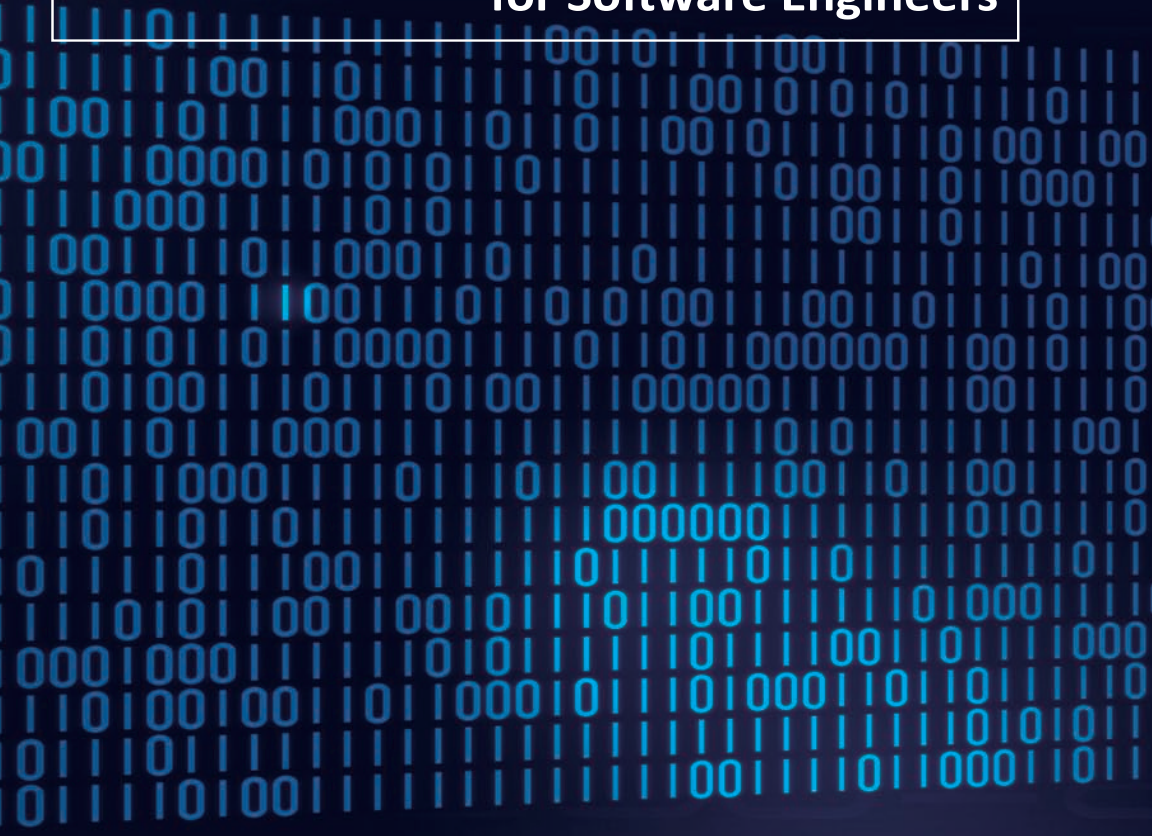
CHAPMAN & HALL/CRC INNOVATIONS IN
SOFTWARE ENGINEERING AND SOFTWARE DEVELOPMENT

Fundamentals of Dependable Computing

for Software Engineers



John Knight

With Foreword by Brian Randell

CRC Press
Taylor & Francis Group

A CHAPMAN & HALL BOOK

Fundamentals of Dependable Computing

for Software Engineers

Chapman & Hall/CRC Innovations in Software Engineering and Software Development

Series Editor
Richard LeBlanc
Chair, Department of Computer Science and Software Engineering, Seattle University

AIMS AND SCOPE

This series covers all aspects of software engineering and software development. Books in the series will be innovative reference books, research monographs, and textbooks at the undergraduate and graduate level. Coverage will include traditional subject matter, cutting-edge research, and current industry practice, such as agile software development methods and service-oriented architectures. We also welcome proposals for books that capture the latest results on the domains and conditions in which practices are most effective.

PUBLISHED TITLES

Software Development: An Open Source Approach
Allen Tucker, Ralph Morelli, and Chamindra de Silva

Building Enterprise Systems with ODP: An Introduction to Open Distributed Processing
Peter F. Linington, Zoran Milosevic, Akira Tanaka, and Antonio Vallecillo

Software Engineering: The Current Practice
Václav Rajlich

Fundamentals of Dependable Computing for Software Engineers
John Knight

CHAPMAN & HALL/CRC INNOVATIONS IN
SOFTWARE ENGINEERING AND SOFTWARE DEVELOPMENT

Fundamentals of Dependable Computing

for Software Engineers

John Knight

With Foreword by Brian Randell

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20120103

International Standard Book Number-13: 978-1-4398-6256-8 (eBook - PDF)

Table of Contents

Foreword

As computer systems have permeated ever more aspects of daily and communal life, so individuals', organizations' and society's dependency on the satisfactory functioning of these systems has become ever greater. This dependency can take many forms. It can arise, for example, from (i) the cost to a manufacturer of recalling a mass-market product of inadequate reliability, (ii) the dangers to life and limb from unsafe actions that are caused by, or fail to be prevented by, a computer system, or (iii) the reputational or financial consequence of failure to protect highly-confidential information.

Computer systems can be employed in many different situations and can fail in many different ways. A system will be judged to be dependable if its failures are neither too frequent, nor too severe. Quite what constitutes failure, and acceptable levels of failure frequency and failure severity, will vary according to the situation and circumstances. And different stakeholders, such as users, operators and system owners may judge these differently.

Just as computer systems can fail in many different ways, so there are many different possible causes of computer system failure, i.e. faults, of various different types. In particular there are hardware operational faults due to component ageing, residual software (and hardware) design faults, and deliberate (or perhaps just accidental) acts by users that trigger little-known vulnerabilities. Moreover, faults in one system may well be the result of failures in some other system, i.e. (i) another system that it is interacting with, (ii) a component sub-system, or (iii) a system that created or modified it. Issues of system boundaries and their careful identification are therefore crucial. Indeed, one cannot expect to achieve high system dependability in systems where the engineers involved do not have a detailed understanding of system boundaries and specifications.

Achieving, and — equally importantly — being able to justify claims of, adequate system dependability from ever more sophisticated computer-based systems is thus a continuing challenge. Unmastered complexity breeds confusion and confusion breeds undependability. Hence as John Knight, the author of this book, rightly emphasizes, the importance of clear concepts and carefully-defined terminology. The concepts and terminology that he describes, and uses carefully throughout this book, are appropriate for all types of system, e.g. (i) a "programmable" washing machine, (ii) a mobile phone incorporating a sophisticated operating system, (iii) a distributed database system supporting the work of a large software design team,

and (iv) a global banking system comprising large numbers of computers, networks and banking personnel.

Conceptually, the dependability issues arising in all these different types of system, and from the various types of fault, are in fact very similar — how and to what extent can one: (i) avoid introducing faults into a system, (ii) find and remove faults that nevertheless exist in the system, (iii) provide acceptable service despite any remaining faults, and (iv) estimate the effectiveness of these various measures. This similarity may however be masked by the use of differing terminology by the different technical communities involved. However terminological differences matter little if there is a common understanding of an adequate set of basic concepts, applying to all types of system, all types of system failure, and all the different possible causes of system failure.

One of the biggest fundamental causes of failure in computer-based systems is their complexity, much of which will for good technical reasons reside in the software. Thus many of the challenges facing those responsible for a computer system's dependability concern software, and hence software engineering. Hence a major strength of this book is the systematic way in which it identifies and discusses the many contributions that good software engineering can make to the task of achieving overall system dependability and the various challenges involved in ensuring that the software itself is adequately dependable.

This book takes full advantage of the extensive work that has been undertaken over many years on the creation of a rich set of system dependability concepts. John Knight makes excellent use of these concepts in producing a very well-argued and comprehensive account, aimed squarely at software engineers, of the variety of dependability issues they are likely to find in real systems and of the strategies that they should use to address these issues. Appropriately qualified students who study this book thoroughly, and computer professionals seeking a greater understanding of the various dependability-related problems that they have encountered already in their careers, should gain much from this book. I therefore take great pleasure in enthusiastically recommending it to both classes of reader.

Brian Randell
Newcastle University
31 July 2011

Preface

We depend on computer systems for much of what we do, and our dependence is greater than most of us realize. Without computers working correctly, our lives would be changed dramatically and mostly for the worse. The spectrum of services that computers help to provide is very diverse and includes banking, education, transportation, energy production, telecommunications, health care, defense, farming, manufacturing, business operations, and entertainment. A passenger railway system, for example, might be prevented from providing service because of a computer problem just as service might be prevented by a loss of power, damage to the track or rolling stock, seriously inclement weather, or lack of a crew.

This book is about computer system dependability, more specifically, those aspects of the subject that are important to the software engineer. The material is suitable for a senior undergraduate course or a first-year graduate course in computer science or computer engineering. The material is also suitable for self study, and the practicing engineer can tackle the subject matter directly.

The book has a single pedagogical goal:

> *To present software and computer engineers with a comprehensive dependability engineering process in which the reasons for and the interrelationships between the various parts are clear and justified.*

Many techniques are introduced, although not in great depth. The book does not cover any specific techniques in depth *except* the process. The intent is to summarize the important topics in sufficient depth that the reader can understand their form and role and have the background to pursue these topics if they wish. With the process clear, the reader will be able to study specific topics in depth and determine the extent to which they apply to his or her engineering activities.

To meet the book's goal, four major items of material are covered:

- Sufficient information about the systems engineering aspects of dependability that the software engineer can (a) understand fully why the software is being asked to do what it is being asked to do and (b) understand why the software is being made to operate on the particular platform specified by the system designers.
- A definitional and conceptual framework within which the engineer can reason and make decisions about software and that software's dependability.

- A comprehensive approach to achieving software dependability.
- A bibliography of the most relevant literature.

Why read this book?

Why should a student or a practicing engineer study this material? There is plenty of technology to study, so how does this fit in? The answers to these questions can be found in the title. The dependability of computer systems is as important as the functionality of the systems, perhaps even more important. Less functionality than expected or planned is often tolerable. But a system that fails at an unacceptable rate is usually intolerable. Unless we take action to prevent them, failures will occur, and, if they occur at a rate that is unacceptable, the associated systems might have to be taken out of service. Practicing engineers and students of engineering (engineers in training) need to master the essential elements of computer system dependability so as to help them to make appropriate engineering decisions. Arguably, studying this material is essential.

The things that you will learn from this book are concerned with the actions needed to reduce the probability of failures to an acceptable level, if that is possible. Specifically, you will learn:

- Why dependability matters.
- What it means for a system to be dependable.
- How to build a dependable system.
- How to assess whether a system is adequately dependable.

In order to be able to master this material, the following background is assumed:

- Experience with high-level language programming in a language like C, C++, C#, or Java.
- A basic knowledge of computer organization including the operation of processors, memories, disk storage systems, and basic communication facilities.
- Exposure to elementary probability theory.
- A working knowledge of discrete mathematics, including propositional calculus, predicate calculus, basic functions, and set theory.
- An understanding of the basic principles of operating systems, including processor management, memory management, peripheral management, and the operation of user interfaces.
- A good knowledge of the principles of software engineering, including requirements analysis, specification, design, testing, documentation and software development processes.

The emphasis in this book is on the software engineering elements of the problem. Thus, the intent is to help software engineers meet the challenge of construct-

ing software systems that are sufficiently dependable, and within budget and time constraints.

How to use this book

Computing system dependability should be part of any degree program in computer science or computer engineering. A course on the fundamental elements of the topic should be the minimum, with more elaborate optional courses in topics such as formal methods, mathematical modeling, dependable computing architecture, and advanced software engineering building on the fundamentals as determined by the goals of the degree program.

This book can form the basis of a one-semester introductory course, and the bulk of the material can be covered in one semester. The material should be presented in the same order as in this book, because later material depends critically on earlier material. The overall development that students should see and with which they should become familiar is:

- What dependability is and why dependability is important.
- The substantial but essential conceptual and definitional structure of the subject.
- The computing platforms upon which critical applications operate and how these platforms affect software.
- The difficulties that arise in software engineering that lead to software failures.
- The mathematically based techniques that can improve the quality of software dramatically and that are becoming available even for large software systems.

For a practicing engineer, the book can be treated as a reference. Topics can be studied in any order provided that for each topic of interest the reader is familiar with the preceding material. My experience in discussing dependability with engineers in industry is that they tend to lack adequate backgrounds in much of the material covered. Starting at the beginning to make sure of things like the definitional structure of the field is generally worthwhile.

The organization of this book

This book is organized into 12 chapters, and the reader is encouraged to work through the chapters in order. Each chapter develops the material from the previous chapters and relies upon their content. Chapter 1 introduces the topic and motivates the study of dependability. The terminology of dependability is presented in Chapter 2, and Chapter 3 introduces the different fault types and the approaches to dealing with faults.

Chapter 4 discusses how to identify the faults to which a system is subject, and Chapter 5 examines the four basic mechanisms for dealing with faults, *avoidance*, *removal*, *tolerance*, and *forecasting*, along with a discussion of dealing with Byzan-

tine faults. Chapter 6 summarizes the issues with degradation faults, a type of fault that arises only in hardware, and Chapter 7 surveys the general issues surrounding software dependability. Chapter 8 and Chapter 9 discuss important topics in software fault avoidance.

Chapter 10 is about software fault elimination, and Chapter 11 is about software fault tolerance. Finally, Chapter 12 examines dependability assessment.

Those to whom I am grateful

I am pleased to have this opportunity to thank many people who were influential in the creation of this book. Brian Randell from the University of Newcastle upon Tyne, who kindly wrote the Foreword, taught me an immense amount over many years in many ways about many things. Premkumar Devanbu of the University of California, Davis, started me down the path of assembling this material when he asked me to present a summary lecture on dependability at the 2001 International Conference on Software Engineering. And Dieter Rombach of the University of Kaiserlauten got me started on organizing this material when he asked me to help with a distance learning class on dependability.

The origins of the detailed material in this book are classes that I have taught at the University of Virginia. I am deeply indebted to the dozens of students who attended those classes, put up with my lecture style, asked me thought-provoking questions, and taught me an immense amount. Thanks to all of you; you know who you are.

I benefited greatly from reviews of the manuscript by M. Anthony Aiello, Tom Anderson, Josh Dehlinger, Michael Holloway, Rich LeBlanc, and Brian Randell. Their combined comments were truly transformative. I also benefited greatly from numerous discussions with Patrick Graydon.

None of this would have happened without the help of many people at the publisher, Taylor & Francis. In particular, I thank Alan Apt and Randi Cohen.

Finally, I owe an immense debt of gratitude to my family — my children, Richard, Abby, and Katie, and my wife Virginia — for the lost weekends and evenings that I have spent writing this book.

Further information

Slides based on the material in this textbook and solutions to exercises are available to instructors at institutes of higher education. Details and further information about this textbook are provided at the following address:

`http://www.dependablecomputing.com/fundamentals.html`

John Knight
Charlottesville, Virginia

For product information, please contact:
The MathWorks,
Inc. 3 Apple Hill Drive,
Natick, MA 01760-2098 USA.
Tel: 508 647 7000
Fax: 508-647-7001
E-mail: info@mathworks.com
Web: www.mathworks.com

This page intentionally left blank

CHAPTER 1

Introduction

The dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable.

Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr

1.1 The Elements of Dependability

1.1.1 A Cautionary Tale

Imagine that you are an expert in *making chains*, and that you are working on a project to suspend a box containing delicate porcelain high above ground. If the chain, the box, the hook that attaches the chain to the box, or the hook that attaches the chain to the ceiling breaks, the box will fall and hit the ground. Clearly, serious damage to the porcelain will probably result. Everybody involved with the porcelain (the owner of the porcelain, the expert in boxes, the expert in hooks, you, etc.) would like to prevent that.

You are a well-educated chain engineer with a degree from a prestigious academic institution. You have had classes in all sorts of chain-engineering techniques, and so you expect to proceed using the education you have. But, before work starts on the project, you become anxious about the following concern:

Concern 1: If chain failure could cause a lot of damage, everything possible has to be done to prevent failure. Porcelain is delicate and expensive. You wonder whether you know all the available techniques that could prevent damage.

FIGURE 1.1 Delicate porcelain in the museum at the Meissen porcelain factory in Meissen, Germany.

If the porcelain is mass produced, the owner might not care if the chain breaks. At least, in that case, the owner might not want to pay for you and other experts to engineer the chain, the box, and the hooks really well. If the box contains antique Meissen porcelain, such as the porcelain shown in Figure 1.1, the owner would want very high levels of assurance that the chain, the box, and the hooks will not break except under the most dire circumstances. Unfortunately, the owner tells you that the porcelain to be suspended is indeed a rare and expensive piece.

As you start to work on the project, you determine quickly that you need to know *exactly* how strong the chain has to be. Suppose the chain you built was not strong enough, what would happen? These thoughts raise a second concern:

Concern 2: Defining the necessary strength of the chain is crucial so that you have a target for the engineering you will undertake on the chain. You cannot make that determination yourself because the definition is really set by the systems engineer in charge of the suspended-box project.

You ask the porcelain's owner how strong the chain needs to be. The owner of the porcelain just says that the chain has to be "unbreakable". You ask the other engineers working with you. None of them understands the question well enough to be able to give you a complete answer. The engineers tell you:

"Use the International Standards Organization chain standard for boxes that hold expensive porcelain high above ground."

But that standard focuses on chain documentation, color, and the shape of the links. Would following the standard ensure that the chain you make will be strong enough?

In order to get this dilemma sorted out, you realize that you need to be able to communicate effectively with others so as to get an engineering answer to your questions. A third concern begins to bother you:

> **Concern 3**: An accurate and complete set of definitions of terms is necessary so that you can communicate reliably with other engineers. You do not have such a set of definitions.

With a set of terms, you will be able to communicate with other chain engineers, metallurgists, experts in box and hook structures, inspectors from the Federal Porcelain Administration (the FPA), physicists who can model potential and kinetic energy, porcelain engineers, and the owner of the porcelain.

As an expert in chains, you know that a chain is only as strong as its weakest link, and this obvious but important fact guides your work with the chain. To protect the porcelain, you search for defects in the chain, focusing your attention on a search for the weakest link. Suddenly, a fourth concern occurs to you:

> **Concern 4**: You need a comprehensive technology that will allow you to find *all* the links in the chain that are not strong enough, and you must have a mechanism for dealing with *all* of the weak links, *not* just the weakest link and *not* just the obvious weak links.

You wonder whether you could estimate how strong the chain is and find ways to strengthen it. You suspect that there are several techniques for dealing with links in the chain that might break, but you did not take the course "*Dealing With Weak Links in a Chain*" when you were in college.

Thinking about the problem of weak links, you decide that you might ensure that the chain is manufactured carefully, and you might examine and test the chain. If you miss a weak link, you think that you might install a second chain in parallel with the first just in case the first chain breaks, you might place a cushion under the box of porcelain to protect the porcelain if the chain breaks, or you might wrap the porcelain carefully to protect it from shock. Other experts would examine the box and the hooks.

Somewhat in a state of shock, a fifth and final concern occurs to you:

> **Concern 5**: You need to know just how effective all the techniques for dealing with weak links in the chain will be. Even if you deal with a weak link in a sensible way, you might not have eliminated the problem.

Because you are a well-educated chain engineer, you proceed using the education you have and you mostly ignore the concerns that have occurred to you. The

chain and all the other elements of the system are built, and the porcelain is suspended in the box. The owner of the porcelain is delighted.

And then there is an enormous crash as the porcelain breaks into thousands of pieces. The chain broke. One link in the chain was not strong enough.

You wake up and realize that all this was just a nightmare. You remember you are not a chain engineer. Phew! You are a *software* engineer. But you remain in a cold sweat, because you realize that the porcelain could have been the computer system you are working on and the "crash" might have been caused by *your* software — a really frightening thought.

1.1.2 Why Dependability?

From this story you should get some idea of what we can be faced with in engineering a computer system that has to be dependable. The reason we need dependability of anything, including computer systems, is because the *consequences of failure* are high. We need precise definitions of how dependable our computer systems have to be. We cannot make these systems perfect. We need definitions of terms so that all stakeholders can communicate properly.

In order to get dependability, we are faced with a wide range of engineering issues that must be addressed systematically and comprehensively. If we miss something — anything (a weak link in the chain) — the system might fail, and nobody will care how good the rest of the system was. We have to know just how good our systems have to be so that we can adopt appropriate engineering techniques. And so on.

This book is about *dependability* of computer systems, and the "cautionary tale" of Section 1.1.1 basically lays out this entire book. The way in which we achieve dependability is through a rigorous series of engineering steps with which many engineers are unfamiliar. Having a strong academic background does not necessarily qualify one for addressing the issue of dependability. This book brings together the fundamentals of dependability for software engineers, hence the name. By studying the fundamentals, the software engineer can make decisions about appropriate engineering for any specific system. Importantly, the software engineer can also determine when the available technology or the planned technology for a given system is not adequate to meet the dependability goals.

Examining the computer systems upon which we depend and engineering these systems to provide an acceptable level of service are important. Most people have experienced the frustration of their desktop or laptop computer "locking up", their hard drive failing with no recent backup, or their computer coming to a halt when the power fails. These examples are of familiar systems and incidents that we would like to prevent, because they cause us inconvenience, often considerable inconvenience. But there are many major systems where failures are much more than an inconvenience, and we look at some examples in Section 1.4.

As in the cautionary tale where the chain engineer was "a well-educated chain engineer with a degree from a prestigious academic institution", most software and computer engineers are well educated in the main issues that they face in their professions. Typically, however, the major issues of dependability are unfamiliar to computer and software engineers.

1.2 The Role of the Software Engineer

Why should a software engineer be concerned about dependability in the depth that the subject is covered in this book? Surely software engineers just write software? There are four important reasons why software is closely involved with everything to do with system dependability:

- **Software should perform the required function.**
  If software performs something other than the required function, then a system failure could ensue with possibly serious consequences. No matter how well implemented a software system is, that system has to meet the requirements. Determining the requirements for software is difficult and usually not within the realm of expertise of other engineering disciplines. The software engineer has to help the entire engineering team with this task.

- **Software should perform the required function *correctly*.**
  If software performs the required function incorrectly, then, again, a system failure could ensue with possibly serious consequences. The software engineer has to choose implementation techniques that will increase the chances that the software implementation will be correct.

- **The software might have to operate on a target platform whose design was influenced by the system's dependability goals.**
  Target platforms often include elements that are not necessary for basic functionality in order to meet dependability goals for the platform itself. Many systems use replicated processors, disks, communications facilities, and so on in order to be able to avoid certain types of fault. Software engineers need to be aware of why the target was designed the way it was. The target platform design is very likely to affect the software design, and software is usually involved in the operation of these replicated resources.

- **Software often needs to take action when a hardware component fails.**
  Software usually contributes significantly to the engineering that provides overall system dependability and to the management of the system following some types of failure. Many things can be done to help avoid or recover from system failures if sufficient care is taken in system design. In almost all cases, software is at the heart of these mechanisms, and so software requirements actually derive

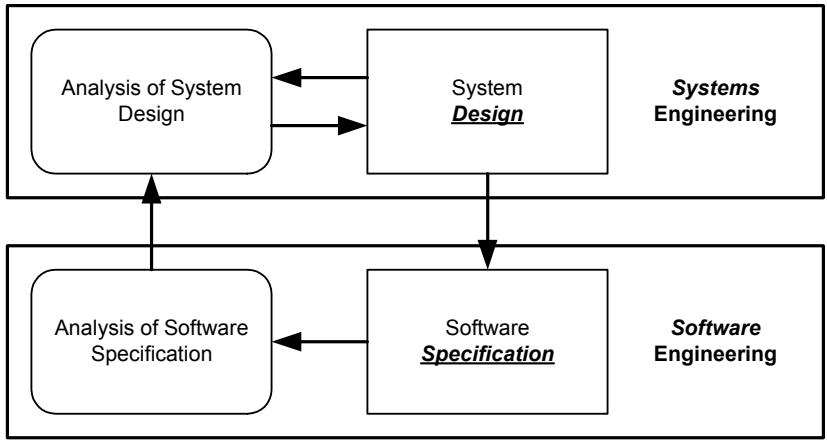


FIGURE 1.2 The interaction between systems and software engineering. System designers develop and analyze their design, and from that determine what the software has to do. When analyzing the software specification, software engineers might need to influence the system design and thereby the software specification because of practical concerns for the software raised by the specification.

from the need to meet certain dependability goals. A simple example is to monitor the input coming from system operators and to inhibit response to inappropriate or erroneous commands. This type of dependability enhancement imposes a requirement that the software has to meet.

These four reasons are a manifestation of the interaction between systems engineering and software engineering. The interaction is illustrated in Figure 1.2. Systems engineers are responsible for the system design, and they use a variety of analyses to model the dependability of their designs. These analyses include fault tree analysis (FTA), Failure Modes Effects and Criticality Analysis (FMECA), and Hazard and Operability Analysis (HazOp). Eventually a system design is created, and the software *specification* derives from that system *design*.

The interaction is not just in the direction from systems engineers to software engineers. A software system is a complex entity that has characteristics and limitations of its own. As a result, information from software engineers needs to be made available to and discussed with systems engineers if the entire system is to perform as desired. Changes to the design of a system might be needed to accommodate software limitations.

Clearly, software is an important component of the systems with which we are concerned. The software engineer has to be prepared to build software that often goes well beyond that needed for primary functionality. The software engineer is far more capable of creating software that meets its objectives if he or she understands something of the systems engineering context and the systems engineer's tools and

techniques. In some cases, those techniques can be adapted usefully and easily to support software engineering.

Two important factors in understanding the role of the software engineer in dependability are the following:

- The software in systems that need high levels of dependability has become involved in far more of the systems' functionality than was the case in the past.
- Software defects have become common causal factors in failures. We have to engineer software carefully if that software is to provide the level of service that modern systems require.

The second of these factors is related, at least in part, to the first. But the crucial point is that the software engineer now plays a much more central role in systems dependability than was the case in the past. As a result, the software engineer needs to understand why the system is asking the software to undertake functionality that is sometimes non-obvious, why the target platform is what it is, and how to achieve the high levels of software dependability that are required.

1.3 Our Dependence on Computers

Whether we like it or not, we depend on computer systems for much of what we do. Without them, our lives would be changed dramatically and mostly for the worse. The spectrum of services that they help to provide is very diverse and includes:

Banking and financial services. Computers keep track of bank accounts, transmit money between banks, even between banks in different countries, provide financial services such as ATMs and credit cards, operate markets and exchanges, and engage in financial transactions on behalf of their owners.

Transportation. Commercial aircraft employ extensive computer systems on-board, and the air-traffic-control system employs computers in a variety of ways. Cars depend on computers for engine management, anti-lock brakes, traction control, and in-car entertainment systems. Passenger rail systems rely on computers for a variety of services, including, in the most sophisticated system, actually operating the train.

Energy production. Computers help control energy production equipment such as electrical generators, help control gas and oil pipelines, manage electrical transmission lines, and control energy systems when demand changes because of weather.

Telecommunications. All of the telecommunications systems that we use, from traditional wired telephone service in our homes to mobile service using smart telephones and iPads, are really just elaborate distributed computing systems.

Health care. Many health care services depend upon computers, including the various forms of imagery, and devices such as drug-infusion pumps, surgical robots, and radiation therapy machines. Information systems play a central role in health care also in areas such as patient records, drug inventories, and facility planning.

Defense. Defense employs extensive computer systems in command and control, in weapons management, within complex systems such as aircraft, ships, and missiles, and in supply-chain management.

Manufacturing. Manufacturing operations rely on computers for robotic production lines, supply-chain management, resource and personnel management, scheduling, inventory management, financial and accounting activities, and parts scheduling.

Business operations. Businesses rely on computers for inventory management, accounting, advertising, customer services, billing, and point-of-sale services.

Entertainment. By definition, computer games use computers. But multi-player games communicate via computer networks, and movies are beginning to be created and presented digitally.

In some cases we receive service without realizing that computers are at the heart of providing that service. Most people are unaware, for example, of how many computers are embedded in appliances in their homes and in their cars. There are computers in thermostats, kitchen appliances, televisions, and so on. These computers do not look like the ones we see on our desks at work, but they are powerful computers nonetheless. These computers are called *embedded* because they are integrated within other equipment. Every such computer has a processor, main memory, and I/O devices. Some are connected to a network, and some have peripherals such as specialized displays and keyboards that allow them to communicate with humans.

Just as with home appliances and cars, most people are unaware of how many computers and networks are involved in authorizing a credit card purchase or processing a check for payment. Even an action as simple as using a credit card at a service station to purchase gasoline requires the actions of dozens of computers and several private networks. The gasoline pump contains a computer that is connected to a private data network which communicates with the bank that issued the credit card. That bank's computers then check for the possibility of fraudulent use and check the card holder's account. Finally, if the transaction does not appear fraudulent and the card holder has the necessary credit, the transaction is authorized and a message to that effect is sent back to the gasoline pump. And all of this is done in a few seconds.

The credit card and check processing systems are examples of national networked infrastructure systems. Another important national infrastructure in the United States. is the freight-rail system, a technology much older than cars and much more complex. Normally all we see are locomotives and freight cars, and we

usually notice them only when we are stopped at a railway crossing. But the movement of freight by rail with modern levels of efficiency and performance would not be possible without a wide range of computers helping to track payloads, operate locomotives, schedule train operation, and optimize overall system performance. Much of this activity relies upon track-side equipment that regularly reports the location of individual freight cars to centralized databases. Importantly, computers help to watch over the system to ensure safe operation.

A careful examination reveals that the computer systems underlying the provision of modern services span a wide spectrum of system characteristics. Some services depend on wide-area networks, some on local-area networks, some on embedded computers, some on large databases, and some on complex combinations of many characteristics.

1.4 Some Regrettable Failures

Before proceeding, it is worth looking at some examples of significant failures that have occurred and in which a computer was one of the (usually many) factors involved in causing the failure. In general, examining failures so as to learn from them and to help prevent similar failures in the future is important. Investigating accidents is a difficult challenge and the subject of study in its own right [73].

1.4.1 The Ariane V

The Ariane V is one of a range of launch vehicles developed by the European Space Agency (ESA). The maiden flight of the Ariane V, known as flight 501, took place on June 4, 1996 at the ESA launch site in Kourou, French Guiana. The vehicle rose from the launch pad and, about 40 seconds later, veered off course, broke apart, and exploded. The reason the vehicle veered off course was that the engines were gimbled to extreme positions, and this caused the vehicle to pitch over. This unplanned pitch over led to excessive aerodynamic loads that caused the self-destruct mechanism to operate as it was designed to do.

An investigation was started immediately by an inquiry board assembled by ESA and composed of international experts. The report of the board was published on July 19, 1996. The report describes many background details of the launch site and the launch vehicle, the steps in the inquiry, and detailed conclusions [89].

Many factors were involved in the accident, but one of the most important factors involved the software in part of the vehicle's Flight Control System called the Inertial Reference System (IRS). The IRS supplies velocities and angles from which calculations are undertaken that set the various control surfaces. The goal is to keep the vehicle on the planned trajectory.

Just prior to the end of the short flight, a software component that is used for alignment while the vehicle is on the ground prior to launch was still executing. This

was not necessary but not viewed as a problem. That particular module was written originally for the Ariane IV, and the module was used on the Ariane V because the required functionality was similar. The module calculated a value related to the horizontal velocity component, but the value to be calculated was higher than the values that occur on the Ariane IV. Unfortunately, the higher value was not representable in the available precision, and that resulted in an exception being raised.

The software was written in Ada, and neither an explicit guard on the value nor a local exception handler was provided to deal with this situation. The exception was propagated according to the Ada exception-handling semantics, and this caused execution of significant amounts of the software to be terminated. The deflection of the engines that caused the vehicle to pitch over just prior to the abrupt end of the maiden flight occurred because test bit patterns were sent to the engine control actuators rather than correct values.

1.4.2 Korean Air Flight 801

On August 6, 1997 at about 1:42 am Guam local time, Korean Air flight 801, a Boeing 747-300, crashed into Nimitz Hill, Guam while attempting a non-precision approach to runway 6L at A.B. Won Guam International Airport. Of the 254 persons on board, 237 of whom were passengers, only 23 passengers and 3 flight attendants survived. The National Transportation Safety Board (NTSB) investigated the accident and classified the crash as a controlled-flight-into-terrain accident [101].

Korean Air flight 801 crashed during its final approach while operating under instrument flight rules (IFR). At the time of the accident, the runway glideslope was out of service, meaning that pilots were not to rely on the glideslope signal when landing at Guam. When the glideslope is unavailable, a non-precision or localizer-only instrument-landing-system (ILS) approach is still possible. In lieu of a glideslope, pilots make a series of intermediate descents using a series of step-down altitude fixes.

Post-accident analysis of radar data indicated that flight 801 began a premature descent on its non-precision approach and violated the 2,000 feet step-down clearance, i.e., the clearance to descend below 2,000 feet. The aircraft proceeded on a steady descent, violating the 1,440 feet step-down clearance before impacting terrain approximately 3.3 nautical miles short of the runway threshold. The NTSB concluded that “the captain lost awareness of flight 801’s position on the [ILS] localizer-only approach to runway 6L at Guam International Airport and improperly descended below the intermediate approach altitudes ... which was causal to the accident.”

During its investigation, the NTSB found that the ground-based Minimum Safe Altitude Warning System (MSAW) had been inhibited. MSAW is a software system that monitors radar data, and, using a map of the local terrain, alerts air-traffic controllers to aircraft that might be flying too low. MSAW is just one of many defenses against CFIT accidents, but it is an important defense. In Guam at the time of this

accident, controllers had been disturbed by false alarms from MSAW, and so MSAW coverage had been disabled intentionally in a circle centered on the airport and with a radius of 54 nautical miles. MSAW's coverage was a circle of radius 55 nautical miles, and so, at the time of the Korean Air 801 crash, MSAW's actual coverage was a ring of width one nautical mile that was entirely over the Pacific ocean.

1.4.3 The Mars Climate Orbiter

The Mars Climate Orbiter (MCO) was a spacecraft designed to orbit Mars and conduct studies of the Martian weather. The spacecraft had a secondary role as a communications relay for the Mars Polar Lander (see Section 1.4.4).

According to the report of the mishap investigation board [94], the MCO was lost on September 23, 1999 as the spacecraft engaged in Mars Orbit Insertion. At that time, the spacecraft was 170 kilometers lower than planned because of minor errors that accumulated during the cruise phase. The spacecraft was lost because of unplanned interaction with the Martian atmosphere.

The source of the minor errors was a series of trajectory corrections in which the calculation of part of the trajectory model was not in metric units. Thus, part of the software was written assuming that the data represented measurements in metric units and part was written assuming that the data represented measurements in Imperial units. Both software parts worked correctly, but the misunderstanding of the data led to errors in calculated values. The cumulative effect led to the difference between the actual and planned altitudes and the subsequent loss of the spacecraft.

1.4.4 The Mars Polar Lander

The Mars Polar Lander (MPL) was a spacecraft designed to land on Mars and which was planned to arrive at Mars several months after the Mars Climate Orbiter. The MPL mission goal was to study the geology and weather at the landing site in the south polar region of Mars.

The spacecraft arrived at Mars on December 3, 1999. Atmospheric entry, descent, and landing were to take place without telemetry to Earth, and so the first communication expected from the spacecraft would occur after landing. That communication never arrived, and nothing has been heard from the spacecraft subsequently.

According to the report of the Special Review Board [118], the exact cause of the failure could not be determined. The most probable cause that was identified was premature shutdown of the descent engine. Magnetic sensors on the landing legs were designed to detect surface contact. However, these sensors also generated a signal when the legs were deployed from their stowed position prior to the final descent to the surface. The spacecraft's software should have ignored this signal, but apparently the software interpreted the signal as surface contact and shut down the descent engine. Since leg deployment occurred at an altitude of 40 meters, the lander would have fallen to the surface and not survived.

1.4.5 Other Important Incidents

A very unfortunate example of how a seemingly benign system can be far more dependent on its computer systems than is obvious occurred in October 1992 with the dispatching system used by the ambulance service in London, England. A manual dispatch system was replaced with a computerized system that was not able to meet the operational needs of the dispatch staff in various ways. Delays in dispatching ambulances to emergencies might have been responsible for several deaths before the computerized system was shut down, although this has not been proven [47].

The Therac 25 was a medical radiation therapy machine manufactured by Atomic Energy of Canada Limited (AECL). The device was installed in numerous hospitals in the United States. Between June 1985 and January 1987, six patients received massive overdoses of radiation while being treated for various medical conditions using the Therac 25. Software requirements and implementation were causal factors in the accidents. A comprehensive analysis of the failures of the Therac 25 has been reported by Leveson and Turner [88].

Finally, an incident that was potentially serious but, fortunately, did not lead to disaster was the launch failure of the Space Shuttle during the first launch attempt in 1981 [49]. The launch was due to take place on Friday, April 10. The Shuttle has two flight-control systems, the primary system and the backup system, each with its own software. During the countdown, these two systems are required to synchronize. Synchronization includes ensuring that they agree on the total number of 40-millisecond, real-time frames that have passed since the initialization of the primary system at the beginning of the countdown. During the first launch attempt, they failed to synchronize because the primary system had counted one more frame than the backup system. At the time, the problem appeared to be in the backup system, because the problem came to light when the backup system was initialized shortly before launch. Efforts to correct the situation focused on the backup system, but the fault was actually in the primary system. The Friday launch had to be canceled, and the launch was finally completed successfully on the afternoon of Sunday, April 12.

1.4.6 How to Think about Failures

In reading about failures (these and others), do not jump to conclusions about either the causes or the best way to prevent future recurrences. There is never a single "cause" and there are always many lessons to be learned. Also, keep in mind that failures occur despite careful engineering by many dedicated engineers. What failures illustrate is the extreme difficulty we face in building dependable systems and how serious the effects can be when something goes wrong.

Finally, when our engineering techniques are insufficient, the system of interest will experience a failure, and sometimes an accident will ensue. As part of our engineering process, we need to learn from failures so that we can prevent their recur-

rence to the extent possible. When we read about a system failure in which a computer system was one of the causal factors, it is tempting to think that the lesson to be learned is to eliminate the identified defect. This is a serious mistake, because it misses the fundamental question of why the defect occurred in the first place.

As an example, consider the loss of the Mars Climate Orbiter spacecraft [94]. The lessons learned certainly include being careful with measurement units. This was identified as a causal factor, and so obviously care has to be taken with units on future missions. But the reason the units were wrong is a more fundamental process problem. What documents were missing or prepared incorrectly? What process steps should have been taken to check for consistency of both units and other system parameters? And, finally, what management errors occurred that allowed this situation to arise?

1.5 Consequences of Failure

The term that is usually used to describe all of the damages that follow failure is the *consequences of failure*, and that is the phrase that we will use from now on. The consequences of failure for the systems that we build are important to us as engineers, because our goal is to reduce losses as much as possible. We have to choose the right engineering approaches to use for any given system, and the choices we make are heavily influenced by the potential consequences of failure.

In this section, we examine the different consequences of failure that can occur. In particular, we examine the non-obvious consequences of failure and the consequences of failure for systems that seem to have essentially none. We will discuss exactly what we mean by a "system" later. At this point, remember the intuitive notions of a system, of a system of systems, and of a system connected to and influencing other systems. Such notions are needed to ensure that we consider all the possible consequences of failure.

1.5.1 Non-Obvious Consequences of Failure

For the examples in the previous section, the damage done was fairly clear. In each case, the losses were significant and came from various sources. A little reflection, however, reveals that the losses might go well beyond the obvious. In the Ariane V case, for example, the obvious loss was sophisticated and expensive equipment: the launch vehicle and payload. There was a subsequent loss of service from the payload that was not delivered to orbit.

What is not obvious are the losses incurred from:

- The environmental damage from the explosion that had to be cleaned up.
- The cost of the investigation of the incident and the subsequent redesign of the system to avoid the problem.

- The delays in subsequent launches.
- The increased insurance rates for similar launch vehicles.
- The loss of jobs at the various companies and organizations involved in the development of the launch vehicle.
- The damage to the careers of the scientists expecting data from the scientific instruments within the payloads.
- The loss of reputation by the European Space Agency.

In practice, there were probably other losses, perhaps many.

1.5.2 Unexpected Costs of Failure

For some systems, the cost of failure seems insignificant. In practice, these *trivial* systems can have significant consequences of failure. In other systems, the consequences of failure seem limited to minor inconvenience. But for these *non-critical* systems, the consequences of failure are not with the systems themselves but with the systems to which they are related. Finally, for *security* applications, the consequences of failure are hard to determine but are usually vastly higher than might be expected at first glance.

We examine each of these system types in turn. As we do so, be sure to note that, for any particular system, the consequences of failure will most likely be a combination of all of the different ideas we are discussing. For example, a text-message system operating on a smart phone might seem trivial, but, if the system is used to alert a population to a weather emergency, failure of the system would leave the population unprotected. Security is also an issue because of the potential for abuse either through malicious alerts or denial-of-service attacks.

Trivial applications. Sometimes, the consequences of failure seem to be minimal but, in fact, they are not. Consider, for example, a computer game that proves to be defective. Such a failure does not seem important until one considers the fact that millions of copies might be sold. If the game is defective, and the defect affects a large fraction of the users, the cost to patch or replace the game could be huge. The loss of reputation for the manufacturer of the game might be important too.

If the game is an on-line, multi-user game, then the potential cost of failure is higher. Many such games operate world wide, and so large numbers of servers provide the shared service to large numbers of users and an extensive communications network connects everybody together.

Access to the game requires a monthly subscription, and the revenue generated thereby is substantial. Supporting such a community of users requires: (1) that the client software work correctly, (2) that the server software work correctly, and (3) that the communications mechanism work correctly. Downtime, slow

response, or defective functionality could lead to a large reduction in revenue for the owners of the game.

Non-critical applications. Some computer systems do not provide services that are immediately critical yet their consequences of failure can be serious because of their impact on systems that do provide critical services. This indirection is easily missed when considering the consequences of failure, yet the indirection is usually a multiplier of the consequences of failure. This multiplier effect occurs because one non-critical application might be used to support multiple critical applications.

As an example, consider a compiler. Compiler writers are concerned about the accuracy of the compiler's translation, but their attention tends to be focused more on the performance of the generated code. A defect in a compiler used for a safety-critical application could lead to a failure of the safety-critical application even though the application software at the source-code level was defect free. And, clearly, this scenario could be repeated on an arbitrary number of different safety-critical applications.

Security applications. In July and August of 2001, the Code Red worm spread throughout large sections of the Internet [143]. Hundreds of thousands of hosts were infected in just a few days. The worm did not carry a malicious payload so infected computers were not damaged, although the worm effected a local denial of service attack on the infected machines. The infection had to be removed, and this worldwide cleanup activity alone is estimated to have cost at least $2,000,000,000.

Many modern information systems process critical data, and unauthorized access to that data can have costs that are essentially unbounded. Far worse than the Code Red worm are attacks in which valuable private information is taken and exploited in some way. Between July 2005 and mid-January 2007, approximately 45.6 million credit card and other personal records were stolen from the TJX Companies [30]. The degree to which this data has been exploited is unknown, but the potential loss is obviously large.

Both the Code Red worm and the attack on the TJX Companies were the result of defects in the software systems operating on the computers involved. Neither was the result of a failure of security technology per se.

1.5.3 Categories of Consequences

Although we all have a general understanding that failure of computer systems might lead to significant losses, the damage that a failure might cause needs to be understood in order to determine how much effort needs to be put into engineering the system.

The various categories of the consequences of failure include:

- Human injury or loss of life.

- Damage to the environment.
- Damage to or loss of equipment.
- Damage to or loss of data.
- Financial loss by theft.
- Financial loss through production of useless or defective products.
- Financial loss through reduced capacity for production or service.
- Loss of business reputation.
- Loss of customer base.
- Loss of jobs.

All of the items in this list are important, but they become less familiar and therefore less obvious as one proceeds down the list. This change is actually an important point. One of the activities in which we must engage is to determine as many of the consequences of failure as possible for the systems that we build. Things like "loss of reputation" are a serious concern yet rarely come to mind when considering the development of a computer system.

1.5.4 Determining the Consequences of Failure

As we saw Section 1.5.2, just because a computer is providing entertainment does not mean that dependability can be ignored. The complexity of determining the cost of failure for any given system means that this determination must be carried out carefully as part of a systematic dependability process. Adding up the costs of failure and seeing the high price allows us to consider expending resources during development in order to engineer systems that have the requisite low probability of failure.

There is no established process for determining the consequences of failure. In general, a systematic approach begins with a listing of categories such as the one in Section 1.5.3. Typically, one then proceeds informally but rigorously to examine the proposed system's effects in each category and documents these effects.

There is no need to compute a single cost figure for a system, and, in fact, assessing cost in monetary terms for consequences such as human injury is problematic at best. Thus, a complete statement about the consequences of failure for a system will usually be broken down into a set of different and incompatible measures including (a) the prospect of loss of life or injury, (b) the forms and extents of possible environmental damages, (c) the prospect of loss of information that has value, (d) services that might be lost, (e) various delays that might occur in associated activities, and (f) financial losses. This list (or an extended or localized version) can be used as a checklist for driving an assessment of consequences of failure.

1.6 The Need for Dependability

We need computer systems to be dependable because we depend upon them! But what do we really need? There are several different properties that might be important in different circumstances. We introduce some of them here and discuss them in depth in Chapter 2.

An obvious requirement for many systems is *safety*. In essence, we require that the systems we build will not cause harm. We want to be sure, for example, that medical devices upon which we depend operate safely and do not cause us injury. Injury, however, can occur because of action or because of *lack* of action, and so making such devices safe is far from simple.

Many systems cannot cause harm and so safety is not a requirement for them. A computer game console, for example, has little need for safety except for the most basic things such as protecting the user from high voltages, sharp edges, and toxic materials. The computer system itself cannot do very much to cause harm, and so, as developers of computer systems, we do not have to be concerned with safety in game consoles.

Another obvious requirement that arises frequently is for *security*. Information has value and so ensuring that information is used properly is important. The customers of a bank, for example, expect their financial records to be treated as private information. Trusted banking officials may view the information as can the owners of the information. But that information is held in a database that is shared by all of the bank's customers, and so security involves ensuring that each customer can access all of his or her own information *and no more*.

A less obvious dependability requirement is *availability*. Some systems provide service that we do not use especially frequently but which we expect to be "always there" when we want to use the service. The telephone system is an example. When we pick up a telephone handset, we expect to hear a dial tone (switched on by a computer). In practice, occasional brief outages would not be a serious problem, because they are unlikely to coincide with our occasional use. So, for many systems, continuous, uninterrupted service is not especially important as long as outages are brief and infrequent.

In examining these requirements, what we see is a collection of very different characteristics that we might want computer systems to have, all of which are related in some way to our intuitive notion of dependability. Intuitively, we tend to think of these characteristics as being very similar. As part of an informal conversation, one might state: "computer systems need to be reliable" or "computer systems need to work properly" but such informality does not help us in engineering. For now, our intuitive notions of these concepts will suffice, but in Chapter 2, we will examine these various terms carefully.

1.7 Systems and Their Dependability Requirements

Many computer applications have obvious consequences of failure. However, as we saw in Section 1.5.2, other applications do not seem to have any significant consequences of failure, although it turns out that they do. In yet others, the detailed consequences of failure are hard to determine, but all the consequences have to be determined if we are to be able to engineer for their reduction.

In this section we look at several systems to see what dependability they require. We also look at some examples of systems from the perspective of their interaction and the consequent impact that this interaction has on the consequences of failure. The primary illustrative example comes from commercial aircraft, and we examine an aircraft system, the production of aircraft systems, and the management of controlled airspace.

1.7.1 Critical Systems

Avionics

The avionics (*avi*ation electr*onics*) in commercial aircraft are complex computer systems that provide considerable support for the crew, help to maintain safe flight, and improve the overall comfort of passengers. This is not a small task, and several distinct computers and a lot of software are involved. Aircraft engines have their own computer systems that provide extensive monitoring, control, and safety services.

In military aircraft, avionics make things happen that are otherwise impossible. Many military aircraft literally cannot be flown without computer support, because actions need to be taken at sub-second rates, rates that humans cannot achieve. Such aircraft have a property known as *relaxed static stability,* meaning that their stability in flight is intentionally reduced to provide other benefits. Stability is restored by rapid manipulation of the aircraft's control surfaces by a computer system.

Commercial and military avionics have different dependability requirements. The overwhelming goal in passenger aircraft is safe and economical flight. Military aircraft, on the other hand, frequently have to be designed with compromises so as to improve their effectiveness in areas such as speed and maneuverability. The damages caused by failure are very different between commercial and military aircraft also. If there is an accident, loss of the aircraft is highly likely in both cases, but death and injury are much more likely in passenger aircraft. Military pilots have ejection seats, but passengers on commercial aircraft do not.

Thus, the consequences of failure of avionics systems are considerable and reasonably obvious, although they differ between aircraft types. When engineering a computer system that is to become part of an avionics suite, we know we need to be

careful. But the engineering approaches used in the two domains will differ considerably because of the differences in the operating environments and the consequences of failure.

Ships

Ships share many similarities with aircraft and employ sophisticated computer systems for many of the same reasons. On-board ships, computers are responsible for navigation, guidance, communication, engine control, load stability analysis, and sophisticated displays such as weather maps. In modern ships, all of these services are tied together with one or more shipboard networks that also support various more recognizable services such as e-mail.

An important difference between ships and aircraft is that stopping a ship is usually acceptable if something fails in the ship's computer system. Provided a ship is not in danger of capsizing or coming into contact with rocks, the ship can stop and failed computer systems can be repaired. Stopping an aircraft is not even possible unless the aircraft is actually on the ground, and engineering of computer systems must take this into account.

An important example of the role of computers on ships and the consequences of failure occurred on September 21, 1997 [153]. The U.S.S. Yorktown, a Navy cruiser, was disabled for more than two hours off the coast of Virginia when a critical system failed after an operator entered a zero into a data field. This zero led to a divide-by-zero exception that disabled a lot of the software, including critical control functions.

Spacecraft

Spacecraft are complex combinations of computers, scientific instruments, communications equipment, power generation devices and associated management, and propulsion systems. Those that leave the vicinity of the Earth and Moon present two intriguing dependability challenges. The first is longevity. No matter where the spacecraft is going, getting there will take a long time, usually several years. That means that the on-board computing systems must be able to operate unattended and with no hardware maintenance for that period of time. The second challenge is the communications time. Many spacecraft operate at distances from the Earth that preclude interactive operation with an Earth-bound operator. Thus, in many ways, spacecraft have to be autonomous. In particular, they must be able to protect themselves and make decisions while doing so with no external intervention.

Medical Devices

In the medical arena, devices such as pacemakers and defibrillators, drug infusion pumps, radiotherapy machines, ventilators, and monitoring equipment are all built around computer systems. Pacemakers and defibrillators (they are often combined

into a single device) are best thought of as being sophisticated computers with a few additional components. Pacemaking involves sampling patient parameters for every heartbeat and making a decision about whether the heart needs to be stimulated.

Pacemakers are an example of the type of system where our intuition suggests, correctly, that safety is the most important aspect of dependability with which we need to be concerned. But safety can be affected by both action and inaction. A pacemaker that stimulates the heart when it should not is very likely going to cause harm. But if the device were to detect some sort of defect in its own operation, merely stopping would not be a safe thing to do. Obviously, the patient needs the pacemaker and so stopping might lead to patient injury or death. Engineering such systems so that they operate safely is a significant challenge, and that engineering has to be done so as to maximize battery life.

Critical Infrastructures

The freight-rail system mentioned earlier is one of many critical infrastructure systems upon which we all depend and which themselves depend heavily on complex computer systems. Many other critical infrastructure systems are in the list of services included at the beginning of this chapter. The computer systems therein do not seem at first sight to have the same potential for serious damage as a result of failure as do systems such as passenger aircraft, medical devices, or spacecraft.

In practice, this is quite wrong. The banking system has to protect our money and make it available as needed. However, were the computing systems within the banking system to fail in some general way, the result would be much more than a minor inconvenience to citizens; an international economic crisis would result. Similarly, failure of the computers within the energy production, transport, or telecommunications industries would have a major impact as service became unavailable.

1.7.2 Systems That Help Build Systems

In areas such as aviation, one tends to think only of the product in operation, an aircraft in flight. But there are two other major and non-obvious areas of computer system engineering for which dependability is important, product design and product manufacturing.

Product design includes many forms of computerized analysis, and if that analysis is not completed correctly, the dependability of the resulting product, safety of an aircraft for example, might be jeopardized. Thus, the development of computer systems that will be involved in the *design* of other artifacts must consider the needs of the target systems as well as the system being developed. A computer system that performs structural analysis on the fuselage and wings of an aircraft is just as important as the avionics system that will fly the aircraft.

During manufacturing of virtually all products, many computers are involved in robotic assembly operations, managing inventories, routing of parts, examining

parts for defects, and managing the myriad of data associated with production. Again, using commercial aircraft as an example, if something such as a robotic welder or the subsequent weld examination system were defective, the safety of the resulting aircraft might be jeopardized.

Often neglected because software takes the spotlight, *data* is a critical item in many systems. For example, in terms of managing data during manufacturing, it is interesting to note that a modern passenger aircraft has more than 100 miles of wire in hundreds of separate wiring harnesses, each of which has to be installed correctly. Labeling alone is a major data management problem. Any mistakes in the data that is used to manufacture, label, locate, or test wiring in an aircraft could obviously lead to serious production delays or failures during operation.

Thus, again we see that computers which do not have obvious high consequences of failure can often be involved with manufacturing such that their failure can lead to defective products. This is far more common than most people realize, and the surprising result is that the consequences of failure of many manufacturing systems are high, although initially this observation is counterintuitive.

1.7.3 Systems That Interact with Other Systems

Air-traffic control is another area in which computer systems play an important role in aviation. These computers are very different from avionics systems yet no less important. Air-traffic-control systems employ radars and other sources of information to present controllers with an accurate picture of the status of the airspace. With this information, controllers are able to make decisions about appropriate aircraft movements that they then communicate to the crews of the aircraft.

The dependability requirements of air-traffic control are primarily in four areas: (1) the data provided by the system to controllers must be accurate; (2) the data must be available essentially all the time; (3) computations that drive displays and other information sources must be accurate; and (4) there must be comprehensive security.

Interactions between systems that are in fact critical do not have to be as complex as something like the air-traffic-control system. For example, pacemakers do not operate in isolation. They have numerous adjustable parameters that physicians set to optimize operation for each patient. This adjustment is carried out by a device called a “programmer” that is itself just a computer system. Pacemakers also capture patient information during operation, and that data can be downloaded for examination by physicians. Both parameter setting and patient data analysis are critical activities, and the associated computer system, the programmer, has to be understood to have significant consequences of failure and correspondingly high dependability requirements.

The important conclusions to draw from these various examples are: (1) that the need for dependability does not lie solely in glamorous, high visibility systems, and (2) the specific dependability requirements that systems present vary widely. The need for adequate dependability is present in practically any system that we build.

1.8 Where Do We Go from Here?

For almost all applications of interest, dependability is not something that can be added to an existing design. Certainly the dependability of a system can probably be improved by making suitable changes to the design, but being able to transform a system developed with just functionality in mind into one that meets significant dependability goals is highly unlikely. Without taking specific steps during the entire development process, systems end up with dependability characteristics that are ad hoc at best.

This limitation appears to present us with a significant dilemma. Does this limitation apply to everything, including the components that we use to build systems? In other words, are we at an impasse? If we can only build dependable systems from dependable components and only build dependable components from smaller dependable components, and so on, then we are indeed facing a serious challenge.

Fortunately, the answer to the question is "no". Building dependable systems relies in part upon our discovering how to meet *system* dependability goals using much less dependable *components* [18, 19, 37]. This concept goes back to the earliest days of computing [6, 144]. The pioneers of computing were able to build computers that could operate for useful periods of time using vacuum tubes (the Colossi, for example [113]), components that were notoriously unreliable.

The path that we will follow is a systematic and thorough treatment of the problem of dependability. As we follow that path, keep the following in mind:

> *The attention paid during system development has to be thorough and orderly. Point solutions are not sufficient.*

For example, knowing that a system requires backup power because the usual power source is "unreliable" is helpful, but far from complete. Having a backup power source seems like a good idea. But, how reliable is the backup power source? How quickly must the backup power source become available if the primary source fails? For how long can the backup power source operate? Can the backup power source supply all of the equipment? What about the myriad other issues such as the possibility of hardware or software failure?

If attention is not paid to *all* of the potential sources of difficulty, the resulting system will not have the dependability that is required. Being sure that attention has been paid to potential sources of difficulty to the extent possible is our goal.

The subject of dependability of computer systems is complex and detailed. Dependability has to be dealt with in a methodical and scientific way. The path we will follow from here is a comprehensive and systematic one. The path mirrors the technology that we need to apply to computer system development. In particular, we will seek general approaches and learn to both recognize and *avoid* point solutions.