the actual hardware that is used in the system, they enable the exchange of hardware without affecting any of the software above the hardware-hiding layer.

Hierarchical structures in aerospace software engineering

Although the A-7E case study by Parnas received considerable attention in the software engineering community – where it is regarded as one of few examples of the strict application of information hiding to a real-life design (Kopetz et al., 1991; Opdyke, 1992, p. 12; Stewart, Volpe, and Khosla, 1997; Shaw, 2001, p. 657) – it appears to have gone largely unnoticed with the aerospace community. Anderson and Dorfman (1991) present an overview of software engineering concepts that are of interest to the aerospace community. Although the topics that are discussed range from development management to implementation practice and programming languages, life cycle philosophies are emphasized and software architecture concepts receive only limited attention. Neither hierarchical layers nor the work by Parnas are explicitly mentioned. How-

† Conforming to the terminology that is used by Parnas (1972), Britton and Parnas (1981) refer to these hierarchical layers as modules.

ever, Bondeli (1991, p. 530) and Jennings (1991, pp. 556-558) briefly discuss the Hierarchical Object-Oriented Design (HOOD) method (Robinson, 1992), which was developed in 1987 under a contract from the European Space Agency. They point out that HOOD defines two types of hierarchical relationships between objects: the uses-services-of relationship and the is-component-of relationship. The former corresponds to hierarchical layers as defined above; the latter relationship corresponds to what is more commonly known as an aggregation.

In the context of the Advisory Group for Aerospace Research and Development (AGARD) Avionics Panel symposium on aerospace software engineering for advanced systems architectures (Advisory Group for Aerospace Research and Development, 1993), Occelli (1993), Micouin and Ubeaud (1993), Mala and Grandi (1993), and Lacan and Colangeli (1993) discuss various HOOD-based software developments. These designs emphasize the top-down approach that is advocated by the method to model the software for large, complex systems. As such, the aggregation-type of hierarchies prevails in the designs. It is also recognized that HOOD does not exhibit the typical characteristics of an object-oriented design method. As Micouin and Ubeaud (1993, p. 7) point out, HOOD does not support crucial object-oriented concepts like inheritance and polymorphism. It is strongly biased towards the procedural-oriented programming language Ada. Thus, although the HOOD method allows to create a design with hierarchical layers, its lack of support for some object-oriented concepts and the emphasis on structural hierarchies make it less suitable for the development of systems that are to apply the concept of information hiding in a hierarchically layered structure. The confusing use of the term hierarchy for an aggregation within HOOD may very well obscure the beneficial characteristics of hierarchical layers and complicate the creation of a truly object-oriented design. Yet, because avionics and flight test instrumentation systems unavoidably interact with various hardware elements that should be exchangeable with little impact, the concept of hierarchical layers is particularly suitable for the design of these systems.

3.3 Interfaces

In order to apply the concept of hierarchical layers to the development of a flight test instrumentation system interface, the scope of an interface must be carefully defined and its constituents must be identified and abstracted. After the layered structure has been determined, the individual elements can be allocated to the various layers. Finally, the hierarchically structured elements can be distributed over the instrumentation system's components that are indicated in figure 3.1.

The three major constituents of an interface are shown in figure 3.2 together with the two most important supporting elements. The sensor or actuator is referred to as the *device*. Since devices interact directly with the environment, they are located at those positions where the physical signals are available for acquisition or where they are

required from an actuator. The interfacial hardware to the signal processor is referred to as the *port*. Because the port is located with the DSP, the distance between the device and the port is covered by wiring that may not always be trivial to create or install. Dedicated software, known as the driver, controls the communication between the port and the signal processor.

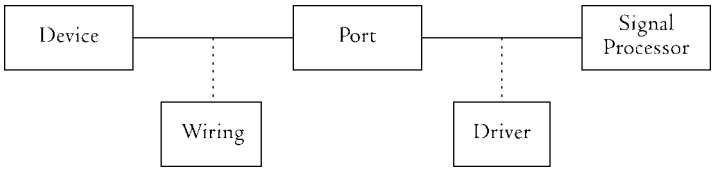


Figure 3.2: Elements of an interface.
Among the three classes and two association classes, the device interacts directly with the environment. It includes any sensor- or actuator-specific signal conditioners and is connected to the port by means of wiring. Communication between the port and the signal processor is governed by driver software.

Interface design and implementation are affected by the characteristics of both the environment side and the DSP side. Therefore, an interface must be designed for a specific combination of a signal processor and an external input or output signal. This may easily threaten the object-oriented requirements for exchangeability and reusability of the processing unit and the interface elements themselves, of which the device is the most important.

Layers

The hierarchical structure for the A-7 software that is presented by Parnas, Clements, and Weiss (1985) contains a few layers, only one of which is used to abstract the hardware components that the system interacts with. Nevertheless, it intends to hide the specifics of the hardware completely and provides virtual devices to the data processing software that runs on top. It is this concept that is extended here to a series of layers for the successful design of interface components for a flight test instrumentation system.

Starting with the driver software that controls the port as indicated in figure 3.2, several abstraction layers are identified that handle the interaction between the signal processor and the device from the lowest level, where the raw communication with the port takes place, to the highest, where the signal processor provides or acquires agreed data in engineering units. As shown in figure 3.3, four layers are defined.

- Layer 1 contains the port driver, which consists of the software that converts a request to the specific implementation of a port into a series of low-level instructions for the digital signal processor. The development of

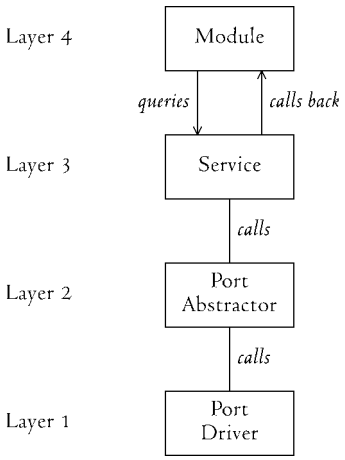


Figure 3.3: Hierarchical layers in interface driver software.
The port driver hides the low-level port commands behind meaningful procedures. The port abstractor hides these port-specific procedures behind a standardized interface for the type of port that corresponds to a certain device. The device specifics are hidden behind the service. It is queried by the module which hides the mere existence of the hierarchy to the user or provider of the device that is communicated with.

port driver software is not an activity in the development of the interface component; the port driver should be nothing more than the standard interface software that is supplied with the port hardware. The responsibility of the port driver layer is to hide the hardware of the port behind a series of software procedures that take care of the basic functionality of the port.

- Layer 2 contains the port abstractor, which consists of the software that converts standardized procedures for a type of port into the implementation-specific procedure calls at the port driver layer. It is the responsibility of the port abstraction layer to hide the peculiarities of the port driver behind standardized software procedures. The port abstractor thus provides a virtual port to the service layer, which can be called without any information on the implementation of the port.
- Layer 3 contains the service, which consists of the software that converts a request or a command to an interface device into procedure calls to the virtual port of the type that the device is connected to. The responsibility of the service layer is to hide the device behind a series of software procedures that represent its data acquisition or distribution functionality. The

service provides a virtual device to the module layer, which can be addressed to acquire data from, or to export data to the environment without knowledge of the actual implementation of the device or its corresponding signal conditioners.

- Layer 4 contains the module, which consists of software that converts the interaction with other data acquisition or data processing components into the communication with the virtual device. Its responsibility is to hide the interaction between the system and the environment behind a standardized element that represents the physical signal outside the instrumentation system. The module layer thus provides a virtual external signal that can be used without knowledge of the type of device that is used to acquire or provide the external signal.

The service layer is generally the most complex of the four interface driver software layers. The conversions that it takes care of, cover everything from the module layer's simple request for the acquisition or export of data, to the possibly multiple calls to the virtual port that are required to complete the request. Because setting up the device or its signal conditioners for communication, and acquiring or converting the signal may consume more time than desirable in a real-time system, the communication between the module layer and the service layer is organized in a challenge-response procedure. The module queries the service, after which program control is returned immediately. When the service has completed the request, it calls back the module with a notification and possibly the outcome.

Hardware-software equivalences in interface layers

The four layers of an interface driver each provide a different abstraction of the interface component. Layer 1 is used to communicate directly with the hardware. On top of this, the abstractions from layer 2, 3, and 4 provide the virtual port, the virtual device, and the virtual external signal respectively. The three abstractions correspond directly to the three physical entities of the same name. This equivalence between hardware and software within the layered structure of an interface component is depicted in figure 3.4. It shows the elements of the hierarchical layers in the driver software on the left-hand side and the hardware elements of the interface, together with the external signal on the right-hand side.

On the hardware side, the port provides an interface† to which the device is connected. The device itself must be implemented in the system in a prescribed way, depending on the individual sensor or actuator. As such, the device provides its own interface, this time towards the port. The two interfaces are connected by the wiring which is typically proprietary. The dedicated driver software that belongs to the port presents the low-level port functions to the port abstractor. Because of the dependence

† Here, the term interface refers to the local interface of the port element as indicated in the UML diagram, not to the complete instrumentation interface of which the port is only a part.

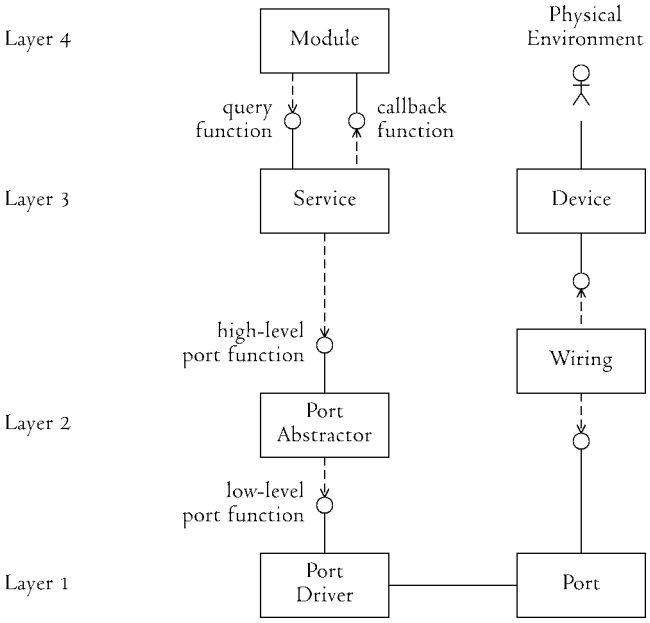


Figure 3.4: Hardware-software equivalences in interface layers.
The hardware elements – shown at the right-hand side – include the device, the port, and the connecting wiring. These elements, plus the external signal as it exists in the environment, are abstracted by the corresponding layers in the interface software – shown at the left-hand side.

of these functions on the specific implementation of the port, the software interface is generally at a lower level of abstraction than the interface at the hardware side which is more or less standardized for the type of port. However, the high-level port functions that form the interface to the port abstractor are usually at a slightly higher level of abstraction than the hardware interface. Nevertheless, port, port driver, and port abstractor are closely related. Together, they interact with the service-layer software and the hardware device through standardized interfaces. This relation will prove important for the further development of the interface component.

The software module and the physical signal are at the same level of abstraction. Likewise, because the service layer is used to provide a virtual device, the service and the hardware device are at a common level. At the layer boundary below the service and the device, there is a difference in abstraction level for the hardware and the software side, caused by the higher abstraction of the virtual port with respect to the hardware port. Therefore, a gap exists in terms of levels of abstraction between the device and the interface of the hardware port. Port abstractors and services are to be developed in

such a way, that they can interact directly. However, a hardware device cannot be plugged into a port without any adapter. The required adaptation is provided by the wiring. The wiring thus bridges the gap in level of abstraction between the device and the port.

The hardware-software equivalences at the various interface layers are important to the maintainability of the complete interface. When one of the hardware components is to be exchanged for a different implementation – for example when a different sensor is used – the impact of the change is restricted to the software at the corresponding layer – in this case the service. Interfaces are assembled from one hardware/software pair for each of the layers. The abstractions of the four layers ensure that a hardware/software pair can be used in various interfaces without modification.

Grouping and allocation of interface elements

As noted above, the port, port driver, and port abstractor form a closely related group of elements. They are jointly referred to as a *port unit*; the composition of a port unit is shown in figure 3.5. Because a port is typically a hardware entity that belongs to a certain type of digital signal processor, the port unit as a whole is strongly associated with the signal processor. Interaction with the device hardware and the service software is controlled through interfaces. The port unit provides its services to the superimposed layers in the hierarchy, without any dependence on or even knowledge of these layers. Therefore, port units must be developed as a part of the platform component for the digital signal processor that they belong to.

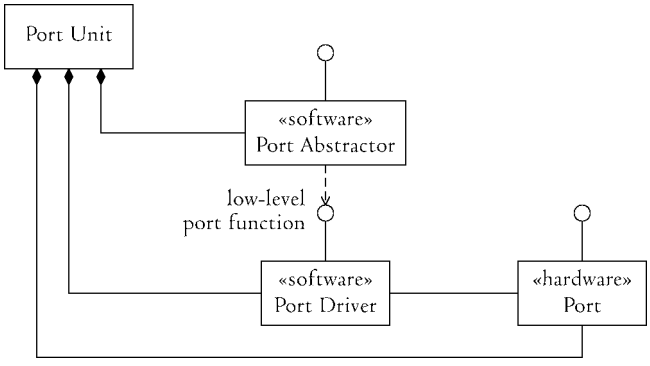


Figure 3.5: Composition of a port unit.
By combining the hardware port, its accompanying port driver, and the proprietary port abstractor, an interfacial unit is created that extends a digital signal processor by two interfaces: a hardware connection for the instrumentation devices, and a virtual port for the corresponding services.

A similar composition of interface elements can be defined for the device and the corresponding service. Because the service is used to hide the particular principle of operation of the device from the user of its results, a service will need to be developed on a one-to-one basis with each device that is included in the instrumentation. The composition of a device and a service is referred to as a *device unit*. Again, interaction with superimposed layers is handled through interfaces that do not require the device or the service to be aware of its users. On the side of the port layers, the service is directly connected to the virtual port that is provided by the port abstractor. The hardware device however only connects to the standard port by means of the wiring. The wiring, although its development is usually a comparatively small task, should therefore be regarded as a part of the device unit as well. The complete composition of a device unit is shown in figure 3.6. As a whole, the device unit must be developed with the device component that also contains the module that uses the device unit.

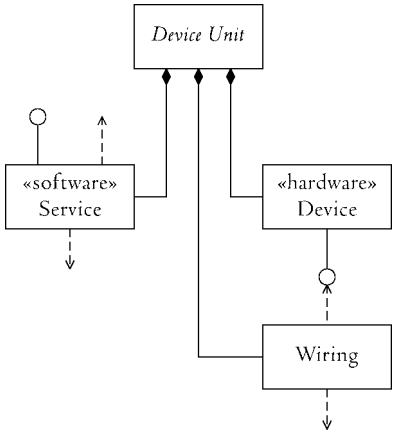


Figure 3.6: Composition of a device unit.
Based on the interfaces that are provided by the port unit, the device unit combines the hardware sensor or actuator with its proprietary wiring and software service in order to provide a virtual signal. The device unit is therefore completely isolated from both the platform-dependent port unit and the application-dependent module.

3.4 Modules in data acquisition and processing

The software modules in the hierarchical interface structure that is described in the previous section, have the responsibility to hide the interaction with the interface's device. They provide a virtual external signal to the other interface and data analysis

components in the instrumentation system. The key characteristic of the virtual signal is that it behaves independently of the particular way the signal is acquired or processed; only its appearance as a representative of a certain physical, external quantity is visible. As such, the concept of a virtual signal can be applied to internal signals as well. A signal thus represents a physical quantity, independent of its origin. Because each signal can be seen as a model of a part of reality, signals allow the various producers and users of information to connect without knowledge of their respective implementation, as long as the original, physical meaning of the signal is known.

Device and analysis modules

As a result of the concept of virtual signals, which guarantees that the use of a signal is not influenced by the way it is produced, all signals in the instrumentation system are hidden behind a module. The module is a software entity that is responsible for the correct acquisition or computation of the signal, and for the correct and timely availability of the signal to its users. Figure 3.7 depicts the classes of modules. The abstract class Module models the basic properties, including the interaction with other modules in the system. Its first derived class is that of a device module. This class corresponds to the module class that occurs in the hierarchical structure for an interface component as shown in figure 3.3 and figure 3.4. A device module communicates through an interface with the service layer of the same interface component. This communication, hence the fact that the module is in the top-most layer of a hierarchical structure, is the distinctive characteristic of a device module. The second class that is derived from the generic module is that of analysis modules. Analysis modules cover all modules in the system that are not device modules. They take care of the computation of intermediate signals that are not acquired from or exported to the environment. Therefore, an analysis module does not take part in a hierarchically layered structure.

Referring to the classes of instrumentation system components as presented in figure 3.1, components that handle data are classified as either an input, an output, or an analysis component. Input and output components are subtypes of the class of device components. Because its main objective is the acquisition or distribution of data between the system and the environment, the inclusion of a device module is essential to any device component. The hierarchical structure of the interface that is responsible for the interaction with the environment is identical for sensor and actuator devices. The composition that links the device module to the component can therefore be modeled at the level of the device component, as shown in figure 3.7. A similar composition exists for an analysis component with respect to an analysis module, since analysis components cannot function without a piece of software that is responsible for the interaction with other modules. However, the composition is defined for the analysis component itself and not at the level of its parent class, the class of data processing components. The latter is also the base for the class of output components; linking analysis modules to data processing components would make them an element of output components as well.

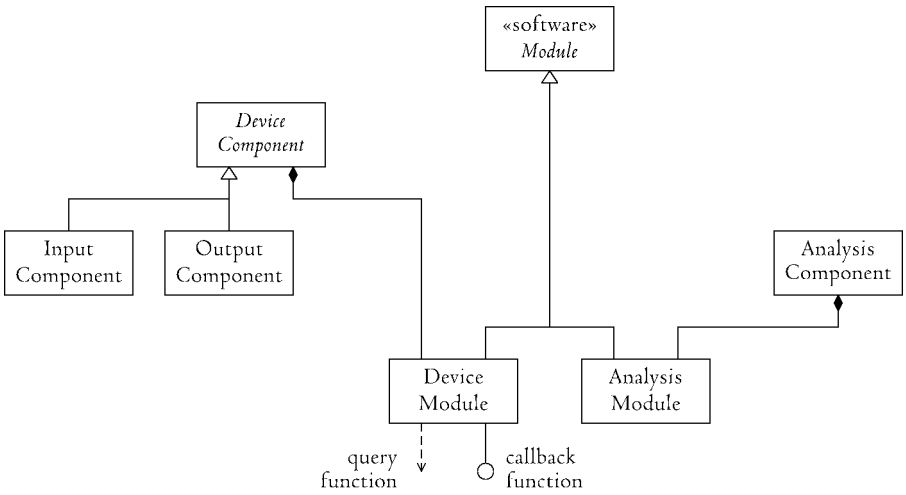


Figure 3.7: Classes of modules in data acquisition and processing components. Derived from the abstract software module type, two classes are used to abstract the signals that are associated with a device component and an analysis component respectively. External signals, acquired by an input component or exported by an output component, are hidden behind a virtual signal that is provided by a device module that interacts with the virtual device. Internal signals, produced by an analysis component, are provided by an analysis module.

Modules and virtual signals

It was stated in the previous subsection that the responsibility of a module is the encapsulation of a signal in a way that its implementation specifics are hidden from its users. The question may be raised whether this is the correct way to model signals according to object-oriented concepts. The tight coupling between modules and virtual signals may be regarded as a violation of the concept of information hiding. In an alternative approach, virtual signals could be modeled as a class of their own. The method that is used to create the signal – in case of input or analysis components – or to export the signal – in case of output components would be amongst the hidden properties of a virtual signal class. In this approach, modules would be obsolete; signals would be fully responsible for constructing themselves from other signals.

Nevertheless, the distinction between modules and signals in a flight test instrumentation system is necessary and correct. Modules are to be regarded as operators; they are the entities that act on the signals. Signals are the operands to the modules, the entities that is operated upon. Important characteristics of a signal are the parameters that are specified during signal modeling (see section 2.1 on page 50) like type,

the sample rate of a discrete signal, range, and accuracy. As such, signals are data type classes. The activities of an operator – including any data processing algorithm that may be applied, or the communication with a complex device – are not necessarily a trivial characteristic of the signal. As such, it is not appropriate to model these activities as an integral part of the signal class. A separate module class for these extensive operators is suitable. Additionally, a concept of virtual signals without modules is difficult to implement for output units. Without modules, all activity is hidden behind virtual signals. The computations that are required to create an internal signal, can be implemented in the software constructor of the virtual signal class. For physical signals that are exported however, this is impossible. A dummy signal would therefore have to be created, which would have the responsibility to send a signal to the environment. Such a dummy virtual signal class is in fact a module.

Therefore, virtual signals are modeled as a class, but they implement only the characteristics that are inherently related to the data that is being exchanged through the signal. They do not include any method that creates or uses the signal; these are implemented in module classes. When an instrumentation system is changed in a way that a certain signal is computed from a different source or with a different method, this ensures that neither the signal nor its users are affected; only the producing module needs to be adapted.

Generally, a module handles multiple signals. Most algorithms that are used in a flight test instrumentation system depend on multiple input signals to calculate one or more output signals. The complex instrumentation that is hidden behind a single data acquisition component often produces more than one output signals. For example, many sensors provide a quality control or error messaging signal that may be exported parallel to the primary signal of interest. Such combinations of multiple signals with a single operator are another reason why data type classes for signals and operator classes for methods should be modeled separately.

3.5 Design and implementation

With the framework of component classes and their relationships that has been set up in the previous sections, the steps that make up the component design and implementation activities can now be defined. For a platform component, these include the assembly of the digital signal processor components, the inclusion of the middleware, and the development of the port units. Data acquisition and data processing components share the development of a device unit and the corresponding device module. Analysis modules are the simplest of all instrumentation components; their development is limited to the implementation of the analysis module.

Platforms

Platform components constitute the digital signal processing systems that are at the heart of the flight test instrumentation. A typical platform component covers more than an isolated computer that is used to coordinate data acquisition or to perform data processing. Instead, it is the assembly of a central processing unit† with all interfacial hardware like signal conditioners, analog-digital and digital-analog converters, network adapters, telemetry equipment, and data recorders. A digital flight test instrumentation system has at least one platform component.

Multiple platform components are usually networked. Network links in such a distributed instrumentation system are the sole connections between platforms. Without them, the distributed system falls apart into several single-platform systems. As an alternative to a networked system, two platforms may be joined in a single instrumentation system by connecting to the same peripheral equipment. In this case, there can be no exchange of processing results between the platforms and – although it is still concurrent – the full system cannot be regarded as a distributed processing environment. However, such instrumentation topologies are uncommon.

As depicted in figure 3.8, a platform component is an aggregation of a processing unit, middleware, and an arbitrary number of port units. The processing unit covers all hardware that belongs to the digital signal processor without being related to a specific interface for flight test sensors and actuator. The latter are assigned to the port units, which combine both the interfacial hardware and the corresponding software. Processing units include only hardware; the operating system is combined with the middleware for the flight test instrumentation system in its own class.

Although platform components as a whole are not derived from a parent class – mainly because their key elements are the hardware components that form the digital signal processor – they always follow the life cycle activities that were shown in figure 1.7, where a specialized component is created on the basis of a generalized one. The reason for this is the position of the middleware. Middleware is common to all the platforms in the instrumentation system, yet its implementation must be adapted to each of the individual platforms. As was explained in section 1.5, this is achieved by developing the generalized middleware in its own life cycle and deriving the platform-dependent specializations from it. All characteristics of the middleware towards the software – hence its interface – should be implemented or at least defined in the generalized version. This ensures the portability of modules and device units amongst different platforms. The derived middleware classes only contain the platform-specific parts of the implementation. Typical examples of this are the way numerical data is stored and exchanged with other platforms – hence what floating-point accuracy and

† In case a platform component is based on a symmetric multiprocessing environment (see page 31), the platform has more than one CPU. Distributed systems however are considered as a collection of platform components, each covering a single processor or symmetric multiprocessing environment.

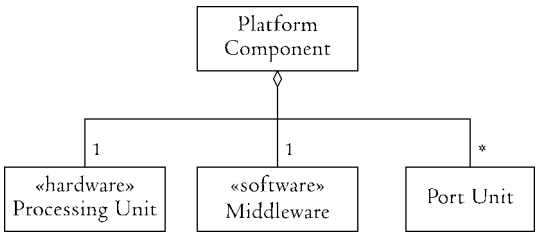


Figure 3.8: Aggregation of a platform component.
The hardware of the digital signal processor is complemented by the middleware that facilitates the communication with other platforms and the application of standardized software components. DSP-specific interfaces that connect the platform to the instrumentation sensors and actuators are added as port units.

which format are used by the platform, or whether numbers are stored as big endian or little endian† – the use of a real-time clock to synchronize the system's behavior, the way interrupts are implemented‡, and the way the network can be accessed to exchange information with adjacent platform components.

Device units

Similar to a platform component, a device unit is dominated by its hardware elements. True inheritance during which the component is derived from a parent class is therefore impossible. The design of a device unit starts with the creation of dynamic models that define the behavior of the device and its service in the form of a statechart and/or a sequence diagram. The next step is the selection of the sensor or actuator that meets the requirements that were specified during component analysis and the previously defined dynamic behavior. Finally, its physical installation in the aircraft and the necessary wiring are designed. As was stated before, it is desirable to set up the designs in a way that reflects the separation between the function of the device – the properties that do not change when a different device is selected – and its implementation.

† The terms big endian and little endian refer to the way data words are distributed over multiple bytes in the computer's memory. Systems that are big endian store the most-significant byte first; little-endian systems start with the least-significant byte. 'Endianness' has been subject to an ongoing debate amongst CPU manufacturers for decades. Only recently, a consensus seems to develop in favor of little endianness. The term 'endian' itself can be traced to *Gulliver's Travels* (Swift, 1726), in which a similar dispute is described on the correct way to open eggs: at the larger end, or at the smaller.

‡ Interrupts are the prime mechanism by which events can lead to the preemption of a thread in a concurrent system. Recall that such a system is a prerequisite for the implementation of a real-time application (see page 33).

Similarities between platform and device unit design and implementation not only exist for the hardware elements, but also for the software part. Like the middleware for a platform component is to be derived from an abstract middleware for the whole instrumentation system, the service that is part of the device unit must be derived from an abstract service that corresponds to the type of sensor or actuator. When a new type of device is implemented, abstract classes have to be designed and programmed for all the levels of abstraction that were defined during the analysis of the component. This is the generalized component implementation activity that is shown in figure 1.7. The instantiated child class for the service in the actual device unit can then be derived from these classes. If an alternative device is subsequently implemented, the generalized implementation is already available and the development can jump directly from component design to component specialization and unit testing. For new devices that are similar to existing ones, one or more of the abstractions will be shared. In these cases, the generalized implementation activity cannot be skipped completely, but will be reduced to the abstraction levels where the devices divert.

Modules

Design and implementation of a device module do not differ from those of an analysis module. In both cases, a software component is developed from a template module that is provided by the instrumentation system's middleware. The template module – indicated as the abstract module class in figure 3.7 – does not implement any method itself, but provides the definition of the interfaces between the module and its environment. Device and analysis methods are derived from the template by adding implementations for the operations that the module must perform.

Whether the instantiated module is derived directly from the template, or whether intermediate abstractions are implemented first depends completely on the individual application. Devices that are part of a group of similar components could have a module of which the tree of ancestors reflects the classification structure that is used for the device's service. Device modules for a less common type of device could be derived directly from the template module. The same applies to analysis modules. Depending on the fact whether the data processing algorithm is a variation to a similar algorithm, the module can be derived from an intermediate class or directly from the module template. In all cases, the decision on module inheritance is one that should be made during module analysis. Design and implementation should only deal with the detailed development of the algorithms themselves, their dynamic behavior in the real-time system, their implementation, and testing.

A common characteristic of all module designs is the creation of a dynamic model. Sequence diagrams are highly suitable to present the signals that the module depends upon and the steps that are taken by the module to complete its computations. For more complicated modules that depend on multiple signals with different arrival patterns, an additional statechart diagram will prove desirable to distinguish between the modes that the module can enter.

4
A Middleware Pattern

Middleware for a digital signal processing system must provide additional functionality with respect to generic middleware standards: It must include a scheduler that actively manages the states of the application modules, and it must provide registration services that connect signal producers and consumers. This chapter presents a new exemplary architectural middleware design that has been optimized for signal processing applications.

The concept of unconfined threads is introduced to model semi-active modules in a concurrent system. Each module can set a schedule interval that defines the window in which it demands activation. Upon completion of its computation, the module can freely choose the new schedule interval. Unconfined threads generalize the thread characteristics in traditional scheduling theory, including periodic, aperiodic, and sporadic threads. In combination with an earliest-deadline-first scheduling policy that is based on beginning-of-completion deadlines and the concept of process time – which is shown to be a stable equivalent of least-laxity-first scheduling – unconfined threads remove the need for additional activation mechanisms in the middleware and facilitate graceful mode changes of the modules.

MIDDLEWARE builds the core of a concurrent flight test instrumentation system. It connects the platform components, which are at the heart of the hardware part of the system, to the software components that take care of the actual data processing. The middleware enables the object-oriented development and maintenance of the applications; it provides essential information-hiding facilities to both the hardware and the software side. Some of these have already been identified in the previous chapters, including a messaging service between the processes in a distributed system, synchronization of clocks on distributed nodes, and a virtual platform that allows software to be executed on any processor.

The use of a middleware to enable distributed applications is not unique to flight test instrumentation. Several middleware standards have emerged for application in industrial and embedded computing. For aerospace applications, the most prominent of these is the Common Object Request Broker Architecture (CORBA). CORBA (Mowbray and Ruh, 1997) focusses on enabling object-oriented software applications on heterogeneous distributed systems. It provides a mechanism for defining the interface of an object in a format that is independent of the programming language, and for connecting distributed client and server objects through proxies that hide location and implementation details. Client and server objects interact with their respective proxies in exactly the same way as they would with any locally implemented object. CORBA would therefore meet one of the requirements for a middleware for concurrent flight test instrumentation systems: It provides transparent communication between distributed objects on various platform types. However, both CORBA and other industrial standard middleware architectures do not address the more specific requirements for an instrumentation system middleware. Neither clock synchronization, nor a scheduling mechanism that is optimized for real-time signal processing applications is inherently supported; an instrumentation system middleware on the contrary should be aware of the temporal requirements for the data that are exchanged.

Flight test instrumentation systems are usually constructed around dedicated digital signal processors that find little application outside a testing environment. For these specific platforms, industrial standard middleware architectures are generally not available. In-house development, or at least the acquisition of a proprietary middle-

ware product is therefore hardly avoidable. Although CORBA and many other architectures are open standards that can be implemented on any dedicated instrumentation platform, the need to create a customized middleware is sufficient reason to abandon industrial standard architectures in favor of a middleware that is optimally adapted to the requirements of a flight test instrumentation system.

The complexity of the middleware and its crucial role in the instrumentation system is the reason for the existence of its own development life cycle. Instead of being integrated with the development of the entire application, middleware should be seen as one of the components that are customized and integrated into the flight test instrumentation system. Like any other off-the-shelf component, the middleware has a life cycle that is external to the development of the instrumentation system.

It is impossible to prescribe the use of a certain type of processor or a certain class of sensors for implementation in all possible flight test instrumentation systems. Similarly, it would be impossible to create a single middleware that can serve the needs of all imaginable instrumentation systems. For this reason, a sample middleware is presented that adheres to the design philosophy that is unfolded in this thesis. It addresses the characteristics and peculiarities of a middleware that must be implemented in a concurrent flight test instrumentation system, without claiming to provide a definitive solution.

4.1 Requirements

Like any other life cycle, the development life cycle for middleware that is shown in figure 1.8 (page 43), starts with an analysis phase in which the requirements are specified. However, no strategy that suits the requirements analysis of middleware has yet been identified in chapter 1. The traditional software engineering strategy of use case modeling relies on the possibility to identify actors that initiate scenarios in which all of the system's tasks are described. In the context of the middleware's responsibility to facilitate the interaction between distributed objects, the client and the server objects can be viewed as actors and use case modeling can be a practical strategy. However, it is impossible to identify actors for many other middleware tasks and use case modeling cannot be used to obtain a comprehensive set of requirements for flight test instrumentation middleware.

Although the requirements for instrumentation middleware are very diverse, their specification can be driven by a single formulation of the middleware's objective: The middleware aims to facilitate the combination and operation of data acquisition and processing software modules in a distributed step-time or real-time application. It is straightforward to analyze this objective and separate it into the various functions of the middleware; a formulation of the functions naturally leads to a specification of the middleware tasks and hence to its requirements.

Instrumentation software modules

The objective of the middleware is to meet the demands of the software modules in the instrumentation system. In order to specify the functions of the middleware accurately, it is necessary to investigate these demands in more detail. The analysis starts at the concept of a data acquisition or processing component in a digital signal processing system and the concept of the software module that is a part of such a component.

A data acquisition or processing component is an instrumentation entity that produces one or more output signals at a certain instant of time – referred to as an *epoch* – based on its own state and the value of zero or more input signals at the same epoch. This definition covers all types of data components. It does not restrict the type of signals in any way; the definition applies both to signals that are internal and to those that are external to the instrumentation system. Because of this, every component must provide at least one output signal. A component that does not, has no meaning for its environment. A component makes itself visible through the signals it produces. If a component can neither internally nor externally be observed, it effectively does not exist. Components that produce external signals are part of the system's outgoing interface; these components are the output components that were described in the previous chapter. Components that produce internal signals on the basis of an external signal are part of the system's incoming interface. These components are the input components; they form the data acquisition part of the instrumentation system. Components for which both the incoming and outgoing signals are internal, are the analysis components from the previous chapter. These are the most common component types in an instrumentation system. Components for which both the incoming and outgoing signals are external do not interact with the rest of the instrumentation system; they can be regarded as stand-alone devices. Components without an input signal are *signal generators*. If the output signal is external, they are typically the signal generators that are found in an excitation chain. Internal signal generators can be used to provide any kind of auxiliary signal that is used in data processing.

A module is a software entity that forms the top-level element of a data acquisition or data processing component. It is the interface of the component to the rest of the instrumentation system. As such, it interacts with the middleware to handle the signals that are internal to the system. With the definition of a data acquisition or processing component, the concept of a module can be defined by its appearance to the middleware: A module is a software entity that produces zero or more internal output signals at a certain epoch, based on its own state and the value of zero or more internal input signals at the same epoch.

It is important to recognize that the application of this definition is by no means restricted to modules in a flight test instrumentation system. The definition also suits the software entities in a simulation. From simple offline simulations to more complicated hardware-in-the-loop and man-in-the-loop simulations, all such digital signal processing systems can be based on software modules that exchange data. For example, the concept of a module can be used in all the research and development resources for

aircraft handling qualities assessment that were shown in figure 3 (page 12). However, simulations and flight test applications share more characteristics than only the definition of a module. From a system development point-of-view, man-in-the-loop simulation, in-flight simulation, and flight test have more in common than offline and online simulation. In-flight simulation combines all the characteristics of a simulation with all those of a flight test; a boundary between the two types of application therefore cannot be defined. At least when a simulation is performed with a human subject in the loop, but usually also for a hardware-in-the-loop simulation, the performance of the modules is time critical. Like flight test instrumentation systems, online simulators are therefore real-time applications†. All these systems are different manifestations of a continuous range of applications that starts with offline simulation and ends with flight test. This is an important inference for the development of a middleware that must allow the reuse of system components through the full range of applications.

Applications as module ensembles

The type of a particular application is completely determined by the modules that make up the application. Ground-based simulation, in-flight simulation, and flight test differ by the specific hardware components that are implemented. To the middleware, each application is nothing more than an ensemble of modules. Its structure follows directly from the dependences between the modules; the way output signals from each module are used by other modules establishes both the connection between all modules and a causal sequence in which the processing in the system can take place. Each module can pose requirements for its deadlines for processing. Apart from such upper bounds to the computation interval, certain modules might put a lower bound on the computation window as well. This will typically be the case for data acquisition modules that cannot be activated before the real time at which the data must be acquired has arrived. The ensemble of modules thus determines not only the structure of the system, but also its temporal behavior.

Because the ensemble of modules implies both the static and dynamic characteristics of the application, there is no need for the middleware to obtain an explicit specification of the system's construction or temporal behavior. Instead, it should support the autonomous interaction of the modules. Interaction between the modules takes place by means of the system's internal signals. An important facility that the middleware must provide, is therefore a mechanism by which the modules can exchange their time-dependent signal data. The mechanism must ensure that the data from multiple signals is properly synchronized. By comparing the availability of a module's input signals for a certain epoch with the module's requirements for those signals, a mechanism can then be created to activate modules automatically in the correct causal order.

† The step from time criticality to real-timeness is enabled by the definition of the latter, which states that a time bound must exist for the computation of results without specifying the bound itself in any way (see page 3).

When the activities of a module are viewed as the tasks of a concurrent thread, the activation of a module by the arrival of all input signals corresponds to the occurrence of an internal event that triggers the transition from a blocked state to the state of awaiting scheduling as shown in figure 1.6 (page 35). Synchronization of multiple input signals that jointly block a module and the release of the module from its waiting state are the first key functions of the middleware.

Another state transition that is indicated in the figure is also the responsibility of the middleware. The unblocking of the state in which the module waits for its next computation epoch is triggered by the real-time clock; the required timekeeping is the responsibility of the middleware. The availability of real time, which must be accurately synchronized over all nodes in a distributed application, is the second key function of the middleware. Finally, figure 1.6 reveals the third key function of the middleware as well. The transition from the state in which the module awaits scheduling to its executing state is triggered by a scheduler. The middleware must provide the appropriate scheduler that fulfills the temporal requirements for the application.

These three key functions of the middleware for a digital signal processing system are different from those of standard middleware implementations for distributed systems. In a signal processing system, scheduling of a module and its release from a blocked state by the arrival of input data or the occurrence of the prescribed time lapse are tightly coupled. Although an integrated environment that meets all the requirements of such a system is therefore desirable, only the first of the key functions roughly resembles the functionality that is provided by existing middleware architectures: the connection of concurrent threads and the exchange of data between them. Timekeeping and scheduling facilities are not provided by standard middleware at all; separate clock synchronizers and schedulers are used instead. Moreover, the brokerage technologies that are used in traditional systems are neither suitable to the typical information histories that are exchanged in a signal processing system, nor do they allow synchronization of multiple data streams. *Remote procedure calls* are the oldest middleware technique. They were introduced by White (1976) and further developed by Nelson (1981). Like CORBA (Mowbray and Ruh, 1997) – or any other object request broker, which can be seen as its object-oriented successor – remote procedure calls use a strict client-server relationship. When data is required, its user makes a request with the provider through a procedure call; the middleware takes care of passing on the request to a remote node and returning the results. Until the request is completed, the user of the data is blocked. This technique is incompatible with the principle of operation of a real-time signal processing system, in which the repetition of such blocking backward dependences leads to unpredictable temporal behavior. In addition, remote procedure calls severely burden providers of data with many users: The provider is called upon by each of the users, resulting in repeated requests for – and possibly in repeated computation of – the same data. Finally, providers and users of data are not anonymous, but interact directly with each other. This complicates the flexible configuration of an application.

During the 1990s, a new type of middleware arose in which data exchange more closely resembles the way signals are exchanged in a signal processing system. This technique is referred to as *message-oriented middleware* (Banavar et al, 1999). Message oriented middleware introduces the concepts of a *publisher*, a *subscriber*, and a *channel*. Publishers are the components that produce data; the data is made available to other components by publication in a channel. Such an information element is referred to as a *message*. Users of data subscribe to the appropriate channel and are notified of any message that arrives. Publication and subscription are completely anonymous; the middleware provides a message cue and handles the notification of subscribers. Paassen, Stroosma, and Delatour (2000) developed a message-oriented middleware for use in a real-time simulation system. The middleware, named Delft University Environment for Communication and Activation (DUECA), contains a publish and subscribe mechanism and includes facilities for message passing between processes and synchronization of data. Additionally, DUECA provides a mechanism for activating a process when all input channels of a data-dependent module contain valid messages, or when the new computation interval for a periodic activity has arrived. The corresponding module state transitions are controlled by event generators that are referred to as triggers. This way, DUECA acknowledges the importance to integrate activation and scheduling facilities with data exchange facilities in a single middleware.

Middleware functions

The primary requirements for the middleware for a distributed digital signal processing system follow from the functionality that the middleware must provide to the modules in the application. They can be grouped around the three key functions that were identified in the previous subsection.

- The middleware must provide a data management mechanism. It must include an anonymous publish/subscribe mechanism that allows data producers and data consumers to exchange the information of discrete signals through channels without knowledge of the other modules. The middleware must take care of synchronization of multiple channels that are used as the inputs of a module and must ensure that valid data is available in each channel when a subscribing module is activated. Additionally, the middleware must construct the application by connecting signal publishers and subscribers without using any additional information.
- The middleware must manage the states of the modules in the application. It must distinguish between the states in which a module awaits a time epoch or the availability of the channels that it subscribes to for a preset time interval; it must also recognize the state in which a module awaits scheduling after it has unblocked from the wait states. The middleware must initiate the state transitions from the two blocked states to the state of being ready for scheduling. In doing so, it must acknowledge a module-specified time interval that defines the window for which input data is

required and during which the module request scheduling. Furthermore, the middleware must provide a scheduler/dispatcher that activates modules which have unblocked. The scheduler must operate in a way that optimally acknowledges the real-time requirements for the modules.

- The middleware must provide a real-time clock. The clock time must be available to all modules and must be accurately synchronized with real time. The latter should ensure that the clock can be used for performing computations in which time plays a crucial role, like mathematical integration or differentiation with respect to time.

These functions must be fully transparent to all the modules in a distributed application. As a result, the channel management mechanism, the state manager and scheduler, and the clock synchronization mechanism must extend over all the nodes of the system. Finally, the middleware must be available for any digital signal processing system that is used in flight test instrumentation systems, real-time simulators, or any related equipment. The middleware therefore must not depend on specific hardware or operating system facilities like a multiprogramming environment.

4.2 Time

In the context of a middleware for real-time applications, detailed analysis of real-timeness and the underlying concepts of time, timescales, and clocks is required. In a signal processing system, time has two equally important functions. First, it provides a measure to chronologically order the events that occur anywhere in the system. This function is not unique to signal processing; it applies to all information processing applications. Second, time is an important parameter to the processing of dynamic signals, for example in differentiation or integration operations. The importance of this function is more or less restricted to signal processing systems. General information processing systems might depend on the ordering of events, but the close relationship with real time is a specific characteristic of a signal.

It is tempting to consider the first time-related responsibility of a middleware, the ordering of events, as a side effect of the second responsibility, the provision of real time. When real time is available to all modules in the system, each event can be time-stamped and a comparison of time stamps allows for a unique ordering of all events. However, in a distributed system, where communication delays between processes are not negligible, the availability of synchronized real time is not straightforward. Small offsets in the clocks of distributed nodes might lead to an incorrect ordering of events and thus, in case certain computations depend on the order of events, to a system failure. It is important to recognize in this respect that many causally related events in a system – where one event triggers the occurrence of the next – occur within microseconds. It is virtually impossible to guarantee clock synchronization with an accuracy that ensures the correct ordering of such events.

Nevertheless, both time-related responsibilities of the middleware are closely related. If both functions are disconnected, it is possible that the logical order of two events differs from the numerical order according to their time stamps. In case the interval between the events is used in further computations, this would introduce a sign error in the time. It is therefore essential that the capability to order events and that to provide a measure of real time are integrated in a single mechanism that explicitly addresses the requirements for both. Thus, the middleware must provide a synchronized real-time clock that includes a mechanism which guarantees the correct ordering of events, even under arbitrarily small clock offsets amongst the nodes in the distributed system.

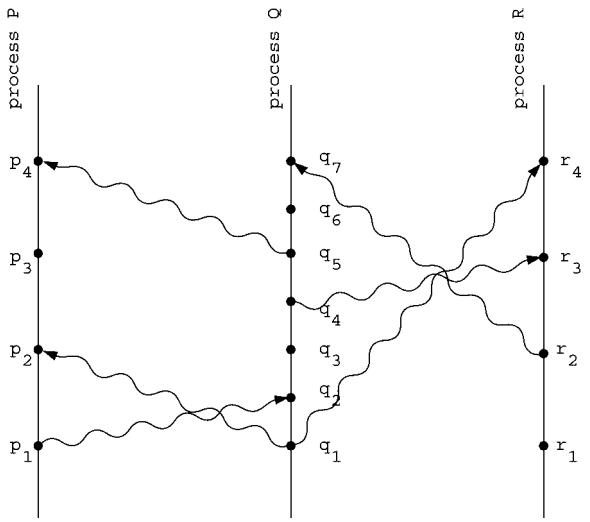
Ordering of events, concurrency, and correct clocks

Lamport (1978) discusses the topic of ordering events in great detail and elaborates the relationship between event ordering and the notion of time. He introduces a partial ordering of events by means of the "happened before" relation á. The relation a á b means that it is possible for event a to causally affect event b. The relation obeys three simple rules. First, if two events a and b occur in the same process and a comes before b, then a á b. Second, if a and b are the sending of a message by one process and the receipt of that message by another process, then a á b. Third, if a á b and b á c, then a á c. Using the "happened before" relation, Lamport defines concurrency as a property of two distinct events a and b for which a à b and b à a; two events are concurrent if neither can causally affect the other. The "happened before" relation is illustrated in figure 4.1. Using the definition, the events p_3 and q_3 are found to be concurrent, since there is no sequence of events that can let one event causally affect the other. Lamport emphasizes that concurrency follows from the ordering of events using messages that are actually sent, not those that could have been sent. In the latter case, figure 4.1a would suggest that q_3 happened before p_3. However, Lamport introduces the clock C_i which assigns a number $C_i\langle a\rangle$ to the event a in the process P_i; the entire system of clocks C assigns the value $C\langle b\rangle$ to any event b, with $C\langle b\rangle = C_j\langle b\rangle$ if b is an event in process P_j. The previously mentioned condition that the ordering of events must resemble the ordering of the times that are assigned to these events, is subsequently formalized (Lamport, 1978, p. 560):

Clock condition. For any events a, b: if a á b, then $C\langle a\rangle < C\langle b\rangle$.

Figure 4.1b shows the clock ticks – the discrete transitions from one clock value to the next – for a clock that satisfies this condition in the same example sequence of events. This view suggests the converse of the previous view: p_3 apparently happened before q_3. This illustrates that a clock system which guarantees the correct ordering of all observable event sequences – subsequent events in a single process and the sending and receipt of a message between processes – provides only a partial ordering. As Lamport explains, a total ordering that also allows to determine the correct order of p_3 and q_3, can only be obtained by introducing a system of clocks that is not uniquely deter-

mined by the clock condition. As a result, different choices of clock systems yield different solutions. Only the partial ordering of events is determined by observable event sequences and message transmissions; events that are not part of a partial ordering remain concurrent.

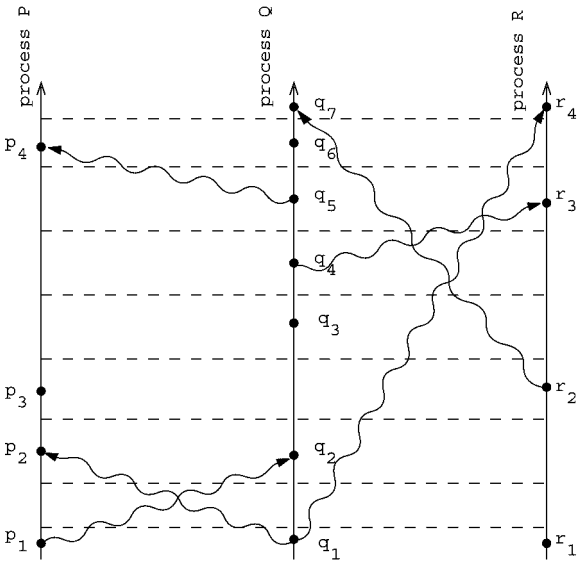


(a) partial ordering without clock

Figure 4.1: Space-time diagrams of events in a distributed system (Lamport, 1978, pp. 559-560).
a. Each vertical line depicts a process; each dot is an event. The vertical direction represents time, with later times being higher than earlier ones. The wavy lines denote messages. *a* á *b* means that it is possible to go from *a* to *b* by moving forward in time along process and message lines. For example, the diagram shows that p_1 á r_4. (continued)

A clock that satisfies the clock condition is referred to as a *correct clock*†. It provides an observable measure for the sequence of events, but it is not necessarily related to physical time. The individual clocks at the nodes of a distributed system can easily be harmonized. A correct system of clocks is obtained when each process increments its clock value in between two events, when each message between processes is stamped

† Lamport (1978) refers to these clocks as *logical clocks*. In the next subsection however, the term logical clock will be applied to clocks that provide a measure of physical time without necessarily satisfying the clock condition. The latter interpretation is also found in more recent literature (Srikanth and Toueg, 1987; Arvind, 1994; Alari and Ciuffoletti, 1997). To avoid confusion, the term correct clock is introduced for a clock that satisfies the clock condition.



(b) alternative view with clock tick lines

Figure 4.1, continued..
b. The dashed horizontal lines indicate ticks of a common clock that satisfies the clock condition. This view, which shows equidistant clock ticks, is equivalent to that in (a), which shows equidistant events in each process. Without the concept of physical time, neither of the two views can be preferred as one that provides a more accurate representation of the actual order of events.

with the clock value of the sending node, and when each process ensures that its clock value is increased – if necessary – to a value larger than the time stamp of a message upon its receipt. This way, the processes of a distributed system are *synchronized*. Synchronization refers to the partial ordering of events across nodes.

As a result of the inability of a correct system of clocks to provide a total ordering of events, any group of events for which knowledge on the correct sequence of occurrence is required must be part of a partial ordering. Thus, the events of such a group must occur in a single process, or synchronization messages must be exchanged between the processes at each relocation of the sequence across the nodes. For example, suppose that two processes communicate by challenge and response. Process A sends a message to process B, which must be acknowledged by a return message. In order to prevent old acknowledgments that might have been delayed in the communication network from being interpreted as a new one, process A might verify the logical time stamp on the acknowledgment to ensure that the response was sent after the

transmission of the challenge message. Such a time stamp can be based on any correct clock and need not be related to physical time. However, if the clock offset between processes A and B is such that the clock in process B is more behind the clock in process A than the transmission delay of the challenge, process B will time-stamp its response with a time prior to that of the challenge. Process A will therefore reject the returning message as a causal impossibility. This is avoided by synchronizing process B with process A upon receipt of the challenge message.

Real time, physical and logical clocks

Although a correct clock as such allows to synchronize activities in a distributed application, it usually does not suffice to support a real-time application. The clock condition only prescribes an increase of the clock value in between any pair of events and possibly at the receipt of a message; the resulting 'time' is by no means related to physical time. However, real-time systems are required to perform within hard temporal limits. Although the limits are not necessarily expressed in terms of physical time either – for example, a system may need to have completed the computations that are triggered by a certain event before a subsequent similar event occurs – most applications must complete selected activities within a predetermined temporal interval with respect to some external reference time. Because the clock condition relates the tick rate of a logical clock to the rate of events in the system, a mere correct clock provides a measure for the progress of the system's computations, but not for its performance if a real-time bound in terms of temporal deadlines is applied.

On the other hand, the use of real time as a reference for the performance of an application does not make the correct clock obsolete. Physical time cannot be observed. If time would be observable†, each process in a distributed system could use physical time to time-stamp its messages; events could be ordered totally without explicit synchronization. It is the unobservability of physical time that causes the need for a clock. Lamport and Melliar-Smith (1985, p. 60) define real time as an assumed Newtonian time frame that is not directly observable, and clock time as the time that is observed on some clock. Thus, a clock is defined as a device that provides an observable measure of time. As was mentioned previously, the clock condition alone does not suffice to create a clock that relates to real time. The tick rate of a real-time clock must additionally be related to the progress of real time by the presence of a *physical clock*. Physical clocks, incidentally referred to as hardware clocks, are devices that include an oscillation mechanism in order to produce periodical ticks that increment a counter. Thus, the counter is a direct measure of real time. The resolution at which time can be observed from a physical clock is the interval in between two increments of the counter. This tick period is referred to as the *granularity* of the clock (Kopetz, 1997). The readout of the physical clock's counter is converted into a time measure by apply-

† At this point, the discussion is limited to Newtonian space-time. Hence, time is assumed a universal property that is invariant to the various hypothetical observers.

ing a scale factor and an offset. The linear function of time on the physical clock counter is referred to as the *logical clock* (Arvind, 1994, p. 475).

In a distributed system, each node has its own physical clock. The corresponding system of logical clocks is not automatically correct. Correctness of a clock that is not linked to a physical clock is most easily achieved by increasing the clock value in between two events and by synchronizing the clock whenever a message is received. Such a procedure however cannot be implemented for a logical clock. The progress of a logical clock is driven by the underlying physical clock, not by rules that depend on events and messages. Therefore, alternative requirements must be specified in order to ensure the correctness of a system of logical clocks†. First, the granularity of each clock must be smaller than the interval between any two events in the corresponding process. This suffices to satisfy the clock condition within a single process. It requires physical clocks to be high-frequency devices that tick at or above the clock frequency of the node's processing unit. Second, the linear time function of a logical clock must be adjusted to ensure proper ordering of message sending and receipt, similar to the synchronization of event-driven correct clocks. This procedure is referred to as *internal clock synchronization*. It extends the concept of synchronization from mere correct clocks to logical clocks, thus ensuring causality between different nodes while maintaining a reference to real time. Apart from ordering events, the resulting synchronized logical clocks can be used to calculate intervals for time-dependent signal operations like integration and differentiation.

Reference timescales

Internal clock synchronization applies small corrections to the time functions of the logical clocks in a distributed system, in order to maintain synchrony between the nodes. Although each of the logical clocks is related to a physical clock, internal clock synchronization does not guarantee that the system of clocks stays synchronized with real time. Neither a common static offset, nor a common rate offset in the logical clocks can be observed. As a result, an internally synchronized system of logical clocks will drift away from real time. In real-time applications where an accurate measure of

† Correctness of a system of clocks depends on the "happened before" relation, which is equally applicable to Newtonian and relativistic space-time. However, the theory of special relativity shows that this relation cannot be related to physical time for events that occur on different locations, because a common reference time does not exist. For nodes with relative velocity greater than zero, observed time from a remote node appears slow with respect to the local time; the time dilation prevents clocks in an arbitrary distributed system from being synchronized. It is impossible to determine a total ordering of events while maintaining the relationship between each logical clock and its local proper time, which is observed through the ticking of the local physical clock. In relativistic space-time, the function of the correct clock – ordering of events – and that of the logical clock – measuring intervals of proper time – must therefore be separated. Relativistic distributed systems in aerospace engineering are not hypothetical: The NAVSTAR Global Positioning System compensates for relativistic effects between its ground, space, and user segments (Ashby and Spilker, 1996).

absolute time or accurate interval measurements are required, the system of clocks must be tied to an independent reference time. This procedure is referred to as *external clock synchronization* (Schmid, 1995, p. 877). All the logical clocks in an externally synchronized system are kept in pace with the reference time. If the deviation between any clock in the system and the reference time is limited, the maximum deviation between any two clocks in the system is limited as well. Thus, externally synchronized systems are always internally synchronized. The converse is not necessarily true.

Most real-time applications, especially signal processing systems like flight test instrumentation systems, must be externally synchronized. A key element in the implementation of such a system is the availability of an observable reference time. It must be provided by a reference clock, which adheres to a predefined timescale. Like logical clocks, timescales are defined in terms of two parameters: a standard interval that determines the progress of time, and an epoch that serves as the origin for the scale. For all timescales that are currently used in science and technology, the former of these parameters is the SI second. It is defined by the adoption of a fixed frequency for the radiation that corresponds to a particular transition of cesium 133 atoms in the absence of a magnetic field (Seidelmann, 1992, p. 40). As a result, the existing scientific timescales differ by definition of their origin epoch only. The prime, worldwide recognized timescale is international atomic time (TAI), maintained by the French Bureau International des Poids et Mesures. TAI is calculated from comparison of about two hundred atomic clocks, including commercial cesium standards, laboratory clocks and national frequency references. International atomic time provides a continuous, or *chronoscopic* timescale. It is defined at a geocentric datum line and should be corrected for general relativity when being extended to any fixed or mobile point near the geoid.

A second important class of timescales is that of sidereal time. It relates a unit of time to the rotation of the Earth with respect to inertial space. Hence, a sidereal day is the time for one complete revolution of the Earth. Because sidereal time is directly related to the actual angle between a point on Earth and the vernal equinox, sidereal time is affected by irregularities in the Earth's rotation. The combination of the Earth's diurnal rotation and its rotation around the Sun results in a difference between the sidereal day and the day as it is observed from the daylight cycle. The difference between both days is one day per year, or almost four minutes per day.

To correct for the irregularities of sidereal time and its incompatibility with a solar day, universal time (UT) is a group of timescales for which a day is defined as the average solar day. It is kept as uniform as possible, despite deviations in the Earth's rotation (Seidelmann, 1992, pp. 50-54). Nevertheless, universal time is directly related to sidereal time in order to be observable from celestial transits. Universal time is used for all civil timekeeping. Because of its astronomical definition, universal time shows slight irregularities with respect to the physical timescale of TAI. This discrepancy is eliminated in coordinated universal time (UTC). UTC is a universal time which uses the SI second as a basis for its rate. The difference between international atomic time and coordinated universal time is by definition an integer number of seconds. By

inserting leap seconds when necessary – approximately once every eighteen months, and usually at the last day of December or June – UTC is kept within 0.9 seconds of sidereal-based universal time. Because of the leap seconds, UTC is a non-chronoscopic timescale. In the past, the term Greenwich Mean Time (GMT) has been used for both universal time and coordinated universal time. Hence, use of this ambiguous term is undesirable.

GPS time, as distributed by the NAVSTAR Global Positioning System satellites, is a chronoscopic timescale that is based on the concept of international atomic time. It adheres to the SI second and does not incorporate the insertion of leap seconds. However, its origin epoch is different from that of TAI. GPS time was set to the same value as coordinated universal time when the GPS timescale was initiated. GPS time is exactly nineteen seconds behind TAI; since both timescales adhere to the same second and are chronoscopic, this difference does not change. Due to the leap seconds that are inserted in coordinated universal time, the offset between GPS time and UTC is growing. TAI and UTC do not specify a standard expression format. A common expression is a monotonic increasing day number and a time in ordinary hours, minutes and seconds, of which only the number of seconds can be fractional. Alternatively, the day number and time can be combined into a single real number: the Julian date. The integer part of the Julian date represents the day since January 1, 4713 B.C.; the fractional part represents the fraction of a day since Greenwich noon. Many applications use the modified Julian date, which is defined as the Julian date minus 2400000.5. A modified Julian day therefore begins at midnight. GPS time is expressed in a week number, starting at zero for the week commencing January 6, 1980, and a week time, which is the fractional number of seconds since midnight between Saturday and Sunday (Parkinson and Spilker, 1996, p. 125).

4.3 Scheduling policies and performance

Unlike other systems, a real-time system has failed to function correctly when it violates its temporal performance requirements. Depending on the type of application, such a breach of one or more operational deadlines may have serious consequences. An in-flight simulation system that misses deadlines in the flight control loop, might turn instable and cause a loss of the aircraft. For many airborne applications, system performance is therefore a safety issue. The performance of a concurrent system is dominated by its scheduling policy. Because of an unsuitable scheduling policy, even a computing system that on average utilizes only a fraction of its capacity, may fail to meet its deadlines if a burst of operations must be completed in a short time span.

There is no scheduling policy that suits all applications equally well. Different types of activities require different scheduling strategies. Additionally, system behavior under transient overload is of paramount importance. Some applications do not require the completion of an activity once its deadline has passed, thus supporting the timely

completion of the remaining computations. Other applications depend on the numerically correct completion of all computations, even if the lateness of one thread leads to a violation of the deadlines of other activities as well. The best scheduling policy for a graceful degradation of such a failing real-time system fully depends on the application and the requirements of its environment.

Performance analysis, during which the schedulability of a collection of threads is verified under the assumption of a certain scheduling policy, is therefore an important responsibility of application development. Although the actual schedulability of an application depends on the combination of the scheduling policy, the available computing power, and the set of activities, the characteristic requirements of a flight test instrumentation system allow for the determination of the most suitable scheduling policy for all signal processing applications.

Thread types

The exact scope of scheduling theory is defined by Ramamritham and Stankovic (1991), who put scheduling in a framework with *allocation* and *dispatching*. Allocation deals with the assignment of the threads and resources in an application to the appropriate nodes and processors; dispatching is the execution of the threads in conformance with the scheduler's decisions. Scheduling itself forms the link between allocations and dispatching. It orders the execution of the threads that have been assigned to a processor or node such that their timing constraints are met and the required resources – for example shared interfaces or memory – are available to the threads that are dispatched. The way the order of dispatching is determined, is what is referred to as the scheduling policy. In a flight test instrumentation system, the threads correspond to the modules that are part of the data acquisition and data processing components. They have been assigned to a platform component during application design. As such, allocation is not a responsibility of the middleware. It is completed off-line, even before the individual components are developed themselves.

Scheduling can be performed off-line as well. Stankovic et al. (1995) describe the differences between off-line and on-line scheduling and emphasize that off-line scheduling does not necessarily result in a static schedule, in which the complete order of dispatching is determined in advance. A scheduling policy that assigns fixed priorities to all threads can be applied beforehand in order to yield a static set of priorities, but not a static schedule. However, off-line scheduling requires a-priori knowledge of all threads in the application and of their exact timing constraints. This information is normally not available. Off-line scheduling should therefore be applied to a worst-case analysis of the application. During operation, the application should use the same scheduling technique online. Stankovic et al. (1995) also address the feasibility of determining an optimal schedule. For a multiprocessing or distributed system, they consider allocation as in integral part of scheduling. Taking into account this complication of the scheduling task, they provide an overview of scheduling possibilities which demonstrates that scheduling in a multiprocessor system is not a generally solv-

able problem. Therefore, heuristic scheduling strategies must be applied in most realistic systems. Some other aspects of practical real-time applications that prevent the determination of an optimal schedule are identified by Klein, Lehoczky, and Rajkumar (1994). They conclude that instead of pursuing an optimal schedule, it is more important to use a policy that is predictable, guarantees acceptably high levels of resource utilization, and addresses practical issues such as operating system overhead, task synchronization, aperiodic events, and transient overload.

The usefulness of a scheduling policy depends not only on the number of nodes or processors, but also on the type of threads. There are three main types: *periodic*, *aperiodic*, and *sporadic threads* (Klein, Lehoczky, and Rajkumar, 1994, p. 24). Periodic threads must be dispatched in a continuous series of regular invocations. If the interval between successive invocations is irregular, the thread is aperiodic or sporadic. Sporadic threads have hard deadlines; in order to make sporadic threads schedulable in the first place, there is a lower bound on the interval between two deadlines in a thread. Aperiodic threads do not have such a lower bound. As a result, timing requirements can only be stated in terms of satisfying an average response time. Sha, Klein, and Goodenough (1991) do not recognize sporadic threads, but introduce another variable to thread periodicity, which they refer to as a *mode change*. Mode changes cover the deletion or addition of a series of threads, or changes in the parameters of a thread such as the dispatch rate. In this thesis, the three types of threads and mode changes are combined into the concept of *unconfined threads*. An unconfined thread is neither bound to a fixed activation rate, nor does it guarantee a priori a minimum interval between activations†. In this respect, the unconfined thread resembles the aperiodic thread. Yet, unconfined threads also include periodic and sporadic threads and can set hard deadlines. In the course of the mission, an unconfined thread determines the interval to the subsequent activation itself; it can adjust its deadlines accordingly. Each of the activations of an unconfined thread is referred to as a *job*. The term unconfined thread refers to the sequence of jobs that correspond to a single software module. Because the behavior of an unconfined thread can be influenced by external events such as measurements, pilot inputs, or operator commands, the concept of unconfined threads prohibits off-line scheduling.

Fixed-priority scheduling

Rate-monotonic analysis is the most widespread scheduling policy for real-time systems (Sha, Klein, and Goodenough, 1991; Lehoczky et al., 1991; Klein, Lehoczky, and Rajkumar, 1994). The main reasons for this are its comparable simplicity as a policy in which each thread is assigned a fixed priority, its optimality among all other fixed-pri-

† The absence of a lower bound on the time between successive deadlines for an unconfined thread does not imply that unconfined threads can meet all their deadlines for any sequence of arbitrarily small intervals. It is merely a requirement to the scheduling policy, which should producing an optimal schedule without expecting a lower limit of two successive activations of a thread, even if the schedule violates the thread's deadlines.

ority strategies, and its stability. The latter means that even during transient overloads, the deadlines of the highest-priority threads are still met. In addition, the efficiency of RMA for most applications is considerably better than the theoretical utilization bound that was derived by Liu and Layland (1973). Instead of the theoretical bound of $\ln 2$ that was already mentioned in section 1.3 (page 36), Lehoczky, Sha, and Ding (1989, p. 171) showed that 88% is a good approximation of the practical schedulability threshold for a large number of tasks. Rate-monotonic analysis is also indirectly suggested by Paassen, Pronk, and Delatour (2000) as a formal test to verify schedulability of the DUECA middleware.

Disadvantages of rate-monotonic analysis are its limitation to periodic threads and the requirement to know the exact computation time of each thread – or a worst-case upper bound – in addition to its period. A solution to the latter issue was presented by Park, Natarajan, and Kanevsky (1993), but it applies only to off-line scheduling. The former issue was already addressed by Liu and Layland (1973): A system that uses rate-monotonic analysis for scheduling of the periodic threads, can perform aperiodic jobs in the idle time in between the periodic computations. Care must be taken that the aperiodic threads do not jeopardize the deadlines of the periodic threads, or that the periodic threads with high priorities cause an unacceptably poor response time for the aperiodic computations (Lehoczky et al., 1991). The preferred solution to this problem is the sporadic server algorithm (Sprunt, 1990); it is based on the inclusion of one or more periodic threads which implement a computation time server for aperiodic jobs at different priority levels. However, the sporadic server algorithm does not include a mechanism to guarantee that sporadic threads actually meet their deadlines; it merely leads to an improvement of the average response time of aperiodic and sporadic jobs when compared to a system where these threads are ordered behind all periodic computations (Sha and Goodenough, 1990). Therefore, the sporadic server algorithm with rate-monotonic analysis is unsuitable for scheduling in a system that relies heavily on unconfined threads, because it does not address their deadlines.

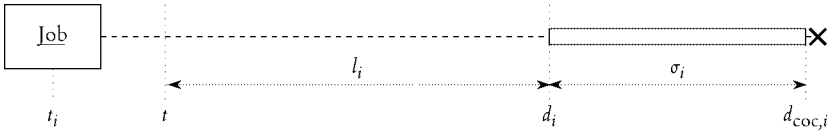
Variable-priority scheduling

Despite the advantages of fixed-priority scheduling as obtained from rate-monotonic analysis, a policy which assigns variable priorities is required to fully utilize the computational capacity of the system. Liu and Layland (1973) showed that the optimum schedule is obtained when fixed priorities for periodic activities are assigned according to the earliest-deadline-first criterion. A similar policy for aperiodic activities that arrive synchronized – hence, that are all known at the time the schedule is determined – has been formulated by Jackson (1955). Jackson's rule, which is referred to as the earliest-due-date (EDD) criterion, applies to any sort of activity. Horn (1974) extended Jackson's rule and the work by Liu and Layland to yield the EDF criterion for jobs that are aperiodic and arrive asynchronous. Dertouzos (1974) showed that it is possible to implement the EDF algorithm in an on-line scheduler on a single-processor machine, under the assumption that the scheduler can preempt all processes.

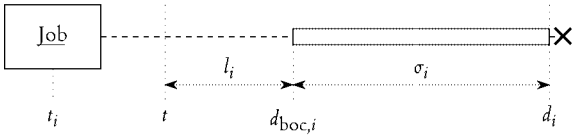
The *least-laxity-first* scheduling policy (LLF) is presented by Mok and Dertouzos (1978) as an alternative to earliest-deadline-first scheduling. Like EDF, least laxity first is an optimal scheduling policy (Liu, Liu, and Liestman, 1982). The laxity of an activity is the difference between the remaining interval until its deadline, and its computation time; it is the maximum time that the execution of the activity can be postponed without violating the deadline. Laxity it is also referred to as *slack time* (Mok, 1983, p. 38). The least-laxity-first scheduling policy assigns the highest priority to the thread with the smallest laxity. This way, it allocates processing time to the thread that is closest to missing its deadline. Because it directly acknowledges the prime criterion that no deadlines are violated, the least-laxity-first policy may appear to be more robust than earliest deadline first. However, both policies are optimal and both produce a schedule that meets all deadlines if such a schedule exists. Differences between the performance of EDF and LLF are found during system overloads. An earliest-deadline-first policy always allocates resources to the thread with the nearest deadline, even if the remaining time is not sufficient for the thread to complete its activities. In least-laxity-first scheduling, this situation is indicated by a negative laxity. Knowledge of threads with negative laxities may be used to drop those threads from the schedule, in order to improve the performance of the remaining threads. Least-laxity-first is therefore the preferred policy if the computation time for each activity is known a priori to the scheduler, and if threads with negative laxity can be removed from the schedule. In all other applications, earliest-deadline-first scheduling is preferred. It does not require knowledge of computation times and generally involves less operating system overhead. Deadlines are fixed; an EDF schedule needs to be preempted only when a new job arrives with a deadline that is nearer than that of the thread which momentarily executes. Laxities however are changing; an LLF schedule will result in excessive context switching when two or more threads have similar laxities.

Least-laxity-first scheduling is therefore of limited importance to applications where deadlines are set for the completion of the thread activities. Kurose, Towsley, and Krishna (1991) however view laxities and deadlines from a different perspective; their approach uncovers the applicability of the least-laxity-first policy to the scheduling of a special set of activities. Kurose, Towsley, and Krishna define t_i as the time at which job i arrives at the system; d_i denotes the corresponding deadline. The computation time for the i-th job is s_i. They then distinguish between two types of systems according to the meaning of the deadlines. For the first type, the deadline is the time at which the computations for the job must commence. For the second type, the deadline specifies the time at which the computations must have completed. In case of a system with beginning-of-computation deadlines, a scheduling policy that always schedules the job with the smallest deadline is the least-laxity-first scheduler; in case of a system with completion-of-computation deadlines, such a scheduling policy is the earliest-deadline-first scheduler. This is illustrated in figure 4.2. Although a least-laxity-first scheduler is undesirable for most applications with the second type of deadlines, it is particularly useful for those where beginning-of-computation deadlines

apply. In a signal processing system, beginning of computation deadlines can typically be defined for data acquisition threads. Each job in such a thread usually represents the acquisition of one sample. The first task in each job is the activation of the interfacial hardware. During signal acquisition and conversion, an unknown time interval can pass. When the raw data is available, the job verifies the result, possibly performs some conversions, and publishes the agreed data. Due to the unknown interval between activation of the hardware and the receipt of the acquired data, the computation time for the thread is unknown. It is also unimportant: To guarantee the time correctness of the acquired data, a meticulous window must be defined for the time that the acquisition hardware is activated. This time relates to the start time for the job, not to its completion. A schedule window is therefore formed by the arrival time of the job and a deadline for the beginning of computation.



(a) beginning-of-computation deadline



(b) completion-of-computation deadline

Figure 4.2: Types of deadlines.
a. The deadline d_i specifies the latest allowed beginning of computation. The completion-of-computation deadline $d_{\mathrm{coc},i}$ is unknown. The laxity l_i equals the distance between the current time t and the deadline.
b. The deadline d_i specifies the latest allowed completion of computation. The computation time s_i is required to calculate the beginning-of-computation deadline $d_{\mathrm{boc},i}$, which yields the laxity as the distance to the current time.

The arrival time t_i as indicated in figure 4.2 marks the transition of a thread into the state in which it awaits scheduling as the creation of a schedulable job object. To the scheduler, the job does not exist before t_i; after t_i, it can be dispatched any time that suits the schedule. Alternatively, the job can be stamped with an additional time r_i which denotes the *release time*. The execution of a job cannot start before the release time. Stankovic et al. (1995) model the arrival time of each thread as a release time in

order to extend the problem of scheduling synchronous threads to that of scheduling asynchronous threads. Although it does not play any role in the feasibility of a schedule or in the selection of an appropriate policy once a system is recognized to have asynchronous activities, the notion of the release time has practical advantages in the modeling of unconfined threads. Instead of having to register schedulable jobs with the scheduler at the exact epoch from which execution is allowed, the preceding job in an unconfined thread can provide a release time and a deadline well in advance. Especially for data acquisition threads with a narrow window between the release time and the beginning-of-computation deadline, this is an important property.

Recognizing the two types of deadlines as defined by Kurose, Towsley, and Krishna, the policies for earliest-deadline-first scheduling and least-laxity-first scheduling are the same. The possibility to accomplish an online EDF scheduler in a single-processor system with full preemption therefore also exists for an LLF scheduler. However, Mok (1983, pp. 41-42) proved that this theorem cannot be generalized. In a system with multiple processors that cannot be controlled by a single scheduler, or in a system where processes make use of binary semaphores to protect mutually exclusive sections of their computations, an optimal schedule can only be determined by a *clairvoyant* scheduler: a scheduler that has knowledge of all future scheduling demands of all processes. Thus, in a distributed system with pre-assigned nodes for each process and with unconfined threads, optimal scheduling is not achievable. This supports the observation by Klein, Lehoczky, and Rajkumar (1994) that the development of a scheduling algorithm should aim for predictability and robustness, rather than optimality.

4.4 Architecture

The requirements specification for the signal processing middleware and the detailed analysis of real-time clocks, synchronization techniques, and scheduling policies, provide the basis for the middleware design. Figure 4.3 shows the architectural design for the exemplary middleware that is presented in this thesis. Its components can be divided into two groups: the application-independent core elements, and abstract classes for application-dependent signal and operator components. The logical clock, the pacer, the scheduler, and the registry form the group of core elements. Each of these is instantiated exactly once in every node of the distributed system. The second group consists of the base classes for signal, module, and pipe components. The number of instances of the classes that are derived from these generalizations differs from application to application.

As was discussed in the previous chapters, the middleware development life cycle produces a generalized version of the middleware that will be used in the development of platform components for the instrumentation system. During the specialization for a new platform, the core elements are adapted to the specifics of the underlying hardware. As long as the same platform is being used, the resulting core element instances

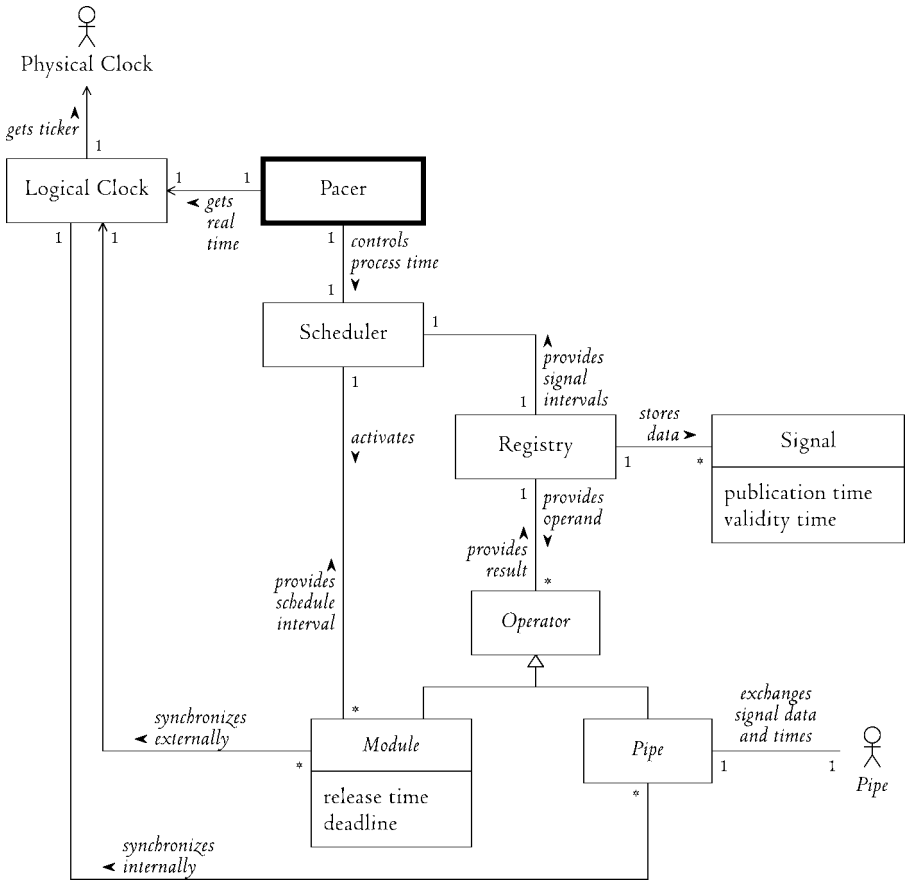


Figure 4.3: Classes in the architectural middleware design.
The main components of the middleware are the pacer, the scheduler, and the registry. Supported by the logical clock, the pacer controls the progress of time for all of the data processes. As such, control resides primarily with the pacer. The scheduler and the registry are responsible for the message-oriented transitions of module states and the activation of modules that await scheduling. The operators, which generalize modules and pipes, are application dependent. Pipes behave like modules, but are actually proxies for remote nodes in the distributed application. Pipes and modules with access to an external reference time handle internal and external clock synchronization respectively.

do not change from one instrumentation system to the next. This is different for the second group of middleware components. The abstract base classes for signals, modules, and pipes may be adapted to the platform on which they will be used, but the specializations will not be instantiated. Instead, the derived classes will serve as a tem-

plate for the application-specific components that are created by the flight test instrumentation engineer during component development. This means that during application development, the signal and operator classes are customized in two phases – specialization of the generalized middleware components to the hardware specifics of the platform component, and specialization of the platform-specific class templates to the application-dependent system components – whereas the core elements undergo only one specialization.

Although the application-specific components are derived from template classes that have been customized for a single type of platform, the portability of modules over different platforms is not affected by this dependence. It is the inheritance mechanism between the parent and child classes which ensures that the application-dependent components are adapted to the hardware platform on which they are deployed. The process of deriving the application components from the platform-dependent base classes itself does not depend in any way on the platform-dependent classes. As was stated on page 87 in section 3.5, the specialized middleware must only change the implementation of the middleware's functionality in order to support the platform; it must not change or add anything to the middleware's appearance towards the modules. Application components that have been developed for any type of platform, can therefore be ported to any other platform by simply joining them with the appropriate specialized middleware.

Core components

Among the middleware's core components that are shown in figure 4.3, the logical clock and the pacer are responsible for the temporal behavior of the application. The registry focuses on the message-oriented behavior. The connection between the temporal and the message-oriented domain is formed by the scheduler. As such, the scheduler occupies a central position in the operation of the application. It is the prime interface of the middleware to the application modules and reflects almost all the middleware's functionality for the modules. The other components can therefore be seen as supporting elements to the scheduler.

Each platform that supports real-time applications must have a logical clock. The primary function of the logical clock is to provide a continuous estimate of real time to the pacer. This is a simple assignment; it might be questioned whether it is justified to identify the logical clock as a separate component in the middleware, or whether it should be integrated with the pacer. However, an additional responsibility of the logical clock is synchronization of the various local clocks in a distributed system, both mutually and with real time. Although this function is secondary to the provision of time to the pacer, the associated tasks are considerably more complicated. The logical clock provides two interfaces through which internal and external synchronization can be accessed respectively. The first of these is used by the pipe components that handle the communication between the nodes in a distributed application. Through the pipes, the logical clocks of all nodes synchronize internally. Modules with

access to an external reference time can use the second synchronization interface of the logical clock to request an adjustment of the clock in order to maintain synchrony with real time.

The pacer controls the actual progress of time in the application. It receives the synchronized estimate of real time from the logical clock and provides the *process time* to the scheduler. Process time is the local notion of time that corresponds to the progress of the computations. It indicates the epoch with which the computations that are being performed are associated. Real time is continuous; process time is discrete. During the finite-length computations of the application, the process time remains constant. When all computations for a time point have completed, the process time is updated to match the time of the next time point. For real-time applications, it is the responsibility of the pacer to maintain process time as close as possible to real time. In step-time applications, the pacer increases the process time as rapidly as possible. This leads to the paradox that real-time applications are generally slower than step-time applications.

The registry manages the signals in the application. The signals in a signal processing application correspond to the channels of any other message-oriented application. They are the carrier of the information between the modules. When modules are to be developed independently from each other, the application must have a facility to connect the input and output signals of the various modules at its disposal. This functionality is provided by the registry. Every internal signal is registered by the module that produces it. Every module that requires a signal, makes a corresponding request with the registry. It would be possible to limit the activities of the registry to this kind of brokerage: Supplier and user of the channel information are informed about the other side and it is left to the modules themselves to arrange the exchange of data. This would be true to the concept of concurrent applications, in which control resides with each of the individual threads and each thread has its own responsibility over its state transitions. However, mere registry brokerage is undesirable for a distributed signal processing system. The set of modules can usually be divided into typical suppliers and typical users. Data acquisition modules are typical suppliers; data processing modules are typical users. Suppliers and users are often located at different nodes. Platforms that host many data acquisition modules usually do not host the data processing modules; typical data processing platforms are high-capacity computing nodes that do not support the interfacing required for data acquisition components. If in such a system a signal from a data acquisition component on one node is used by many processing modules on another node, each of the users will request the same information from the supplier, and the supplier must notify all users of the availability of new data. This is inefficient and can easily lead to overload of the information provider or the network that connects the two nodes. The registry of figure 4.3 therefore has a more extended role. Apart from maintaining an overview of all internal signals in the system, it also actively supports the data exchange through these signals. Whenever a supplier publishes the results of its activities in a signal, the registry stores the new

data in a signal object. It automatically notifies the scheduler of the new data, providing a complete list of all modules that subscribe to the signal. When one of the subscribers is subsequently activated by the scheduler and requests the data as its operand, the registry retrieves the data from the signal object and makes it available to the user module.

Like the registry manages the signals, the scheduler manages the states of the modules. Because the registry sends notifications about new signal data to the scheduler rather than to the subscribing modules, the scheduler must monitor the blocked states of the modules to check whether they can be lifted. The scheduler can then combine this information with the temporal information – the process time – which is received from the pacer to determine which modules are awaiting scheduling. Finally, the scheduler activates the appropriate modules. It is the joint responsibility of the registry and the scheduler to synchronize the data in the various signals and to guarantee that correct data is available when a user is activated, even if publishing and subscribing modules are scheduled at different process times.

Application-dependent components

Figure 4.3 shows three classes of application-dependent components: signals, modules, and pipes. Signal components are created by the registry whenever a module registers a publication channel. The application developer who specifies the modules for the application therefore does not explicitly compose an overview of all the signals, although such an overview could easily be extracted from the application's signal diagram. The ensemble of internal signals in the system is implicitly defined by the collection of modules. A comprehensive overview of the latter is therefore indispensable.

Signal components are accessed only by the registry. By handling all data exchange between the operators, the registry can synchronize the data in the various signals. To support this, each data packet in a signal is stamped with an interval that determines its temporal applicability: the *signal interval*. It is provided by the module that produces the signal. The signal interval starts with the *publication time* and ends with the *validity time*. The publication time inclusively marks the epoch at which the data packet becomes valid. The packet maintains its validity until the epoch that is marked by the validity time; the validity time itself is not included in the interval. Usually, the validity time for a data packet is equal to the publication time of the subsequent data packet in the signal†. The signal component stores the most recent data packet together with its signal interval as the state of the signal.

Similar to the signal interval that is associated with each signal, each module is stamped with a *schedule interval*. The schedule interval defines the first upcoming process time frame in which the module must be activated. It starts with the release time and ends with the deadline. The interval allows the scheduler to compare the temporal

† The continuous coverage of subsequent signal intervals is typical for periodic signals. Aperiodic signals are often triggered by unpredictable events and do not adhere to this concept.

demands of the module with the registry-provided availability of the data in the signals it subscribes to; it replaces the ticker that would be required otherwise to initiate the activation of the module. Schedule intervals are defined by the module itself: Every time that a module finishes its computations and moves into a blocked state, it specifies a new schedule interval. Since there are no limitations on either the release time or the deadline, the module has total control over its scheduling demands. Dispatching does not necessarily occur at a single epoch and intervals do not need to be equidistant. The module-specified schedule window is the key mechanism behind unconfined threads. Because a module can adjust its activation rate to its state, it enables graceful mode changes. For example, a module can be in a 'sleeping state' in which it is activated whenever new data in a signal arrives. If the new data serves as a 'wake-up call', the module can subsequently demand scheduling at more frequent, regular intervals.

Module and pipe components belong to the common parent class of operators. Pipes are proxies that connect one node in the distributed system to the next; they replicate the signals, modules, and logical clocks of the remote nodes as if they were local. With respect to producing and reading signals, pipes exhibit the same behavior as modules. It is this behavior that is generalized in the operator class; to the registry, there is no difference between a module and a pipe. When a local module subscribes to a signal that is produced remotely, the pipe end at the remote node subscribes to the signal. By specifying an infinite schedule interval, the pipe is activated whenever new data arrives in the signal. The remote pipe then sends the data packet to the local pipe end through the network. The local pipe end publishes the data in a duplicate signal; the registry will treat the replicated signal and the arriving data just like data that is produced locally. This mechanism ensures that each new data packet is sent through the network only once, independent of the number of subscribers on the receiving side. In addition to replicating signals, pipes ensure the internal synchronization of the system by regularly exchanging their respective local time estimates and activating the synchronization mechanism of the logical clock.

4.5 Application topology

The middleware architecture that is presented in the previous section facilitates the assembly of an object-oriented, distributed signal processing system primarily by two components: the generalized pipe and the registry. Pipe components handle the connection of the nodes in the system through the network. As such, they determine the physical structure of the application. The set of registry components on all the nodes manages the connection of the input and output signals from the modules, independently of the physical application topology. Because ambiguities may arise when multiple instances of a single module are used, or when different modules produce similar signals, the registries must provide a logical application topology that resolves such conflicts.

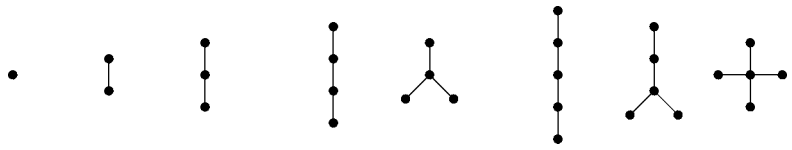
The physical application structure is determined during the architectural design of the application, by selecting the platform components and assigning the data acquisition and processing components to the platforms. Within the application, the physical structure is only visible to the middleware. The logical application structure resides at a higher level of abstraction. It is determined during application analysis and implicitly recorded in the signal diagram. The logical structure specifies exactly in what way signal processing components must be connected by projecting groups of components and subsystems on the overall application. The set of platform components is unaware of the application's logical structure; the logical topology is completely a property of the functional side of data acquisition and processing components. The encapsulation of the physical application topology – that determines which signal processing components execute at which node of the system – in the logical topology – that determines which signals are visible to each component – is an important characteristic of the middleware.

Physical structure

The connection of nodes in the distributed application as indicated in figure 4.3 is a strict one-to-one relation. Each connection is referred to as a pipe; it is implemented by two pipe components on the respective nodes of the application. Each pipe end represents the remote node on the local application. The network is a facility that is used by the pipe to perform the actual communication between the nodes; it is invisible to the rest of the middleware. As such, pipes cannot be used to represent the network as an independent application component through which multiple remote nodes may be reached. This concept guarantees complete independence from the capabilities of the various types of networks that can be encountered in an instrumentation system. The network is regarded as nothing more than a facility that enables two platform components to communicate. Such a network can be a standard type of network that allows the connection of many nodes, but also a private communication channel between two platforms like shared memory or a proprietary data bus.

The complete distributed application with n nodes is constructed with exactly $n-1$ of these one-to-one network connections. Because the network connections are the $n-1$ edges of a connected, undirected n-node graph (Euler, 1736; Harary, 1969; Diestel, 2000), the distributed system forms a *tree graph* (Cayley, 1857). Tree graphs have two prime characteristics that are desirable for the structure of an instrumentation system network. First, the number of edges is minimal. If an edge is removed, the graph becomes disconnected. Second, there exists exactly one path between any two nodes in the graph. This is equivalent with the absence of any loops – in graph theory referred to as circuits – in the undirected graph. If an edge is added to a tree graph without adding a node, a circuit is created. The tree structure is therefore the least-redundant topology for a network that is based on one-to-one pipes.

Generally, the nodes in a tree graph are unordered. A tree with unordered nodes is called a *free tree* (Cormen et al., 2001, pp. 1085-1091). Alternatively, a tree can have a



All free tree graphs for 1, 2, 3, 4, and 5 nodes. In free tree graphs, all nodes are equivalent.

single *root node*; the nodes in a rooted tree are ordered. The nodes that are adjacent to the root node are referred to as its children. Each child can act as a parent itself for any number of children. This way, a hierarchy of nodes is obtained. A tree with ordered nodes optimally suits a computer network structure for a distributed instrumentation system. The known parent-child relationship between any two adjacent nodes allows for different roles of the nodes at both pipe ends. The difference is most prominent during start-up of the application. A node that acts – or can act – as a parent, contains a server implementation of the pipe. A node that acts as a child on the other side of the same network link, implements the corresponding client. The child node connects to the parent's server during application start-up.

Because the parent node only needs to offer the pipe services for the type of networks it supports, the parent node must only be aware of those types of networks, rather than the actual instances of child nodes. A child node must know its exact parent; the parent does not need to know about the children until they have connected to the pipe service. Some services allow only one child to connect. A typical example is a network that is implemented by shared memory to which two processing units have access. Other services can allow any number of children to connect, for example on a standard network like Ethernet. A single service is then offered for establishing a pipe. Whenever a child connects to the parent, a new pipe is created for the actual communication between the nodes.

The procedure in which child nodes connect to a single parent and in which each node can act as a parent for any number of children, automatically results in a rooted tree network. The root node is the single node in the system that does not connect to a parent. Because every other node is associated with a unique pipe to its parent, the number of edges in the system equals the number of nodes minus one. Circuits cannot be created. This will prove important for the clock synchronization technique that is described in the next section. If more than one node in the system does not connect to a parent, the graph of nodes becomes disconnected. From each root, a separate tree is created between which there can be no communication. An instrumentation system with such independent subsystems is generally undesirable, since the middleware would not be able to fulfill its role as a central interface between the platform components and the data acquisition and processing components. Instead, the information that is

to be exchanged between the subsystems must be sent through external signals between input and output components, which will create a dependence between those components. This affects the flexibility to change the application or reuse the components.

Logical structure

As discussed on page 114 and shown in figure 4.3, all internal signals in the application are managed by the registry. Modules that rely on the output from other modules interact only with the registry; they have direct access to neither the signals, nor the modules that are responsible for producing them. The middleware must therefore provide a mechanism that allows consuming modules to uniquely identify the signals that they depend upon. Such a mechanism can easily interfere with the flexibility of module development. A signal identifier – such as a describing name – must be agreed by the producing and all the consuming modules. Modules that are originally developed for different applications, generally do not share a single identifier for the same signal. New applications thus require a modification of those modules, which affects their backward compatibility. A second limitation of using unique identifiers arises when a signal-producing module is instantiated more than once in an application. The multiple occurrence of signals with the same identifier conflicts with the requirement to uniquely identify each of them.

Both problems can be solved by the practice to create standardized components and to derive specializations for each application. Legacy modules can be used to derive application-dependent versions that only differ from their ancestors in the signal identifiers. Modules that are to be used repeatedly, can be specialized into multiple versions with mutually different signal identifiers. However, specializing components only to match signal identifiers can become cumbersome for large applications. Signal identifiers themselves run the risk of becoming counterintuitive in the process of finding distinguishing descriptions. At the same time, there is usually no need for all signals to have a unique, globally valid identifier. Many signals are only used by a few modules that are topologically close to its producer. It is therefore desirable that the middleware can connect such signals locally; multiple signals are then allowed to have the same identifier, as long as they are located in different parts of the application.

The middleware can resolve the problem of connecting the correct input and output signals in a system with redundant signal identifiers by recognizing different signal scopes. Each scope forms a subspace within the whole application in which signals must have unique identifiers. A scope groups the modules and the signals that are logically related. Usually, scopes coincide with certain physical or logical subsystems of the full application. For this reason, such a scope is referred to as an *entity*. Entities are organized as a rooted tree; consequently, they have hierarchical relationships. Module objects are assigned to a single entity in the system. The signals they produce are automatically visible in the entity where the module is located and in all its child entities. By default, signals cannot be seen from entities higher in the tree or from the other child entities of such a parent entity. Figure 4.4 shows an example of an entity tree.

Each entity can be regarded as a part of its parent entity. In the example, the systems are part of the aircraft; both the left and the right engine are part of the aircraft systems. This part-of relationship is typically depicted as a UML composition between objects.

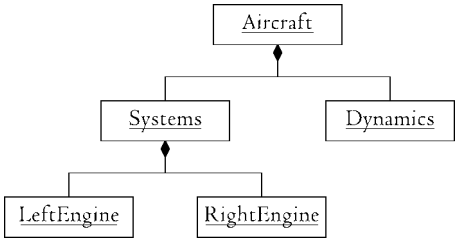


Figure 4.4: Example of entities.
Entities form a tree of subsystem objects in the complete application. Entities can correspond to physical subsystems like the systems of an aircraft, but also to logical groups of signals such as the aircraft's dynamics. The child entity named systems is the parent of two children: one for the left-hand, and one for the right-hand engine. In the latter two entities, local spaces are available for the modules and signals that relate to a single engine.

To enable the connection of signals with different identifiers across entity boundaries, an *alias* can be defined. A normal alias does nothing more than providing an alternative identifier for an existing, visible signal. This way, signals from legacy modules with signal identifiers that do not match can be connected without adapting the modules. More importantly, an alias allows to change the name of a unique signal identifier from a parent entity into a redundant identifier in the local entity. This allows to connect multiple instances of a single module in different entities – which all require the same input signal – to different signals in the parent entity. Normal aliases only provide an additional identifier under which a signal that is already visible can be addressed; they do not extend the scope of a signal. A *deep alias* makes a signal visible in one of its ancestors and thus automatically in all of the ancestors children. Similar to a normal alias, a deep alias can change the name of the signal it refers to.

For a middleware as the one presented here, entities and aliases should be defined during application analysis. The application's signal diagram contains the signal names that are used as the signal identifiers. The signal diagram therefore allows to find any mismatch in input and output signal identifiers or conflicting identifiers due to the repeated use of modules. Entities and aliases are therefore best recorded in the signal diagram. For this purpose, the symbology of the signal diagram that was presented in figure 2.1 (page 52) must be extended to depict an entity, an alias, and a deep alias. The new elements are shown in figure 4.5.

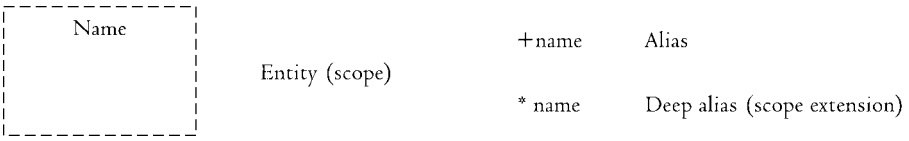


Figure 4.5: Topological elements of a signal diagram.
Although entities are objects, the solid-line class and object symbols in the signal diagram are preserved for operators. For entities, a new stereotype symbol is introduced as a dashed box. Similar to the way signal names are recorded in the signal diagram (see figure 2.1), aliases are shown as the new signal name next to the signal arc. A normal alias is characterized by a plus sign in front of the additional signal name; a deep alias is preceded by an asterisk.

An example of the use of entities and aliases in a signal diagram is shown in figure 4.6. Two signal paths are shown that start at two producing modules. The two signals are used as an input for two instances of the same data processing component. To enable the two instances to be connected to the correct input signal, the objects are located in separate entities. The input signal crosses a single entity boundary in the direction from the parent to the child entity. Such a crossing does not require an alias to make the signal visible. A normal alias is used to match the identifier for the processing module's input signal to the respective signals from the parent entity. The output signal of the data processing components is local to the entity in which it is produced. The signal path that is shown crosses two entity boundaries in the direction from the child to the parent. Such a crossing always requires a deep alias because signals are not visible in their parent entities. At the same time, the deep alias resolves the ambiguity conflict that would arise when both output signals from the processing modules are made visible in the parent entity without changing their identifiers.

4.6 Activities

The middleware's four core components are the logical clock, the pacer, the scheduler, and the registry. Figure 4.3 shows that control resides with the pacer; this comparatively small component initiates almost all the activities in the system. The only deviations from the pacer-initiated flow of control occur when a module responds to an interrupt from an external system. This will typically be the case for data acquisition modules that import aperiodic signals. For example, discrete packages that are read from a databus arrive in irregular patterns. The interfacial hardware to the databus will usually receive a complete packet and subsequently trigger an interrupt with the corresponding data acquisition component. Similar to such active external signals, the

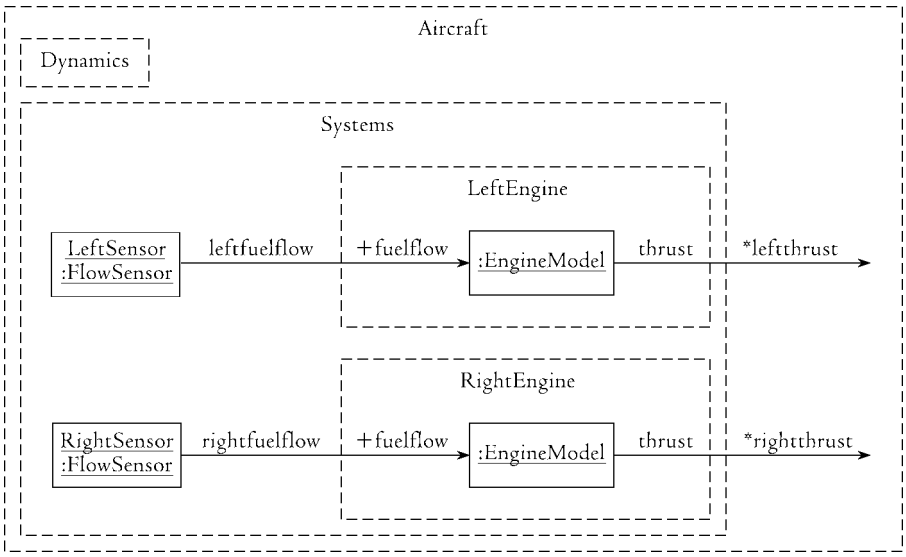


Figure 4.6: Example of topology in a signal diagram.
The entity tree from figure 4.4 is depicted as five nested entities in a signal diagram with four modules. The class FlowSensor measures the fuel flow of an engine; two instances of the class measure the flow for the left and the right engine respectively. They are not part of the entities LeftEngine and RightEngine, because these modules are input components that acquire external signals which do not adhere to the middleware topology. The class EngineModel estimates the thrust for an engine based on the fuel flow. The instances for the two engines are in different entities; aliases provide the appropriate signal identifier 'fuelflow' for each of the engines. Each engine model object produces a signal named thrust. Since these are visible only within the two respective engine entities, deep aliases are needed to reveal the thrust signals under distinguishable names in the entities Systems and Aircraft.

remote nodes in a distributed system behave as an actor with respect to the local node. Pipes can thus initiate activity in the system that is not controlled by the pacer.

In the scope of dynamic modeling, the middleware can be seen as a single process for each node in the distributed application. Within that process, the pacing thread is the most important. Most activities take place within the pacing thread. Even when activities are performed by the logical clock, the registry, the scheduler, or one of the passive modules, control is handed over from the pacer and the activities therefore take place within the pacing thread. The pacing thread interacts with a number of threads to which it is concurrent: one for each remote node and one for each active module.

Pacing

The maintenance of process time is the most important responsibility of the pacer. The corresponding task is comparatively simple. Although the pacer thread is the primary thread in an application, most of the activities are carried out by the registry, the scheduler, and the modules. The pacer only maintains the process time and activates the scheduler in order to dispatch any module activity that is due. Keeping the process time in a real-time application differs significantly from that in a step-time application. In the latter case, the pacer has only a single state: the one in which modules are dispatched through the scheduler. The logical clock is not used. In a real-time application, the pacer can also enter an idle state, in which the progress of time is controlled by the logical clock.

Figure 4.7 shows the activities for the pacer in both real-time and step-time applications. The initial state is that in which modules are dispatched by the scheduler. The state is left when the smallest of all local release times – the earliest time at which a module may start computation – is larger than the current process time. The smallest of all local release times is referred to in the figure as the dispatch time. In a step-time application, the process time is increased to the current dispatch time and the dispatch state is reentered. The application thus progresses through simulated time at the rate that computation times allow. In a real-time application, the idle state is entered upon leaving the dispatch state. For the duration of the idle state, the process time is continuously kept in synchrony with real time as estimated by the logical clock. The idle state is left when the process time is no longer smaller than the dispatch time. Most often, this is caused by the progress of process time. Alternatively, a module may be activated by some external event outside the pacing thread. Through the registry and the scheduler, the signals that are published by such a module may unblock other modules and reduce their release times accordingly.

The middleware's logical clock has the primary function to provide an estimate of time to the pacer in a real-time application. In addition, the logical clock must synchronize with the clocks on adjacent nodes and possibly to one or more locally available external reference times. Because of the close relation between the logical clock and the pacer, the clock synchronization activities are organized as a second fiber in the pacing thread. The primary fiber – formed by the activities that are shown in figure 4.7 – is the active one. In order to achieve real-time performance, the activities in the dispatch state have priority over all other computations; the primary fiber must therefore retain control over the pacing thread. Both the pacer's idle state and the logical clock are exclusive to real-time applications. When the pacer is in its idle state, the secondary fiber can be activated to enable the logical clock activities. Clock synchronization consists of communication between nodes and short-duration adjustments of the clocks; there are no activities in clock synchronization, but only actions and waiting states. For the algorithm that is presented in this thesis, the states and actions of clock synchronization are shown in figure 5.7 (page 156). Between the state transitions of quasi-zero duration, the secondary fiber is therefore permanently in a blocked state.

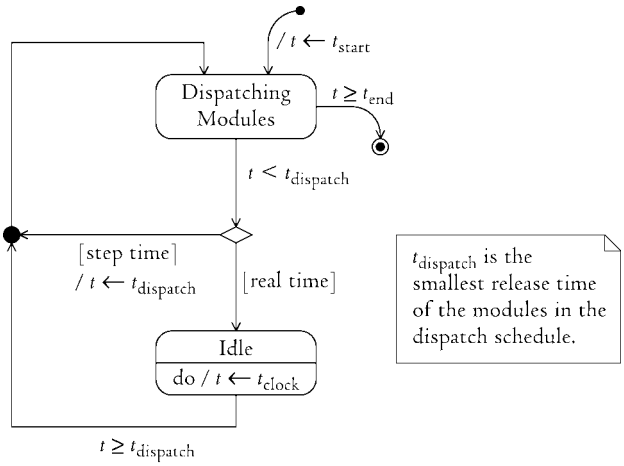


Figure 4.7: Pacing activities.
The pacer activities are centered around the process time t. The time is initialized with the process start time; the pacing – and thus the application – ends when the end time is reached. In step-time applications, the process time is increased with the completion of module computations to the next dispatch time. In real-time applications, the process time is connected to the logical clock time and the pacing idles until the release time for one of the local modules has arrived.

This allows it to return control to the primary fiber any time the pacer should leave the idle state and enter the dispatch state.

Registration

The signal management activities of the registry can be divided into three major tasks: connecting producers and consumers, managing subscriptions, and notifying subscribers when a signal contains new data. From the registry's point of view, subscription management and consumer notification about new data are completely local. Only during the connection of signals, the activities of the registry are distinguishable for producers and consumers that both reside at the local node, and for signals that cross the node boundaries in a distributed application.

Figure 4.8 contains the sequence of messages for the connection of a signal. The consuming operator, either a module or a pipe end, places a request for a signal by providing a signal identifier. According to the rules described in section 4.5, the registry resolves the scope of the requesting operator and any applicable aliases. If the requested signal exists locally in the scope of the operator, a notification is sent and the connection is complete. Otherwise, the registry relays the request to the adjacent nodes. At a remote node, the pipe end repeats the original request with the remote reg-

istry just like a local module would. If the signal is found, either because it is locally available in scope, or because it was obtained from a recursive relay to nodes further in the network tree, the pipe end subscribes to the signal as a consuming operator and returns the positive answer to the first node. The local pipe end then registers the signal as a producing operator with the local registry and the originally requesting operator can be notified. If the signal is not found – which means that it is not in scope anywhere in the subtree of the application that starts at the particular node – the registry is notified of the negative outcome of the search. Finally, if all adjacent nodes return negative responses, the local operator is notified of the unavailability of the requested signal.

Once they have connected, modules can quickly subscribe and unsubscribe to a signal. Subscription only means that the registry adds the module to the list of subscribers for the particular signal; unsubscription is a removal from the list. This allows modules to flexibly change their subscriptions and hence switch through various modes of operation. Mode and subscription changes never involve communication with remote nodes. Pipe ends that have been involved in a successful signal connection, subscribe to the signal once and remain in the subscribed state, regardless of the subscription state of the final consumers of the signal. Whenever new data is published by the original producer of a signal, the registry notifies the scheduler of all operators in the subscribers list. When a pipe operator is subsequently activated by the scheduler, it reads the new data and sends it through the pipe. The receiving pipe end acts as a producing operator and publishes the data with the local registry. During operation of the application, pipe ends are therefore proxies of remote modules, including both consumers and producers.

A registry can receive only one positive response to the signal connection request from all of its adjacent nodes. If a single signal would be available at two adjacent nodes without having already been registered locally, the two adjacent nodes must be connected by a path that does not go through the local node. This means that the two nodes are connected by at least two different paths, which violates the requirement that the network is a tree. The absence of circuits in the network also allows each node to pass on the connection request to other nodes without causing an endless loop of requests through a series of nodes where the signal is not found. Finally, the tree structure of the network and the connection procedure of figure 4.8 guarantee minimal proliferation of signals. Signals are only present at those subtrees of the network that are necessary to connect all the subscribers to the producer. If a new subscriber is added, a new path is connected to the closest node in the existing tree between producer and consumers.

Scheduling

The scheduler produces the module activation schedule and dispatches the scheduled modules in close interaction with the pacer. As explained in section 4.1, the scheduler for a digital signal processing middleware should also control the various states of the

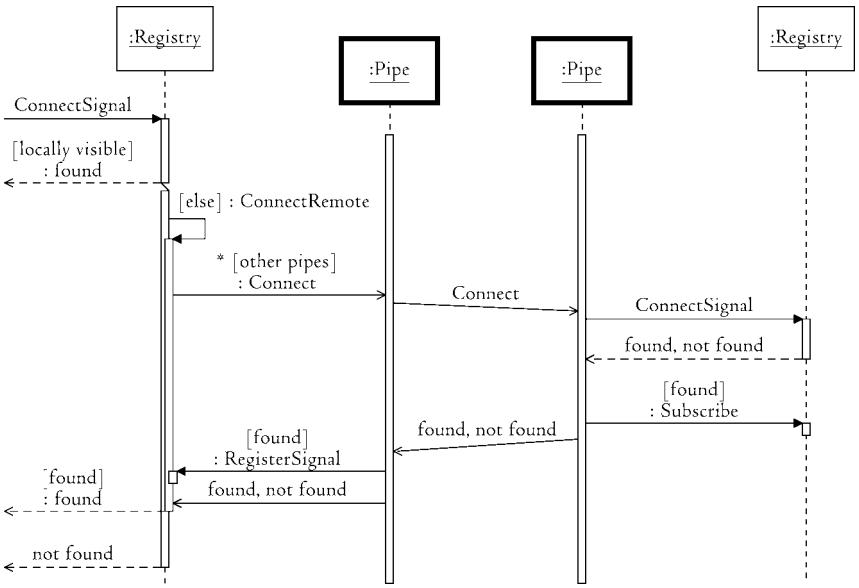


Figure 4.8: Registration sequence for connecting signals.
A synchronous query with the passive registry component is first processed locally. If the requested signal has already been registered and is within the correct scope, the initiating operator – a pipe end or a local module – is notified. Otherwise, the request is passed on to the adjacent nodes in the system by iteratively sending a request to each of the pipe components. In case the original request came from a pipe end, the originating pipe is excluded from the remote search. Like the local node, the remote nodes can pass on the request to other adjacent nodes. If the search for the signal with a remote registry is successful, the corresponding pipe end subscribes to the signal and notifies the local pipe. The local pipe then registers the signal with the local registry and the remote search is ended. Otherwise, the iteration continues with the next local pipe end. If the remote search is unsuccessful, the initiating operator is notified that the signal was not found.

modules that it manages. For this purpose, the scheduler maintains a finite-state object for each of the modules at the node. The object for module i contains the release time r_i and the deadline d_i as provided by the module itself, and a list of all the signals that the module has subscribed to. The module states as modeled by the object are shown in figure 4.9 as the left-hand orthogonal substate in the scheduler's waiting state. For each of the modules, such a substate exists. The right-hand substate is unique; it models the behavior of the scheduler itself.

As unconfined threads, the modules themselves specify their schedule interval that starts with the release time and ends with the deadline. All deadlines are treated as

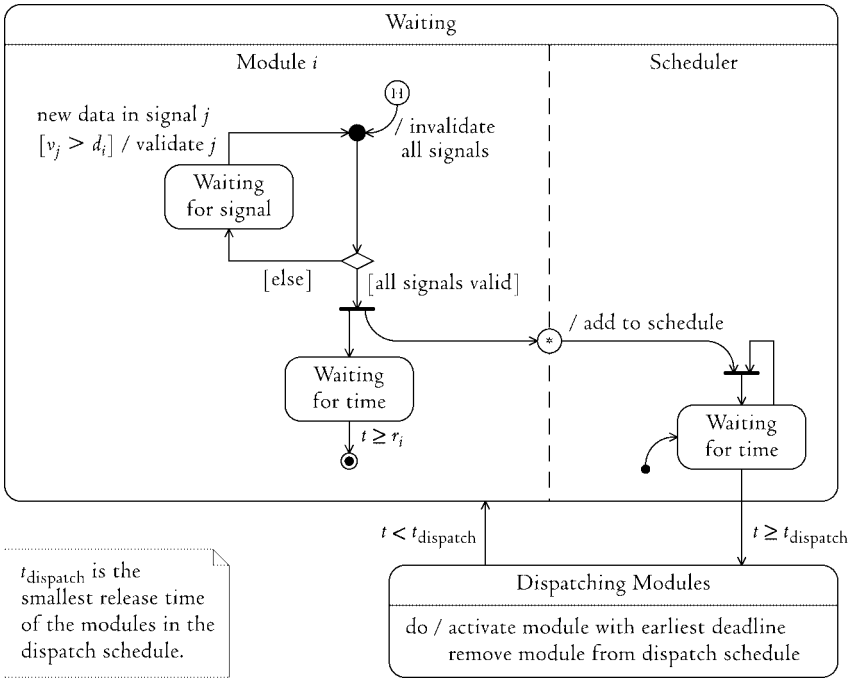


Figure 4.9: Scheduling activities.
The two primary states of the scheduler correspond to the states of the pacer as shown in figure 4.7. As long as modules can be dispatched for the current process time, they are sequentially activated in order of their deadlines. When the dispatch schedule has been exhausted for the current process time, the scheduler transits into a waiting state. The waiting state contains a single substate for the management of the dispatch schedule and one orthogonal substate for each module that resides at the node. New modules and those that immediately before have been dispatched, invalidate all their subscribed signals and wait for the arrival of new data in each of them. When all signals are valid for the schedule interval, the release time and deadline are communicated to the substate that manages the schedule and the module enters the state in which it awaits its release time. If the scheduler triggers a transition to the Dispatching Modules state after the release time for a module has passed, the module is part of the schedule; its substate has reached the termination pseudostate and is restarted when dispatching completes. Otherwise, the module substate resumes the waiting state it was in when dispatching started.

beginning-of-computation deadlines. The schedule is constructed according to the earliest-deadline-first criterion. As described in section 4.3 on page 109, EDF scheduling for beginning rather than completion of computation results in a least-laxity-first schedule that is particularly suitable to signal processing applications. In an applica-

tion with unconfined threads, only the modules themselves can be expected to have any knowledge on the duration of their computation if this knowledge is available at all. Any module that requires scheduling for completion of computation can thus provide a schedule interval that ends with a beginning-of-computation deadline which equals the difference of the required completion deadline and the expected computation time. Scheduling is performed using the process time that is provided by the pacer, not using the real time as estimated by the logical clock. Process time is discrete; it does not change during module computations for a single epoch in the process. Consequently, the laxity of a job does not change during the computation of another job and context switching between jobs with similar laxities is avoided. The use of process time for an EDF schedule with beginning-of-completion deadlines therefore results in an LLF schedule with the stable behavior of a completion-of-computation deadline schedule that is based on real time.

After each activation, the scheduler updates the release time and deadline of the module and marks all subscribed signals as invalid. Each time new data arrives in one of the signals, the scheduler is notified by the registry. The scheduler compares the validity time of the new data to the module's deadline. If the data's validity exceeds the deadline – indicating the applicability of the data to the complete interval the module may be activated – the signal is marked as valid. When all signals for a module have become valid, the module is added to the activation schedule and dispatched when the process time exceeds the release time.

The message-oriented scheduling technique that is based on signal and schedule intervals inherently supports the various module and signal types that are managed by tickers, periodic threads, and sporadic servers in other scheduling concepts. Signals are basically divided into *stream signals* and *event signals*. A stream signal contains a continuous, unbroken flow of data. Each data element is stamped with a publication time and a validity time that indicate the interval during which it is guaranteed that no new data arrives. Most often, the publication time matches the validity time of the previous element in the signal. There is no need for the intervals to be of equal duration; stream signals can thus convey both periodic and aperiodic series of data elements. The distinguishing feature of a stream signal is that upon publication of an element, the epoch that the following element arrives is at least approximately known. The application can wait for the following elements to arrive if the present validity does not extend over the required interval; the additional elements may actually provide new data for the required interval, or indicate in retrospect through their publication and validity times that the previous data remained applicable beyond its signal interval. The predictability of a stream signal contrasts to the unpredictability of an event signal. Data in event signals is published without knowledge of the publication time of the next element. Because modules with a deadline cannot wait for data in an event signal – new data may not arrive for a longer period of time – an event signal must be regarded as valid for any interval after the publication time. Hence, event data is characterized by a validity time stamp of infinity.

Organized according to the signals to which they subscribe, modules can be divided into *free modules*, *stream modules*, and e*vent modules*. Free modules do not subscribe to any input signal. After having been dispatched, they are immediately added to the schedule again and await their release time. Free modules are typically part of data acquisition components. The schedule interval is usually small: Only the interval determines the resulting activation epoch and activation most often involves polling or activating a device unit, which should occur at accurately preset intervals. Stream modules rely on one or more stream signals for input data. The schedule interval usually starts at the previous activation process time and ends after a finite interval. This results in activation as soon as data for the end of the schedule interval is available in all input signals. Unlike free modules in data acquisition components, stream modules can therefore be dispatched well before the epoch for which they produce new data. Finally, event modules should be scheduled when new data arrives on the event signals to which they subscribe. Because the arrival of an event is unpredictable, the event module cannot set a deadline. Event modules are therefore characterized by a schedule interval from the previous activation process time to infinity†. The module is then scheduled as soon as a new element arrives on the event signal. By letting the scheduler check the validity of all signals that a module subscribes to when new data arrives in one of them, an event module can subscribe to multiple event signals, yet be scheduled whenever new data arrives on one of them. Hybrid modules that subscribe both to stream and event signals can be handled the same way.

† To enable an event signal to be valid for the schedule interval of an event module, infinity as a validity time is defined to be larger than infinity as a deadline.

5
Clock Synchronization

A novel clock synchronization algorithm is presented that allows the nodes in a distributed system to determine a robust and accurate common estimate of real time. The new algorithm neither depends on any special hardware, nor does it impose a specific network structure. The key to the probabilistic peer-to-peer algorithm is simultaneous Bayesian estimation of local and remote time errors, of local and remote clock rate errors, and of the communication delay between the synchronizing nodes from a single, underdetermined equation.

Statistical accuracies of all parameters are estimated in parallel to allow for convergence to a near-optimal estimate of global time. The algorithm thus causes the set of clocks to converge towards the time indicated by the good clocks at the cost of adjusting poor clocks. As a result, the presence of one or more externally synchronized nodes anywhere in the network – which represents a good clock – inherently leads to external synchronization of the full network. The stochastic nature of the peer-to-peer algorithm and the recognition of the information content in the full set of clocks result in both more accurate and more precise time estimates.

INTERNAL and external clock synchronization in a distributed application is the responsibility of the middleware's logical clock. Internal synchronization is performed in interaction with the logical clocks on the adjacent nodes in the system; pipe components take care of the communication between the nodes. If a module with access to an external reference time is present, external synchronization is performed in direct interaction with that module. The logical clock must provide an interface to both the pipe components and the local modules through which the synchronization can be performed.

The presence of such interfaces for ensuring both internal and external synchronization is an inherent requirement on the middleware for a signal processing system. However, the functional requirements do not prescribe or even suggest a specific implementation of clock synchronization. Various algorithms for clock synchronization in distributed systems have been developed since the 1980s. Because these developments were not aimed at signal processing applications, only few resulting algorithms can provide synchronization in the range below one millisecond. Such accuracies have been achieved for specific real-time networks (Schossmaier et al., 1997) that are not generally available for any instrumentation system. The high demand on time accuracy for applications that perform dynamic computations like differentiation and integration, therefore requires a new clock synchronization algorithm to be developed.

5.1 Probabilistic peer-to-peer synchronization

Existing clock synchronization techniques emphasize internal synchronization; external synchronization is generally dealt with by designating a reference clock as the master in a master-slave structure. Such an approach is incompatible to an instrumentation system in which the availability of a reference time cannot be ensured. Instead, the reference time – nowadays typically acquired by a GPS receiver – must be treated like any other data acquisition signal in the application. The availability of new data in the signal should trigger an external synchronization of the system of clocks. The new syn-

chronization algorithm should therefore primarily maintain internal clock synchrony in an optimal way and must be robust against reference time dropouts. Whenever external synchronization is available, the system of clocks should follow the reference time, although it must also recognize the uncertainty that is part of any acquired signal.

Deterministic and probabilistic algorithms

The key problem in internal clock synchronization is the removal of skew due to time delay in the communication between the nodes. This uncertainty is eliminated for some systems by the introduction of specific distributed clock hardware (Schossmaier et al., 1997; Schmid, Horauer, and Kerö, 1999) or specific high-speed networks (Abali, Stunkel, and Benveniste, 1997). These approaches lack general applicability. Clock synchronization for instrumentation systems should only rely on the generic concept of a network by which nodes can exchange messages. Because of the stochastic nature of network communication delays (Christian, 1989, p. 534), the travel time of the synchronization messages cannot be neglected in a system that relies on a notion of real time. Attiya, Herzberg, and Rajsbaum (1996) focus on the effect of various message delays, but assume that clocks do not drift. Because an instrumentation system cannot be expected to have access to a sufficiently accurate clock to neglect clock drift in comparison to network delays, an algorithm is required that handles the effects of both clock drift and network uncertainties, without imposing any physical network structure or requiring specific clock synchronization hardware. Ostrovsky and Patt-Shamir (1999) propose such an algorithm. It is an extension of the method by Attiya, Herzberg, and Rajsbaum (1996) and acknowledges both message delay uncertainties and clock drift. However, the algorithm is *deterministic* and regards nodes with external synchronization as a master clock towards which all other nodes must be synchronized as closely as possible. The latter prevents a system from being tied to multiple reference times. Both are prohibiting disadvantages for application in high-accuracy instrumentation systems.

As Schmid (1995, p. 877) notes, the trade-off between accuracy and precision† of a system of clocks is a natural threat to the determination of a global notion of time. Synchronization is achieved by adjusting the clocks to correct for mismatches that are caused by communication delays. This causes an internally synchronized system of clocks to drift away from real time and counteracts the effects of external synchronization. The drift is more severe when communication delays are not properly acknowledged. Probabilistic synchronization algorithms, as suggested by Christian (1989), Arvind (1994), and Olson and Shin (1994), are generally more suitable for dealing with the accuracy-precision trade-off in externally synchronized systems than more traditional deterministic algorithms (Lamport and Melliar-Smith, 1985; Srikanth and

† In this context, precision refers to the maximum deviation between two clocks in the system; accuracy refers to the maximum deviation of real time. As such, precision is the goal of internal clock synchronization; accuracy is the goal of external synchronization.

Toueg, 1987). Probabilistic algorithms model communication delays as a stochastic process, often assuming that the delays exhibit a Gaussian distribution. As a result, the global estimate of time will be a random variable that is more accurate than the estimate from a deterministic algorithm.

The probabilistic method that was presented by Christian (1989) is based on a challenge-response procedure where a node, the slave, polls another node, the master, for its time. By determining the round trip delay of the message, a measure for the accuracy of the obtained time is obtained. Christian's method thus implies a strict hierarchical system of masters and slaves. Because the method does not recognize the uncertainty in the master's node local clock, and because the time delays between the nodes introduce additional uncertainties, each slave will have a time estimate that is less accurate than that of its master. In case of a master failure, all underlying nodes in the network will loose their synchronization. The probabilistic synchronization method by Arvind (1994) is a variation to Christian's method. It uses an improved time transmission protocol, but preserves the master-slave structure of Christian's method and therefore has the same disadvantages. This also applies to the synchronization algorithm that was developed for the internet, the Network Time Protocol (Mills, 1991), and its further developments (Mills, 1994; Mills, 1995). Abali, Stunkel, and Benveniste (1997, p. 119) found that NTP-synchronized nodes on a distributed system exhibit remaining clock errors in the range of 1 to 3 ms. Such a discrepancy is unacceptable for a signal processing system†.

Olson and Shin (1994) present a probabilistic method that does not rely on a master-slave structure. Nodes are divided into so-called synchronization groups, through which messages travel in a cyclic path; clock synchrony is achieved for all nodes that take part in the group. Because the selection of synchronization groups breaks down a distributed system into smaller subsystems, the method is well suited for application in large distributed systems. However, the method does not allow for external synchronization. Since a synchronization group does not prioritize amongst its nodes, there is no mechanism for forcing all clocks to converge to the time observed from a single, externally synchronized node. Additionally, the selection of synchronization groups and the cyclic organization of each group do not allow for arbitrary network topologies, in which certain nodes might be visible to only a single other node.

Schmid (1995) proposes a concept for a method that inherently allows for external synchronization or for the interconnection of multiple synchronizing subsystems. The underlying idea is to update a local clock only if the correction constitutes an improvement. To determine whether this is the case, the low-accuracy but high-reliability local clock is used to verify the integrity of the incoming synchronization messages. The method thus fuses the information from remote clocks with that of the local clock, resulting in a more accurate and more robust time estimate. Although Schmid recog-

† As an example, attitude estimation and navigation algorithms are often processed at a rate of 1 kHz, requiring synchrony of distributed nodes in the range of $10\ ms$.

nizes the need for a probabilistic version of the method, its implementation uses deterministic criteria. The times obtained from remote nodes are expressed in hard intervals; the intersection of the intervals is a measure for the accuracy of the time estimate. Because of these hard boundaries and because the known accuracies of remote clocks are not taken into account, the method does not exploit the stochastic information that is available in the system to the full extent.

Concept of peer-to-peer synchronization

External synchronization of a system of clocks can be improved with *probabilistic peer-to-peer clock synchronization*. Because it does not assign a master-slave structure, a peer-to-peer algorithm can optimally combine the stochastic clock information from all nodes. The cumulative uncertainty of the master clock and the communication delay for each child is eliminated. Especially when the network topology prevents all nodes from accessing a single master clock directly, the peer-to-peer algorithm results in more accurate internal synchronization. The key element of the method is a Bayesian parameter estimator. It distributes the required clock corrections according to the estimated accuracy of the individual clocks. Each synchronization step results in updated clock parameters and the corresponding updated accuracy estimates. This way, the procedure recognizes the stochastic knowledge on the parameters as obtained from all of the preceding optimization steps: The algorithm will converge towards good time estimates at the cost of adjusting poor ones. This holds for mutual synchronization of clocks in the network, but also for synchronization with an external reference time. The basic method is therefore applicable to both internal and external synchronization. When an external reference time is of better quality than the common notion of time in the network, the reference time will represent a good clock and the optimization will converge towards it. The presence of multiple reference times provides the system with additional information, which is utilized by the algorithm to further improve the common notion of time.

Bayesian parameter estimation algorithms are particularly suitable for application to the problem of clock synchronization. During each synchronization step, multiple parameters in the system of clocks must be estimated from a single, scalar observation: the indicated time. The system is therefore strongly underdetermined. However, parameter estimation methods are basically intended for overdetermined systems (Huffel and Vandewalle, 1991, p. 1). They provide a unique solution for the parameters, separating the system into orthogonal subspaces for the estimate and the estimation error (Sage and Melsa, 1971, p. 234; Huffel and Vandewalle, 1991, pp. 34-38). Such a unique solution does not exist for an underdetermined system. Common parameter estimation methods, including maximum-likelihood and least-squares algorithms, therefore fail to provide a reliable parameter estimate for underdetermined systems. Bayesian estimators are a generalization of maximum-likelihood and least-squares estimators (Eykhoff, 1974). They reflect additional stochastic information on the unknown parameters in the computation of the estimate. This additional informa-

tion allows to determine a unique solution for the clock synchronization parameter estimation problem where the specialized methods fail.

The use of a Bayesian estimator in clock synchronization results in a strong dependence on the stochastic correctness of the information that is injected in the synchronization process (Sage and Melsa, 1971, p. 223). Because the system of clocks is synchronized completely on the basis of peer-to-peer messages between adjacent nodes, there is no global management of the stochastic information in the system. Nevertheless, repeated use of the same information must be avoided. Reusing stochastic information results in excessive confidence in the resulting estimates; overconfidence in certain clocks will disturb the balance between good clocks and poor clocks. For this reason, loops of synchronizing clocks are not compatible with a stochastic algorithm. The Bayesian clock synchronization technique thus relies on the tree network topology for the middleware that was presented in the previous section.

Probabilistic peer-to-peer clock synchronization is particularly suitable for external synchronization to a high-accuracy, low-continuity time service like the one provided by the Global Positioning System. When the external service is available, the stochastic analysis of the logical clocks and the reference time results in an adjustment of both the local time and the corresponding clock rate. When external synchronization subsequently fails, the system continues to run at the adjusted clock rate until the external service is reacquired. In addition, the possibility to connect a probabilistic peer-to-peer synchronization system to multiple reference clocks can provide redundancy of the external synchronization. The learning behavior of the system of clocks and the provision for redundant synchronization to real time are clear advantages of the peer-to-peer method over deterministic or probabilistic master-slave algorithms.

Bayesian parameter estimation

The clock synchronization algorithm must estimate the parameters for each logical clock in the system. A logical clock consists of a linear function of the corresponding hardware clock. The nondimensional counter $t_i(t)$ of the clock i is combined with its granularity G_i and offset L_i to yield the clock time $C_i(t)$:

$$C_i(t) = G_i \cdot t_i(t) + L_i. \tag{5.1}$$

The granularity and the offset are the unknown clock parameters; both have the dimension of time.

The probabilistic peer-to-peer clock synchronization technique applies a single Bayesian parameter estimation algorithm to both internal and external synchronization. It relies on the model structure that is depicted in figure 5.1. The system consists of a linear map of the multivariate input $\mathbf{u}$ to the scalar output x. The dashed line indicates the system boundary. The input vector is measured without distortion; the output measurement is affected by noise. The system represents a pair of clocks for which the times must be synchronized; it yields the difference of the indicated times, corrected for the average communication delay between the clocks. The clocks

are synchronized by estimating all of the system parameters – grouped in the parameter vector $\mathbf{b}$ – from an observation of the difference of the two indicated times. Because the parameters should be chosen such that the time difference approaches zero, the observation on the system output is the pseudomeasurement $y = 0$. The Gaussian component to the communication delay between the clocks acts as measurement noise. If the communication delay between a pair of clocks would be constant, the system with the true clock parameter and average delay values would indicate a time difference of zero. In reality, the system output x equals the Gaussian delay component d which cannot be corrected for by the model. This error appears to the observer as the measurement noise $-d$ on the pseudomeasurement of the output.

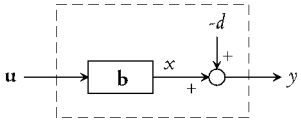


Figure 5.1: System model for clock parameter estimation.
The input vector $\mathbf{u}$ is multiplied by the parameter vector $\mathbf{b}$. The resulting system output x equals the communication delay d. The stochastic delay appears as the noise $-d$ on the pseudomeasurement $y = 0$.

The Bayesian parameter estimate for the system in figure 5.1 is obtained from the exactly known input vector $\mathbf{u}$, the a-priori parameter vector estimate $\mathbf{b}$ and corresponding covariance matrix $C_{\mathbf{bb}}$, and the variance for the random component to the communication delay C_{dd}:

Proposition. The optimal a-posteriori estimate $\mathbf{b}|y$ for the vector of model parameters in terms of maximum probability satisfies

$$\mathbf{b}|y = C_{\mathbf{bb}|y} C_{\mathbf{bb}}^{-1} \mathbf{b} \tag{5.2}$$

with

$$C_{\mathbf{bb}|y} = \left(C_{\mathbf{bb}}^{-1} + \frac{\mathbf{u}\mathbf{u}^{\mathsf{T}}}{C_{dd}} \right)^{-1}, \tag{5.3}$$

if $\mathbf{u}^{\mathsf{T}}\mathbf{b}$ is a scalar with an expected value of zero in case of synchronized clocks, $\mathbf{b}$ is the a-priori parameter vector estimate, and d is the zero-mean Gaussian communication delay between the clocks.

Proposition (5.2)/(5.3) follows directly from substitution of the output pseudomeasurement into the Bayesian parameter estimate for a generic linear map with measurement noise, the proof for which is given in appendix C. The scalar linear map $\mathbf{u}^{\mathsf{T}}\mathbf{b}$ is referred to as the *synchronization criterion*.

During an internal synchronization step, the clock parameters for two logical clocks are adjusted to make the indicated times converge. The synchronization criterion for the parameter estimation is based on the *internal time equation*, which states that the local and remote clock times differ by the delay D between the epochs at which the underlying hardware counters are read:

Internal time equation. For local node l and remote node r: $C_l(t_2) - C_r(t_1) = D$.

The time t_1 indicates the epoch at which the remote counter is read; t_2 indicates the epoch at which the local counter is read. If the remote counter is read immediately before the synchronization message is sent, and the local counter is read immediately upon receipt of the message, the delay equals the communication delay in the network. The total delay D is replaced by the sum of the average delay between the nodes D_{lr} and the Gaussian random variation d. Combination of the internal time equation and the clock model from (5.1) yields the synchronization criterion

$$G_l \cdot t_l(t_2) + L_l - G_r \cdot t_r(t_1) - L_r - D_{lr} = d, \tag{5.4}$$

which satisfies the requirement that $E\{d\} = 0$. The parameter vector $\mathbf{b}$ contains the four logical clock parameters and the average communication delay:

$$\mathbf{b} = \left[G_l \;\; L_l \;\; G_r \;\; L_r \;\; D_{lr} \right]^T. \tag{5.5}$$

The input vector $\mathbf{u}$ contains the counter values $t_l(t_2)$ and $t_r(t_1)$, and three constant coefficients:

$$\mathbf{u} = \left[t_l(t_2) \;\; 1 \;\; -t_r(t_1) \;\; -1 \;\; -1 \right]^T. \tag{5.6}$$

The updated clock parameter estimate can now be computed by substituting the old parameter values and the counter observations into (5.5) and (5.6), and applying the estimator (5.2) subsequently.

The principle of operation of the Bayesian estimator is illustrated in figure 5.2. It shows the joint probability density function $p_{y,\mathbf{b}}$ and the way it is constructed from the a-priori probability density of the parameter and the conditional probability density of the measured output. In order to be able to visualize the joint probability density, the multivariate parameter $\mathbf{b}$ is depicted as a one-dimensional quantity. Along the cut that is provided by the measurement on the output y, the a-posteriori probability density function of the parameter is found by dividing the joint probability density by the unconditional probability of the output. Because the latter does not depend on the parameter, the division only scales the resulting function to have an enclosed area of 1; it does not change the location of the maximum.

For external synchronization, the time equation states that the difference between the time indicated by a local clock at epoch t and an external reference time T_e equals the transmission delay between the epoch at which the reference time is observed and the epoch t. Although the communication delay can be modeled as the sum of an aver-

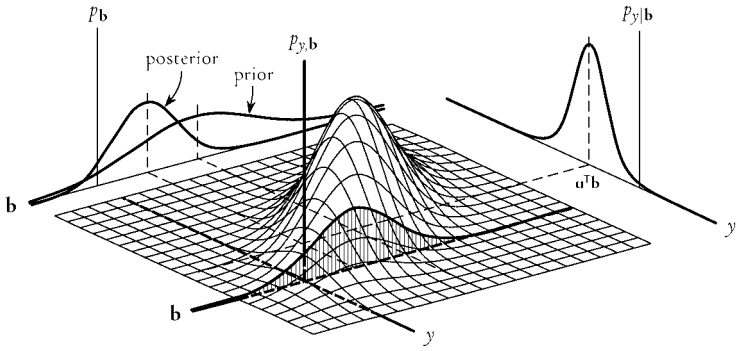


Figure 5.2: Application of the Bayesian estimator.
The joint probability density function $p_{y,\mathbf{b}}$ is the product of the a-priori probability density function for the parameter $p_{\mathbf{b}}$ and the conditional probability density function for the output given the parameter $p_{y|\mathbf{b}}$. Because $y = \mathbf{u}^T\mathbf{b} - d$, $p_{y|\mathbf{b}}$ equals the Gaussian distribution for the random delay variation; its maximum coincides with the deterministic system output $\mathbf{u}^T\mathbf{b}$. The dependence on $\mathbf{b}$ results in a skewed joint density function in the $y, \mathbf{b}$-domain. The synchronization criterion provides a cut through the joint probability density function that reduces the dimension of the domain by one. Because the criterion specifies that $y = 0$, the cut coincides with the parameter space $\{\mathbf{b}\}$. Along the cut, the a-posteriori conditional probability density function for the parameter given the measurement $p_{\mathbf{b}|y}$ is found. The new parameter estimate is chosen as the value of $\mathbf{b}$ for which the a-posteriori probability density function reaches its maximum.

age value and a Gaussian, zero-mean variation – similar to the modeling of internal communication delays – the average delay for external synchronization messages cannot be observed from within the system. Therefore, the consistent component to the total delay does not appear in the *external time equation*. Instead of being estimated with the clock parameters, any consistent communication delay – assuming that the average value is known – must be removed from the reference time beforehand.

External time equation. For an externally synchronized node l: $C_l(t) - T_e = d$.

Because the remaining random variation has an expected value of zero, the external time equation and the clock model from (5.1) directly yield the synchronization criterion:

$$G_l \cdot t_l + L_l - T_e = d. \tag{5.7}$$

In the form $E\{\mathbf{u}^T\mathbf{b}\} = 0$, the parameter vector contains the local clock parameters and the corrected reference time:

$$\mathbf{b} = \left[G_l \;\; L_l \;\; T_e \right]^{\mathsf{T}}; \tag{5.8}$$

the input vector contains the exactly known coefficients:

$$\mathbf{u} = \left[t_l \;\; 1 \;\; -1 \right]^{\mathsf{T}}. \tag{5.9}$$

The output of the Bayesian parameter estimator (5.2) for external synchronization provides an update for the two logical clock parameters, but also for the external reference time. The difference between the observed reference time and its updated value is the synchronization residual between the logical clock and the reference time. The existence of this residual is distinctive for the probabilistic peer-to-peer synchronization algorithm; it represents the acknowledged uncertainty of a reference time.

Stochastic properties

The Bayesian clock synchronization technique is a recursive algorithm. In each synchronization step, the parameters for the logical clock and the corresponding covariance matrix are updated using the previous values and the new observations on the physical clocks. The clock parameters, the communication delays, and all associated auto and cross covariances must therefore be initialized upon start-up of the procedure. Because the balance of the optimal estimator depends on it, it is important that the initial values are stochastically correct.

Initialization of a clock's granularity is straightforward; it follows directly from the tick frequency of the hardware clock for which a nominal value is known. The granularity variance C_{GG} can be estimated from the expected accuracy of the physical clock by taking the square of the counter's granularity, premultiplied by the expected drift ratio. For a clock with a typical drift ratio of 10^{-6}, the variance should be initialized at $10^{-12}G^2$. When the real time is approximately known, the initial clock offset L can be computed and the corresponding variance can be estimated from the accuracy of the initial time estimate. When no time information is available, the offset can simply be set to zero. The offset variance should then be initialized at a high value to indicate the absence of an accurate time estimate. For many clocks, a value of 10^{20} is appropriate when seconds are used as unit of time, indicating an uncertainty of about 300 years. This value matches the maximum range of a 32-bit second-based clock that rolls over after 136 years. The cross covariance between granularity and offset should usually be initialized to zero, which indicates that the initial values for the two values – representing the time and the clock rate – are independent upon start-up. Although it might be possible to provide initial estimates for the average time delays that occur during the communication between two nodes, it is not advantageous to do so. An initial value of zero, combined with an appropriately large variance, will cause the time delay to be estimated automatically during the initial convergence of the clocks.

The a-priori covariance matrix for internal synchronization contains the covariances for the local and remote granularity and offset estimates, and for the average

communication delay estimate D_{lr}. The parameter estimates for the local clock are assumed to be uncorrelated with those for the remote clock; in addition, the average communication delay estimate is assumed uncorrelated to all four clock parameters. The corresponding covariances are therefore set to zero:

$$C_{\mathbf{bb}} = \begin{bmatrix} C_{G_lG_l} & C_{G_lL_l} & 0 & 0 & 0 \\ C_{G_lL_l} & C_{L_lL_l} & 0 & 0 & 0 \\ 0 & 0 & C_{G_rG_r} & C_{G_rL_r} & 0 \\ 0 & 0 & C_{G_rL_r} & C_{L_rL_r} & 0 \\ 0 & 0 & 0 & 0 & C_{D_{lr}D_{lr}} \end{bmatrix}. \tag{5.10}$$

The a-priori covariance matrix for external synchronization is constructed from the local clock covariances and the variance of the reference time $C_{T_eT_e}$:

$$C_{\mathbf{bb}} = \begin{bmatrix} C_{G_lG_l} & C_{G_lL_l} & 0 \\ C_{G_lL_l} & C_{L_lL_l} & 0 \\ 0 & 0 & C_{T_eT_e} \end{bmatrix}. \tag{5.11}$$

The updated covariance matrices $C_{\mathbf{bb}|y}$ do not have the same structure as the a-priori matrices from (5.10) or (5.11). The synchronization step introduces additional cross covariance terms that mutually relate the estimates for the local and remote clock parameters and for the average delay or the external reference time. However, these cross covariances cannot be preserved and the covariance matrix for each following synchronization step will be filled as shown in (5.10) and (5.11). This procedure effectively resets the algorithm to its initial state† and destroys part of the stochastic information in the system. It affects the proper balance between all logical clocks and the external reference times; the stochastic error that is introduced will result in biased estimates of the clock parameters. The probabilistic peer-to-peer synchronization technique is therefore sub-optimal. However, it is impossible for a peer-to-peer algorithm to consider inter-node cross covariances. To achieve true optimality, quasi-simultaneous counter and reference time measurements on all clocks and central processing of the optimization equations would be required. Apart from practical complications for large systems, truly optimal synchronization thus puts a requirement on the availability of external reference times that cannot be guaranteed.

† The original assumption that parameter estimates for separate logical clocks are uncorrelated may be unjustified as well. If a single reference time or even if a similar procedure for initializing the logical times is used on several nodes, the clock offset estimates are dependent.

The bias on the logical clock parameter estimates has the same effect as drift of the physical clocks. Both cause a mismatch between the true granularity of the hardware clock and the estimated value that is used in the logical clock. Probabilistic synchronization algorithms require special measures to cope with such discrepancies between varying system parameters and the model parameters that are assumed constant. During repeated synchronization, confidence will be gained in the various estimates for the clock parameters and the average communication delays; the corresponding estimated variances will be monotonically decreasing. In the long term, this will cause a fixation of the clock parameters. The algorithm will not be able to react to changes in the true clock parameters caused by a drifting physical clock. Instead, all of the observed clock differences will be attributed to the random communication delays between the nodes and to the uncertainty in the reference times. In order to model nonstationarity of the hardware clocks, process noise must be added to the clock and communication models. The process noise handles both true drift of the physical clocks and the granularity bias that is introduced by the peer-to-peer synchronization algorithm itself. Process noise represents an increasing uncertainty on the parameters with the progress of time. When no synchronization takes place for a long time, confidence in the clock parameters is eventually lost. By comparing the clock with those of neighboring nodes, the clock and communication delay parameters are corrected and confidence is gained. Process noise is therefore implemented as a periodic increase of the parameter variances. The size of these increases should depend on the actual stability of the clock. Values of $10^{-8}\,[\mathrm{s}^2/\mathrm{s}]$ for clock offset and average time delay variances, and $10^{-6}\,[1/\mathrm{s}]$ per s^2 of granularity variance seem appropriate from practical experience.

The random delay variance C_{dd} serves as a tuner to the filtering effect of the confidence matrix in (5.3). By increasing the delay variance, the communication delay is regarded as less predictable and most of the error that needs to be compensated to satisfy the synchronization criterion is attributed to the random delay term. As a result, the clock parameters are adjusted less and a smoother time estimate is obtained. Conversely, decreasing the delay variances leads to larger adjustments to the clock parameters. The expedited convergence of the clocks to a common notion of time comes at the cost of more noisy behavior after convergence. In external synchronization, the variance for the reference time $C_{T_e T_e}$ has the same influence as the random communication delay variance; both provide a way to tune the convergence rate of the local clock towards the external time. When the a-posteriori variance estimate for the reference time is not preserved for future synchronization, both variances can be combined into a single reference time uncertainty. However, maintaining the structure as used here allows for easy modeling of the correct variances. Choosing realistic values for both $C_{T_e T_e}$ and C_{dd} will enhance synchronization performance by ensuring rapid convergence towards the reference time while optimally resisting transmission delay variations. Additional considerations with respect to the accuracy and precision of the probabilistic peer-to-peer clock synchronization algorithm are presented in the next section.

Integration of a probabilistically synchronized logical clock

The probabilistic peer-to-peer synchronization algorithm adjusts the logical clock in order to meet the synchronization criterion in an optimal way. Optimality in this respect is defined as the most probable constellation of logical and remote clock parameters and incidental communication delays. The method does not and cannot respect additional conditions without losing its favorable stochastic properties. In a signal processing system, a typical example of such a condition would be the requirement to provide a chronoscopic time estimate with a bounded clock rate. A probabilistically synchronized logical clock does not provide such a continuous time estimate; at each synchronization, a discontinuity is caused by the instantaneous change of the clock parameter estimates. This is referred to as instantaneous synchronization (Kopetz and Ochsenreiter, 1987, p. 936). Clock jumps due to instantaneous synchronization can occur both forward and backward in time. The output timescale is therefore neither chronoscopic nor guaranteed to be consistent. Since this is unacceptable for interval measurements as used for computation with dynamic data, the logical clock's output must be postprocessed by a second logical clock with filtered and bounded clock rate. The second clock should converge towards the output of the synchronized clock while maintaining a chronoscopic output. Since the demand for a chronoscopic timescale is related to the specific characteristics of a signal processing system, the filtering clock is modeled in the middleware architecture of figure 4.3 as part of the pacer rather than as an extension to the logical clock.

Although the raw output of the logical clock is not chronoscopic, it is the best estimate of real time based on the observations. Real time is continuous; if the observations that are used as a reference time are not chronoscopic, they do not match the logical clock's prediction and the Bayesian estimator cannot correctly be applied. It is therefore essential that throughout the signal processing system a single, chronoscopic timescale is used. TAI is the most suitable scale for this purpose. Observations on another scale, for example UTC, should be converted to TAI before being used for external synchronization. The problem of using UTC in signal processing applications and the desire to use TAI instead is also recognized by Levine and Mills (2000); they suggest a standard for providing leap second information along with UTC synchronization messages. As discussed in section 4.2, the constant shift between GPS time and TAI makes such a standard redundant when GPS is used to obtain reference time observations.

5.2 Practical aspects

Physical clocks are implemented as a counter of ticks that arrive at a regular interval. Because a counter in a computer system has a finite range, counter rollovers will occur after each *clock period*, which equals the range of the physical clock's counter multiplied by its granularity. The linear function that forms a logical clock must be prepared to

handle such counter rollovers. In the probabilistic peer-to-peer synchronization algorithm, the occurrence of a counter rollover has consequences for the stochastic properties of the logical clock as well. With the corrections that are required to handle counter rollovers, artificial rollovers can be introduced to the algorithm. The use of artificially rolled counter values leads to a numerically better conditioning of the parameter estimation problem; computational flaws due to numerical limitations may thus be avoided.

Counter rollovers

To handle counter rollovers, the logical clock compares each observation on the hardware counter to the previously obtained value. In case the former is smaller than the latter, the offset for the logical clock is increased by the known physical clock period. If N denotes the range of the hardware counter, the rollover is described as a reduction of the original counter t_1 by N, to yield the new counter value t_2:

$$t_2 = t_1 - N. \tag{5.12}$$

The time that is indicated by the logical clock from (5.1) must not be changed by this event. This provides the new clock offset L_2:

$$L_2 = t_1 \cdot G + L_1 - (t_1 - N) \cdot G = L_1 + N \cdot G. \tag{5.13}$$

The definition of the second-order statistics of a stochastic variable yields directly that

$$\begin{aligned} C_{(x+y)z} &= C_{xz} + C_{yz} \\ C_{(x+y)(x+y)} &= C_{xx} + 2C_{xy} + C_{yy}. \end{aligned} \tag{5.14}$$

When applied to the new clock offset L_2, this provides the updated clock parameter covariances:

$$\begin{aligned} C_{GL_2} &= C_{GL_1} + NC_{GG} \\ C_{L_2L_2} &= C_{L_1L_1} + 2NC_{GL_1} + N^2C_{GG}. \end{aligned} \tag{5.15}$$

Only through these transformations, the second-order statistics of the time estimate according to (5.1) are unaffected by a counter rollover, which is important for successful application of the Bayesian parameter estimator.

Numerical conditioning

The Bayesian estimator in the peer-to-peer clock synchronization algorithm is extremely susceptible to numerical inaccuracies. The squared occurrence of numbers that are of varying order of magnitude – for example small granularity values in combination with large clock offsets and counter values – leads to ill conditioning of the matrices in (5.3). Numerical problems should be prevented by a combination of two measures. First, the probabilistic peer-to-peer clock synchronization algorithm should

be applied with normalized clock parameters. Second, a formulation of the Bayesian estimator of (C.12)/(C.14) should be used that does not involve matrix inversions.

Before every synchronization step, the normalized parameters are computed by scaling the counters with their own granularity and by shifting the clock offsets to a common origin, for example the offset of the local clock:

$$\begin{aligned} \hat{t}_i &= t_i \cdot G_i \\ \hat{G}_i &= 1 \\ \hat{L}_i &= L_i - L_l \\ C_{\hat{G}_i\hat{G}_i} &= C_{G_iG_i} / G_l^2 \\ C_{\hat{G}_i\hat{L}_i} &= C_{G_iL_i} / G_i \end{aligned} \tag{5.16}$$

in which the hat indicates a normalized parameter. The a-posteriori parameters are obtained through the inverse transformation on the normalized results from the Bayesian estimator:

$$\begin{aligned} G_{i|y} &= \hat{G}_{i|y} \cdot G_i \\ L_{i|y} &= \hat{L}_{i|y} + L_l \\ C_{G_iG_i|y} &= C_{\hat{G}_i\hat{G}_i|y} \cdot G_i^2 \\ C_{G_iL_i|y} &= C_{\hat{G}_i\hat{L}_i|y} \cdot G_i \end{aligned} \quad . \tag{5.17}$$

(5.16) and (5.17) eliminate the influence of the clock granularity; the normalized counter has the dimension of time, which is the same for all clocks. To further improve the conditioning of the confidence matrix, it is desirable that the offsets for the logical clocks are approximately the same as well. This can be achieved by introducing artificial counter rollovers using the equations that were presented in the previous subsection. If a rollover is inserted once every second, each logical clock consists of a second counter L and a normalized sub-second component $t' \cdot G$, in which the prime denotes the artificially rolled counter values. All parameters now have the dimension of time; once the clocks are approximately synchronized, their offsets are unlikely to deviate more than one second. The normalized offsets from (5.16) are thus of the same order of magnitude as the normalized counters.

According to the matrix inversion lemma, the inverse of the confidence matrix as defined by (C.12) and used in (5.3) can be written as

$$\left(C_{\mathbf{bb}}^{-1} + \frac{\mathbf{u}\mathbf{u}^{\mathsf{T}}}{C_{mm}} \right)^{-1} = C_{\mathbf{bb}} - \frac{C_{\mathbf{bb}}\mathbf{u}\mathbf{u}^{\mathsf{T}}C_{\mathbf{bb}}}{\mathbf{u}^{\mathsf{T}}C_{\mathbf{bb}}\mathbf{u} + C_{mm}}. \tag{5.18}$$

With (5.18), the Bayesian parameter and covariance estimators can be reformulated as a three-step propagation. The first step is the computation of the transition matrix K:

$$K = I - \frac{C_{\mathbf{bb}}\mathbf{u}\mathbf{u}^{\mathsf{T}}}{\mathbf{u}^{\mathsf{T}}C_{\mathbf{bb}}\mathbf{u} + C_{mm}}. \tag{5.19}$$

Both the a-posteriori parameter estimate and the corresponding covariance matrix are subsequently obtained from a single multiplication with the transition matrix:

$$\begin{aligned} \mathbf{b}|_y &= K\mathbf{b} \\ C_{\mathbf{bb}|_y} &= KC_{\mathbf{bb}}. \end{aligned} \tag{5.20}$$

Because the remaining inverse in (5.19) is scalar, computation of the clock synchronization estimates does not require a matrix inversion.

5.3 Simulations

The behavior of the probabilistic peer-to-peer clock synchronization algorithm is illustrated with three simulation experiments of a distributed system. The experiments cover the system's nominal behavior during internal and external synchronization, and the effect of a violation of the assumption that the communication delay of a network link is symmetric. The simulated system contains four nodes that are connected in a network as shown in figure 5.3. The granularity of each hardware clock is approximately 1 ms, corresponding to a tick rate of 1 MHz. This value is representative for the built-in counters that are found in most digital processing system. The stability of such hardware clocks is poor with respect to that of dedicated clock components. Where the latter typically achieve a stability of $1 \cdot 10^{-6}$, built-in counters can drift as much as one second per hour. Because the peer-to-peer synchronization technique aims to exploit the stochastic information on the clocks at all nodes, does not assign master and slave clocks, and does not expect that each node has access to an accurate clock, the typical stability of $3 \cdot 10^{-4}$ for a built-in counter is used in the simulations.

The logical clocks all start with an initial granularity estimate G_0 of 1 ms. The error of this estimate is indicated for each node in figure 5.3 by the ratio of the true granularity G and the initial estimate. The difference between the initial offset estimate L_0 and the true offset L is the initial time error. At start-up, the time is expected to be known with a precision amongst the nodes of 1 ms. The assumption of such a quasi-synchronized start-up only serves the visualization of the simulation results. Larger time differences between the nodes are eliminated at the first synchronization step, as long as the offset variance is properly modeled and counter values are small. The latter prerequisite minimizes the influence of the granularity estimates on the gross offset synchronization. Each logical clock should therefore reset its counter at start-up; this is automatically achieved by the inclusion of artificial counter rollovers as suggested in the previous section.

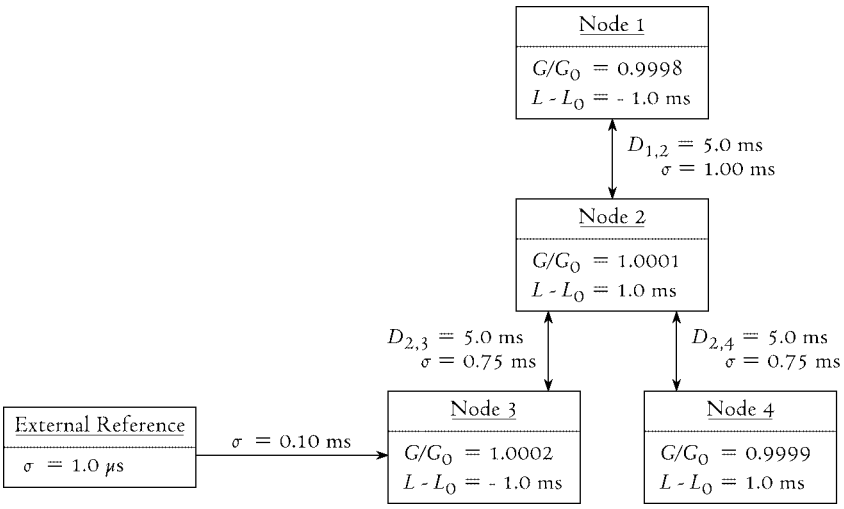
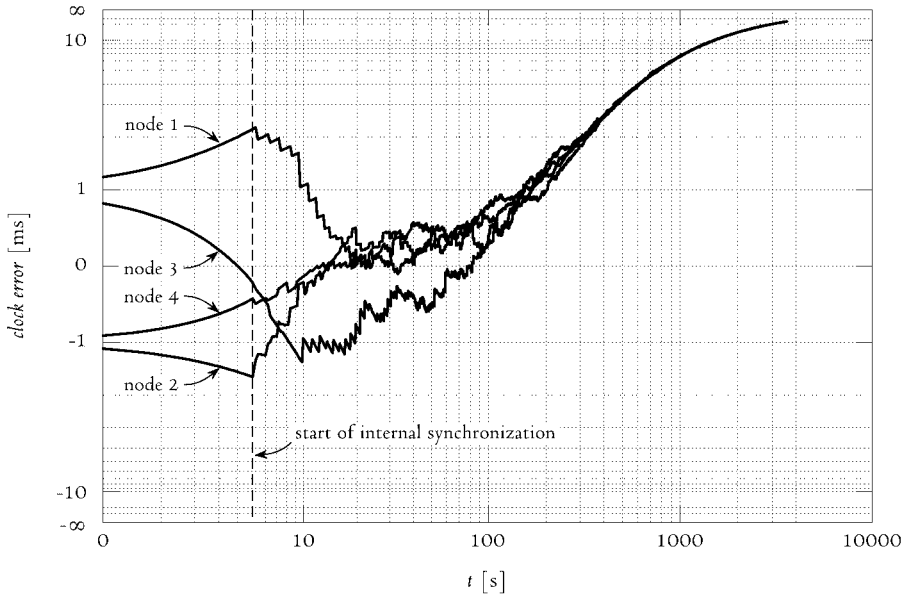


Figure 5.3: Topology of the simulated distributed system.
The four node objects each represent a logical clock. Node 1 is the parent of node 2; node 2 is the parent of nodes 3 and 4. Node 3 has access to an external reference time.

Figure 5.3 also shows the average and the standard deviation of the communication delay for each network link in the system. The external reference time is assumed to have comparatively stable access to node 3. The assumed standard deviation of 0.1 ms is a conservative estimate for a system where the messages from a reference time trigger an interrupt. The reference time itself is assumed to be accurate in the range of 1 ms, which corresponds to the accuracy that is achieved by a GPS receiver.

Internal synchronization

Figure 5.4a shows the total clock errors for each of the four nodes during an internally synchronized run without external synchronization. During the first five seconds, the nodes are not synchronized. The four indicated times run away linearly from each other and from real time due to the combination of the time error and the rate error on the initial granularity and offset estimates. When synchronization is started, mutual synchronization messages are exchanged between the directly connected nodes at a rate of one per second and the clocks rapidly converge to a common notion of time. Within 15 seconds, a precision of 1 ms is reached; a precision of 0.1 ms is achieved within 5 minutes. It is noteworthy that the steady-state precision of the time estimate amongst the nodes eventually becomes better than the precision of the communication delay between the nodes.



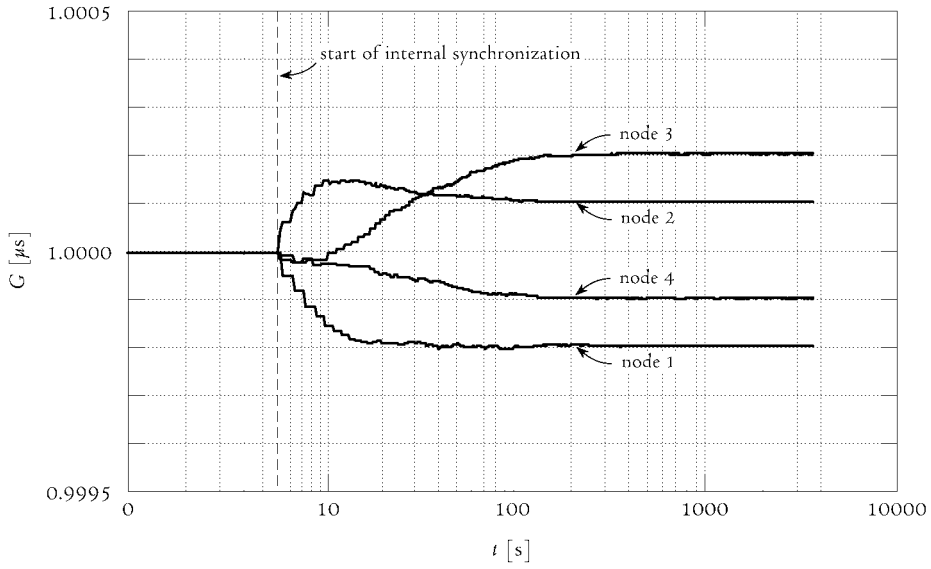
(a) development of total clock errors

Figure 5.4: Simulation results of internal synchronization.
a. The clock processes start 1 second before simulation begin. As indicated by the dashed line, internal synchronization starts 5 seconds after simulation begin. The converged time estimates run away linearly from real time. The curvature of the clock error is the result of the nonlinear scales in the diagram. (continued)

The diagram clearly shows that the joint time estimate from the four nodes is more accurate than each of the individual time estimates, which can be pictured as a continuation of the free-running time estimates before synchronization. In the absence of a reference time however, the common notion of time still drifts away from real time. The improved accuracy of the system of clocks is the result of the synchronization amongst peers without assigning master and slave clocks. The improved precision of the system of clocks when compared to the communication delay precision is primarily the result of the probabilistic approach.

Figure 5.4b shows the development of the granularity estimates for each of the four nodes. The steady-state values are close to the true granularities that are specified in figure 5.3. Internal synchronization guarantees that the clocks in the system indicate the same time, without relating the common time to real time. Once the clocks have synchronized, this means that the ratio of the various granularity estimates equals the corresponding ratio of true counter granularities. The proportional error of each gran-

ularity estimate with respect to the true value is therefore the same for all nodes. It determines the rate at which the system of clocks drifts away from real time. Nevertheless, the difference of each steady-state granularity estimate and the corresponding true value is smaller than the error for the initial estimate. The improvement of the granularity estimates increases with the number of nodes in the system and with a more symmetrical spacing of the initial values around the true clock rates. However, even with strongly asymmetric clock drift, the common notion of time represents an averaged value which removes the larger clock errors at the cost of increasing small ones. The ensemble of clocks is therefore guaranteed to be more accurate in a least-squares sense than the individual clocks without internal synchronization.

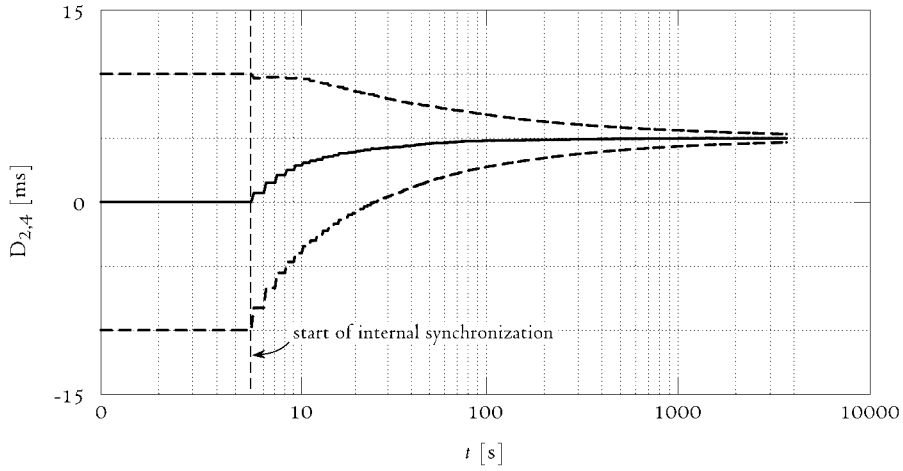


(b) development of granularity estimates

Figure 5.4, continued.
b. The granularity estimates converge from the initial estimate of 1 ms to the steady-state estimate that keeps the clocks synchronized as shown in (a). The ratio of the final estimates equals the ratio of the true granularities of the hardware clocks. (continued)

Figure 5.4c depicts the development of one of the three communication delay estimates in the system. The dashed lines indicate the auto covariance of the average delay estimate that is determined together with the delay estimate itself. In this simulation, the initial estimate was set to zero and the covariance was initialized at the square of 10 ms, representing the expected range of communication delays. Alternatively, the

delay may be initialized at a positive value with reduced initial variance, in order to indicate that negative communication delays are impossible. However, with the assumption of a Gaussian distribution, the possibility of a negative delay estimate cannot be eliminated.



(c) development of average delay estimate between nodes 2 and 4

Figure 5.4, continued.
c. The estimate for the average delay between node 2 and node 4 is initialized at zero with a 65% confidence bound of 10 ms. During synchronization, the delay estimate approaches the true symmetric value of 5 ms. The confidence bound is obtained as the square root of the corresponding variance; it decreases with continued synchronization.

Independent of the initial estimate, the synchronization procedure takes the estimate in approximately one hundred message exchanges to a steady-state value in direct vicinity of the true value of 5 ms. Unaffected by the remaining granularity and offset errors for the clocks, the communication delays are fully observable from within the system and are consequently estimated without bias, as long as they are expressed in terms of the granularity estimates. The latter form the basis for time measurement in the system; granularity bias as present in internally synchronized systems is therefore directly reflected in the delay estimates.

External synchronization

Figure 5.5 shows the development of the four total clock errors when node 3 is synchronized externally as indicated in figure 5.3. Especially in the time range between 10 and 100 seconds, the periodic corrections to the indicated time at node 3 are appar-

ent. Meanwhile, internal synchronization forces the other three nodes 1, 2, and 4 to converge to a common notion of time.

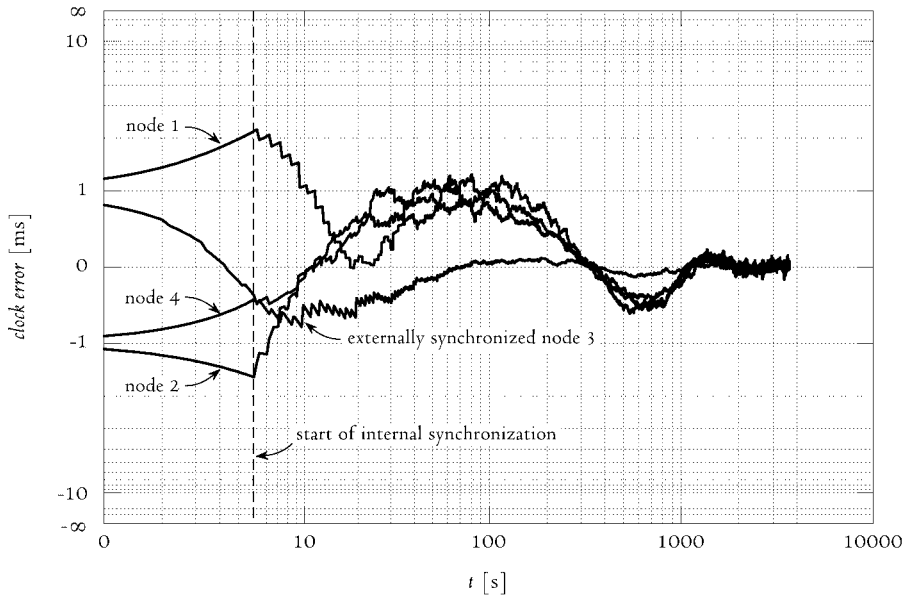


Figure 5.5: Simulation results of external synchronization.
Until internal synchronization starts, the development of the total clock errors for the three nodes that are not externally synchronized equals that in figure 5.4a. Node 3 is bound to real time; both its offset and clock rate are corrected at one update per second. Internal synchronization lets the other three nodes converge towards the more stable node 3. Once the nodes have synchronized internally, they are also in synchrony with real time.

The comparatively large discrepancy between the time for node 3 and the common time for the remaining nodes is an illustration of the trade-off between accuracy and precision during clock synchronization as mentioned in section 5.1 (page 133). The synchronization between nodes 1, 2, and 4 is not affected by additional clock adjustments from external synchronization. The corrections on node 3 lead to improved accuracy of the time estimate at the cost of reduced precision when compared to the neighboring nodes. External synchronization thus slows the internal synchronization. A stable precision amongst the nodes of 1.0 ms is only reached after approximately 100 seconds; 0.1 ms precision requires more than one thousand seconds of synchronization.

After 150 seconds, the externally synchronized system of clocks as a whole is more accurate than the internally synchronized system as shown in figure 5.4a. Because the externally synchronized node has converged to real time at that point, accuracy and

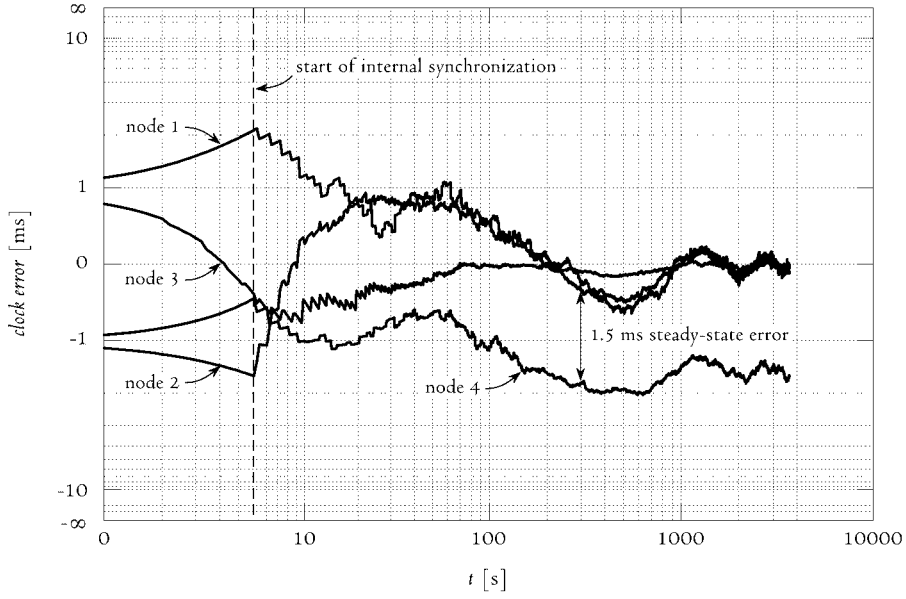
precision of the full system of clocks are more or less equal; both achieve a steady-state value around 0.1 ms. This value is directly related to the process noise that is added to the granularity variance in the logical clocks. If the process noise intensity is decreased, the clock rates become less dynamic and the steady-state precision and accuracy of the system of clocks is improved. However, the reduced mobility of the clock granularities has a damping effect on the adjustment of the clocks towards real time. The discrepancy between the externally synchronized node and the remaining nodes can therefore become larger and can persist for a longer time; the steady state is reached at a later time point. Conversely, an increase of process noise leads to improved convergence at the cost of a less stable steady-state time estimate. The optimal balance is found when the process noise matches the actual instability of the hardware clocks; this is the situation shown in figure 5.5. Within ten seconds from synchronization start, both accuracy and precision are in the range of the communication delay variance between the nodes. With the progress of time, the result is improved about a factor of ten.

Asymmetric communication delay

One of the assumptions that underlie the probabilistic peer-to-peer clock synchronization algorithm and that might prove difficult to satisfy in practice, is that of symmetric communication delays. Especially in the case of synchronizing a multi-platform system in which various processors cover a broad spectrum of computing performance, asymmetries can be introduced by the different response times of the nodes to incoming synchronization messages.

The effect of a violation of the communication symmetry assumption is demonstrated by the simulation that is shown in figure 5.6. The average downward delay for a message from node 2 to node 4 is increased to 8 ms; the upward delay from node 4 to node 2 is kept at the original value of 5 ms. The round-trip delay is therefore increased by 3 ms. Under the assumption that the communication delay is symmetric, this increase can be compensated by shifting the clocks on the two sides of the network link by 1.5 ms with respect to one another. This leads to an overestimation of the communication delay in one direction and an underestimation in the other; the sum of both delays however exactly matches the true round-trip delay.

Because the delay from node 2 to node 4 is larger than the delay from node 4 to node 2, the downward delay must be underestimated and the upward delay must be overestimated in order to obtain a symmetric estimate of the average delay. This is achieved when the time that is indicated at node 4 is behind the time indicated at node 2. The steady-state offset of 1.5 ms with node 4 running behind node 2 is clearly seen in figure 5.6a. Because node 2 is connected to the externally synchronized node 3, the relative error of 1.5 ms completely appears as an absolute time bias on node 4. The distribution of the total asymmetry of 3 ms over the clock errors is confirmed in figure 5.6b. The average delay estimate converges to the mean value of the upward and the downward delays; the respective error is half of the total asymmetry.

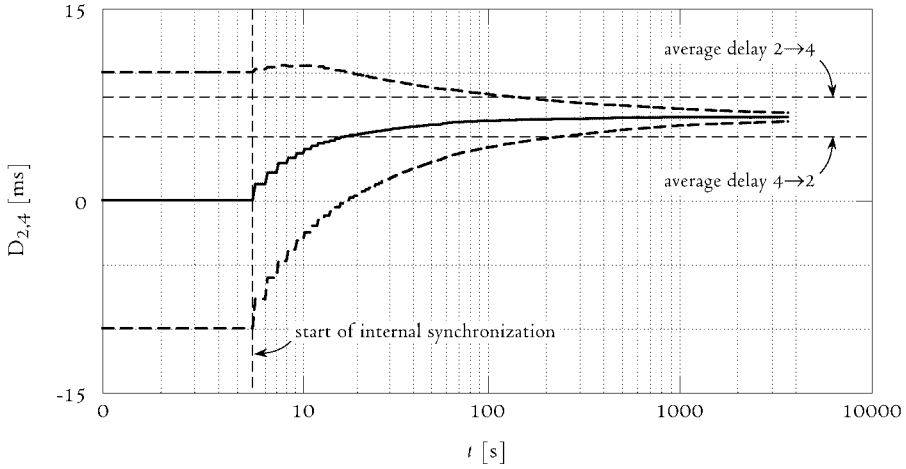


(a) development of total clock errors

Figure 5.6: Simulation results of external synchronization with communication delay asymmetry between nodes 2 and 4 of 3 ms.
a. The skewed sending and receipt of synchronization messages between nodes 2 and 4 results in a steady-state clock error of 1.5 ms at the far side from the node that is externally synchronized. (continued)

5.4 Activities in the middleware

The function of a logical clock, providing an optimal estimate of real time to the pacer, requires the clock to perform a number of tasks. Converting the physical clock counter into the time estimate is the easiest. This completely passive operation – performed upon request from the pacer – involves nothing more than the evaluation of the linear equation (5.1). The other tasks are internal synchronization with the other nodes in the application, and external synchronization if a module with access to a reference time is available. These synchronization tasks are implemented in the logical clock fiber that runs in the idle time of the pacer fiber; the activities for the latter were shown in figure 4.7 (page 124). As was discussed in section 4.6, clock synchronization consists of only wait states and actions. The actual computation time that is



(**b**) development of average delay estimate between nodes 2 and 4

Figure 5.6, continued.
b. The estimated delay between nodes 2 and 4 converges to the average of the true upward and downward delays.

required by the algorithm is negligible in comparison with the communication delays that arise during message exchange between internally synchronized nodes. Each time such an exchange occurs, the logical clocks on both sides of the communication link must enter a wait state. The activities in clock synchronization are therefore a system of wait states with actions at the state transitions; each action forms a part of the synchronization algorithm. The complete procedure consists of a round trip through various wait states with all intermediate actions.

States and actions

A logical clock in a distributed application regularly performs internal synchronization steps with each of the adjacent nodes, during which adjustments to both its own clock and that of the remote node are estimated. Because the adjustments for the remote clock need to be communicated to the parent or child node where the clock is located, a synchronization step involves a challenge-response type of communication. Using the fact that all nodes are organized in a network tree, the interaction is initiated by the parent node. It reads its hardware clock and sends the counter value that is obtained to the child node. In the same message, the parent node also reveals its clock model parameters and the associated stochastic accuracies. Upon receipt of the message, the child node reads its own hardware clock. It now has available all clock parameters and accuracies, as well as the counter values for both clocks. If the child node keeps track

of the estimated time delay between the two nodes, all data for performing a Bayesian clock parameter estimation as presented in section 5.1 are available. Afterwards, the child node sends a return message to the parent, in which the parent is notified of its updated clock parameters and the common communication delay.

Thus, unlike most challenge-response communications, the key action is not performed at the arrival of the response message, but at receipt of the initial message at the child node. The response message is only used to notify the initiating node of the changes to its estimated clock parameters. As a result, communication delays cannot be estimated from a single synchronization step: No round trip information is available when the computations are made. The algorithm therefore only functions correctly when the procedure is applied in both directions. Every synchronization step that is initiated by the parent and performed by the child node, must be followed by a step that is initiated by the child and performed by the parent. The result is a bouncing series of synchronization messages between two nodes. There is no need for the synchronization messages between the nodes to follow up the previous message without pause, nor should the pause in both directions be the same. A practical procedure is therefore to have the parent node initiate a double step at a more or less fixed rate. The child node performs the first synchronization and responds with the return message and the converse synchronization request. The parent node then performs the second synchronization and returns the results to the child in conclusion.

Since external synchronization does not involve the estimation of a time delay, the procedure for binding a node to a reference time is simpler. When a message with a time observation is received from a local module that has access to a reference time, the logical clock only has to read the physical clock in order to be able to compute the Bayesian clock parameter estimate. Because the synchronization is completely unilateral – the reference time is not adjusted to the local time estimate – there is neither a converse synchronization step nor a response message.

The states and actions of the clock synchronization fiber are shown in figure 5.7. The initial state of the fiber is the idle state. It is left when one of three events has occurred: the pipe thread that communicates with the parent node has received an initiating synchronization message, the thread of an active module has received an external synchronization message, or a preset interval for the synchronization with one of the child nodes has elapsed. In the latter case, the synchronization step is initiated by reading the local physical clock and sending the initial message through the pipe for the child node. The local fiber then waits for the response from the child with the results from the first synchronization step, subsequently performs the converse step, and sends the terminating message. At the child node, the first synchronization step is performed in response to the receipt of the initial message from the parent. Subsequently, the fiber waits for the parent's terminating message and stores the results. An external synchronization involves only the computation of the local Bayesian parameter estimation.

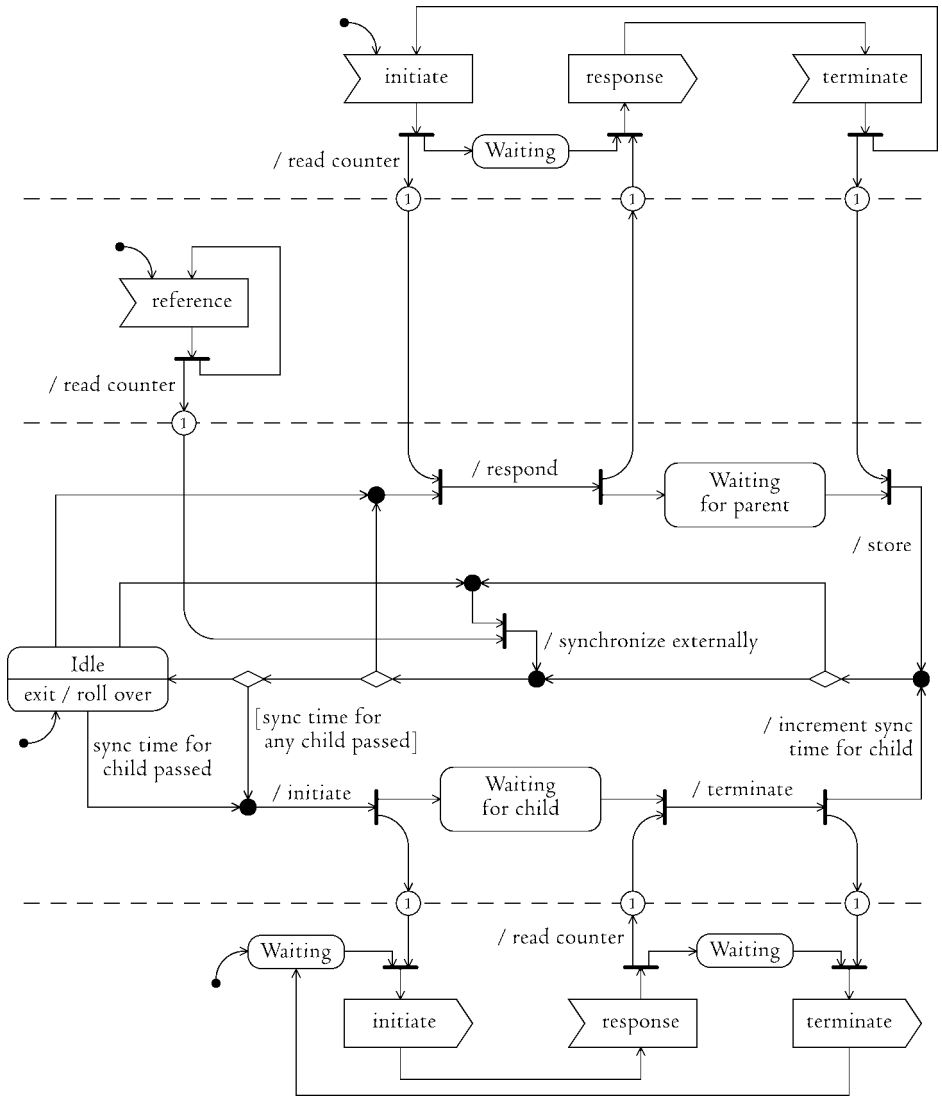


Figure 5.7: Clock synchronization activities.
The synchronization fiber in the pacer thread interacts with three other thread types that run concurrently. The threads shown at the top and at the bottom are the pipe threads for communication with the parent node and a single child node respectively. Second from the top is the thread for a module with access to an external reference time. The synchronization fiber has only three non-instantaneous states: idle, waiting for parent, and waiting for child. State transitions are mainly triggered by synchronization pseudostates with the supporting threads.

Message sequence

From the logical clock fiber, synchronization with each of the child modules is initiated and incoming messages from the parent and any number of externally synchronizing nodes are handled. The sequence of messages and state transitions for both internal and external synchronization is depicted in figure 5.8. It shows how the counter observations are performed synchronously by the active threads of the pipe ends and an externally synchronizing module, immediately before an asynchronous message is sent to an object in a different thread.

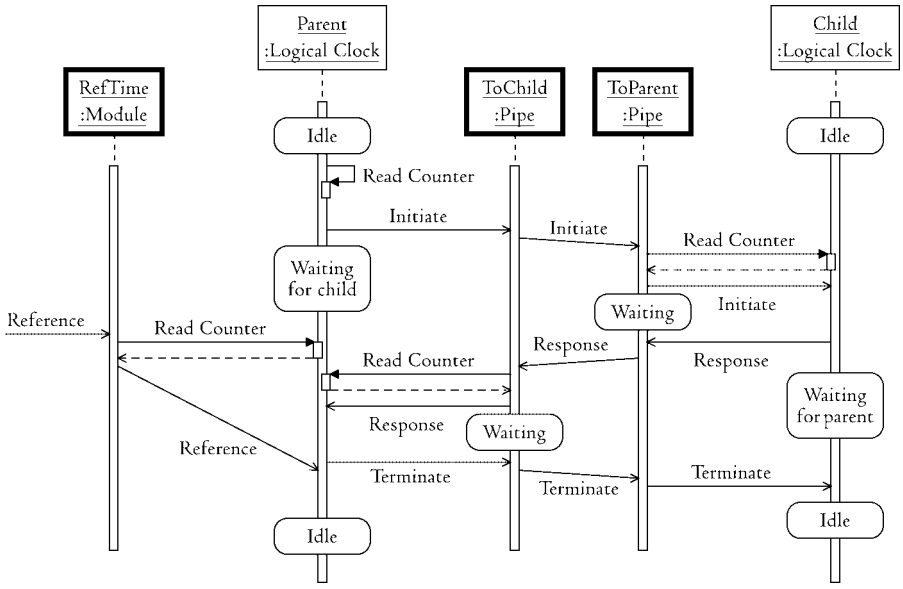


Figure 5.8: Clock synchronization message and state sequence.
A complete two-way synchronization step between a parent and a child node starts with the reading of the parent's hardware clock and ends with the receipt of the terminate message by the child. Apart from the counter observations, all messages are asynchronous. The receipt of an external reference time by the parent node is stacked until the logical clock completes the synchronization with the child node.

The incoming messages are processed sequentially; one synchronization cannot be preempted by another. Internal synchronization between nodes involves the adjustment of clock parameters for both nodes by the process running at one of them. Consequently, the other node temporarily has to hand over its clock parameters and must refrain from changing their values locally until the updated values are returned from the node performing the synchronization. Therefore, a node cannot handle more than one internal or external synchronization simultaneously. However, there is no objec-

tion to temporarily stacking a synchronization request and processing it when a running synchronization has been completed. As long as hardware counter values are read at the correct epoch and are stored along with the pending synchronization, stacking synchronizations does not affect their outcomes. The latter is reflected in the location of the actions that read the local counter at the receipt of a message. By performing them in the thread that deals with the communication rather than in the synchronization fiber, a time-correct observation of the counter upon arrival of the message is ensured irrespective of the state of the synchronization fiber.

When multiple events occur during the processing of a synchronization sequence – for example the arrival of both an external synchronization message from a module and an internal synchronization message from the parent node – an ambiguity arises for the transition from the idle state. This is avoided by prioritizing the servicing of the concurrent threads, in which external synchronization precedes a requests from the parent node; initiation of a synchronization sequence with a child node has lowest priority. The prioritization is implemented as a number of decision pseudostates along the transition back to the idle state. External synchronization only consists of an action. Because it does not involve a blocking state, external request are processed with highest priority. Synchronization with a child node can be initiated at any time that suits the local node; it is therefore postponed until all synchronizations with the parent node and external reference times have been completed. The practice of stacking synchronization requests is the main reason for initiating internal synchronization from the parent node. Because each node cannot have more than one parent and because external synchronization is addressed with highest priority, the number of waiting synchronizations is minimized. The assignment of parents and children in the system of clocks is therefore purely practical; it does not imply a master-slave relation between any pair of clocks.

6
Case Study in Human-Factors Testing

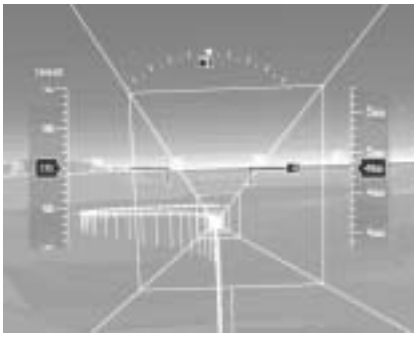
The in-flight assessment of a tunnel-in-the-sky synthetic vision display is a characteristic application of a human-factors testing system. Summer 2001, the Faculty of Aerospace Engineering at the Delft University of Technology performed a flight test program as the first of a series of experiments on the cybernetics of a tunnel-in-the-sky display in actual flight. The flight test instrumentation system for the program was developed according to the methodology that is presented in this thesis.

The application is analyzed and designed in steps, starting with a signal diagram that models the instrumentation requirements, producing a context model and several design views of the application, and ending with an integration design that shows the installation of the system in the aircraft. The components are individually modeled, implemented, and tested before the application is synthesized in the laboratory and integrated in the aircraft. In preparation of the Delft University flight tests, a throwaway prototype was created for a new hardware component; unit testing revealed an implementation flaw that could be corrected easily. Based on pilot comments, the symbology of the tunnel-in-the-sky display was optimized during application synthesis. The system was then successfully integrated and used in flight test.

THE distinctive advantage of a DSP-based flight test instrumentation system lies in its applicability to closed-loop flight testing. Traditional data acquisition systems cannot feed back processing results to the system under test in real time. At the center of the closed-loop system, one or more digital signal processors are required to collect the sensed information from the aircraft, the pilot, and the environment, to perform the computations, and to deliver the results to the excitation components. If the pilot is in the loop or if the closed loop involves the dynamic behavior of the aircraft, the real-time demands on the systems are extremely stringent: The deadlines must be such that latencies are noticeable to the pilot and do not qualitatively affect the dynamic response of the aircraft. Human-factors testing systems and in-flight simulation systems as defined in chapter 2 are therefore among the most demanding flight test instrumentation systems.

The in-flight assessment of a synthetic-vision flight guidance display is a typical example of a closed-loop flight test with the pilot and the aircraft in the loop. Mulder (1999, p. 14) identifies the *tunnel-in-the-sky display* as a prominent candidate for becoming the primary flight display of future aircraft. The tunnel display is a perspective flight path display that presents the pilot with the primary information for guiding the aircraft in a single image. The image contains a projection of the desired flight trajectory and the aircraft's current three-dimensional orientation and position. To flight-test a tunnel-in-the-sky display, the test aircraft is equipped with an experimental display, several sensors and processors that produce the display imagery, sensors that acquire the pilot's inputs to the aircraft inceptors to allow evaluation of the pilot's control behavior, and a data recorder. The instrumentation system thus closes the loop from the aircraft's dynamic response to the image that is presented to the pilot.

As yet, human-factors research into tunnel-in-the-sky displays has mainly been conducted using simulated environments. Assessment of a tunnel display in true flight has been limited to a few trials (Theunissen, 1997; Alter et al., 1998; Funabiki et al., 1999; Barrows and Powell, 2000; Sachs and Sperl, 2001) that focussed on demonstrating the concept feasibility. The pioneering nature of the flight tests with a tunnel display resulted in an emphasis on the technical aspects of implementing the display, and on the pilot's basic capability to follow the indicated flight path. The cybernetics



Tunnel-in-the-sky display as used by Mulder (1999).
The tunnel display is a perspective flight path display that presents the pilot with a single three-dimensional image of the desired flight trajectory and the aircraft's actual orientation and position. As in this case, the image can be enhanced by airspeed and altitude information and synthetic visualization of the outside world, including terrain characteristics or traffic in the vicinity of the aircraft.

of tunnel-in-the-sky displays as studied by Mulder (1999) on a fixed-base simulator, have not yet received the same amount of attention in flight test. To initiate fundamental research on the human-factor aspects of the tunnel display in real flight, a flight test program was successfully conducted by the Faculty of Aerospace Engineering at the Delft University of Technology during the summer of 2001 (Mulder, Kraeger, and Soijer, 2002). The flight test instrumentation system for these experiments was developed according to the methodology of this thesis; some implementation details on the project are summarized in appendix D.

6.1 Application modeling

Application modeling – covering both analysis and design – starts with the classification of the application as an open-loop testing, an adaptive testing, a human-factors testing, or an in-flight simulation system. The in-flight assessment of a tunnel-in-the-sky display is a closed-loop application in which a single loop closure is achieved by feeding back information from the response of the aircraft to the information that is presented to the pilot. The pilot controls the aircraft by means of the standard flight control system; the instrumentation system does not excite the aircraft directly. In line with the naming convention of the various archetypes, the application is therefore a case of human-factors testing as described on page 48. The remaining activities in application modeling are the development of a signal model, a context model, and static and dynamic models for the incremental design.

Requirements analysis: the signal model

The signal model records the various requirements for the application in the form of a description of all the signals in the system, starting from the external inputs and ending with the system outputs. The model is best constructed from the back. Back and front are identified by opening the closed-loop application at the location where the signals are external to the instrumentation system: at the pilot and the airframe. The output of the instrumentation system is therefore the image that is presented to the pilot; the inputs are determined by analyzing the process of constructing that image. In an alternative approach, the data that is recorded for post-flight processing can be regarded as the output of the flight test. Pilot and aircraft would then be part of the chain of components that is the source of these signals. The latter approach is less general than the former, because it cannot be used for human-factors testing applications without data recording for which the final output is the pilot's opinion. However, the second approach is necessary for signals that are required for post-flight analysis, but that are not a part of the closed loop.

The signal diagram for the in-flight assessment of a tunnel-in-the-sky display is shown in figure 6.1. The diagram contains both types of output signals: those that are found at the artificial opening of the closed-loop application, and those that are not part of the closed loop, but that are required for post-flight analysis. By opening the application loop at the point where the instrumentation system sends one or more signals to the pilot, the image that is seen by the pilot is identified as the primary output signal. As a continuous visual cue, typical signal requirements like accuracy or update rate do not apply. An important requirement on the signal is its brightness, which should allow to see the image in the cockpit at all weather conditions. The minimal brightness of 250 cd/m^2 is specified as a UML constraint. Secondary to the closed-loop signal, the performance of the pilot must be analyzed after the flight test and must be compared with the pilot comments on handling qualities and workload. The data recorder is therefore an important end point of signals as well. The signals that are deemed necessary for post-flight analysis are the geometric position and orientation of the aircraft, the barometric altitude and the airspeed, and the pilot's control activity. The latter is represented by the control surface position of the elevator, the rudder, and the aileron. Each of these is required as a discrete, digital signal at an update rate of at least 10 Hz; the accuracy of a surface deflection angle must be 0.1°. These requirements are indicated along the signal lines. The other parameters that must be recorded are not subject to requirements for update rate or accuracy. The requirements that will follow from the use of these signals in the closed loop – during the construction of the display image – are deemed suitable for post-flight analysis as well. These requirements are driven by the expected characteristics of the experimental cockpit display.

The visual cue that is produced by the display is a continuous, analog signal. The display receives a discrete description of the image from the image generator. The latter is a typical example of a data processing component that combines hardware and

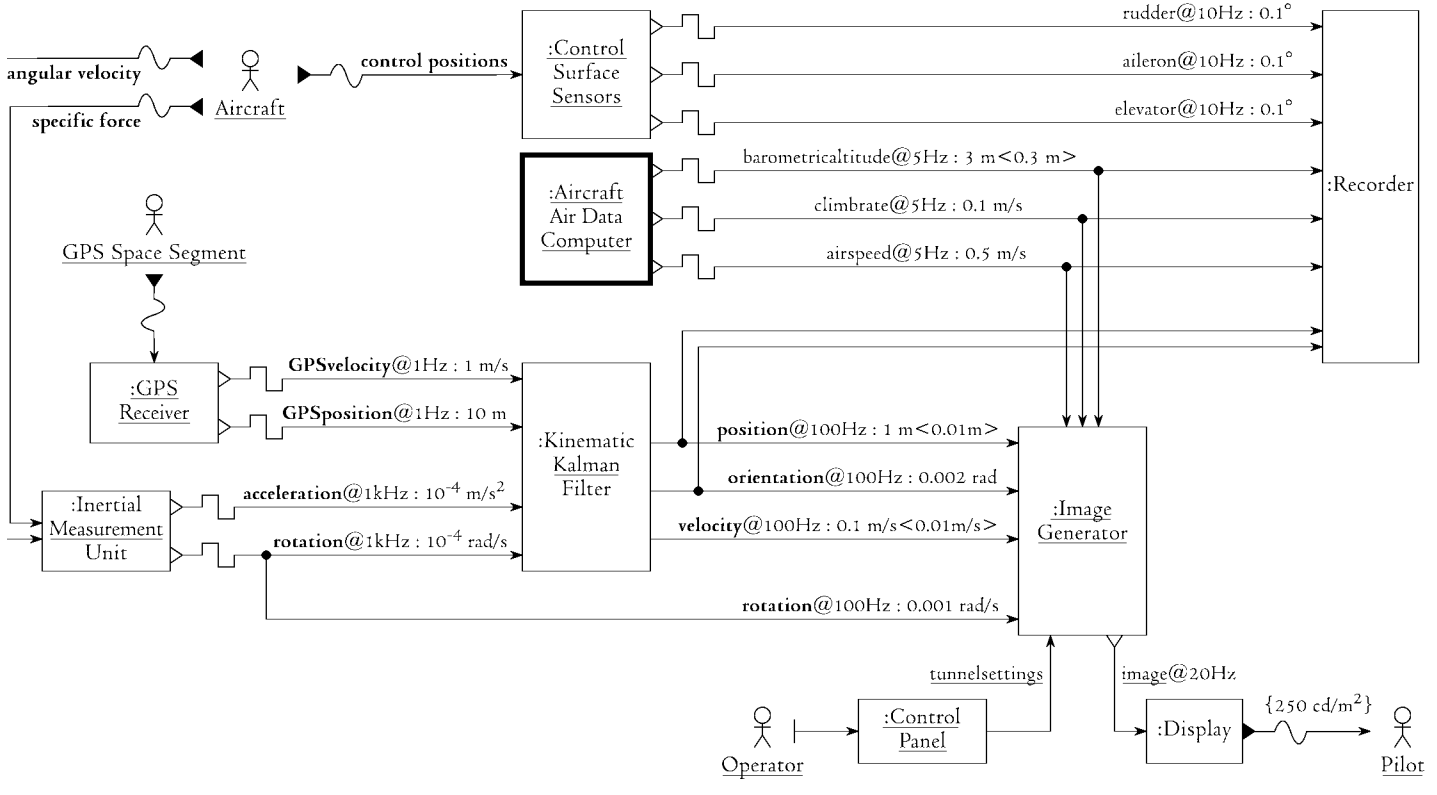


Figure 6.1: Signals for tunnel-in-the-sky assessment.
The tunnel image as seen by the pilot is the primary output signal; the three control surface positions and the aircraft's position and orientation are the secondary outputs, required for post-flight analysis. The instrumentation components deliver these signals ultimately from external inputs that stem from the aircraft, its air data computer, the system operator, or the GPS space segment.

software. In the signal diagram, the signal generator serves as an operator that converts the positional, orientational, and dynamic data of the aircraft into the image that is sent to the display. It thus includes the software that creates the graphics of the tunnel-in-the-sky and the image generator hardware that produces the video signal. The update rate of the image is specified at 20 Hz. This ensures clear separation in bandwidth of the aircraft and the display dynamics. Display delays up to 250 ms do not have a noticeable effect on the flying qualities of a fighter aircraft (Bailey, 1989), a result which is assumed to hold for transport aircraft as well. With twenty frames per second, the tunnel-in-the-sky image will be updated every 50 ms. When the additional delay in the hardware part of the display – which has a refresh rate between 60 and 80 Hz – is taken into account, the average delay of the image will equal approximately 27 ms, which is well below the critical value.

The identification of the discrete image signal that is sent from the image generator to the display, is the first iteration step in the signal modeling of the tunnel-in-the-sky application. A signal that is invisible outside the instrumentation system is identified and its requirements are specified, based on the needs of the operators that have been recorded earlier. This procedure is repeated until the signal diagram is complete. The image generator is the most complex operator in the application. It must create the tunnel-in-the-sky graphic with all desired symbology. The basic information that is required are the aircraft position and orientation. They determine the synthetic vision image itself, assuming that the exact location and shape of the tunnel is stored with the image generator. Because no significant delays should be introduced between the aircraft dynamics signals and the display image, the position and orientation are required at a rate of 100 Hz. Their accuracies are based on the navigation performance that is required during a tunnel-in-the-sky approach; in this application, a positional accuracy of 1 m and an orientational accuracy of 2 mrad (0.1°) are chosen. For the position, an additional precision is specified. The required precision is based on the resolution of the tunnel display: A remaining uncertainty of 1 cm does not cause a significant change to the tunnel image. This guarantees a stable graphic that is insensitive to the noise on the navigational data.

In addition to the primary navigational data, the image generator requires five signals that are used to create the symbology on the display. The aircraft velocity and rotation vector are used to create a flight path vector symbol in the tunnel display. The requirements for these signals are similar to those on the primary navigational data. The barometric altitude, the rate of climb, and the indicated airspeed of the aircraft are used to create to tape-like symbols at the side of the tunnel display. They are secondary to the pilot and hardly influence the handling qualities of the aircraft during a tunnel approach. Therefore, the bandwidth and accuracy requirements for these signals are an order of magnitude less stringent than on the other parameters. Finally, the image generator requires a set of operator settings. Because the tunnel-in-the-sky display is experimental, many of its parameters are adjustable. In this application, the parameters that can be adjusted during the experiment include the steepness of the approach, a

selection of additional symbology, the interval over which the flight path vector is predicted ahead, and anti-clutter settings of the display. These parameters are provided by an operator using a control panel. The settings are sent to the image generator in an application-specific format, which is indicated in the diagram by underlining the signal name. The control panel transmits a package of settings to the image generator whenever the operator makes a change. The control panel thus receives irregular input from one source. By including the operator as an actor in the signal diagram and by indicating the aperiodic nature of his inputs with the barred signal arc, this path of the signal flow has been completed.

The navigational input to the image generator is produced by a Kalman filter for aircraft kinematics. It fuses inertial navigation with GPS data. The requirements for the GPS position and velocity vector signals and the inertial acceleration and rotation vector signals are derived from the requirements for the Kalman filter's output. The rotation vector is the same signal that is used by the image generator. However, the requirements for the Kalman filter are more stringent. This is a clear demonstration of the signal diagram as a model to specify signal requirements. The diagram shows the double use of the rotation signal, together with the respective requirements. The effect of any change to the application is easily identified and the appropriate resulting adaptations can be incorporated. When for example the Kalman filter is removed from the application because a different source of navigational information will be used, it is immediately clear from the signal diagram that the requirements for the rotation vector signal can be relaxed, but that it cannot be removed altogether. The GPS and acceleration signals would disappear completely.

At this stage, the signals that are the front of the signal diagram all stem from data acquisition components. The control surface deflections are measured by sensors that are physically connected to the controls; barometric altitude and indicated airspeed are obtained from the aircraft's air data computer; GPS position and velocity are acquired by a GPS receiver and an inertial measurement unit is used to obtain the aircraft's acceleration† and rotation. The air data computer is regarded as an autonomous system that is external to the instrumentation. It is therefore indicated as an active component in the signal diagram. A GPS receiver acquires signals from the satellites that are jointly referred to as the space segment of GPS. The remaining data acquisition components receive their input directly from the aircraft. With these signals, the signal diagram is complete. For each of the instrumentation system's operators, all the input signals are indicated; each of the signals stems either from another instrumentation component, or from an actor that is external to the application.

† Although an inertial measurement unit acquires specific force rather than acceleration, the term acceleration is used in harmony with the name 'accelerometer' for the corresponding sensor. The effect of gravity is compensated for by the Kalman filter.

Context analysis

For a human-factors testing application, context analysis is limited to the development of a context model in which the interfaces between the application and the environment are worked out in more detail than in the signal model. The context model is based on the archetype that is shown in figure 2.4 (page 56). Centered around the pilot and the aircraft, the operators that are at the system boundary as identified in the signal diagram of figure 6.1 are brought into the context model. The function of each operator is analyzed in terms of the internal and external signals that it connects. Most often, this leads to a differentiation of the operator into multiple system components.

A prerequisite for context analysis is detailed knowledge of the application environment. Only when the exact characteristics of the external systems are known, the signals that cross the application boundary are known and the interface components can be specified. A generic designation like 'aircraft' in figure 6.1 is inadequate. For example, the control position vector varies for aircraft with different configurations of control surfaces; the type of sensor that must be used to acquire the surface positions also depends on the type of aircraft. Therefore, at this point in the development of the flight test instrumentation system, the platform that will be used for the tests must be selected if it is not prescribed by the flight test itself.

The tunnel-in-the-sky assessment flight tests have been conducted with the Cessna Citation II laboratory aircraft that is jointly operated by the Delft University of Technology and the National Aerospace Laboratory in Amsterdam. The aircraft has a flight envelope that is representative for civil transport aircraft, making it a suitable platform to assess the handling qualities for an operational environment of modern airliners. The Cessna Citation II has a conventional, unaugmented flight control system with a single rudder, a single elevator, and a pair of coupled ailerons. The aircraft is normally operated by two pilots, but flight deck layout provides for single-pilot operation. This is an important aspect for the tunnel-in-the-sky assessment experiments. The tests are flown from the right-hand seat, using the experimental display in the instrument panel. A safety pilot in the left-hand seat monitors the experiment using the standard aircraft instruments. As single-pilot operation is possible from the left-hand seat, the safety pilot can take control anytime without difficulty.

The selection of the Cessna Citation II leads to the context model as shown in figure 6.2. The Citation's conventional flight control system means that surface positions cannot be acquired from a databus or from an electric signal at an actuator. Instead, they must be measured from the actual surface rotation by means of synchros. Because the two elevator surfaces and the two aileron surfaces are mechanically coupled, only a single deflection is measured for each. With the measurement of rudder deflection, the aircraft must therefore be equipped with three synchros. The typical electric two-input/three-output interface of a synchro is not compatible with the digital data recorder, so each synchro output must be connected to a dedicated interface.

The Cessna Citation II has a digital air data computer that provides indicated airspeed, barometric altitude – subject to the altimeter setting at the flight deck – and

Delft University-operated Cessna Citation II (edited; original photograph National Center for Atmospheric Research, USA).
The Delft University and the National Aerospace Laboratory (NLR) jointly operate a Cessna Citation II twin-engine business jet as a research and education platform. The Citation II carries a maximum payload of 1400 kg up to altitudes of 13 km with an endurance of 2h30. The laboratory aircraft provides for flexible cabin layout, instrumentation racks, and high-current instrumentation power.

rate of climb over an ARINC 429 databus. A corresponding databus interface must therefore be included in order to acquire the altitude and airspeed information that is used by the image generator. In addition to these three parameters, the digital air data computer provides Mach number and temperature measurements over the same databus. Although these are not required for the current flight test, they can be recorded if the interface is developed in a way that it recognizes the additional parameters as well.

The inertial measurement unit that is indicated in figure 6.1 must acquire specific force and angular velocity. Since the Delft University aircraft is not equipped with an inertial navigation system, these signals cannot be obtained from the aircraft directly. Therefore, a proprietary three-axis accelerometer and three-axis rate sensor are included as interfaces in the context model. The inertial sensors must be attached to the aircraft, which is indicated in the diagram by the connecting arc. This illustrates the use of context modeling to identify any kind of association between components. The strapdown installation of inertial sensors to the airframe differs from the installation of other instrumentation components like the interfaces or the data recorder. For the inertial sensors, the rigidity and the exact location of the installation play an important role; this importance is signalled by the inclusion of the installation in the context model. Additional information, such as requirements for the accuracy of the

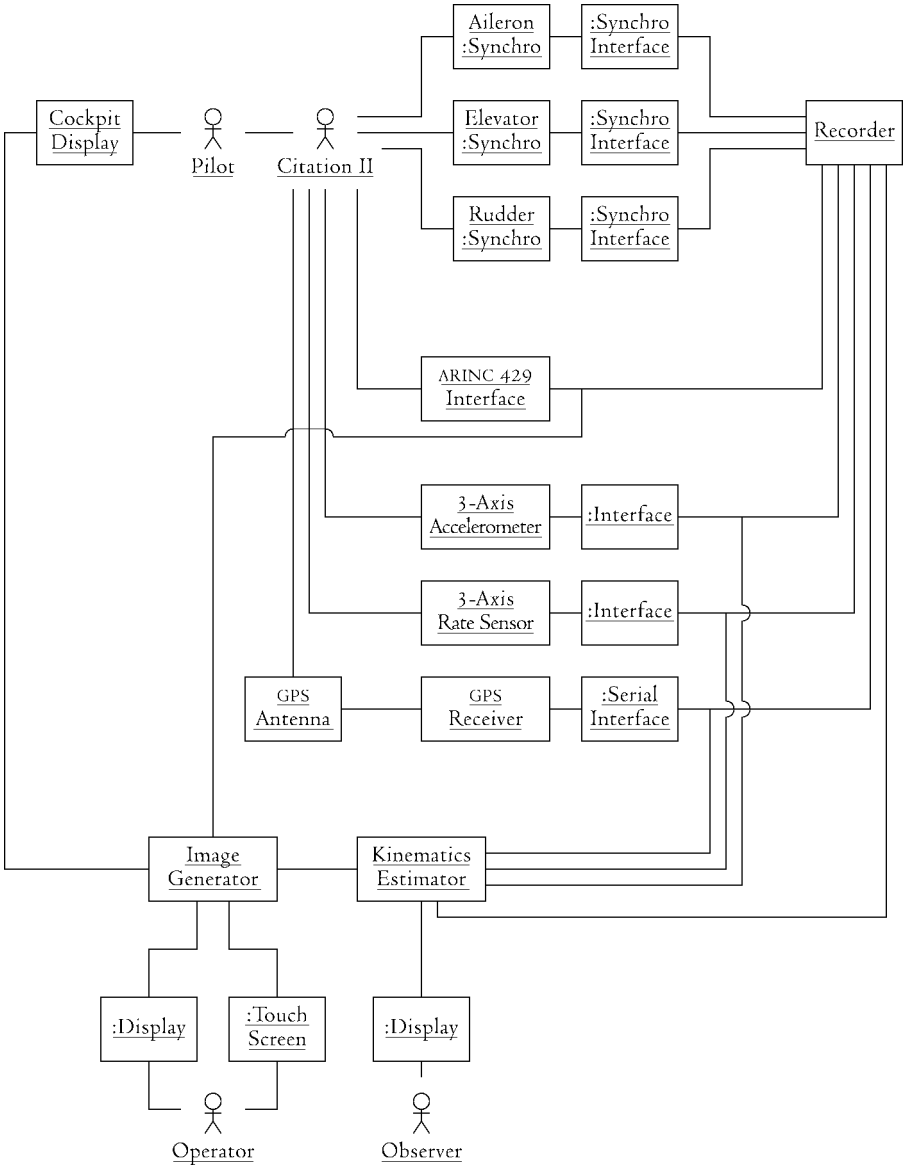
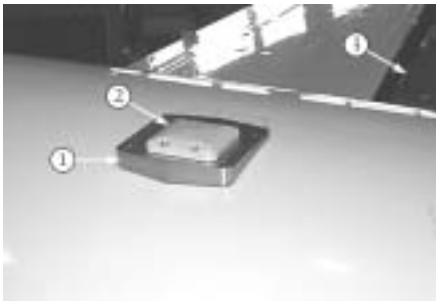


Figure 6.2: System context for tunnel-in-the-sky assessment.
The object diagram emphasizes the interfaces to the environment, which consists of the aircraft, the pilot, the operator, and the observer. Interaction with the three human actors occurs through displays and a touch screen; interaction with the aircraft is either electrical or physical. The latter indicates that sensors must be attached to the airframe in a specific position in order to function correctly.

location and the alignment of the installation, can be marked in the model as constraints or as notes.

Similar to the installation of the inertial sensors, the installation of the antenna for the GPS receiver is important to its function in the application. Both the relative position of the antenna with respect to the inertial measurement unit, and the field-of-view for the antenna on the outer airframe surface – avoiding areas where parts of the airframe block or reflect the satellite transmissions – need to be considered when the instrumentation GPS antenna is located on the aircraft. An instrumentation GPS receiver and antenna are necessary because the laboratory aircraft does not provide a standard receiver. Serial interfaces have become the standard for communicating with GPS receivers; the instrumentation system must therefore be equipped with a serial interface as well.



GPS antenna on top of the laboratory aircraft fuselage.
The mechanical interface (1) aligns the off-center antenna (2) with the aircraft *xy*-plane. Located approximately at the wing's leading edge (3), the antenna is clear of the fin and the stabilizer.

The man-machine interfaces in the application are the tunnel display for the test pilot and the control panel for the system operator. The dominant characteristic of the tunnel display as an instrumentation component, is the fact that it must be located in the right-hand instrument panel at the flight deck. The control panel for the system operator serves the combined purpose of showing the tunnel image in the cabin, showing all the switches and parameters that determine the appearance of the tunnel image, and receiving the operator's input to change these settings. The control panel therefore consists of two interfaces to the operator: the display and a touch screen. Finally, an interface is added with respect to the signal model of figure 6.1. The state of the kinematic Kalman filter that produces the navigational data for the image generator should be visible during the experiments. Hence, an additional observer display is directly connected to the filter.

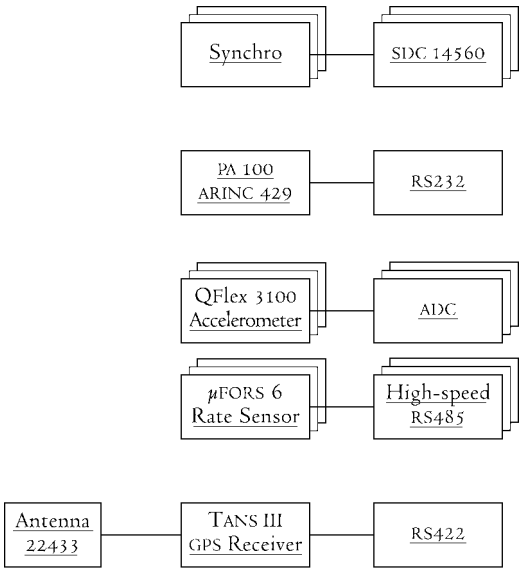
Incremental design: static models

Following context analysis, application design takes the model from the problem to the solution domain. In close cooperation between the flight test engineer and the instrumentation engineer, devices are selected that meet the criteria of the signal model, platforms are composed that can host the data acquisition and data processing modules, and an architectural design is made, including the network for a distributed application. Due to the increasing amount of detail, the size of the models grows rapidly during application design. The application design should therefore not be recorded in a single diagram. Instead, it is desirable to create several diagrams that focus on different aspects of the design, or that capture a subset of the components. The latter also suits the incremental development life cycle; as long as the available subset forms a functional application, component development and application synthesis can benefit from a stepwise approach.

In the application for assessing a tunnel-in-the-sky display, an incremental approach can be applied to the open-loop and closed-loop parts of the application. It is possible to develop the tunnel application without addressing the recording of pilot control activity; measuring the control surface deflections and storing them for post-flight analysis can be seen as an independent application. However, the latter part is small in comparison with the closed-loop part. The application is therefore developed in a single increment.

The resulting application design is shown in figure 6.3. Figure 6.3a contains a subset of the components from the context model of figure 6.2: Only the interface components with the aircraft and the corresponding instrumentation-side interfaces are shown. These are the devices and ports of the application respectively. The design is started by specifying the exact type of each device. In this case, synchros, an ARINC bus receiver, accelerometers, rate sensors, and a GPS receiver with antenna need to be selected. Similarly, specific devices are identified for the other interfaces to the application: the displays and the touch screen. Using the manufacturer's documentation on the devices, the port units that will be required to connect the devices to the instrumentation system are added to the model. Connecting the synchros requires a proprietary port that is based on synchro-digital converters (SDCs); the other devices are connected through either a standard serial interface, or an analog-digital converter (ADC). For the displays, the port unit consists of a graphics processor/video card.

Figure 6.3b shows the port units again, but this time in combination with the appropriate platform component. The processing unit for the acquisition platform is a dSPACE DS1003 real-time processor board that provides a dedicated high-speed databus for connecting various input and output interface boards in combination with an extensive prioritized interrupt system. Standard boards for serial interfaces and analog-digital conversion are available from the processor board manufacturer. In addition, custom interfaces can be developed using a standard board that provides all the electronics for accessing the high-speed real-time bus. Using the possibility to develop a custom board for the synchro interface, the DS1003 can be used to acquire all the

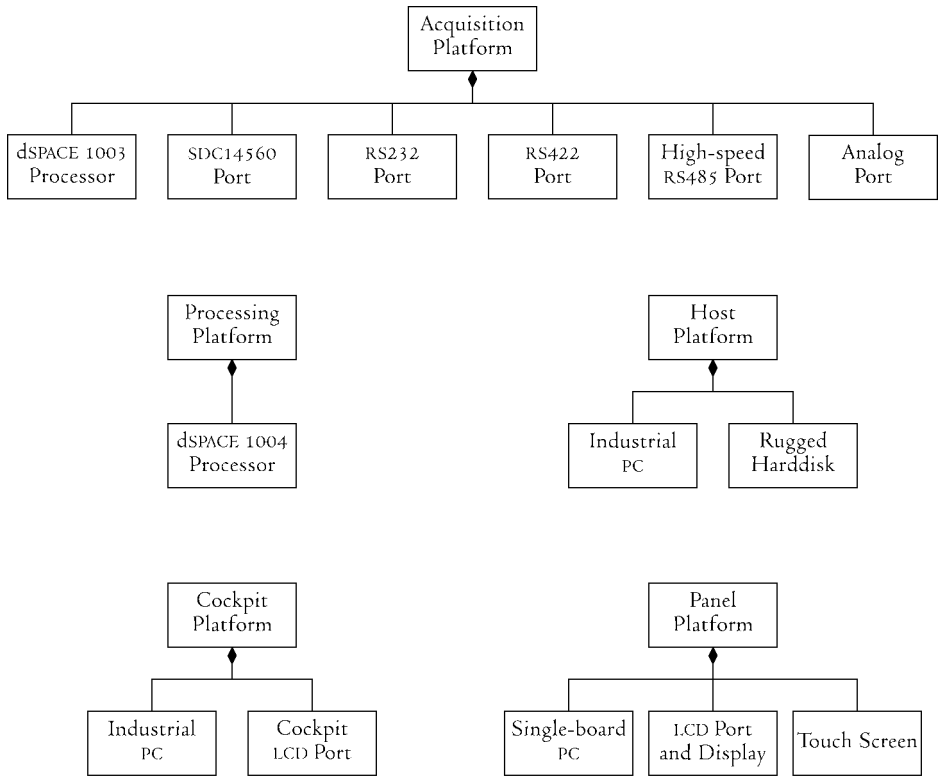


(a) devices and ports

Figure 6.3: Application design for tunnel-in-the-sky assessment.
a. Based on the signal and context models of the application, sensors and interfaces are chosen that meet the application requirements. Each of the components in the right-hand column must be implemented as a port unit; each of the components in the left-hand column is a device unit. The GPS receiver and the corresponding antenna are considered a single device. (continued)

aircraft signals in the tunnel-in-the-sky assessment application. The DSPACE DS1004 is a processing unit that can only be used in combination with the DS1003. It contains a 64-bit DEC Alpha processor that runs at 500 MHz. The DS1004 processor cannot be connected to any interface or a network; it communicates with the DS1003 by means of dual-port memory. The processing platform is included as the host of the Kalman filter component that provides the navigational data for the image generator; the processing rate of 1 kHz that was specified in figure 6.1 justifies the use of a dedicated platform for these computations.

Both dSPACE platforms are embedded-computing systems for which start-up code is provided by the manufacturer and that do not use a full operating system. Both platforms require a personal computer system as a host that loads the start-up and the application code on the dSPACE boards. The host is therefore the third platform in the application. By installing a rugged harddisk with the host platform, it can serve as the



(b) platforms

Figure 6.3, continued.
b. The acquisition platform component hosts all the port units for the application interface to the aircraft. The processing unit is chosen for its suitability to acquire data from many ports. The CPU for the processing platform is a high-capacity board that can host the software component for the kinematic Kalman filter. Both these platforms require a host PC, which can also be used as the data recorder when a rugged harddisk is added. The man-machine interfaces are hosted by a proprietary cockpit platform and a more standard panel platform for the operator and observer stations. (continued)

data recorder for the whole application. Finally, two types of platforms are defined for the cockpit display and the observer and operator panels respectively. The cockpit platform combines a medium-capacity processing unit that is capable of producing the display graphic, with a port for the liquid crystal display (LCD) at the flight deck. The panel platform consists of a single-board processing unit, a touch screen, and an LCD

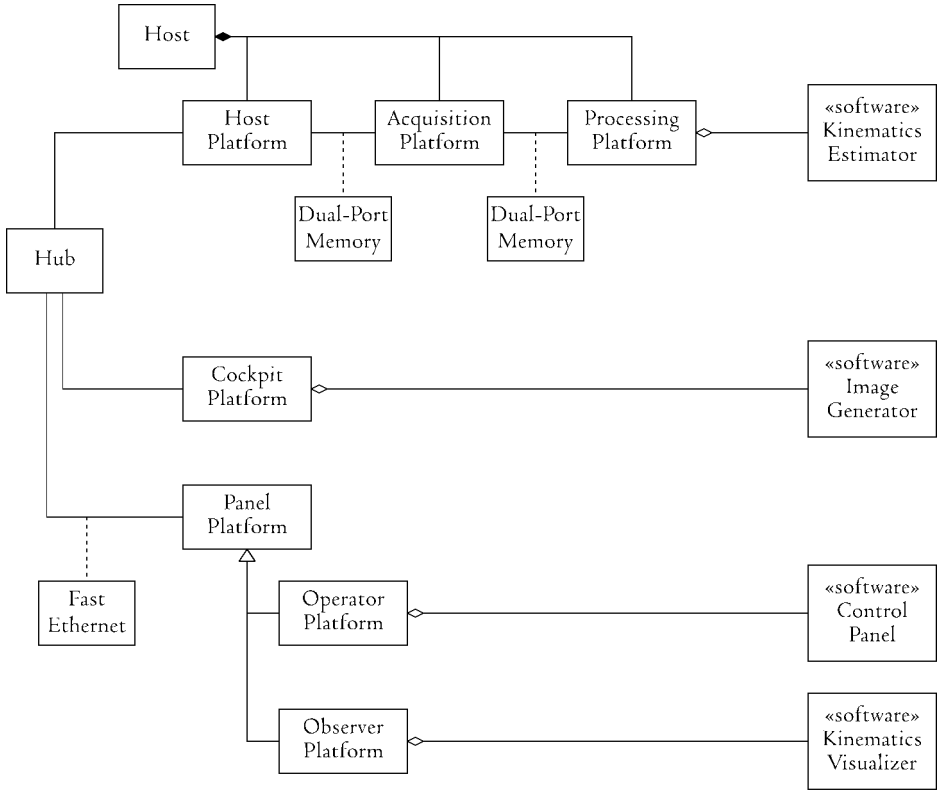


Figure 6.3, continued.
c. Both the connection from the host platform to the acquisition platform, and that from the acquisition to the processing platform is achieved by means of dual-port memory. All other network connections are implemented as fast Ethernet. The application therefore requires a hub. The software parts of the device units are by default located with the same platform as the corresponding port units. The remaining data processing components are explicitly assigned to a platform.

port with display. For the panel platforms, the display is regarded as part of the platform because it must be mounted in a single housing with the processing unit and the touch screen. They must therefore be assembled and tested as a single component; assembly cannot be delayed to application synthesis. The host platform, the cockpit platform, and the panel platforms are Intel Pentium-based personal computers on which Microsoft Windows 2000 is used as the operating system. Although Windows 2000 does not provide the preemption services that are required for a real-

time operating system, it was deemed suitable for the application. Time-critical activities in the application are all performed by the dSPACE system. The utilization of the remaining platforms is low. In combination with the time-stamping services of the middleware and its robustness against transient overloads, violations of deadlines by the PC components are both incidental and harmless.

Figure 6.3c gives an overview of the network in the distributed application and the allocation of the software components to the various platforms. The network connections between the host platform, the cockpit platform, and the panel platforms are implemented as fast Ethernet. Together with the dual-port memory to the acquisition and processing platforms, all platforms are thus joined in the network. The Ethernet portion of the network requires a central hub. Because it is not directly involved in the signal flow, the hub was not identified during application analysis. The hub is an essential hardware component to support the previously identified platform components in the distributed instrumentation system. The hub is therefore a typical part of the solution domain rather than the problem domain.

The network and deployment design of figure 6.3 builds the bridge over the component development to application synthesis. It shows the allocation of the software components to the various platforms. In addition, it gives a rough impression of the physical layout of the instrumentation system in the aircraft. For example, the network and deployment diagram is the first in which the kinematics estimator and its visualizer are not connected through a direct association. The estimator is located at the processing platform – only connected to the Ethernet through two dual-port memories – and the visualizer resides at the observer panel. This topological separation has nothing to do with the logical structure of the application as discussed in section 4.5 (page 119), but only with its implementation on platforms and the installation of those platforms at different locations in the aircraft. The installation of the instrumentation system is important to the mechanical and electric design of the platform and device components. Therefore, the integration design of the application is made before component development starts. Figure 6.4 shows the simple result for the display assessment application. It extends the information in the deployment diagram for the components that are not at an inherently prescribed location. Because their location is fixed and known, the synchros at the control surfaces are omitted.

Dynamic models for selected components

The final activity in application modeling is the development of dynamic models for those components that exhibit state-dependent behavior. The tunnel-in-the-sky image generator is such a component. It distinguishes between three states: idle, acquiring tunnel, and flying approach. The three states and their transitions are shown in figure 6.5. Initially, the components enters the idle state, in which the pilot is informed about the inactivity of the system by the display of a red cross. As soon as valid data is available from the kinematics Kalman filter – which occurs normally immediately after system startup – the image generator enters the acquisition mode. It shows the complete

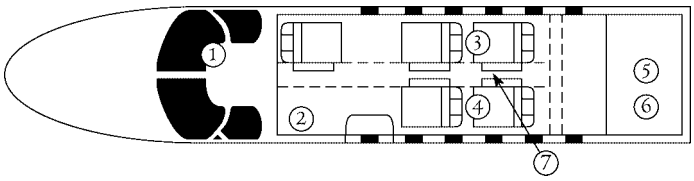


Figure 6.4: Integration design for tunnel-in-the-sky assessment.
The location of the various components in the aircraft serves as a reference during their mechanical design. The cockpit display (1), the cockpit platform (2), the operator panel (3), the observer panel (4), the inertial measurement unit (5), and the host (6) are inside the cabin; the GPS antenna (7) is on top of the fuselage.

synthetic vision and primary flight data, but leaves out certain tunnel symbology. The tunnel itself however is shown when it is in the field of view. When the aircraft is in the vicinity of the tunnel and the velocity vector is roughly aligned with the longitudinal axis of the tunnel, a 'tunnel captured' event is triggered. In the resulting approach mode, the full symbology is displayed. A transition back into the acquisition mode is triggered by a go-around. This behavior reveals the design of the image generator as an experimental component: An approach is not expected to end in a landing.

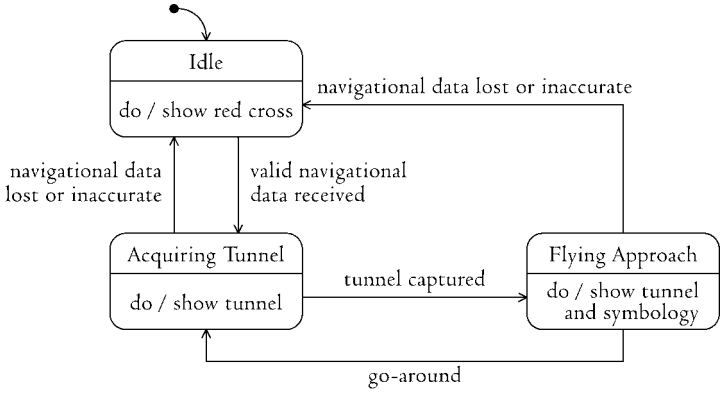


Figure 6.5: States of the image generator.
The image presented to the pilot has three main modes. When no navigational data is available or when the Kalman filter indicates that the navigation estimate is inaccurate, a red cross is displayed. Otherwise, the synthetic vision with primary flight data and the tunnel is shown. When the tunnel has been captured, additional symbology is shown during the approach.

6.2 Component development

The platform, device, and analysis components that make up the tunnel-in-the-sky assessment application can all be found in the architectural design of figure 6.3. The activities in the subsequent component development cover the creation of static and dynamic models, prototyping for hardware components, incremental design and implementation, and specialization and unit testing. It is neither practical nor necessary to perform all these activities for each component. Yet, the analysis, design, implementation, and testing activities should be distinguishable for each component in order to ensure the correct abstractions in the component's design, and thus to guarantee is reusability and extensibility. For example, platform components without port units hardly require any analysis. When the processing unit has been selected, only a mechanical design must be made and the middleware must be adapted to the new system. In this application, using three different types of processing units – two different dSPACE boards and Intel Pentium PCs – middleware specialization is limited to two minor adaptations. First, the logical clock function that reads out the hardware clock must be adjusted to the available counters on the different boards. The DS1003 contains a dedicated real-time clock; on the other two platforms, the clock signal for the CPU itself is used. Second, pipe ends must be programmed for each type of network connection. A total of five pipe ends are required: one for the Ethernet on a personal computer board, and four for both ends of each dual-port memory. Because the DEC Alpha processor is big endian and the other platforms are little endian, a converter is included in the pipe end for the DS1004. For the remaining components, development is illustrated by discussing two of them in more detail: the rate sensor component and the synchro port unit in the acquisition component.

Fiber optic rate sensor

Figure 6.3a showed that the Litef *m*FORS 6 fiber optic rate sensors are accessed through an RS485 serial interface. As this type of interface is not unique to fiber optic rate sensors, the component should be abstracted to isolate the specific properties of the *m*FORS 6 from those of any other device that uses an RS485 interface. This abstraction is part of the service element in the input component. Figure 6.6 shows how the actual service class is based on two abstract base classes: one for the generic serial interface, and one for a generic rate sensor. The first abstract service models the interaction between the service and an RS485 port unit. It can be reused in any service that models an RS485 device. Because RS485 is very similar to other serial protocols, the RS485 service itself is derived from a generic serial port service class. The latter can also be used to derive abstract base classes for different protocols. In this application, this is the case for an RS232 service that is used for the ARINC interface and the RS422 service for communication with the GPS receiver.

The second generalization of the *m*FORS service is the abstract class for a generic rate sensor service. It defines the interface between the service and the rate sensor

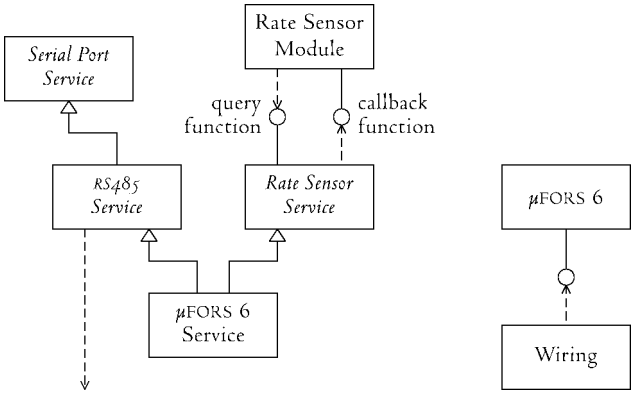


Figure 6.6: Classes of a *m*FORS 6 rate sensor component.
The component consists of the sensor itself, the corresponding wiring, the rate sensor module, and the *m*FORS 6 service. The latter is derived from two abstract classes that model the interface to the module and to the RS485 port unit. The abstract rate sensor service does not change when a different kind of sensor is used; the abstract serial port services can be used in other components as well, for example in the service for the ARINC interface and the GPS receiver.

module. The specialized service for the *m*FORS 6 sensors does little more than combining the interface to the module with that to the RS485 port unit. A measurement query from the module is translated into the correct message for the rate sensor and transmitted through the serial interface. When the message with the measurement result is received, the data is converted into engineering units and any calibration corrections are applied. The agreed data is then returned to the module by means of the callback function. The principle of operation for the device component has now been formulated and the component analysis is complete.

The design of the rate sensor component consists of dynamic modeling of the software elements and a mechanical and electric design for the installation of the hardware elements. The dynamic behavior of the component is specified in a sequence diagram that formalizes the principle of operation that was described above. Figure 6.7 shows the module, service, and device element of the rate sensor component together with the port unit. The interaction between the component and the rest of the instrumentation system is handled by the module. It is activated by the scheduler and returns the schedule interval for the next job at the end of its activation. The job that is activated by the scheduler only sends the data request to the fiber optic sensor; it does not await the response. This asynchronous behavior avoids that the rate sensor module blocks during the comparatively slow serial communication with the sensor; it also prevents the module from blocking indefinitely when no return message from the sensor arrives

at all. When the port receives the raw measurement data from the device, it generates an interrupt that is handled by the rate sensor service. If the received data is found to be correct, the service passes the calibrated value to the module; the module finally publishes the new signal value through the middleware's registry.

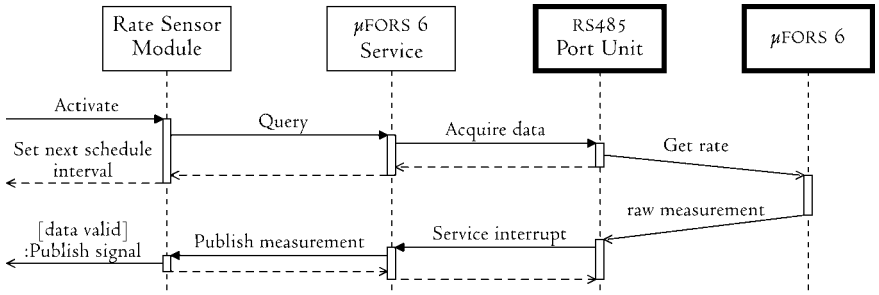
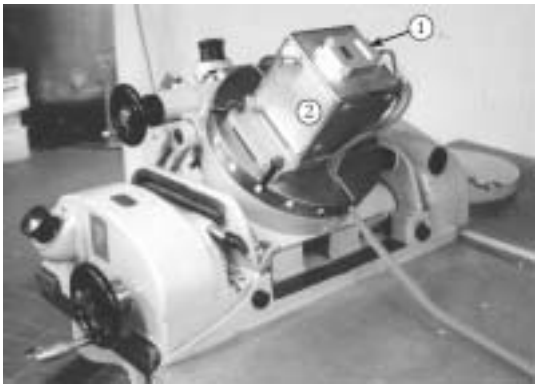


Figure 6.7: Data acquisition sequence for a *m*FORS 6 rate sensor component. The module is activated by the middleware's scheduler when a rate measurement is requested. The module passes the request to the service, which activates the sensor through the port unit. When the raw measurement is returned, the port unit generates an interrupt and sends the result to the service. The raw data is converted – including calibration corrections – and the module callback is activated to publish the new signal value with the registry.

The mechanical and electric design of the device component does not differ from the practice for a conventional instrumentation system. After all, the rationale for the object-oriented development methodology is the desire to use existing instrumentation in a more flexible way, without having to convert to proprietary smart sensors. The way that devices are installed in the aircraft and how they are electrically connected therefore remains unchanged. Similarly, device implementation is largely unaffected by the new methodology as well. After the component has been assembled, the most noticeable change is the need to program the device's service and module before it can be tested. Unit testing must include a verification of the software elements in the component. Finally, the completed component can be calibrated easily in a software environment that is based on the same middleware as the instrumentation system. When the resulting corrections are included in the conversion from raw measurements into agreed data by the service, the development of the device component is complete.

Synchro port unit

The port unit for connecting synchros to the acquisition platform is the most demanding hardware component in the development of the instrumentation system. Unlike the devices, platforms, and other interfaces, it is not based on a few high-level



Calibration of a fiber optic rate sensor.
The device (1) is mounted on the milled frame (2) that joins the accelerometers and rate sensors in a single unit. The sensor's sensitive axis is aligned with the Earth's rotation axis. Bias, drift, and noise intensity are estimated from a 24h run.

off-the-shelf components. The synchro port unit is a custom development that integrates synchro-digital converters – available as integrated circuits – with a dSPACE custom interface board. The connecting electronics must be designed as part of the component development and the port unit's software requires not only an abstractor, but also a driver to be programmed.

Static modeling of the synchro port unit is comparatively easy. The proprietary nature of the component is the cause for the absence of any abstractions with respect to the standard port unit composition that was shown in figure 3.5 (page 82). Although the unit's hardware part as shown in figure 3.5 could be separated into the dSPACE board, the synchro-digital converters, and the interfacial electronics, the port hardware is best considered a single element and the componential breakdown is best left to the electronic design. Dynamic modeling on the other hand is of paramount importance. Not only must the procedures for communication with the synchro-digital converters be specified, the exact timing of these interactions is required to be able to properly design the interfacial electronics.

First, the behavior of the synchro-digital converters is analyzed. The DDC SDC 14560 integrated circuit has sixteen tristate outputs, two inputs to enable the output lines, an inhibit that freezes the internal synchro-digital conversion process, and two resolutions selection lines. Internal conversion must be inhibited before the conversion result can be read from the 16-bit databus. The SDC 14560 thus has two primary states, the transition between which is triggered by a change of the inhibit signal. Unfortunately, the transition that stops the internal conversion process cannot be

assumed instantaneous: a duration of 500 ns is specified in the converter's documentation. This interval is depicted in figure 6.8 as a third state.

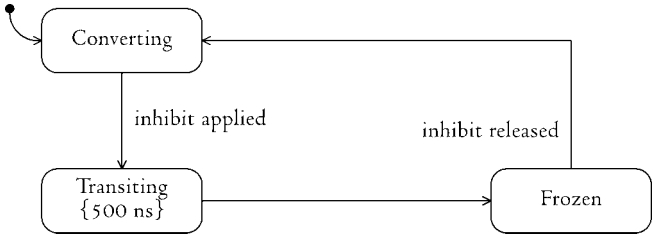


Figure 6.8: States of the SDC 14560 synchro-digital converter.
The converter has two consistent modes: converting and frozen. Transitions are triggered by the inhibit signal. The transition to the frozen state however takes 500 ns; the transition interval is modeled as a third state.

In addition to the 500 ns delay that must be accounted for when changing the synchro-digital converter to its readable state, a 150 ns delay occurs when the output lines are enabled. The comparatively slow behavior of the converter has important consequences for its integration with the dSPACE interface board; the latter system only allows delays up to 70 ns. The electronic hardware therefore has to *latch* the output from the synchro-digital converters: additional circuits must block the converter's output during the transitions and rapidly enable it when the data is to be read by the electronics on the interface board. At the side of the port's device driver, the inhibit signal to freeze conversion, the enable signal to activate the converter output and the reading of the databus must be separated in time. This is indicated in the sequence diagram of figure 6.9, where the timing requirements are shown as constraints.

The key activity in the development of the synchro port unit is the electronic design of the hardware. It should result in the behavior that has been specified in the dynamic model of figure 6.9. In order for the unit to function dependably, the previous analyses and dynamic design must prove correct and the temporal characteristics of the integrated circuits in the resolution selection and output latches must be correctly analyzed. In addition, the speed-critical components must be optimally arranged on the board: When certain connections exceed a maximum length, signal delays prevent the successful exchange of information. To mitigate the risk of implementing a port unit that is based on an erroneous design, the synchro port is first implemented as a throwaway prototype.

In the throwaway prototype, all custom electronics are *wire-wrapped* on a secondary board that is connected to the dSPACE interface board by means of two large connectors. The integrated circuits are mounted in sockets to avoid any soldering. The implementation costs of such a prototype are minimal. In preparation for the 2001 flight

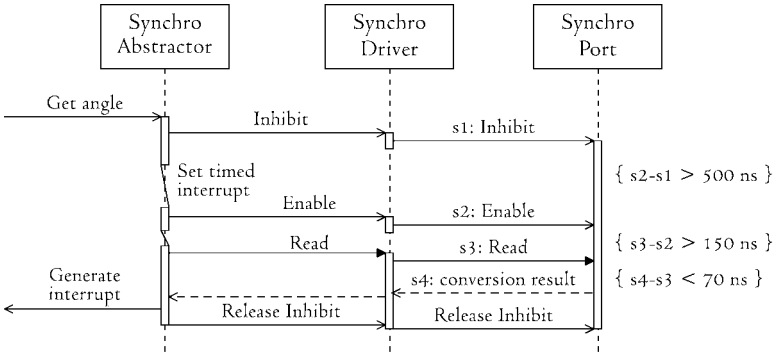
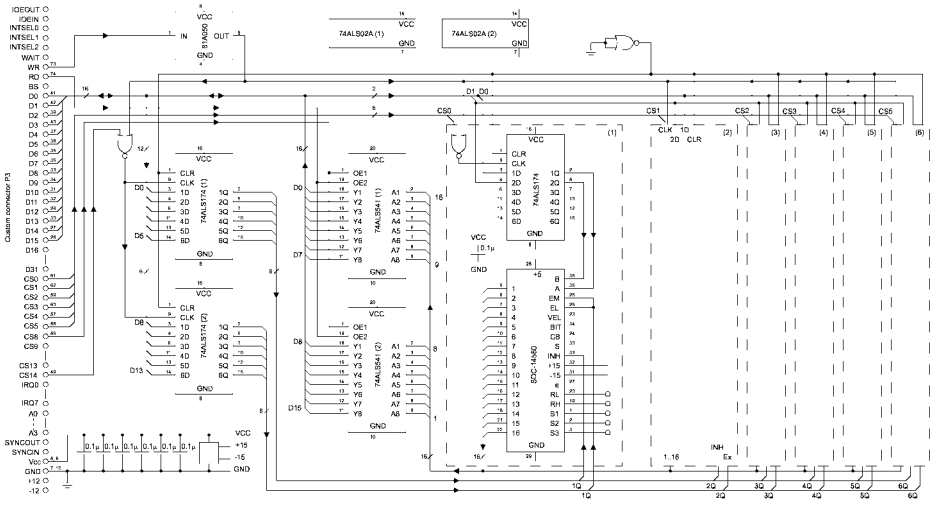
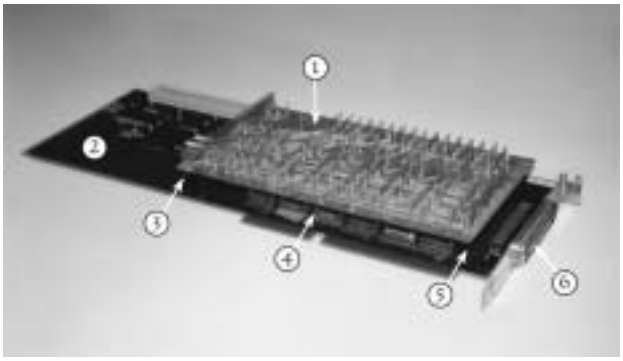


Figure 6.9: Data acquisition sequence for an SDC 14560 synchro converter port. The request for acquisition of the synchro angle from a service is handled asynchronously. First, the inhibit signal is sent and a timer is set that must ensure a 500-ns delay. Then, the enable signal is sent, followed by the read request after 150 ns. The hardware must respond with the conversion result in less than 70 ns. The port abstractor then generates an interrupt with the service and releases the inhibit signal.



Electronic design of the SDC 14560 synchro-converter port.
The port provides converters for six synchros behind a single latch.

test program at the Delft University of Technology, the synchro port prototype was assembled in one day and unit tested for two days. A single wire routing design flaw was discovered during the tests: Different path lengths for two mutually dependent signals prevented the correct timing of a data read. The routing was corrected in a five-minute effort with no additional costs. The prototype was then successfully used in the incremental prototyping phase of the application.



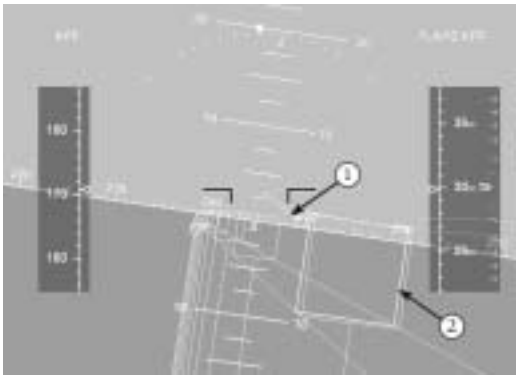
Throwaway prototype of the synchro port unit hardware.
The wire-wrapped auxiliary board (1) is connected to the dSPACE board (2) by means of the two connectors (3) and (5); the six converters (4) are in between. The synchros are connected to the main board's standard connector (6).

6.3 Application synthesis

With the completion of the application's components, the last phase of instrumentation development is begun: the synthesis into the complete system. Even when the application is not developed as an incremental prototype, it is desirable to perform the integration as an incremental process: Various combinations of selected components are integrated to verify their correct interaction, without possible interference from the rest of the application. A laboratory environment that can simulate the contribution of the missing application components is a prerequisite for such an approach. The signal processing middleware that governs the distributed system during its application provides this simulation environment automatically. All internal communication between components is handled by the middleware; as long as signal identifiers match, it is irrelevant to a signal consumer where the signal originates. When the laboratory environment is complemented by testing equipment that simulates the external signals for the application – in this case for example a simulating air data computer which pro-

vides an ARINC databus, or a set of synchros – integration testing can be performed for any subset of the application.

For the tunnel-in-the-sky assessment application, it is desirable to separately test the interaction of the data acquisition components with the kinematics Kalman filter, and that of the image generator with the control panel first. The Kalman filter can subsequently be joined with the image generator and the control panel, but without integrating the data acquisition components. The latter are replaced by recorded measurements from an earlier flight. This allows to evaluate the performance of the filter and the image generator in a realistic environment. In preparation of the 2001 flight tests, the cockpit and the host platform were set up in the laboratory. Software modules that replayed GPS data, inertial measurements, and ARINC recordings from an earlier landing of the Cessna Citation II aircraft, were used instead of the data acquisition platform. The data replay software modules are standardized software components that read a signal's time history from a personal computer harddisk and feed the data in real time to the process. This way, the correct interaction of the filtered positional and orientational estimates with the generation of the tunnel graphic was verified, both in terms of accuracy, precision, and bandwidth. Using the same setup, the tunnel image was evaluated by one of the pilots that took part in the test program. His comments resulted in an extension of the display's symbology before beginning the flight tests.



Tunnel-in-the-sky display as used after pilot comments during integration testing. The flight path vector symbology (1) is complemented by a reference frame (2) that moves along the tunnel at the same distance in front of the aircraft as the tip of the flight path vector. The frame thus allows to locate the vector exactly inside or outside the tunnel.

The instrumentation system for the assessment of a tunnel-in-the-sky display is not an application that is repeatedly used without modification. After the evaluation of a single display configuration, the following test program will generally not use the same

setup, but will incorporate changes to the display, to the way the data are processed, or even to the type of signals that are used as a source for the navigational estimate. The application is therefore a typical example of an experimental flight test instrumentation system that is operated as an incremental prototype; formal system testing and operation as elaborated in section 2.4 does not apply. With the installation of the system in the aircraft, the development activities have reached the ultimate stage in the development life cycle. However, the life cycle is not terminated. With the completion of each flight test program, the system and all of its documentation serve as the basis for developing the next incremental prototype.

System integration of the core instrumentation components.
The host (1) contains the data acquisition, the data processing, and the corresponding host platform; it is connected to the panels and the cockpit platform by the hub (2). The GPS receiver (4) is behind the inertial measurement unit (3). The previously unmodeled display (5) was added to show the status of the host.

For the flight test program that was carried out by the Delft University, the instrumentation system was integrated in the Cessna Citation II laboratory aircraft using existing instrumentation racks for the host, the sensors, and the cockpit platform. The installation mechanism of the cockpit display was specifically designed and produced for the tunnel-assessment project. It allows to easily attach the display to and remove it from the right-hand pilot's instrument panel during flight. Thus, the experimental display can be installed for the actual display assessment portion of the flight, while

System integration of the cockpit platform.
The cockpit platform is integrated in a compact-size industrial computing case (1). The pilot's display (2) can be parked on top of the rack.

System integration of the cockpit display.
The approximately 6 cm thick display unit is mounted in front of the instrument panel. The display can be easily installed and removed during flight, in order to allow the right-hand pilot to use the conventional instruments and assist the left-hand pilot as usual before and after the display assessment flight phase.

System integration of the panel platforms.
The observer panel (1) and the operator panel (2) are fully equal in terms of hardware, although cabling is routed in a way that matches left-hand or right-hand installation in the aircraft.

allowing the right-hand pilot to use the original Citation II instrumentation during the other flight phases or in case of an emergency. Together with the layout of the Citation's flight deck that allows for single-pilot operation, the installation of the experimental display had no considerable safety implications. Finally, the two panel platforms were based on an earlier design to mount a display at the back of a cabin seat. The implementation as a platform component for the evolutionary instrumentation life cycle was performed in the year before the tunnel-in-the-sky display assessment program. These components did not undergo any change, thus confirming the potential to reuse similar system components across different applications.

Discussion

DURING the first century of powered flight, aerospace engineering has become a driver behind the development and application of advanced technologies. Digital computer and software engineering may be the most prominent of these. Yet, the development of information technology has detached from aerospace engineering since the 1970s. This thesis has presented a way to make optimal use of computer and software technologies in a particular field of aerospace engineering: the development of a flight test instrumentation system.

Application development

Flight test instrumentation systems are not produced in series. Although there exist many companies that specialize in the production of instrumentation components, the system as a whole is the typical product of a flight test department, for whom the design and implementation of an instrumentation system is an almost accidental activity. The interval between the development of consecutive systems is large – up to several years – and most often, only one system is built from a certain design. Even though the development process itself is highly structured and well-described in the literature, the process is traditionally not embedded in a life cycle concept. As a result, such an instrumentation system is likely to be unsuitable to serve as the basis for a successor system. Moreover, the lack of standardized documentation and the long interval between instrumentation system developments can pose a threat to the passing of knowledge from one development team to the next.

Life cycle

The use of a life cycle model as proposed in this thesis is to improve this situation. Its main contribution does not lie in the different phases and activities with respect to the development processes found in the literature; though deemed important, the differences are small. The main contribution is the meticulous separation of application and component development, aimed at applying the instrumentation system in evolutionary prototyping. Each step in the development process intends to produce reusable components. The system as a whole should not be abandoned when the flight test program is complete; it should be adapted for the next application.

The life cycle concept will not reveal its advantages until an instrumentation system is significantly changed, or until it is actually reused in another application. In fact, the use of the evolutionary life cycle will slow down initial system development. The large number of analyses and designs that are made before implementation starts can easily conceal the progress of the project or result in a loss of overview during the modeling phase. Both can threaten the entire project. It is therefore of paramount importance that the development of an evolutionary instrumentation system is carefully managed, especially during the initial phases. This increased need for management and the prolonged initial development are recognized as the prime disadvantages of the life cycle concept. Nevertheless, the improved maintainability of the system will generally pay off at the end of even a single development cycle.

The effect of the life cycle concept was demonstrated in the Delft University flight test program for tunnel-in-the-sky display assessment that was performed in 2001. The project was initiated in January; application modeling was completed in February. March and April were used for component development and the first flight tests took place in May. Considering the new developments that were incorporated in the system and comparing the time schedule to that of previous projects, this constitutes a significant reduction of development time. Moreover, the methodology of unit testing and incremental system integration resulted in a more reliable system: No problems occurred after the planned testing phases and the original time plan was kept. The instrumentation system for tunnel-in-the-sky display assessment was implemented as a new application. However, many of its components – including the inertial measurement unit, the GPS receiver, and the host and panel platforms – had been developed for a previous application already. The preceding application was the first to be developed with the life cycle concept that is presented in this thesis. Its development lasted slightly less than a year, which is longer than the time that would have been required to develop the same application in the traditional way.

Modeling and documentation

The development methodology of this thesis is believed to be the first comprehensive concept for all types of applications in flight test and the process that leads up to flight test. Enabled by a common middleware, step-time simulations, real-time simulations, and flight test are consecutive steps in a single process. They can be regarded – and therefore developed – as the various increments of an evolutionary system. Renewed development of components with the same function is avoided. This improves the development time of the systems that are used for the later test phases, but also helps to prevent unintentional differences between the phases. The latter is the most important contribution of using a single life cycle: It enables to “test what you fly, fly what you test”. Especially for handling qualities testing with an in-flight simulation system, this opportunity can reduce the risk of wasting a test flight or even endangering the aircraft and its crew because of a faultily implemented component.

The evolutionary development methodology for flight test instrumentation systems is an adaptation of software engineering concepts for object-oriented applications to the specific nature of a signal processing system. The signal diagram is intended to fill the gap between the modeling of real-time computer systems as they are traditionally encountered in software engineering case studies, and the demands of a signal processing system. Requirements modeling in traditional software engineering is use-case oriented, but the unpredictable and aperiodic nature of a use case does not match continuous data flow requirements. The signal diagram strictly focuses on requirements and allows to separate the signal's characteristics that are actually critical, from those that are merely implied. Unlike a measurand list – which is the conventual starting point for requirements modeling in an instrumentation system – the signal diagram does not record any information that is not essential to the desired function of the application. More importantly, the graphic nature of the signal diagram in comparison with the tabular numbers of a measurand list, helps to convey the model's true information to the user, albeit at the cost of loosing compactness.

A further key concept of the methodology that is presented in this thesis, is the maintenance of normalized documentation. Redundancies should be avoided at all cost, because they easily lead to anomalies and contradicting information. The drawback of normalized documentation lies in the distribution of the information that is required at one point during the development process over multiple resources. A documentation system should therefore be facilitated by digital information management that allows to synthesize the data from various sources as desired. It should be possible to access such an information database with a portable system that can be used anywhere in the laboratory and on-site with the aircraft. Only then, the undesirable composition of summary documents with redundant information can be avoided.

Component development

Whereas the flight test instrumentation system as a whole is the unique creation of a flight test department, its components are mainly off-the-shelf. It is therefore comparatively difficult to adapt the components to a new development concept. When a instantaneous migration from conventional to intelligent instrumentation is to be avoided – which is not only undesirable because of the large risk that is involved, but also often impossible because not every instrument is available as a smart sensor – a component development concept is needed that embeds the existing standard elements in a shell that is compatible with the application concept. This leads to an independent development life cycle for each flight test instrumentation component, during which the shell is created in a generalized form and adapted to the specific instrument.

Hardware

Hierarchical layers are the key to the integration of standard instrumentation components in a physically centralized system†, while sharing the advantages of decentralized intelligent instrumentation. After the use of the signal diagram instead of a measurand list, the implementation of hierarchical layers for every device in the application constitutes the largest change to the practice of instrumentation development. Rather than creating a complete channel in the measurement chain, separate elements are created for the actual device, its preprocessors, and its interfaces to the digital processing system. Each element is usable in not only the combination for which it is originally developed, but also in any other measurement chain where the same function of the individual element is to be fulfilled. In order to make all parts compatible, the development of instrumentation hardware components with hierarchical layers relies heavily on both the application and the component design. The latter ensures that the aggregate of elements actually functions as a single measurement chain channel; the former ensures that the channel fulfills its role in the application. The increased complexity and importance of component modeling with respect to traditional instrumentation development may obscure the application design and result in components that do not meet their requirements. This should be avoided by letting the application's signal diagram play an important role in hardware component development.

The use of prototyping for new hardware components as proposed in this thesis is nothing new. Throwaway and evolutionary prototyping are concepts from software engineering, but it should be recognized that software engineering originally adopted the concept of prototyping from mechanical engineering. Especially because the implementation of a flight test instrumentation system is usually a unique event, there can hardly be a difference between the production of a custom component and the creation of a prototype. Hence, the contribution of this work does not lie in the introduction of prototyping to instrumentation development, but rather in its formalization as part of the evolutionary life cycle. By identifying the points in the development process where prototyping may be considered, its effective use can be increased. Moreover, treating an evolutionary prototype as a valuable first increment of the component or system contributes directly to its further development and reduces the overhead that is caused by implementing the prototype.

Software

The middleware that is presented in this thesis is intended as a pattern. It is an exemplary design that can be used as the starting point for a more specific development, or that can be replaced by an alternative system altogether. Nevertheless, the present

† Centralization of the instrumentation system refers to the installation of the interfaces to the sensors and actuators with the core platforms for data recording and data processing, as opposed to the approach of intelligent instrumentation, in which all interfacing is performed at the location of the device. A centralized instrumentation system therefore does not exclude a distributed system, in which multiple core platforms are installed at different locations.

design describes a complete middleware that can be used in most signal processing applications. In addition, it addresses various characteristics of a flight test instrumentation system that do not receive the same amount of attention in common middlewares. These include the concept of unconfined threads and the emphasis on clock synchronization.

Unconfined threads generalize the concepts of periodic, aperiodic, and sporadic threads as they are used in concurrent computing. An unconfined thread groups a sequence of jobs that belong to a single operator in one thread; the jobs in the unconfined thread are strictly ordered and mutually non-concurrent. The responsibility to register one job at a time with the middleware's scheduler is relayed to the thread; the thread must also convert any completion-of-computation deadline into a beginning-of-computation deadline. This enables the middleware to determine an optimal and robust schedule for the jobs in various unconfined threads, without putting any restriction on the intervals after which the jobs need to be activated.

The probabilistic peer-to-peer clock synchronization algorithm that is introduced as part of the middleware's logical clock constitutes a major improvement over existing clock synchronization algorithms. It is believed to be the first technique to fully exploit the stochastic information that is available in the system of clocks. The method enhances a distributed real-time system in a dynamic environment where time plays an important role as a physical quantity, instead of serving only as a measure to order events. The increased accuracy of the estimate of time makes probabilistic peer-to-peer synchronization most suitable to signal processing systems that handle dynamic data, for example in flight simulation, handling qualities and control law analysis, or system identification. A disadvantage of the method is the instantaneous adjustment of the clock parameters, which results in a non-chronoscopic and inconsistent time scale. An additional logical clock that filters instantaneous time changes is therefore required. The probabilistic clock synchronization algorithm may be further developed by adding mechanisms to recognize and handle faulty clocks; for traditional synchronization methods, such algorithms are readily available.

Outlook

Although the enabling technologies for future aerospace projects will continue to push frontiers, the momentum of new developments does not appear as strong as it was during the first seventy years of powered flight. The oil crisis in the early 1970s marks the turning point in the tumultuous development of aerospace technology: 1969 is not only the year of the first manned flight to the moon, but also that of the first flights of the Boeing 747 and Concorde. From today's point of view, Concorde and the first generation Boeing 747 are exponents of obsolete technology; yet, these aircraft have set the standards in size and velocity of commercial aircraft for the last thirty-five years. New developments are no longer aimed at continuously achieving higher alti-

tudes, larger ranges and payloads, or greater velocities. Instead, a rationalization started that focuses on improving the quality of flight in terms of efficiency and environmental aspects. Aerospace technology has matured.

A similar trend can be observed in information technology. The past decades proved Moore's law for computer hardware development: The circuit density or capacity of semiconductors doubles every eighteen months. However, the exponential increase seems to flatten. Not only are the inherent boundaries of present-day semiconductor materials reached, the demand for ever-increasing capacities fades as well. The focus now shifts to a more effective use of existing technology. Information technology has so far been dominated by hardware developments; software engineering and rationalization of information technology will govern its application in the coming decades. Having detached from aerospace engineering as its primary driving force after the 1960s, the state of the art in information technology should now be reintroduced to the developments in aerospace technology. Being part of research and development and having the opportunity to implement new ideas in a confined environment, flight test instrumentation development seems the right place to start.

Acknowledgments

The work that is presented in this thesis was supervised by Bob Mulder, who's continuous enthusiasm and flexibility never excluded any solution, no matter how unconventional. During the preparation of the manuscript, René van Paassen acted as advisor. His experience in aerospace-related information technology put many of the ideas in the right perspective. This thesis would not be the same without their inputs.

Great gratitude is also owed to the pilots and engineers in the flight department at the Delft University of Technology, Faculty of Aerospace Engineering, who contributed to the development of the instrumentation system for the Cessna Citation II laboratory aircraft. During this phase, the role of Kees van Woerkom – both as a source of information and in terms of implementing hardware solutions – has been invaluable.

References

Abali, B., Stunkel, C.B., and Benveniste, C. (1997). "Clock Synchronization on a Multicomputer" *Journal of Parallel and Distributed Computing 40* (1), 118-130.

Adolph, C.E. (1994). "Planning of a Flight Test Programme" in *AGARDograph 160 (1) 2nd ed: Basic Principles of Flight Test Instrumentation Engineering* Borek, R.W. and Pool, A., eds. (Advisory Group for Aerospace Research and Development, Neuilly sur Seine), ISBN 92-835-0731-2.

Advisory Group for Aerospace Research and Development (1993). *Conference Proceedings 545: Aerospace Software Engineering for Advanced Systems Architectures* (Neuilly sur Seine), ISBN 92-835-0725-8.

Agresti, W.W. (1986). *New Paradigms for Software Development* (IEEE Computer Society Press), ISBN 0-444-70124-9.

Alari, G. and Ciuffoletti, A. (1997). "Implementing a Probabilistic Clock Synchronization Algorithm" *Real-Time Systems 13* (1), 25-46.

Alter, K.W., Barrows, A.K., Enge, P., Jennings, Ch.W., Parkinson, B.W., and Powell, J.D. (1998). "Inflight Demonstrations of Curved Approaches and Missed Approaches in Mountainous Terrain" Institute of Navigation *GPS-98*, 1165-1172.

Anderson, Ch.M. and Dorfman, M., eds. (1991). *Aerospace Software Engineering: A Collection of Concepts* (American Institute of Aeronautics and Astronautics, Washington DC), ISBN 1-56347-005-5.

Arvind, K. (1994). "Probabilistic Clock Synchronization in Distributed Systems" IEEE *Transactions on Parallel and Distributed Systems 5* (5), 474-487.

Ashby, N. and Spilker Jr., J.J. (1996). "Introduction to Relativistic Effects on the Global Positioning System" in *Global Positioning System: Theory and Applications 1* Parkinson, B.W. and Spilker Jr., J.J., eds. (American Institute of Aeronautics and Astronautics, Washington DC), ISBN 1-56347-106-X, pp. 623-697.

Attiya, H. Herzberg, A., and Rajsbaum, S. (1996). "Optimal Clock Synchronization Under Different Delay Assumptions" SIAM *Journal on Computing 25* (2), 369-389.

Bacon, J. (1992). *Concurrent Systems: An Integrated Approach to Operating Systems, Database, and Distributed Systems* (Addison-Wesley), ISBN 0-201-41677-8.

Bailey, R.E. (1989). "Effect of Head-Up Display Dynamics on Fighter Flying Qualities" *Journal of Guidance, Control, and Dynamics 12* (4), 514-520.

Banavar, G., Chandra, T., Strom, T., and Sturman, D. (1999). "A Case for Message Oriented Middleware" Springer *13th International Symposium on Distributed Computing*, 1-18.

Barrows, A.K. and Powell, J.D. (2000). "Flying a Tunnel-in-the-Sky Display within the Current Airspace System" American Institute of Aeronautics and Astronautics *38th Aerospace Sciences Meeting & Exhibit*, paper 2000-1059.

Basili, V.R. and Turner, A.J. (1975). "Iterative Enhancement: A Practical Technique for Software Development" IEEE *Transactions on Software Engineering 1* (4), 390-396.

Berard, E.V. (1993). *Essays on Object-Oriented Software Engineering 1* (Prentice Hall), ISBN 0-13-288895-5.

Boehm, B.W. (1976). "Software Engineering" IEEE *Transactions on Computers 25* (12), 1226-1241.

—— (1981). *Software Engineering Economics* (Prentice Hall), ISBN 0-13-822122-7.

Bondeli, P. de (1991). "Aerospace Software Engineering in France" in *Aerospace Software Engineering: A Collection of Concepts* Anderson, Ch.M. and Dorfman, M., eds. (American Institute of Aeronautics and Astronautics, Washington DC), ISBN 1-56347-005-5, pp. 525-544.

Booch, G. (1994). *Object-Oriented Analysis and Design with Applications 2nd ed* (Addison-Wesley), ISBN 0-8053-5340-2.

Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *The Unified Modeling Language User Guide* (Addison-Wesley), ISBN 0-201-57168-4.

Borek, R.W. and Pool, A., eds. (1994). *AGARDograph 160 (1) 2nd ed: Basic Principles of Flight Test Instrumentation Engineering* (Advisory Group for Aerospace Research and Development, Neuilly sur Seine), ISBN 92-835-0731-2.

Britton, K.H. and Parnas, D.L. (1981). "A-7E Software Module Guide" Naval Research Laboratory *Memorandum Report 4702*.

Cayley, A. (1857). "On the Theory of Analytic Forms Called Trees" *Philosophical Magazine 13*, 19-30.

Christian, F. (1989). "Probabilistic Clock Synchronization" *Distributed Computing 3*, 146-158.

Coad, P. and Yourdon, E. (1990). *Object-Oriented Analysis 2nd ed* (Prentice Hall), ISBN 0-13-629981-4.

—— (1991). *Object-Oriented Design* (Prentice Hall), ISBN 0-13-630070-7.

Cook, W.R., Hill, W.L., and Canning, P.S. (1990). "Inheritance is Not Subtyping" Association for Computing Machinery *Annual Symposium on Principles of Programming Languages*, 125-135.

Cormen, Th.H., Leiserson, Ch.E., Rivest, R.L., and Clifford, S. (2001). *Introduction to Algorithms 2nd ed* (MIT Press, Cambridge), ISBN 0-262-53196-8.

Crounse, D.R. (1995). "Flight Test Instrumentation" in *AGARDograph 300 (14): Introduction to Flight Test Engineering* Stoliker, F.N., ed. (Advisory Group for Aerospace Research and Development, Neuilly sur Seine), ISBN 92-836-1020-2.

Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R. (1972). *Structured Programming* (Academic Press), ISBN 0-12-200550-3.

Dahl, O.J. and Nygaard, K. (1966). "SIMULA: An ALGOL-Based Simulation Language" *Communications of the ACM 9* (9), 671-678.

Dertouzos, M.L. (1974). "Control Robotics: The Procedural Control of Physical Processes" International Federation for Information Processing *6th Congress on Information Processing*, 807-813.

Diestel, R. (2000). *Graph Theory 2nd ed* (Springer), ISBN 0-387-95014-1.

Dijkstra, E.W. (1968a). "Co-operating Sequential Processes" in *Programming Languages* Genuys, F., ed. (Academic Press), pp. 43-112.

—— (1968b). "The Structure of the 'THE'-Multiprogramming System" *Communications of the ACM 11* (3), 341-346.

Douglass, B.P. (1998). *Real-Time UML: Developing Efficient Objects for Embedded Systems* (Addison-Wesley), ISBN 0-201-32579-9.

Eccles, L.H. (2000). "IEEE P1451.3: A Standard for Networked Transducers" Society of Flight Test Engineers *31st Annual Symposium*, paper I-04.

Eykhoff, P. (1974). *System Identification: Parameter and State Estimation* (Wiley), ISBN 0-471-24980-7.

Euler, L. (1736). "Solutio Problematis ad Geometriam Situs Pertinentis" *Commentarii Academiae Scientiarum Imperialis Petropolitanae 8*, 128-140.

Fowler, M. and Scott, K. (1999). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (Addison-Wesley), ISBN 0-201-65783-X.

Funabiki, K., Muraoka, K., Terui, Y., Harigae, M., and Ono, T. (1999). "In-flight Evaluation of Tunnel-in-the-Sky Display and Curved Approach Pattern" American Institute of Aeronautics and Astronautics *Guidance, Navigation and Control Conference and Exhibit*, 108-114.

Gibbs-Smith, Ch.H. (1960). *The Aeroplane: An Historical Survey of its Origins and Development* (Her Majesty's Stationery Office, London).

Gibson, J.C. (1999). *Development of a Methodology for Excellence in Handling Qualities Design for Fly by Wire Aircraft* (Dissertation, Delft University Press), ISBN 90-407-1841-5.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and its Implementation* (Addison-Wesley), ISBN 0-201-11371-6.

Gomaa, H. (1986). "Software Development of Real-Time Systems" *Communications of the ACM 29* (7), 657-668.

—— (2000). *Designing Concurrent, Distributed, and Real-Time Applications with UML* (Addison-Wesley), ISBN 0-201-65793-7.

Harary, F. (1969). *Graph Theory* (Addison-Wesley).

Höhne, G. (2001). "Roll Ratcheting: Cause and Analysis" Deutsches Zentrum für Luft- und Raumfahrt *Forschungsbericht 2001-15*, ISSN 1434-8454.

Horn, W.A. (1974). "Some Simple Scheduling Algorithms" *Naval Research Logistics Quaterly 21*, 177-185.

Huffel, S. Van and Vandewalle, J. (1991). *The Total Least Squares Problem: Computational Aspects and Analysis* (Society for Industrial and Applied Mathematics, Philadelphia), ISBN 0-89871-275-0.

Institute of Electrical and Electronics Engineers (1997). *1451.2 Standard for a Smart Transducer Interface for Sensors and Actuators: Transducer to Microprocessor Communication Protocols and Transducer Electronic Data Sheets (TEDS) Formats* (Piscataway), ISBN 1-55937-963-4.

—— (1999). *1451.1 Standard for a Smart Transducer Interface for Sensors and Actuators: Network Capable Application Processor (NCAP) Information Model* (Piscataway), ISBN 0-7381-1767-6.

International Civil Aviation Organization (1996). *Annex 10 to the Convention on International Civil Aviation: Aeronautical Telecommunications 1: Radio Navigation Aids* (Montreal).

International Organization for Standardization (1981). "Data Processing - Open Systems Interconnection - Basic Reference Model" Elsevier *Computer Networks 5* (2), 81-118.

Jackson, J.R. (1955). "Scheduling a Production Line to Minimize Maximum Tardiness" University of California, Los Angeles, Management Science Research Project *Technical Report 43*.

Jacobson, I. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach* (Addison-Wesley), ISBN 0-201-54435-0.

Jacobson, I., Booch, G., and Rumbaugh, J. (1999). *The Unified Software Development Process* (Addison-Wesley), ISBN 0-201-57169-2.

Jennings, N. (1991). "Aerospace Software Engineering in the United Kingdom" in *Aerospace Software Engineering: A Collection of Concepts* Anderson, Ch.M. and Dorfman, M., eds. (American Institute of Aeronautics and Astronautics, Washington DC), ISBN 1-56347-005-5, pp. 545-559.

Kelley, C.R. (1968). *Manual and Automatic Control: A Theory of Manual Control and its Application to Manual and to Automatic Systems* (Wiley).

Kelly, R.J. and Davis, J.M. (1994). "Required Navigation Performance (RNP) for Precision Approach and Landing with GNSS Application" ION *Navigation 41* (1), 1-30.

Klein, M.H., Lehoczky, J.P., and Rajkumar, R. (1994). "Rate-Monotonic Analysis for Real-Time Industrial Computing" IEEE *Computer 27* (1), 24-33.

Knight, V.H. and Dove, B.L. (1994). "Principles of Instrumentation System Design" in *AGARDograph 160 (1) 2nd ed: Basic Principles of Flight Test Instrumentation Engineering* Borek, R.W. and Pool, A., eds. (Advisory Group for Aerospace Research and Development, Neuilly sur Seine), ISBN 92-835-0731-2.

Kopetz, H. (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications* (Kluwer Academic), ISBN 0-7923-9894-7.

Kopetz, H. and Ochsenreiter, W. (1987). "Clock Synchronization in Distributed Real-Time Systems" IEEE *Transactions on Computers 36* (8), 933-940.

Kopetz, H., Zainlinger, R., Fohler, G., Kantz, H., Puschner, P.P., and Schutz, W. (1991). "An Engineering Approach to Hard Real-Time System Design" Springer *3rd European Software Engineering Conference*, 166-188.

Kurose, J.F., Towsley, D., and Krishna, C.M. (1991). "Design and Analysis of Processor Scheduling Policies for Real-Time Systems" in *Foundations of Real-Time Computing: Scheduling and Resource Management* Tilborg, A.M. van and Koob, G.M., eds. (Kluwer Academic), ISBN 0-7923-9166-7, pp. 63-89.

Laban, M. (1994). *On-Line Aircraft Aerodynamic Model Identification* (Dissertation, Delft University of Technology), ISBN 90-6275-987-4.

Lacan, Ph. and Colangeli, P. (1993). "Software Engineering Methods in the Hermes On-Board Software" in *Conference Proceedings 545: Aerospace Software Engineering for Advanced Systems Architectures* (Advisory Group for Aerospace Research and Development, Neuilly sur Seine), ISBN 92-835-0725-8.

Lamport, L. (1978). "Time, Clocks, and the Ordering of Events in a Distributed System" *Communications of the ACM 21* (7), 558-565.

Lamport, L. and Melliar-Smith, P.M. (1985). "Synchronizing Clocks in the Presence of Faults" *Journal of the ACM 32* (1), 52-78.

Lehoczky, J.P., Sha, L., and Ding, Y. (1989). "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior" IEEE *10th Real-Time Systems Symposium*, 166-171.

Lehoczky, J.P., Sha, L., Strosnider, J.K., and Tokuda, H. (1991). "Fixed Priority Scheduling Theory for Hard Real-Time Systems" in *Foundations of Real-Time Computing: Scheduling and*

Resource Management Tilborg, A.M. van and Koob, G.M., eds. (Kluwer Academic), ISBN 0-7923-9166-7, pp. 1-30.

Levine, J. and Mills, D. (2000). "Using the Network Time Protocol (NTP) to Transmit International Atomic Time (TAI)" US Naval Observatory *32nd Annual Precise Time and Time Interval (PTTI) Meeting*, 431-439.

Liu, C.L. and Layland, J.W. (1973). "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment" *Journal of the ACM 20* (1), 46-61.

Liu, C.L., Liu, J.W.S., and Liestman, A.L. (1982). "Scheduling with Slack-Time" *Acta Informatica 17*, 31-41.

Mala, W. and Grandi, E. (1993). "Experiences with the HOOD Design Method on Avionics Software Development" in *Conference Proceedings 545: Aerospace Software Engineering for Advanced Systems Architectures* (Advisory Group for Aerospace Research and Development, Neuilly sur Seine), ISBN 92-835-0725-8.

McCracken, D. and Jackson, M. (1982). "Life Cycle Concept Considered Harmful" ACM *Software Engineering Notes 7* (2), 28-32.

Meyer, B. (1987). "Reusability: The Case for Object-Oriented Design" IEEE *Software 4* (2), 50-64.

Micouin, P.M. and Ubeaud, D.J. (1993). "Hierarchical Object Oriented Design: Possibilities, Limitations and Challenges" in *Conference Proceedings 545: Aerospace Software Engineering for Advanced Systems Architectures* (Advisory Group for Aerospace Research and Development, Neuilly sur Seine), ISBN 92-835-0725-8.

Mills, D.L. (1991). "Internet Time Synchronization: the Network Time Protocol" IEEE *Transactions on Communications 39* (10), 1482-1493.

—— (1994). "Precision Synchronization of Computer Network Clocks" ACM *Computer Communication Review 24* (2), 28-43.

—— (1995). "Improved Algorithms for Synchronizing Computer Network Clocks" IEEE/ACM *Transactions on Networking 3* (3), 245-254.

Mok, A.K. (1983). *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment* (Dissertation, Massachusetts Institute of Technology, Cambridge).

Mok, A.K. and Dertouzos, M.L. (1978). "Multiprocessor Scheduling in a Hard Real-Time Environment" Institute of Electrical and Electronics Engineers/Association for Computing Machinery *7th Texas Conference on Computer Systems*, pp. 5.1-5.12.

Mowbray, Th.J. and Ruh, W.A. (1997). *Inside Corba: Distributed Object Standards and Applications* (Addison-Wesley), ISBN 0-201-89540-4.

Mulder, M. (1999). *Cybernetics of Tunnel-in-the-Sky Displays* (Dissertation, Delft University Press), ISBN 90-407-1963-2.

Mulder, M., Kraeger, A.M., and Soijer, M.W. (2002). "Delft Aerospace Tunnel-in-the-Sky Flight Tests" American Institute of Aeronautics and Astronautics *Guidance, Navigation, and Control Conference and Exhibit*, paper 2002-4929.

Nelson, B.J. (1981). *Remote Procedure Call* (Dissertation. Carnegie Mellon University, Pittsburgh).

Occelli, P. (1993). "Object Versus Functional Oriented Design" in *Conference Proceedings 545: Aerospace Software Engineering for Advanced Systems Architectures* (Advisory Group for Aerospace Research and Development, Neuilly sur Seine), ISBN 92-835-0725-8.

Olson, A. and Shin, K.G. (1994). "Probabilistic Clock Synchronization in Large Distributed Systems" IEEE *Transactions on Computers 43* (9), 1106-1112.

O'Neil, P. and O'Neil, E. (2001). *Database: Principles, Programming, and Performance* (Morgan Kaufmann, Academic Press), ISBN 1-55860-438-3.

Opdyke, W.F. (1992). *Refactoring Object-Oriented Frameworks* (Dissertation, University of Illinois, Urbana-Champaign).

Ostrovsky, R., and Patt-Shamir, B. (1999). "Optimal and Efficient Clock Synchronization Under Drifting Clocks" Association for Computing Machinery *18th Annual Symposium on Principles of Distributed Computing*, 3-12.

Paassen, M.M. van, Pronk, C., and Delatour, J. (2000). "Middleware for Real-Time Distributed Simulation Systems" Society for Computer Simulation *12th European Simulation Symposium*, 14-22.

Paassen, M.M. van, Stroosma, O., and Delatour, J. (2000). "DUECA - Data-Driven Activation in Distributed Real-Time Computation" American Institute of Aeronautics and Astronautics *Modeling and Simulation Technologies Conference*, paper 2000-4503.

Park, D.W., Natarajan, S., and Kanevsky, A. (1993). "Fixed-Priority Scheduling of Real-Time Systems Using Utilization Bounds" American Institute of Aeronautics and Astronautics *9th Computing in Aerospace Conference*, 375-382.

Parkinson, B.W. and Spilker Jr., J.J., eds. (1996). *Global Positioning System: Theory and Applications 1* (American Institute of Aeronautics and Astronautics, Washington DC), ISBN 1-56347-106-X.

Parnas, D.L. (1972). "On the Criteria to Be Used in Decomposing Systems into Modules" *Communications of the ACM 15* (12), 1053-1058.

—— (1979). "Designing Software for Ease of Extension and Contraction" IEEE *Transactions on Software Engineering 5* (2), 128-138.

Parnas, D.L., Clements, P.C., and Weiss, D.M. (1985). "The Modular Structure of Complex Systems" IEEE *Transactions on Software Engineering 11* (1), 259-266.

Ramamritham, K. and Stankovic, J.A. (1991). "Scheduling Strategies Adopted in Spring: An Overview" in *Foundations of Real-Time Computing: Scheduling and Resource Management* Tilborg, A.M. van and Koob, G.M., eds. (Kluwer Academic), ISBN 0-7923-9166-7, pp. 277-305.

Robinson, P.J. (1992). *Hierarchical Object-Oriented Design* (Prentice Hall), ISBN 0-13-390816-X.

Royce, W.W. (1970). "Managing the Development of Large Software Systems: Concepts and Techniques" Institute of Electrical and Electronics Engineers *Western Electronic Show and Convention 14*, paper A-1.

Rumbaugh, J., Blaha M.R., Lorensen, W., Eddy, F., and Premerlani, W. (1991). *Object-Oriented Modeling and Design* (Prentice Hall), ISBN 0-13-629841-9.

Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling Language Reference Manual* (Addison-Wesley), ISBN 0-201-30998-X.

Sachs, G. and Sperl, R. (2001). "Experimental Low-Cost 3D-Display for General Aviation Aircraft" American Institute of Aeronautics and Astronautics *Atmospheric Flight Mechanics Conference and Exhibit*, paper 2001-4195.

Sage, A.P. and Melsa, J.L. (1971). *Estimation Theory with Applications to Communications and Control* (McGraw-Hill).

Schmid, U. (1995). "Synchronized Universal Time Coordinated for Distributed Real-Time Systems" IFAC *Control Engineering Practice 3* (6), 877-884.

Schmid, U., Horauer, M., and Kerö, N. (1999). "How to Distribute GPS-Time over COTS-based LANs" US Naval Observatory *31st Annual Precise Time and Time Interval (PTTI) Meeting*, 545-560.

Schossmaier, K., Schmid, U., Horauer, M., and Loy, D. (1997). "Specification and Implementation of the Universal Time Coordinated Synchronization Unit (UTCSU)" *Real-Time Systems 12* (3), 295-327.

Seidelmann, P.K., ed. (1992). *Explanatory Supplement to the Astronomical Almanac* (University Science Books), ISBN 0-935702-68-7.

Serlin, O. (1972). "Scheduling of Time Critical Processes" American Federation of Information Processing Societies *Sprint Joint Computer Conference*, 925-932.

Sha, L. and Goodenough, J.B. (1990). "Real-Time Scheduling Theory and Ada" IEEE *Computer 23* (4), 53-62.

Sha, L., Klein, M.H., and Goodenough, J.B. (1991). "Rate Monotonic Analysis for Real-Time Systems" in *Foundations of Real-Time Computing: Scheduling and Resource Management* Tilborg, A.M. van and Koob, G.M., eds. (Kluwer Academic), ISBN 0-7923-9166-7, pp. 129-155.

Shaw, M. (2001). "The Coming-of-Age of Software Architecture Research" Institute of Electrical and Electronics Engineers *23rd International Conference on Software Engineering*, 656-664.

Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline* (McGraw-Hill), ISBN 0-13-182957-2.

Silberschatz, A. and Galvin, P.B. (1998). *Operating System Concepts 5th ed* (Wiley), ISBN 0-471-36414-2.

Snyder, A. (1986). "Encapsulation and Inheritance in Object-Oriented Programming Languages" Association for Computing Machinery *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86)*, 38-45.

Sommerville, I. (2000). *Software Engineering 6th ed* (Addison-Wesley), ISBN 0-201-39815-X.

Sprunt, B. (1990). *Aperiodic Task Scheduling for Real-Time Systems* (Dissertation, Carnegie Mellon University, Pittsburgh).

Srikanth, T.K. and Toueg, S. (1987). "Optimal Clock Synchronization" *Journal of the ACM 34* (3), 626-645.

Stankovic, J.A. and Ramamritham, K., eds. (1993). *Advances in Real-Time Systems* (Institute of Electrical and Electronics Engineers), ISBN 0-8186-3792-7.

Stankovic, J.A., Spuri, M., Natale, M. Di, and Buttazzo, G.C. (1995). "Implications of Classical Scheduling Results for Real-Time Systems" IEEE *Computer 28* (6), 16-25.

Stewart, D.B., Volpe, R.A., and Khosla, P.K. (1997). "Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects" IEEE *Transactions on Software Engineering 23* (12), 759-776.

Swift, J. (1726). *Gulliver's Travels* (Benj. Motte, London).

Tanenbaum, A.S. (1996). *Computer Networks 3rd ed* (Prentice Hall), ISBN 0-13-349945-6.

Theunissen, E. (1997). *Integrated Design of a Man-Machine Interface for 4-D Navigation* (Dissertation, Delft University Press), ISBN 90-407-1406-1.

United States Department of Transportation and Department of Defense (1992). *1992 Federal Radionavigation Plan - DOT-VNTSC-RSPA-92-2/DOD-4650.5* (Washington DC).

White, J. (1976). "A High-Level Framework for Network-Based Resource Sharing" American Federation of Information Processing Societies *National Computer Conference*, 561-570.

Wirfs-Brock, R., Wilkerson, B., and Wiener, L. (1990). *Designing Object-Oriented Software* (Prentice Hall), ISBN 0-13-629825-7.

Appendix

A
Unified Modeling Language

THE Unified Modeling Language (UML) provides a standardized graphical notation for describing object-oriented models. This appendix contains a summary of the diagrams and the elements that are used throughout this thesis. A complete definition of the language is presented by Rumbaugh, Jacobson, and Booch (1999). Details on the application of UML to the design of real-time systems are discussed by Gomaa (2000).

Class diagrams

A class diagram shows classes and their relationships in a logical view of a system. It identifies the type of entities that form the system, as well as the relationships between them. Class diagrams are used for *static modeling* of the system. They describe which classes exist, which have a mutual relationship, and what the characteristics of these relationships are, but not how the cooperation between the classes is realized and how it changes with time during application of the system.

Class diagrams can be drawn from three different perspectives: the conceptual perspective, the specification perspective, and the implementation perspective. The conceptual perspective focusses on the role of the system and its subsystems in the environment, and the function that each of the subsystems has to fulfill. There need not be a direct mapping between the classes in the diagram and the objects that form a realization of the concept. A more accurate representation of the elements that exist in the realization, is given in class diagrams that are drawn from the specification and implementation perspectives. The specification perspective focusses on the interface of the classes. It defines the function of each class and how it interacts with its environment. Details on the task of each class, hence on the activities that the class deploys in order to achieve the fulfillment of its function, is shown only from the implementation perspective.

Classes are depicted as boxes containing the class name as shown in figure A.1. Optionally, attributes and operations of the class can be shown in two additional compartments. The top compartment always holds the class name. When both attributes and operations are shown, the center compartment holds the attributes and the bot-

tom compartment holds the operations. Operations are also referred to as methods. Class names are capitalized; attribute and operation names are shown in lowercase.

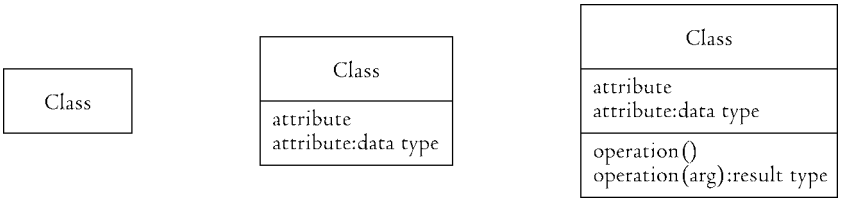


Figure A.1: UML notation for classes.
Attributes and operations of a class are shown in two compartments below the one holding the class name. They can contain both formal specifications of the implementation and informal conceptual descriptions of the properties and activities of the class.

Associations

An association between two classes is shown as a line that connects the two class boxes as shown in figure A.2. Associations which are strictly one-directional are drawn with a simple arrowhead in the direction of the navigability. Associations that are navigable in one direction occur when one class in the association is not aware of the existence of the other class. Optionally, an association can be labeled with any combination of multiplicity indicators and names. Association names describe the type of association between the two classes; an optional filled arrowhead indicates the direction in which the description is to be interpreted. By default, association names appear on the left side of the connecting line when following the line in the direction for the description. Multiplicities indicate how many instances of the class can have the specified association with the other class. Multiplicities are indicated by a number at the end of the line that connects the classes. Alternative values are separated by commas; a range of values is indicated by two dots. The asterisk is used to indicate an arbitrary number, or infinity when used in a range.

An association class is a class that models an association between two other classes. As shown in figure A.3, it is depicted as a class that is connected to an association by a dashed line. Association classes are particularly useful when the relationship between two classes exists by means of a physical entity that has his own parameters, characteristics and behavior.

Aggregations and compositions

Aggregation and composition are relationships between classes where one class is a part of the other. Both an aggregation and a composition are shown in figure A.4. Aggrega-

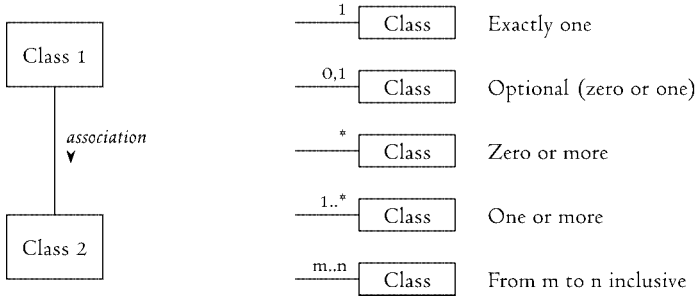


Figure A.2: UML notation for class associations.
When no direction is indicated, an association name applies in the direction for which it is located left of the association arc. Exceptions are indicated by a filled arrowhead. Multiplicity identifiers can be used at any association end to indicate the number of class instances that can participate in the association.

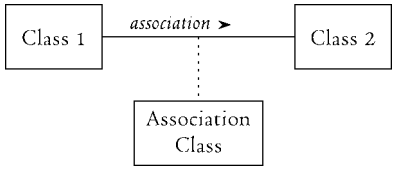


Figure A.3: UML notation for association classes.
The association class indicates a relationship between class 1 and class 2 through the association class. The logical interaction takes place between the two top classes; the practical interaction takes place by means of the association class.

tion is depicted as an open diamond on the side of the aggregate class. It models the loose association of classes in a single group. Composition is shown by a filled diamond on the side of the composite class. Composition is a stronger form of aggregation. Part class instances generally can belong to only one composite class instance; they are created and destroyed at the same time as the composite class.

Specializations, abstract classes and interfaces

A specialization is a relationship between a class and a subclass, which describes that the subclass is a modified or extended variation to the original class. The specialization relationship is an *is-a* relationship: the subclass is a type of the superclass. As shown in figure A.5, a specialization is depicted as an arrow between the classes with

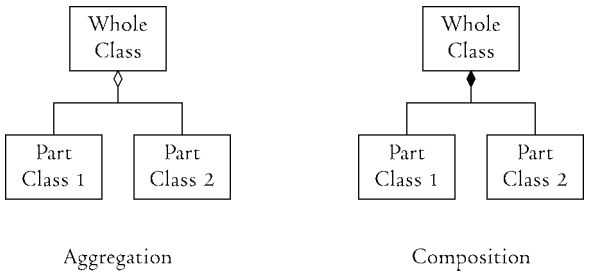


Figure A.4: UML notation for aggregations and compositions.
An aggregation is depicted as an association with an open diamond at the side of the aggregate class. Composition is a stronger form of aggregation, depicted by a filled diamond. The parts of a composition can participate in only a single composite class.

an open arrowhead pointing to the superclass. When viewed in the other direction, the specialization relationship is referred to as a generalization. Two classes that are derived from a common superclass, share a number of attributes or operations that are generalized as the model of the superclass.

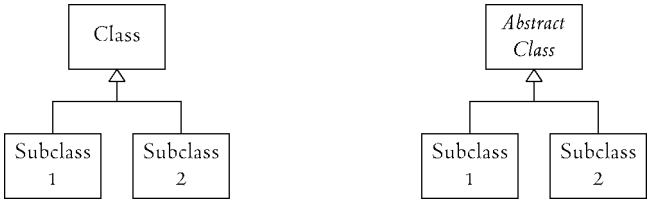


Figure A.5: UML notation for generalization/specializations.
Derived classes are connected to their parent class by association with a triangle at the side of the parent class. If the parent class cannot be instantiated itself but only serves as a template for the child classes, it is referred to as an abstract class and identified by an italicized class name.

An *abstract class* is a class that cannot be instantiated; no objects can be derived from an abstract class. Instead, an abstract class is used to derive subclasses. This way, the abstract class can define the behavior that is common to all of the derived classes without the need for a complete implementation of the class so that it can be instantiated. Abstract classes are indicated by a class name in italics.

A special type of abstract class is the *interface class*. An interface class does not contain any implementation. It is used to define the external functionality of the derived classes only. Interface classes are depicted as a lollipop symbol with the interface name printed next to it, as shown in figure A.6. The class that implements the interface, is connected to it by means of a solid line. Classes that apply the functionality that is provided by the interface, are connected to the interface by a dashed line with a simple arrowhead pointing to the interface. This indicates a dependency of the class that uses the interface. The association is one-directional, because an interface class is not aware of the classes that may use its functionality.

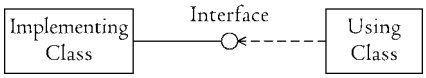


Figure A.6: UML notation for interface classes.
Interfaces are abstract classes that specify the operations of the child classes which implement those operations. The interface class is depicted as a lollipop symbol that is connected to the specialized class. The association of classes that use the implementing class through the interface is referred to as a dependency, depicted as a dashed one-way association.

Active classes and actors

All objects in a system are either passive or active. Passive objects wait for a message from another object before performing any task and never initiate any action. Active objects on the contrary have their own thread of control; they exist in parallel to other active objects and can initiate system activity. Active objects that run in parallel are referred to as *concurrent tasks* (Gomaa, 2000). Passive and active objects are instances of passive and active classes respectively. Passive classes are indicated in class diagrams by a thin border. Active classes are indicated by a thick border, as shown in figure A.7.

Figure A.7: UML notation for active classes and actors.
Active classes and actors are a special type of class: Both have their own thread of control. Active classes are located within the boundaries of the system under design; actors are active classes outside the system.

An actor is an outside user who interacts with the system. Although the word suggests that an actor is a human operator, an actor can in fact be any system or group of

systems that is external to the system under investigation. Because an actor must interact with the system, actors need to be of an active class type. They cannot be restricted to pure response-type behavior. Actors are depicted as a stick figure with a name written next to the symbol.

Object diagrams

A special form of the class diagram is the object diagram. It shows the interaction of objects from various classes at the instance level. Objects are depicted as a named box that is similar to the symbol that indicates a class. To distinguish an object from a class, object names are underlined as shown in figure A.8. Both the object name and the name of class to which the object belongs are optional. Object and class names are separated by a colon.

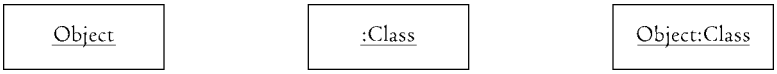


Figure A.8: UML notation for objects.
The object diagram is a special form of the class diagram in which the typical interaction of the instances of various classes is shown at a lower level of abstraction than in the normal class diagram. Objects are identified by an underlined name. Object and class name are separated by a colon.

Statechart diagrams

A statechart diagram shows the states that a system can be in, and the possible transitions between those states. As such, statechart diagrams depict the dynamic behavior of the system with time. Statechart diagrams are therefore used in *dynamic modeling*, which is also referred to as *behavioral modeling*.

In a statechart diagram, each state is shown as a rounded box with a descriptive name as shown in figure A.9. Additional to the name, a second compartment can be shown which contains an activity that is being executed for the duration of the state, or actions that are executed upon entry of or exit from the state. All three need not be defined; any combination of actions and an activity can be used.

Transitions between states are displayed as an arrow with a simple arrowhead pointing towards the new state. Next to the transition, an event name with an optional condition and an associated action can be specified. As shown in figure A.10, a condition is enclosed by square brackets and an action is preceded by a slash. An event is the cause for a state transition. It is an occurrence at a single epoch with conceptually zero duration. A conditional transition only takes place when the event occurs while the

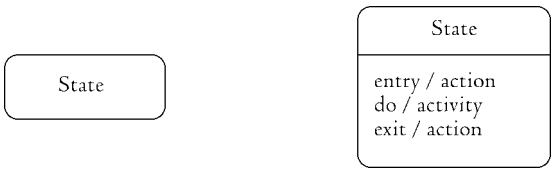


Figure A.9: UML notation for states.
States are depicted as a rounded box. A second compartment can contain entry or exit actions and an activity. Actions are assumed instantaneous; they cannot be interrupted. An activities is conducted as long as the system is in the state that contains the activity. When the state is left, the activity is interrupted.

condition is met; otherwise, the originating state is maintained. Actions are performed as the result of a transition. They take place at the epoch that the state transition occurs. Like the event that causes the transition, an action has conceptually zero duration.

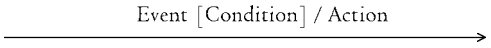


Figure A.10: UML notation for transitions.
A transition from one state to another is indicated by an arc with an open arrowhead. If the transition is triggered by an event, the event is printed with the transition. If an action is indicated, it is assumed to be performed instantaneously during the state transition. Transitions without a triggering event occur when the activity of the state that is left has completed.

Pseudostates

Pseudostates are additional elements in a statechart diagram. Like states, they are located at the beginning or end of a transition, but they are not an actual system state with finite duration. Instead, they are used to model decision points or to mark the beginning or end of the lifetime of the statechart.

Common pseudostates are shown in figure A.11. The initial pseudostate marks the state of the system upon its creation. The transition that exits the initial pseusostate is associated with the creation event itself. Thus, the initial pseudostate has a single transition that points towards the first true state that the system will be in immediately after it has been created. Similarly, the terminal pseudostate marks the end of the lifetime of a system. When a system reaches its terminal pseudostate, it no longer has a state and thus effectively ceases to exist. Not all statechart diagram contain a terminal

pseudostate. Branch pseudostates have a single transition that enters the pseudostate, and multiple transitions that exit it. Exactly one of the exiting transitions occurs at the same epoch at which the pseudostate is entered. A branch pseudostate is used to simplify modeling of multiple transitions that depend on a single event and on various outcomes of one or more conditions. The transition that enters the branch pseudostate is associated with the event; each of the transitions that leaves the pseudostate, is associated with a different outcome of the condition. Because the branch pseudostate is not a real state that can exist for a finite interval, one of the conditions for the outgoing transition must be true. A junction pseudostate is a simple connection of two or more incoming transitions and one outgoing transition. It can always be replaced by connecting the incoming transitions directly to the target of the junction pseudostate. Fork and join pseudostates are used in concurrent systems. At a fork pseudostate, concurrency is created by activating two new states at the same time. A join pseudostate does the opposite: two threads are joined into a single one. Incoming transitions to a join pseudostate must be triggered by the same event or synchronized through a synchronization pseudostate. A synchronization pseudostate is always located at the boundary between concurrent subsystems. It is connected to a join pseudostate on one side, and a fork pseudostate on the other. The synchronization pseudostate blocks the state of the concurrent thread on the side of the fork pseudostate in the state that enters the fork, until the thread can publish a token in the pseudostate. Similarly, it blocks the transition into the join state on the other side until a token can be collected. This way, transitions in the two concurrent threads can be synchronized. The number in the synchronization pseudostate identifies the number of transition tokens that can be stored. Transitions on the fork side – the publishing side – are not blocked until the synchronization state is full; transitions on the join side are not blocked as long as the synchronization pseudostate is not empty. Finally, two special types of pseudostates are the signal send and receive states. They are used to indicate transitions during which a signal is sent to or received from another system. Send pseudostates are instantaneous. Receive states are wait states with no action or activity; the transition out of a receive state is triggered by the receipt of the signal.

States can be nested. Nested states are depicted as state boxes inside the rounded box that indicates the superstate. When the system is in the superstate, it must be in one of the substates that are indicated. Inside a superstate, an initial pseudostate indicates which substate is entered when the superstate is entered for the first time. When the superstate is reentered, the initial pseudostate will reactivate the same initial substate as when the superstate was entered for the first time. Alternatively, a system can preserve the substate it was in during a period of inactivity of the corresponding superstate. When the superstate is reentered, the system will be in the same substate as when the transition out of the superstate occurred. This behavior is modeling using the history pseudostates as indicated in figure A.11. When a superstate is entered for the first time, the history pseudostate acts as an initial pseudostate, indicating the first substate to be activated. When the superstate is reentered, the history pseudostate is

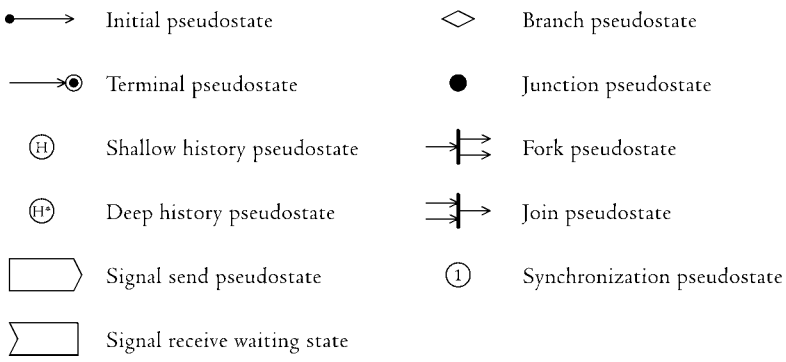


Figure A.11: UML notation for pseudostates.
Except for the signal receive waiting state, all states that are not indicated by a rounded box are passed instantaneously. Such pseudostates model more complex state transitions by separating them into a series of transitions that are connected by pseudostates, each with its own conditions or actions.

ignored and the preserved substate is reentered. A shallow history pseudostate preserves the state of the level at which it is used only; a deep history pseudostate also preserves the states at all of the deeper levels of nested states.

Sequence diagrams

A sequence diagram displays the interaction of a group of objects in a time sequence. It uses pictorial elements from both the class diagram and the statechart diagram. As the sequence diagram defines in what way multiple objects communicate by showing which messages are sent between them and in which order, the use of the sequence diagram belongs to dynamic modeling.

Objects and lifelines

Figure A.12 shows the basic form of a sequence diagram. At the top, a series of objects is shown. Sequence diagrams are drawn for objects, not for classes. The left-most object in a sequence diagram is often an actor that initiates the sequence of events in the diagram. From each object, a dashed line runs downwards. Along the line, time runs from top to bottom. This line is referred to as the lifeline for the object. When the object does not merely exist, but is taking part in the sequence of activities, a solid box is shown along the lifeline. Optionally, state markers can be inserted on the lifeline to indicate the momentary state of the object at the corresponding time point.

Although the vertical orientation of a sequence diagram is by far the most common representation, sequence diagrams are also drawn with the objects on the left and the lifelines running from left to right.

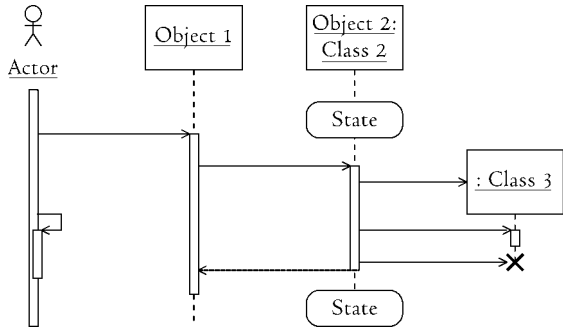


Figure A.12: UML notation for sequence diagrams.
The exact order of events and message exchanges between a series of objects is shown along lifelines for each object. Object symbols and state indicators are taken from class diagrams and statechart diagrams respectively.

Arrows between any two objects indicate messages that are sent between the objects. The sequence in which the messages are shown, corresponds to the actual sequence in which they are sent. The spacing between the messages in the diagram is not significant. Specific time intervals between two events must be indicated by inserting a constraint in the diagram.

The creation of an object by another one is shown by moving the object box for the object that is created downwards to the point where it is created. The destruction of an object is depicted by ending its lifeline and displaying a cross at the point where the message that terminates the object is received. When an object sends a message to itself that starts a second activity for the object, this is shown by adding a second box to its lifeline. This does not necessarily mean that classic recursion takes place. The second activity does not have to be a call to the same function; it only indicates a second activity for the same object.

Messages

Messages are annotated according to the rules as shown in figure A.13. The minimal description for a message is a name. Messages that are only sent when a certain condition is met, are shown with the condition preceding the message name; the condition is enclosed in brackets. A number can be printed in front of the name and separated from it by a colon, indicating the sequence in which the messages are sent. An asterisk at the position of a number indicates the repeated sending of the message in a loop or itera-

tion. A condition in brackets then specifies the criterion for continuing the iteration. If an operation is invoked with certain parameters, the parameter list is enclosed by parentheses and shown directly after the operation name.

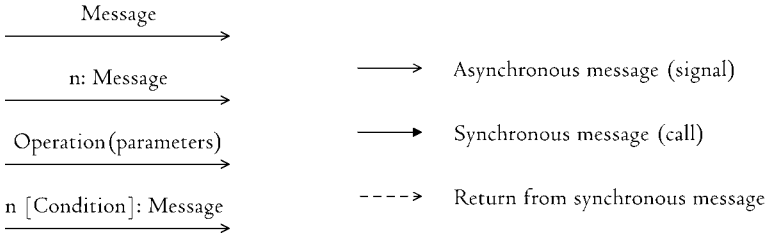


Figure A.13: UML notation for messages.
Messages are identified by numbers and names next to an arc in a sequence diagram. Similar to a statechart diagram, conditions can be indicated in brackets. The line type and arrowhead determine the type of message. Asynchronous messages are also referred to as a signal; synchronous messages are also referred to as a call. Calls are always followed by a return message, indicated by a dashed line.

Messages are either synchronous or asynchronous. Synchronous messages start an operation on the receiving object that runs in the thread of the calling object. The calling object blocks until the receiving object returns control. Synchronous messages are shown with a solid arrowhead; the return message is shown with a dashed line and a simple arrowhead. Asynchronous messages start an operation that runs in the thread of the receiving object. Meanwhile, the calling object continues executing in its own thread. Asynchronous messages are depicted with a simple arrowhead. Because they do not cause the calling object to block, there is no formal return message. When a return message is sent, the return message is a separate asynchronous message in the direction opposite to the message that initiated the operation.

General

The graphical elements that are used in the various types of UML diagrams are not strictly separated. Actor and class symbols are used in class diagrams and in sequence diagrams; state symbols are used in statechart diagrams and sequence diagrams. Apart from these symbols that are applied across various types of diagrams, a few additional notations are of general use for annotating UML diagrams.

Constraints and notes with explanatory text can be inserted in any type of diagram. Notes are depicted as rectangles with a folded upper right corner; they contain free-format text. Constraints are enclosed by braces and can be expressed in plain text,

pseudo code or a formal programming language. Both notes and constraints, shown in figure A.14, can be placed anywhere in the diagram. Constraints normally relate to a specific class, object, transition, or association and are then located next to that item.

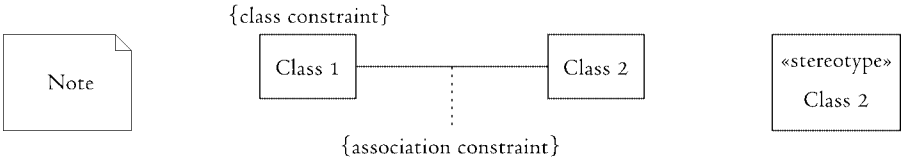


Figure A.14: UML notation for notes, constraints, and stereotypes.
Notes can be included in any type of UML diagram. They contain free text that provides additional information on the diagram as a whole or on specific elements in it. Constraints can only be related to the elements in a diagram. Enclosed by braces, they provide additional restrictions on the element. Stereotypes are another way to provide additional information on an element. Unlike constraints, they do not specify a restriction.

Stereotypes (Wirfs-Brock, Wilkerson, and Wiener, 1994) are used to derive a new building block from an existing UML element (Rumbaugh, Jacobson, and Booch, 1999). With stereotypes, the Unified Modeling Language itself can be extended by the user to match the peculiarities of a specific problem. Stereotypes are depicted as a descriptive text, enclosed by guillemets (« »), next to or in the modeling element. As described by Pinet and Lbath (2001), stereotypes in class diagrams are mainly used to define an additional classification of the objects, independent from the classification by classes. As such, stereotypes can be used as an alternative representation of inheritance.

References

Gomaa, H. (2000). *Designing Concurrent, Distributed, and Real-Time Applications with UML* (Addison-Wesley), ISBN 0-201-65793-7.

Pinet, F., Lbath, A. (2001). "Semantics of Stereotypes for Type Specification in UML: Theory and Practice" Springer *20th International Conf. Conceptual Modeling*, 339-353.

Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling Language Reference Manual* (Addison-Wesley), ISBN 0-201-30998-X.

Wirfs-Brock, R., Wilkerson, B., and Wiener, L. (1994). "Responsibility-Driven Design: Adding to Your Conceptual Toolkit" *Report on Object Analysis and Design 1* (2), 27-34.

B
Activities in Instrumentation Development

THE evolutionary development life cycle for flight test instrumentation systems that is shown in figure 1.7 on page 41 consists of three phases: application modeling, component development, and application synthesis. Grouped by the covering phases, each activity in the life cycle can be divided into a series of steps that lead to the accomplishment of the activity's goal.

Application modeling

Application modeling consists of requirements analysis, context analysis, and incremental design.

Requirements analysis

The steps in application requirements analysis are the following:

- Classify the application as an open-loop testing system, an adaptive testing system, a human-factors testing system, or an in-flight simulation system.
  The type of application will help to determine the correct requirements for and relations between signals in the signal model.
- Develop a signal model. Signal modeling starts at the back end of the system and iterates towards the front. This consists of:
  (a) Identify the output signals.
  (b) Classify the output signals as analog or digital, continuous or discrete; specify the signal types, and specify requirements such as accuracy, update rate, range, and resolution.
  (c) Identify the operators that produce the signals, identify the related input signals, and classify and specify the input signals.
  (d) Iterate step (c) until no operators are required to produce the input signals.
  The iteration has reached its end when the model no longer contains loose signals. This is the case when all signals in the model either stem from an operator, or enter the application as inputs from the environment.

Context analysis

The steps in application context analysis are the following:

- If the application is an in-flight simulation system, classify the system as a single-loop or double-loop fly-by-wire system, or a single-loop or double-loop man-machine system.
- Develop a context model. This consists of:
  (a) Identify the interfaces between the system and the environment.
  (b) Create an object diagram from the specification perspective. The model must contain all interfaces between system and environment and their mutual relationships. It depicts all instantiated excitation, measurement, and processing components.

Incremental design

The steps in incremental application design are the following:

- Develop a static model. This consists of:
  (a) Isolate a part of the context model that allows for implementation as an evolutionary prototype.
  (b) Create a class diagram from the implementation perspective, including all the objects from the context model that will be implemented in the current development cycle.
  (c) Select the sensors, actuators, digital signal processors and their peripheral hardware.
  (d) Identify generalizations to model the logical structure of the application.
  (e) Identify aggregations and compositions to model the physical structure of the application.
- Develop a dynamic model. This consists of:
  (a) Create a sequence diagram for those components that exhibit state-dependent behavior for known input sequences.
  (b) Create a statechart diagram for those components that exhibit state-dependent behavior for unpredictable input sequences.

Component development

Component development consists of analysis and throwaway prototyping, incremental design, generalized implementation, and specialization and unit testing.

Analysis and throwaway prototyping

The steps in component analysis and throwaway prototyping are the following:

- Develop a static model. This consists of:
  (a) From the static application model, isolate the groups of component

classes that are linked by generalization/specialization relationships.
(b) For each group, identify abstractions to separate various levels of application-specific properties.
(c) For each group, create a class diagram from the implementation perspective, including all abstractions and generalizations for the component.
(d) Specify the principle of operation for each component and extend the class diagrams to reflect the dependences on supporting classes.

- If appropriate, create throwaway prototypes for new hardware components. This consists of:
  (a) Create a rapid prototype design and implement it.
  (b) Unit-test the prototype.
  (c) If appropriate, use the prototype in incremental integration and prototyping of the application.
  (d) Feed back the results from the prototype to the component analysis.

Incremental design

The steps in incremental component design are the following:

- Classify each component as a platform, a device, or an analysis component.
- For each device and each analysis component, create sequence and/or statechart diagrams for the module.
- For each device component, create a design according to the following:
  (a) Create sequence and/or statechart diagrams for the device and the service.
  (b) Specify the interfaces to the service.
  (c) Create a mechanical design for the installation of the devices and the corresponding wiring.
- For each platform component, create a design according to the following:
  (a) Specify the interfaces to the port units based on the designs of the device components.
  (b) Create sequence and/or statechart diagrams for the port units.
  (c) Create a mechanical design for the installation of the hardware.

Generalized implementation

The steps in generalized component implementation are the following:

- If necessary, update the generic middleware according to its own life cycle.
- If necessary, implement the abstractions from the static component model.

Specialization and unit testing

The steps in component specialization and unit testing are the following:

- Implement new platform components. This consists of:
  (a) Assemble the digital signal processor and its peripherals.

(b) Develop a specialization of the middleware.
(c) Install the middleware and unit-test the platform in simulation.
- Implement new port units. This consists of:
  (a) Program the port abstractor.
  (b) Assemble and unit-test the port unit in a simulated environment.
- Implement new device units. This consists of:
  (a) Assemble the device and the wiring.
  (b) Program the service.
  (c) Assemble and unit-test the device unit in a simulated environment.
  (d) Calibrate the device and include the resulting corrections in the service.
- Implement new modules. This consists of:
  (a) Program the methods of the module.
  (b) Unit-test the module in simulation.

Application synthesis

Application synthesis consists of incremental integration, incremental prototyping, and system testing and operation.

Incremental integration

The steps in incremental application integration are the following:
- Assemble the application increment from the unit-tested components.
- Integration-test the application increment against the application analysis. This consists of:
  (a) Verify the correct interaction between all components.
  (b) Verify the achievement of the application requirements.

Incremental prototyping

The steps in incremental prototyping are the following:
- Operate the tested increment in its intended environment.
- If any deficiencies with respect to the requirements are revealed, identify the cause and return to the appropriate application design, component analysis, or component design activity.
- If no deficiencies are revealed and the increment is a subset implementation of the application analysis, return to the incremental application design activity to develop the next application increment.

System testing and operation

The steps in system testing and operation are the following:
- If appropriate, design and conduct a system test in which the application is tested against the user expectations. This consists of:

(a) Verify the achievement of the application requirements in the system's true operational environment.

(b) Validate the application for the fulfillment of its function.

- If any deficiencies with respect to the requirements are revealed, identify the cause and return to the appropriate application design, component analysis, or component design activity.
- If any deficiencies with respect to the fulfillment of the system's purpose are revealed, identify the cause and return to the application requirements analysis.
- Operate the system until its function changes.

C
Bayesian Estimation

THE probabilistic peer-to-peer clock synchronization algorithm that is presented in this thesis is an application of Bayesian parameter estimation. Its primary characteristics and the way it is integrated in a flight test instrumentation system middleware are described in chapter 5. The algorithm depends on a special case of parameter estimation: the determination of coefficients for a linear, multiple-input single-output system with a Gaussian a-priori parameter estimate vector, exactly known inputs and an output that is affected by Gaussian measurement noise. The derivation of the appropriate closed-form parameter estimator from Bayes' theorem for probability densities is the proof of proposition (5.2)/(5.3).

Linear multiple-input systems

Figure C.1 shows a block diagram for the linear map $f: \mathrm{R}^n \rightarrow \mathrm{R}$ which takes the input vector $\mathbf{u} \in \mathrm{R}^n$ to the scalar output $x = S\, b_i u_i$. The coefficients b_i are the unknown system parameters for which an estimate is sought. When the parameters are grouped in the vector $\mathbf{b}$, the system output equals the inner product of the input and parameter vectors $\mathbf{u}^\mathrm{T}\mathbf{b}$. The input $\mathbf{u}$ is observed without disturbance; the corresponding observation y of the system output x is affected by the measurement noise m.

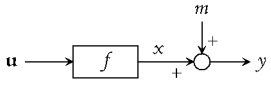


Figure C.1: Linear map with scalar output and measurement noise.

In case an a-priori estimate is available, the optimal estimate for the system parameters depends on Bayes' rule for joint probability density functions. The joint probability density theorem states that for any two events, the joint probability density equals the conditional probability density for one, multiplied by the unconditional probability density for the other:

Joint probability density theorem. $p_{a,b}(a, b) = p_{a|b}(a|b) \cdot p_b(b)$. (C.1)

Based on (C.1), Bayes' theorem for probability densities provides an expression for the conditional probability density function $p_{\mathbf{b}|y}(b|u)$, which represents the a-posteriori probability density of the unknown parameters $\mathbf{b}$, given the measurement u of the output y:

Bayes' theorem for probability densities. $$p_{\mathbf{b}|y}(b|u) = \frac{p_{y|\mathbf{b}}(u|b) \cdot p_{\mathbf{b}}(b)}{p_y(u)}.$$ (C.2)

The Bayesian estimate for the system parameters is found by evaluation of the right-hand side of (C.2).

Lemma. For the system in figure C.1: $p_{y|\mathbf{b}}(u|b) = p_m(u - \mathbf{u}^T b)$.

Proof. Because the system output x and the measurement noise m are fully correlated through the observed output y, the probability density function for the measurement is determined by a single integral:

$$p_y(u) = \int_{-\infty}^{\infty} p_x(x)\, p_m(u-x)\, dx.$$ (C.3)

Through partial integration, the density function for the system output x is eliminated from (C.3):

$$p_y(u) = \Big[F_x(x) p_m(u-x)\Big]_{x=-\infty}^{x=\infty} - \int_{-\infty}^{\infty} F_x(x)\; p_m'(u-x)\, dx$$ (C.4)

in which F_x denotes the probability distribution function for the system output and p_m' denotes the derivative of p_m with respect to x. With the exactly known input vector $\mathbf{u}$, the probability distribution of the output is a step function which depends on the system parameters $\mathbf{b}$:

$$F_x(x) = \begin{cases} 0 & x < \mathbf{u}^T\mathbf{b}, \\ 1 & x \geq \mathbf{u}^T\mathbf{b}. \end{cases}$$ (C.5)

Combination of (C.4) and (C.5) yields the conditional probability density function for the measurement when the system parameters are given:

$$p_{y|\mathbf{b}}(u|b) = \Big[F_x(x) p_m(u-x)\Big]_{x=-\infty}^{x=\infty} - \Big[p_m(u-x)\Big]_{x=\mathbf{u}^T b}^{x=\infty} = p_m(u - \mathbf{u}^T b).$$ (C.6)

The measurement noise m is assumed Gaussian with zero mean and variance C_{mm}:

$$p_{y|\mathbf{b}}(u|b) = \frac{1}{\sqrt{2\pi C_{mm}}}\; e^{(-1/2)(u-\mathbf{u}^T b)^2 / C_{mm}}.$$ (C.7)

The multivariate probability density function for the n-element parameter vector is modeled by a Gaussian distribution on the a-priori estimate $\mathbf{b}$:

$$p_{\mathbf{b}}(b) = \frac{1}{\left[(2p)^n \left|C_{\mathbf{bb}}\right|\right]^{1/2}} e^{(-1/2)(b-\mathbf{b})^T C_{\mathbf{bb}}^{-1}(b-\mathbf{b})} \tag{C.8}$$

in which $C_{\mathbf{bb}}$ is the covariance matrix of the a-priori parameter vector estimate. Finally, the probability density of the output is expressed as the n-fold integral over the parameter space – denoted as $\mathbf{B}$ – of the joint probability density of y and $\mathbf{b}$:

$$p_y(u) = \int_{\mathbf{B}} p_{y,\mathbf{b}}(u,b)\, d^n b. \tag{C.9}$$

The new estimate for the parameter vector $\mathbf{b}$ is chosen at the value for which the a-posteriori probability density function $p_{\mathbf{b}|y}(b|u)$ reaches its maximum. Because the evaluation of (C.9) does not depend on the actual parameter $\mathbf{b}$, the a-posteriori estimate is found by determining the maximum of the joint probability density function $p_{y,\mathbf{b}}(u,b)$ in the parameter subspace that is defined by the measurement on the output. Thus, the maximum is sought for the product of (C.7) and (C.8):

$$\max_b p_{y,\mathbf{b}}(u,b) = \max_b \text{const}\; e^{(-1/2)\left[(b-\mathbf{b})^T C_{\mathbf{bb}}^{-1}(b-\mathbf{b}) + (u-\mathbf{u}^T b)^2 / C_{mm}\right]}. \tag{C.10}$$

Due to its quadratic nature, the exponent in (C.10) has only one maximum. The Bayesian estimate of the parameter vector is therefore easily found by partially differentiating the logarithm of the joint probability density function to all of the components of $\mathbf{b}$, and setting the resulting Jacobi matrix to zero:

$$\frac{\partial}{\partial b} \ln p_{y,\mathbf{b}}(u,b) = 0 \quad \Leftrightarrow \quad 2(b-\mathbf{b})^T C_{\mathbf{bb}}^{-1} + \frac{-2}{C_{mm}}(u-\mathbf{u}^T b)\mathbf{u}^T = 0. \tag{C.11}$$

This yields an expression for the a-posteriori parameter vector $\mathbf{b}|y$ in terms of the a-priori estimate $\mathbf{b}$ and covariance matrix $C_{\mathbf{bb}}$ of the parameters, the known input vector, the output measurement, and the variance for the measurement noise:

$$\mathbf{b}|y = \left(C_{\mathbf{bb}}^{-1} + \frac{\mathbf{u}\mathbf{u}^T}{C_{mm}}\right)^{-1}\left(C_{\mathbf{bb}}^{-1}\mathbf{b} + \frac{u}{C_{mm}}\mathbf{u}\right) = B^{-1}\left(C_{\mathbf{bb}}^{-1}\mathbf{b} + \frac{u}{C_{mm}}\mathbf{u}\right). \tag{C.12}$$

(C.12) is the Bayesian estimator for the linear map of figure C.1. It is the basis for the Bayesian clock parameter estimator of proposition (5.2).

Confidence matrix

The random variables at the right-hand side of (C.12) are the a-priori parameter estimate $\mathbf{b}$ and the output measurement u. Thus, the covariance matrix of the a-posteriori parameter estimate is:

$$C_{\mathbf{bb}|y} = B^{-1}E\{\mathbf{x}\mathbf{x}^{\mathsf{T}}\}B^{-1}, \quad \mathbf{x} = C_{\mathbf{bb}}^{-1}(\mathbf{b} - E\{\mathbf{b}\}) + \frac{u - E\{u\}}{C_{mm}}\mathbf{u}. \tag{C.13}$$

Because $u - E\{u\} = m$, and the measurement noise m is uncorrelated with the a-priori parameter estimate, $E\{\mathbf{x}\mathbf{x}^{\mathsf{T}}\}$ reduces to B. This means for the covariance estimate of the a-posteriori parameter:

$$C_{\mathbf{bb}|y} = B^{-1}. \tag{C.14}$$

For the matrix B, the term *confidence matrix* is introduced. As the inverse of the a-posteriori covariance matrix, it represents the combined information on the accuracy of the a-priori parameter estimate, represented by $C_{\mathbf{bb}}^{-1}$, and the reliability of the new observation. The matrix $\mathbf{u}\mathbf{u}^{\mathsf{T}}$ contains information on the conditioning of the model parameters in the observation of the system; this measure for parameter observability is scaled by the measurement accuracy C_{dd}^{-1}. Right-multiplied by the inverse of the a-priori parameter covariance matrix, the confidence matrix provides a transition matrix that propagates the a-priori parameter estimate to the a-posteriori value.

The results of (C.12) and (C.14) are equivalent to the corrector equations of a discrete-time Kalman filter through a conversion by the matrix inversion lemma. For the Kalman filter model, the system in figure C.1 is regarded as the output equation of a linear state space model with measurement noise, for which $\mathbf{b}$ is the unknown state vector and $\mathbf{u}^{\mathsf{T}}$ is the output matrix. For analysis of the peer-to-peer clock synchronization algorithm, the version that is presented here is preferred because it reveals the role of the various quantities that appear in the parameter estimation problem more naturally. For numerical reasons, the state estimation formulation of the Kalman filter is better suited for actual implementation of the model. The computationally optimal form of the algorithm is presented in chapter 5.

D
Case Study Implementation Details

ALL of the instrumentation development activities that are listed in appendix B were applied during the human factors flight test program that is described in chapter 6. The middleware incorporates all of the features that are presented in chapter 4, including the probabilistic peer-to-peer clock synchronization algorithm that is introduced in chapter 5. The middleware was developed during the main instrumentation system upgrade for the Delft University's laboratory aircraft that was completed in the year before the human factors flight test program. Below are some details on both programs, as well as an outlook to future application of the resulting instrumentation system.

Preceding project: flying classroom instrumentation upgrade

From 1998 through 2000, the instrumentation system on board the Delft University's Cessna Citation – which is known as the *flying classroom instrumentation system* – was upgraded to use six improved student-observer stations, an inertial-GPS hybrid navigation system, and a more flexible network layout. The development methodology and the middleware that are presented in this thesis were developed during this time.

- In 1998, the instrumentation system layout was designed and the hardware components were selected in an effort of approximately four man-months by the project leader. The engineers in the flight department were involved in an advisory role.
- In 1999, new hardware components were acquired, integrated, and tested in a joint effort of approximately one man-year by the project leader and two engineers in the flight department.
- A first version of the middleware – which did not include the probabilistic peer-to-peer clock synchronization and did not support entities as described in section 4.5, was implemented in C and C++ in an effort of approximately half a man-year by the project leader.
- In 2000, the migration of the previous flying classroom instrumentation system to the new system was completed in a joint effort of approximately one man-year by the project leader and three engineers in the flight depart-

ment. Simultaneously, additional visualization software components for the application were developed by an engineer in the technical department in a six-month effort. In another two-month effort, the final version of the middleware, including clock synchronization and entities, was implemented by the project leader.

In summary, thirty six man-months were invested in the project, eight man-months of which were used for developing the middleware.

Case study and future projects

Following the flying classroom instrumentation system upgrade and reusing the middleware and parts of the application, the case study project was initiated January 2001 and completed in May of the same year.

- A team of seven people, including a pilot, four engineers from the flight and technical department, the head of the flight department, and the project leader, invested a total effort of approximately one man-year in the project. The main contributions to this total are two four-month efforts by the project leader and an instrumentation engineer, developing the application and its software, and implementing the hardware components respectively.
- Two flight test campaigns were completed – one in May and one in August – with a total of fifteen flight hours.

Since summer 2001, the instrumentation system has repeatedly been used in the flying classroom configuration. In 2003, a new project was initiated in which the system will be used for aerodynamic parameter estimation. In this project, the middleware, the platforms, and the data acquisition components will be reused without modification.

Glossary

Abstract class
An abstract class is a class that cannot be instantiated. As such, it is used as a template for deriving subclasses instead of as a template for creating objects.

Abstraction
Abstraction is the process of isolating information on an entity that is essential from a certain perspective and leaving out all other, irrelevant information. Abstraction is used to identify what information can be hidden (see *information hiding*) and thus how an *encapsulation* should be performed.

Accuracy
The accuracy of a measurement is a quantification of the difference between a measured value and the true value of the measurand.

RNP accuracy specifies the risk of not achieving the navigation requirements due to an excessive total system error (Kelly and Davis, 1994).

In clock synchronization, accuracy refers to the external synchrony of an ensemble of clocks.

Action
An action is an operation with assumed zero duration.

Activity
An activity is an operation with non-zero, possibly infinite duration.

Actor
An actor is an active object, or a group of active objects, that is external to the system and that interacts with it. Active objects are autonomous objects that exhibit behavior that is independent from their environment. Active objects can either be human users, or systems or software components with their own thread of control.

Address space
Address space is the logical memory that is visible to a process.

Agreed data
Agreed data is the result of raw data measurement, preprocessing, conversion to SI units, and validation for plausibility. An agreed data element has been judged to be a correct image of the corresponding real-time entity (Kopetz, 1997).

Atom
An atom is an indivisible operation. It cannot be interrupted or preempted, nor is it possible that two atoms in different threads overlap.

Binary semaphore
A binary semaphore is a two-state variable used to control mutual exclusion. Access to a binary semaphore is limited to an atomic acquire operation and an atomic release operation.

Channel
A channel is the communication link between multiple modules under control of a *message-oriented middleware*, by which data between *publishers* and *subscribers* is exchanged. See also *signal*.

Class
A class is a type of object. It is used to define the characteristics and the behavior of the objects that are instantiated from it, or the subclasses that are derived from it.

Concurrency
Concurrency is the possibility for simultaneous occurrence of multiple activities or events. A concurrent system is a system with more than one threads of control.

Context analysis
Context analysis is the process of modeling the environment of an application and specifying the interfaces between the application and its environment. Context analysis is an activity in the analysis phase of a development life cycle.

Context switching
Context switching is the activation of the appropriate address space for the process that is being activated in a *multiprogramming* or a *symmetric multiprocessing environment*. Context switching is necessary because these environments have a single memory space, while each process must execute in its own address space.

Control
Control is the authority over the states of a process, thread, or fiber in a concurrent application.

Coordinated universal time (UTC)
Coordinated universal time is a non-chronoscopic timescale that is based on the SI second, while being kept in synchrony with *universal time* by the insertion of leap seconds.

Counter
See *physical clock*.

Data anomaly
A data anomaly is a difference between multiple occurrences of the same information in a database. Data anomalies are introduced when a non-normalized database is updated without addressing all data redundancies.

Data redundancy
Data redundancy is the multiple occurrence of a single information fragment in a database, allowing for inappropriate changes to each of the individual occurrences.

Deadlock
A deadlock is a failure of a concurrent system in which multiple threads are infinitely blocked due to a mutual dependency.

Device
A device is a hardware component in a flight test instrumentation system that is used for interaction between the system and its environment. Sensors are input devices, actuators are output devices.

Distributed processing environment
A distributed processing environment is a concurrent system with two or more CPUs that do not share any memory. Communication between the nodes of a distributed system can only take place through a network.

Encapsulation
Encapsulation is the strategy of grouping related information into a single entity. Encapsulation is a means to achieve *information hiding*, by packaging public interfaces with hidden implementations into a single object.

Epoch
An epoch is an instant of time.

Evolutionary prototyping
Evolutionary prototyping is the development of a preliminary version of the application with reduced reliability, durability, or performance, in order to obtain early feedback that can be applied in an iteration of the complete development life cycle.

External clock synchronization
External clock synchronization is the process in which a clock or an ensemble of clocks is synchronized with an external reference time.

Fiber
Fibers are concurrent activities within a single thread, invisible to the operating system and scheduled by the thread itself.

Flight technical error (FTE)
The flight technical error is the difference between the desired position of an aircraft, and the position reported by a navigation system. In general, the flight technical error is fed back to and minimized by the pilot or automatic flight control system that guides the aircraft. With the navigation sensor error (NSE), the FTE is part of the total system error (TSE).

Granularity
The granularity of a physical clock is the interval between two ticks; the resolution of a physical clock equals its granularity. Granularity has the unit of time; it is the inverse of the clock's tick frequency.

Graph
A simple graph is a set of vertices V and a set of unordered pairs of elements of V that are called edges; the vertices are also referred to as nodes. A graph that contains multiple edges for a single pair of vertices is a multigraph. A graph with one or more edges that connect a vertex with itself, is called a pseudograph. Graphs are usually depicted as a set of dots that represent the vertices and that are connected by arcs that represent the edges.

Inceptor
An inceptor is a pilot's control device that is used to command the flight control system.

Incremental prototyping
Incremental prototyping is the development of a preliminary version of the application with reduced reliability, durability, or performance, in order to obtain early feedback that can be applied in an iteration of the development life cycle from the design phases, but not including the analysis phase. See also *evolutionary prototyping*.

In-flight simulation
In-flight simulation is a piloted simulation of an aircraft in flight, performed on a host aircraft that is controlled indirectly, in order to match the dynamics of the simulated aircraft as closely as possible.

Information hiding
Information hiding is the concept of separating the characteristics of an entity that are required by its environment from those that are not, and making the latter invisible to anything outside the entity itself. Information hiding can be regarded as the rationale behind object orientation.

Inheritance
Inheritance is the concept of letting new classes which are derived from a so-called parent class, obtain all of the behavior and characteristics of the parent class. By extending a derived class or by changing parts of it, modified behavior with respect to the parent class can be obtained while reusing as much of the parent class implementation as possible.

Instance
An instance is a realization or implementation of a type, based on a generic description. An object is an instance of a class.

Integration testing
Integration testing is the verification of the correct interaction and cooperation of the unit-tested system components that are assembled during the incremental integration phase of a development life cycle. Integration testing takes place in a simulated environment and is the final verification before the system is tested in its real environment. See also *unit testing* and *system testing*.

Intelligent instrumentation
An intelligent instrument combines a sensor or actuator with the associated intelligence in a single object, in order to produce or accept agreed data.

Internal clock synchronization
Internal clock synchronization is the process in which an ensemble of clocks in a distributed real-time system is mutually synchronized.

International atomic time (TAI)
International atomic time is the chronoscopic international time standard that adheres as closely as possible to the SI second. TAI is maintained by comparing over two hundred time standards worldwide.

Iron bird
An iron bird is a hardware-in-the-loop simulation facility in which flight-worthy aircraft systems are tested. An iron bird is usually a steel test rig that resembles the shape of the aircraft.

Job
A job is an activity of finite duration in a concurrent system, with a clearly defined beginning and end. Only a job can be subject to a deadline.

Logical clock
A logical clock is an abstract entity from which time can be observed. A logical clock is based on the output of a physical clock by applying a scale factor and an offset, with the intent to convert the physical clock's counter into the same units as the reference time, and to compensate for synchronization discrepancies.

Measurand
The measurand is the physical quantity that is being observed in the process of measurement.

Memory space
Memory space is the physical memory in a computer system. See also *address space*.

Message-oriented middleware
A message-oriented middleware (Banavar et al, 1999) is a middleware that links the components of a distributed application by means of messages that are exchanged between them. Messages are sent through *channels* by modules that are referred to as *publishers*; the arrival of a message in a channel triggers the activation of a *subscriber*.

Middleware
Middleware is a layer of software that sits above the heterogeneous operating system to provide a uniform platform above which distributed applications can run (Bacon, 1992). See also *message-oriented middleware*.

Module
A module is a software entity that is responsible for the correct acquisition or computation of a signal, and for the correct and timely availability of an outgoing signal to its users. It provides a virtual signal that is independent of the module's implementation.

Multiprogramming environment
A multiprogramming environment is a concurrent system in which there is only a single CPU. Concurrency is achieved by scheduling.

Navigation sensor error (NSE)
The navigation sensor error is the difference between the true position of an aircraft, and the position reported by a navigation system. With the flight technical error (FTE), the NSE is part of the total system error (TSE).

Object
An object is an instance of a class.

Physical clock
A physical clock or hardware clock is a time measurement device that consists of an oscillation device and a counter. The oscillator produces ticks, which increment the counter; observations of the counter provide the measurements of time.

Polymorphism
Polymorphism is the object-oriented concept of assigning different implementations to a single entity, depending on its context.

Port
A port is a hardware component that connects directly to a digital signal processor in a flight test instrumentation system. Ports are used to interface with *devices* for the purpose of interaction between the system and its environment.

Precision
The precision of a measurement is a quantification of the difference between successive measurements of the same measurand. Precision is an indication for repeatability: the closeness at which a measurement can be repeated.

In clock synchronization, precision refers to the internal synchrony of an ensemble of clocks.

Process
A process is an activity with one or more threads of control that executes in its own address space.

Process time
Process time is an observable, discrete timescale that corresponds to the progress of the computations in a signal processing system.

Proxy
A proxy is a software entity that represents an entity at a another node of a distributed application. It reproduces the behavior of the remote entity in order to let local entities interact with the remote entity without knowing about its remote location.

Publisher
In an application under control of *message-oriented middleware*, a publisher is the module that produces data that is used by other modules. The data is exchanged between the publisher and the *subscriber* by means of a *channel* or *signal*.

Race condition
A race condition is a failure of a concurrent system in which the outcome of activities unintendedly depends on the relative timing of events. Race conditions occur when the correct functioning of an operation depends on the previous completion of an activity in another thread, whereas the sequence of activities between threads is not properly synchronized.

Rate-monotonic analysis (RMA)
Rate-monotonic analysis is the ordering of scheduling priorities for the threads in a concurrent application, by which higher priorities are assigned to periodic activities with shorter computation periods (Liu and Layland, 1973).

Real time
Real time is an assumed Newtonian time continuum that is not directly observable.

Real-time processing
Real-time processing is the execution of computations or actions for which success or failure not only depends on the correctness of the results, but also on the time at which these results are delivered (Stankovic and Ramamritham, 1993).

Required navigation performance (RNP)
Required navigation performance (Kelly and Davis, 1994) is the aircraft containment surface about a nominal flight path, referred to as the tunnel, that is used to define a maximum total system error that will not result in unsafe operation.

Requirements analysis
Requirements analysis is the process of specifying the functionality of an application. It is the first activity in the analysis phase of a development life cycle.

Resolution
The resolution of a measurement is the smallest change in value that can be observed from the measurement system used.

Scheduling
Scheduling is the act of allocating processing time to the *threads* of a concurrent application when the total number of threads exceeds the number of available processors. Within a thread, scheduling also refers to the allocation of processing time to one of multiple *fibers*.

Semaphore
See *binary semaphore*.

Sensor fusion
Sensor fusion is the process in which information is gathered from the combination of observations from multiple sensors.

Signal
A signal is a time-dependent variation of a detectable quantity, the state of which is used to convey information or to excite a system.

Within a digital signal processing application, a signal is the time-aware equivalent of a *channel*.

Subscriber
In an application under control of *message-oriented middleware*, a subscriber is a module that depends on data that is produced by another module, the *publisher*. The subscriber is activated by the middleware when new data arrives in the *channel* that links the publisher and the subscriber. In a signal processing system, the middleware is time aware and only activates the subscriber when valid data is available in all signals it subscribes to.

Symmetric multiprocessing environment
A symmetric multiprocessing environment is a concurrent system with two or more CPUs that share a single memory space. A symmetric multiprocessing environment is the only system on which true concurrency of multiple threads in a single process can be achieved.

System testing
System testing is the verification and validation of a system in its real environment.

Task
A task is the activity that an object must complete to fulfill its function in the system. For an active object in a concurrent system, the term *thread* is used. A limited activity inside a thread or fiber, possibly with a deadline, is referred to as a *job*.

Thread
A thread is a concurrent activity within a *process*. Threads are scheduled by the operating system. A thread is the smallest concurrent element that is visible to the operating system; concurrency within a thread is realized by means of *fibers*.

Throwaway prototyping
Throwaway prototyping is the implementation of a simplified instance of the application in terms of durability, reliability, or performance, with the goal to test the correctness of the application models during the analysis phase. Throwaway prototypes are abandoned when the analyses have been completed.

Ticker
A ticker is a time-controlled entity in a signal processing system that publishes an event upon the arrival of certain epochs, with the aim to activate other entities at those epochs by means of a message-oriented middleware.

Total system error (TSE)
The total system error is the difference between the desired position of an aircraft and its actual position. The TSE is the sum of the navigation sensor error (NSE) and the flight technical error (FTE). TSE is an important parameter to required navigation performance (RNP).

Tree
A tree is a connected, simple *graph* without circuits. In a tree, the number of edges equals the number of vertices minus one. A tree without a root is called a free tree. If one vertex of the tree is designated as the root, the tree is hierarchically ordered. In a rooted tree, each edge is separated into a parent and a child vertex, in a way that the parent is the vertex that is closer to the root.

Unit testing
Unit testing is the verification of the correct functioning of a component of a system in a simulated environment. Unit testing is the final activity in a component development life cycle. See also *integration testing*.

Universal time (UT)
Universal time is a set of chronoscopic timescales that are based on the mean diurnal motion of the Sun, while being as uniform as possible. See also *coordinated universal time*.

Validation
Validation is the act of confirming the intended behavior of a system with respect to its functional requirements. Software validation was defined by Boehm (1981) as “to establish the fitness or worth of a software product for its operational mission”.

Verification
Verification is the act of confirming the correct behavior of a system with respect to its design. Software verification was defined by Boehm (1981) as “to establish the truth of correspondence between a software product and its specification”.

References

Bacon, J. (1992). *Concurrent Systems: An Integrated Approach to Operating Systems, Database, and Distributed Systems* (Addison-Wesley), ISBN 0-201-41677-8.

Banavar, G., Chandra, T., Strom, T., and Sturman, D. (1999). "A Case for Message Oriented Middleware" Springer *13th International Symposium on Distributed Computing*, 1-18.

Boehm, B.W. (1981). *Software Engineering Economics* (Prentice Hall), ISBN 0-13-822122-7.

Kelly, R.J. and Davis, J.M. (1994). "Required Navigation Performance (RNP) for Precision Approach and Landing with GNSS Application" ION *Navigation 41* (1), 1-30.

Kopetz, H. (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications* (Kluwer Academic), ISBN 0-7923-9894-7.

Liu, C.L. and Layland, J.W. (1973). "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment" *Journal of the ACM 20* (1), 46-61.

Stankovic, J.A. and Ramamritham, K., eds. (1993). *Advances in Real-Time Systems* (Institute of Electrical and Electronics Engineers), ISBN 0-8186-3792-7.

Index

A
A-7 aircraft 75
abstract class 71, 112, 208
– *defined* 229
abstraction 27, 60, 75, 80
– *defined* 229
– of components 70, 73, 176
abstractor, port 79
accuracy
– *defined* 229
– in clock synchronization 133, 148, 151
– in signal diagram 51, 162
action 154, 155, 158, 210
– *defined* 229
activities, middleware 121–129
activity 210
– *defined* 229
actor
– *defined* 229
– *notation for* 209
– primary 47
Ada 77
adaptive testing system 48, 55
adder 51
address space 31
– *defined* 229
aggregation 59, 77
– *notation for* 206
agreed data 9, 42, 78, 110, 178
– *defined* 230
alias 120–121
allocation 106
analysis module *See* module
analysis phase 20
anomaly *See* data anomaly
aperiodic thread 107
application analysis 40
– *See also* requirements analysis *and* context analysis
application synthesis *See* synthesis
application type 48–50, 95, 160
application-dependent components, middleware 115–116
architecture, middleware 111–116
arrival time 109
association 61
– *notation for* 206
– class
– *notation for* 206
atom 37
– *defined* 230
atomic time *See* time
augmentation 59

B
Bayesian parameter estimation 135, 136–142, 223–226
bias 141, 150, 152
binary semaphore 37, 111
– *defined* 230
boundaries
– in system analysis 53
– of logical topology entities 121

C
callback 80, 177
campaign 65
carefree handling 11
case study 160–186
causality
– in module ensembles 95
central processing unit 31, 35, 87, 170
chain
– excitation 2, 3, 42, 53, 94
– measurement 2, 3, 53
– processing 2, 3, 53
– reproduction 2

chains
– in instrumentation systems 2–3, 53–54
– *See also* chain
channel 97, 114
– *defined* 230
child class 28
chronoscopic timescale 104, 143
clairvoyant scheduler 111
class 26, 28
– *defined* 230
– *notation for* 205
– abstract 71, 112, 208
– *defined* 229
– active
– *notation for* 209
– association
– *notation for* 206
– interface 209
class diagram 54, 59, 74, 205–210
– middleware architecture 111
– perspective *See* perspective
clock 96, 98, 99, 102, 176
clock condition 99, 102
clock period 143
clock synchronization 96, 98, 113, 132–158
– activities 153–158
– delay 138
– simulation 152
– external 104, 138–140
– *defined* 232
– simulation 150–152
– internal 103, 138
– *defined* 233
– simulation 147–150
closed-loop instrumentation system 5, 60, 160
COMET (Concurrent Object Modeling and Architectural Design Method) 40
component analysis 40, 72–74
component design and implementation 86–89
component development 40, 70–89
– in case study 176–182
composition 59
– *notation for* 206
computation time 109, 123, 128
concept 18
– information hiding versus encapsulation 28
conceptual perspective *See* perspective
concurrency 8, 29–38, 99
– *defined* 230
concurrent system 34
concurrent tasks 30, 209
conditioning, clock synchronization 144–146
confidence 136, 142
confidence matrix 142, 226
constraint 180
– *notation for* 216
context analysis 40, 46, 53–59
– *defined* 230
– in case study 166–169
context switching 31, 109, 128
– *defined* 230
control 121
– *defined* 230
– for scheduling 32, 123
control augmentation 59
coordinated universal time *See* time
CORBA (Common Object Request Broker Architecture) 92, 96
core components, middleware 113–115
correct clock 100
correctness
– of stochastic information 136, 140–142
counter 136, 140
counter rollover *See* rollover, counter
CPU *See* central processing unit
criterion, synchronization *See* synchronization criterion

D

data acquisition components 42, 71, 94, 129
data anomaly 7, 66
– *defined* 231
data preprocessing 10
data processing components 42, 71, 94
data redundancy 7, 66
– *defined* 231
data replay 2, 183
deadline 105–111, 115, 126

deadlock 36
– *defined* 231
decentralization 9
deep alias 120
delay 180
– in clock synchronization *See* clock synchronization
departure 50
derived class 28, 70
design concept 18
design notation 18
design phase 20
– application *See* incremental application design
deterministic clock synchronization 133
development method 18
development phases 20
device 77, 170
– *defined* 231
– unit 83, 88–89, 129
device module *See* module
differentiation
– paradox of computer platforms 4
digital signal processor 4, 54
dispatching 98, 106, 123, 125–128
display augmentation 59
distributed processing environment 31
– *defined* 231
– scheduling 106
distributed system 8, 61, 87, 116
documentation 7, 65–67
drift 140, 142, 146, 149
driver 78, 179
DUECA (Delft University Environment for Communication and Activation) 97
dynamic modeling 38, 61, 88, 89
– in case study 174, 177, 179

E
earliest deadline first 36, 108, 127
efficiency
– of a flight test 4
encapsulation 27–28
– *defined* 231
– of topology by middleware 117
endianess 88, 176
ensemble, module 95
entity 119–121
epoch
– *defined* 231
equivalences, hardware-software
– in interface layers 80–82
event 35, 99
– module 129
– signal 115, 128
event-driven system 33
evolutionary prototyping *See* prototyping
exchangeability 78
excitation chain *See* chain
external clock synchronization *See* clock synchronization
external time equation 138

F
feedback
– during system development 20–26, 43
– of processing results 7, 48–49, 55, 60
fiber 30, 155
– *defined* 232
fixed-priority scheduling 107–108
flight envelope protection 11
flight technical error 13
– *defined* 232
flight test engineer 7, 53, 65
flight test instrumentation 2
– life cycle 39–44
fly-by-wire 49
– double-loop system 59
– single-loop system 57
free module 129
free tree 117
frequency
– of discrete signal 51, 162
function
– in object-oriented modeling 8
– of components 73

G
generalization 28, 59, 60, 73, 87
– *notation for* 207
– of components 70
GPS 165, 169
– time 105, 136, 147
granularity 102, 136, 140
– *defined* 232
graph 117
– *defined* 232

H
happened before relation 99
hardware-software equivalences *See* equivalences
heuristic scheduling 107
hierarchical layers 75–83
HOOD (Hierarchical Object-Oriented Design) 77
human-factors testing system 48, 56, 161

I
identifier *See* signal identifier
implementation perspective *See* perspective
implementation phase 20
inceptor 6, 48, 49, 160
– *defined* 232
incremental application design 40, 46, 59–61, 170–175
incremental integration 64, 182
incremental prototyping *See* prototyping
in-flight simulation 11
– *defined* 232
in-flight simulation system 49, 56
– *See also* fly-by-wire *and* man-machine
information hiding 26–28, 75
– *defined* 233
inheritance 26, 28–29, 71, 88
– *defined* 233
– *notation for* 207
– of middleware elements 113
input components 71, 94, 176
installation 174
instance 28
– *defined* 233
instantaneous synchronization 143
instrumentation engineer 7, 66, 113
integration design 174
integration testing 20, 61
– *defined* 233
– in case study 182
integration, system 43, 61, 183
intelligent instrumentation 9–11
– *defined* 233
interface 27, 53, 73
– *notation for* 209
– class 209
– in object-oriented modeling 9
– to instrumentation system *See* interface components
interface components 71, 75, 77–83, 166
internal clock synchronization *See* clock synchronization
internal time equation 138
international atomic time *See* time
iron bird 64
– *defined* 233

J
job 107, 109–111, 128
– *defined* 234
Julian date 105

K
Kalman filter 165, 226

L
laboratory testing 64, 183
laxity 109
layers *See* hierarchical layers
leap second 105, 143
learning behavior 136
least laxity first 109–111, 127
least-squares parameter estimation 135
life cycle 18–26, 184
– of flight test instrumentation 39–44
– of middleware 42, 93
linear map 136, 223
logical clock 100, 103, 113, 136
– *defined* 234
logical topology 119–121

M
man-machine
– double-loop system 59
– interface 169
– single-loop system 59
maximum-likelihood parameter estimation 135
measurand
– *defined* 234
measurand list 7, 66
measurement chain *See* chain
memory space 31
– *defined* 234
message 31, 37, 97, 124, 133, 138, 139, 152, 154, 177, 214
– *notation for* 215
– in ordering of events 99–103

message-oriented middleware 97
– *defined* 234
method 18, 73
middleware 32, 42, 87, 92–129, 176
– *defined* 234
– activities 121–129
– in aerospace software 92
– architecture 111–116
– requirements 93–98
mode change 107, 116, 125
module 80, 83–86, 89, 94–95, 115
– *defined* 234
– free, stream, and event 129
multiprocessing environment, symmetric 31
– *defined* 237
– scheduling 106
multiprogramming environment 31
– *defined* 234
mutual exclusion problem 37

N
navigation sensor error 13
– *defined* 234
negator 51
network 31, 87, 117, 133, 174
Newtonian time 102
normalization, clock parameter 145
normalization, database 7, 66
notation 18

O
object 28, 126
– *defined* 235
– *notation for* 210
– diagram 54
object-oriented modeling 8–9, 26–29
offset 136
open-loop instrumentation system 5, 60
open-loop testing system 48, 54
operation, system 64
operator 51, 85
– in signal diagram 51, 164
– middleware component 116
ordering of events 99–102
output components 71, 94
overdetermined system 135
overload
– scheduling 109

P
pacer 114, 123–124
paradox
– of computer platform differentiation 4, 11
– of real-time and step-time processing 114
parameter list *See* measurand list
partial ordering *See* ordering of events
peer-to-peer clock synchronization 135–136
– *See also* clock synchronization
performance
– of real-time systems 102, 105–111
periodic thread 107
perspective
– flight path display 160
– of class diagram 205
– conceptual 54
– implementation 59, 60, 74
– specification 54, 66
physical clock 102
– *defined* 235
physical time 102
physical topology 117–119
pipe 116, 117, 176
platform components 42, 71, 87–88, 111
– in case study 170–174
polymorphism 29
– *defined* 235
port 78, 170, 172
– *defined* 235
– unit 82, 87, 170, 176
– unit, in case study 178
precision 51, 164
– *defined* 235
– in clock synchronization 133, 147, 151
preemption 32, 108
preprocessing 10
primary actor 47
priority scheduling 36, 106
probabilistic clock synchronization 133
– *See also* clock synchronization
problem domain 53, 65
procedure-driven system 32
process 30
– *defined* 235
process noise 142, 152

process time 114, 123, 128
– *defined* 235
processing chain *See* chain
producer/consumer modules 119
producer/consumer problem 37–38
proper time 103
prototyping 22–24
– evolutionary 22, 39
– *defined* 231
– incremental 22, 40, 64, 74, 184
– *defined* 232
– throwaway 22, 40, 70, 74, 180–182
– *defined* 237
proxy 92, 116, 125
– *defined* 235
pseudostate 158
– *notation for* 211
publication time 115, 128
publisher 97, 114
– *defined* 235

Q
query 80
quick-look devices 54

R
race condition 37
– *defined* 236
range
– in signal diagram 51
rate-monotonic analysis 36, 107
– *defined* 236
real time 98, 102–103
– *defined* 236
real-time processing 3, 60, 95, 114, 123
– *defined* 236
redundancy *See* data redundancy
reference time 103–105, 135, 138
registry 114
– activities 124–125
relativity 103, 104
release time 110, 115, 126
remote procedure call 96
reproduction chain *See* chain
required navigation performance 13
– *defined* 236
requirements analysis 40, 46, 47–53
– *defined* 236
– of components 74
– of middleware 93–98
residual 140
resolution
– *defined* 236
– in signal diagram 51
reusability 70, 78, 176, 186
RMA *See* rate-monotonic analysis
RNP *See* required navigation performance
rollover, counter 144, 146
root, of tree graphs 118
round-robin scheduling 35

S
safety 4, 48, 49, 65, 105, 186
safety pilot 50, 166
scenario 38, 47
schedule interval 115, 126–129
scheduler
– as middleware component 115
– *See also* scheduling
scheduling 32–36, 96, 98, 105–111
– *defined* 236
– activities 125–129
scope 119
second
– as basis for timescales 104
secondary signal
– in signal diagram 53
semaphore *See* binary semaphore
sensor fusion 165
– *defined* 237
sequence diagram 61, 89, 213–215
– clock synchronization 157
– in case study 177, 180
– registration 124
service 79, 89
shortest job first 36
sidereal time 104
signal 51, 85–86, 94, 95, 114, 115, 162
– *defined* 237
signal diagram 50, 115
– *notation for* 50
– in case study 162–165
– logical topology in 120
signal generator 94
signal identifier 119, 124

signal interval 115, 128
signal modeling 47, 50–53, 85
signal type 50, 86, 128–129
simulation 94
– in-flight 11
– *defined* 232
– probabilistic clock synchronization 146–152
SJF *See* shortest job first
slack time 109
smart sensors *See* intelligent instrumentation
solution domain 59, 66, 170
specialization 28, 87
– *notation for* 207
– of components 70
specification perspective *See* perspective
sporadic server 108
sporadic thread 107
stability 109
– of scheduling 108, 128
stability augmentation 59
stacking synchronization messages 158
state 96, 97, 115
– *notation for* 210
statechart diagram 61, 89, 210–213
– clock synchronization 155
– in case study 175, 179
– pacing 123
– scheduling 126
static modeling 38, 53, 60–61, 170
– in case study 179
step-time processing 3, 60, 114, 123
stochastic correctness 136, 140–142
strategy 18
– encapsulation versus information hiding 28
stream signal and module 128
subscriber 97, 115
– *defined* 237
substitutability 29
subtyping 29
symmetric multiprocessing environment 31
– *defined* 237
– scheduling 106
synchronization 95, 97, 101
– *See also* clock synchronization
synchronization criterion 137, 138, 139, 142
synthesis 43, 61–65
– in case study 182–186
system context model 53
system synthesis *See* synthesis
system testing
– *defined* 237
system testing phase 20, 64

T

TAI *See* time
task 30
– *defined* 237
– in object-oriented modeling 8
– of components 73
temporal behavior 95
tested increment 64
testing phase, system 20, 64
thread 30
– *defined* 237
– types 106–107
throwaway prototyping *See* prototyping
ticker 35, 116
– *defined* 238
time 98–105
– sidereal 104
– TAI 104, 143
– *defined* 233
– UT 104
– *defined* 238
– UTC 104, 143
– *defined* 231
time equation
– external 138
– internal 138
time slice 33, 36
timescale 103–105
topology 116–121
total ordering *See* ordering of events
total system error 13
– *defined* 238
transition 96, 97
– *notation for* 210
tree 117–118, 125
– *defined* 238
trigger 35, 97
tuning of clock synchronization 142
tunnel-in-the-sky display 160
– *See also* perspective flight path display

type 29, 205
– of application 48–50
– of signal 50, 86

U
UML *See* unified modeling language
unconfined thread 107, 116, 126
underdetermined system 135
unified modeling language 38–39, 205–216
unit testing 20, 89, 178
– *defined* 238
universal time *See* time
use case modeling 38, 47, 93
UT *See* time
UTC *See* time
utilization 108

V
validation 20, 65
– *defined* 238
validity time 115, 128
variable-priority scheduling 108–111
verification 20, 64
– *defined* 238
– of schedulability 106
virtual device 76, 80
virtual port 79
virtual signal 83–86
visibility
– of signals 119–121
visualization devices 54
– in case study 169

W
waterfall model 20–21
wiring 78, 82, 83
– in case study 176

Curriculum Vitae

Marco W. Soijer was born January 9, 1973 in Nijmegen, the Netherlands. He attended the Christelijk Lyceum Alphen aan den Rijn from 1985 and the Dominicus College Nijmegen from 1987, where he obtained the Gymnasium *b* certificate in 1991.

September 1991, Marco Soijer enrolled as a student with the Faculty of Aerospace Engineering at the Delft University of Technology. He obtained his private pilot licence in 1994 as a participant in the student pilot qualification program of the disciplinary group for stability and control. Under the supervision of professor Mulder, he completed a thesis on the subject of system identification for rotary wing aircraft. December 1996, he graduated cum laude with a Master of Science degree in Aerospace Engineering.

Marco Soijer then joined the same section – which had now been renamed to the control and simulation department – at the Faculty of Aerospace Engineering as a junior researcher. He focussed his activities on flight test programs with the Delft University laboratory aircraft, including both aeronautical applications and atmospheric and geophysical research. The development of a new instrumentation system for the laboratory aircraft was initiated in 1998; it forms the basis for this thesis.

Since July 2001, Marco Soijer has been with the flight test technology and data analysis department of EADS Military Aircraft in Manching, Germany, as a system specialist for handling qualities and aerodynamic model estimation in the Eurofighter program.