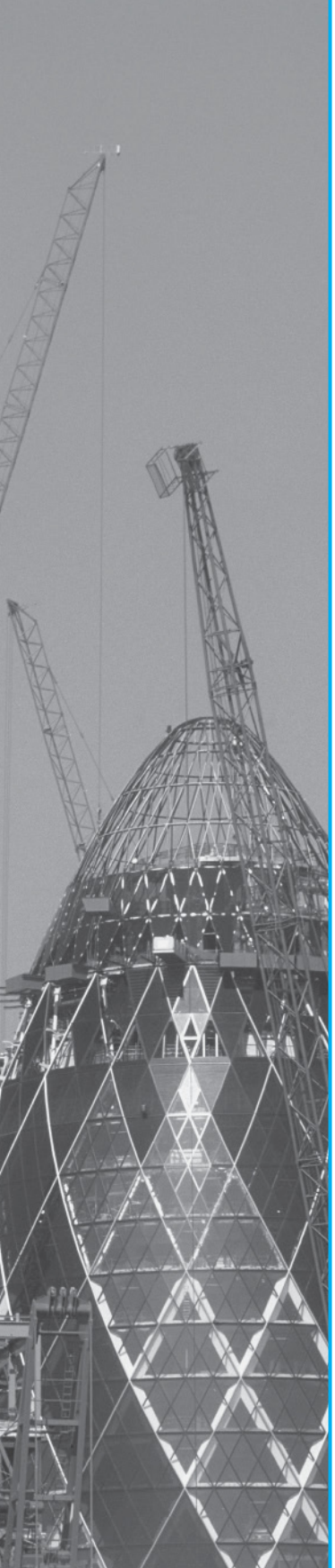
Software Engineering

TENTH EDITION

Ian Sommerville

ALWAYS LEARNING

PEARSON



19 Systems engineering

Objectives

The objectives of this chapter are to explain why software engineers should understand systems engineering and to introduce the most important systems engineering processes. When you have read this chapter, you will:

- know what is meant by a sociotechnical system and understand why human, social, and organizational issues affect the requirements and design of software systems;
- understand the idea of conceptual design and why it is an essential first stage in the systems engineering process;
- know what is meant by system procurement and understand why different system procurement processes are used for different types of system;
- know about the key systems engineering development processes and their relationships.

Contents

A computer only becomes useful when it includes both software and hardware. Without hardware, a software system is an abstraction—simply a representation of some human knowledge and ideas. Without software, a hardware system is a set of inert electronic devices. However, if you put them together to form a computer system, you create a machine that can carry out complex computations and deliver the results of these computations to its environment.

This illustrates one of the fundamental characteristics of a system: It is more than the sum of its parts. Systems have properties that only become apparent when their components are integrated and operate together. Furthermore, systems are developed to support human activities—work, entertainment, communication, protection of people and the environment, and so on. They interact with people, and their design is influenced by human and organizational concerns. Hardware, human, social, and organizational factors have to be taken into account when developing all professional software systems.

Systems that include software fall into two categories:

1. *Technical computer-based systems* are systems that include hardware and software components but not procedures and processes. Examples of technical systems include televisions, mobile phones, and other equipment with embedded software. Applications for PCs, computer games, and mobile devices are also technical systems. Individuals and organizations use technical systems for a particular purpose, but knowledge of this purpose is not part of the technical system. For example, the word processor I am using (Microsoft Word) is not aware that it is being used to write a book.

2 *Sociotechnical systems:* include one or more technical systems but, crucially, also people, who understand the purpose of the system, within the system itself. Sociotechnical systems have defined operational processes, and people (the operators) are inherent parts of the system. They are governed by organizational policies and rules and may be affected by external constraints such as national laws and regulatory policies. For example, this book was created through a sociotechnical publishing system that includes various processes (creation, editing, layout, etc.) and technical systems (Microsoft Word and Excel, Adobe Illustrator, InDesign, etc.).

Systems engineering (White et al. 1993; Stevens et al. 1998; Thayer 2002) is the activity of designing entire systems, taking into account the characteristics of hardware, software, and human elements of these systems. Systems engineering includes everything to do with procuring, specifying, developing, deploying, operating, and maintaining both technical and sociotechnical systems. Systems engineers have to consider the capabilities of hardware and software as well as the system's interactions with users and its environment. They must think about the system's services, the constraints under which the system must be built and operated, and the ways in which the system is used.

In this chapter, my focus is on the engineering of large and complex software-intensive systems. These are "enterprise systems," that is, systems that are used to support the goals of a large organization. Enterprise systems are used by government and the military services as well as large companies and other public bodies.

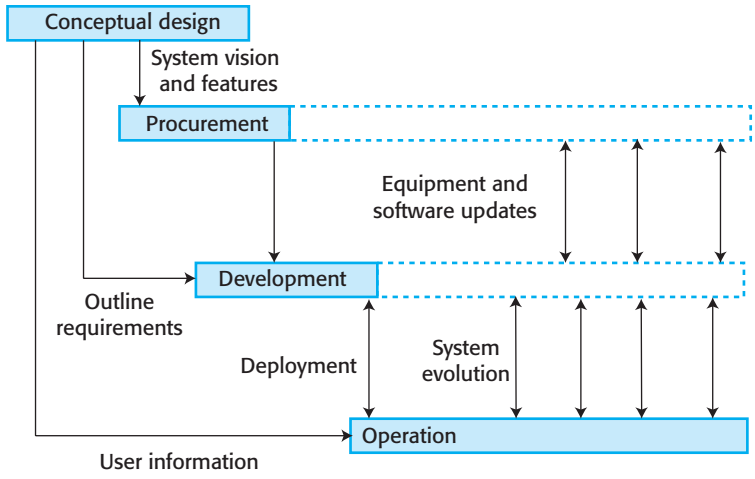


Figure 19.1 Stages of systems engineering

They are sociotechnical systems that are influenced by the ways that the organization works and by national and international rules and regulations. They may be made up of a number of separate systems and are distributed systems with large-scale databases. They have a long lifetime and are critical for the operation of the enterprise.

I believe that it is important for software engineers to know about systems engineering and to be active participants in systems engineering processes for two reasons:

1. Software is now the dominant element in all enterprise systems, yet many senior decision makers in organizations have a limited understanding of software. Software engineers have to play a more active part in high-level systems decision making if the system software is to be dependable and developed on time and to budget.
2. As a software engineer, it helps if you have a broader awareness of how software interacts with other hardware and software systems, and the human, social, and organizational factors that affect the ways in which software is used. This knowledge helps you understand the limits of software and to design better software systems.

There are four overlapping stages (Figure 19.1) in the lifetime of large, complex systems:

1. *Conceptual design* This initial systems engineering activity develops the concept of the type of system that is required. It sets out, in nontechnical language, the purpose of the system, why it is needed, and the high-level features that users might expect to see in the system. It may also describe broad constraints, such as the need for interoperability with other systems. These limit the freedom of systems engineers in designing and developing the system.
2. *Procurement or acquisition* During this stage, the conceptual design is further developed so that information is available to make decisions about the contract for the system development. This may involve making decisions about the distribution of

functionality across hardware, software, and operational processes. You also make decisions about which hardware and software has to be acquired, which suppliers should develop the system, and the terms and conditions of the supply contract.

3. *Development* During this stage, the system is developed. Development processes include requirements definition, system design, hardware and software engineering, system integration, and testing. Operational processes are defined, and the training courses for system users are designed.

4. *Operation* At this stage, the system is deployed, users are trained, and the system is brought into use. The planned operational processes usually then have to change to reflect the real working environment where the system is used. Over time, the system evolves as new requirements are identified. Eventually, the system declines in value, and it is decommissioned and replaced.

Figure 19.1 shows the interactions between these stages. The conceptual design activity is a basis for the system procurement and development but is also used to provide information to users about the system. Development and procurement overlap and further procurement during development, and operation may be needed as new equipment and software become available. Once the system is operational, requirements changes are inevitable; implementing these changes requires further development and, perhaps, software and hardware procurement.

Decisions made at any one of these stages may have a profound influence on the other stages. Design options may be restricted by procurement decisions on the scope of the system and on its hardware and software. Human errors made during the specification, design, and development stages may mean that faults are introduced into the system. A decision to limit testing for budget reasons may mean that faults are not discovered before a system is put into use. During operation, errors in configuring the system for deployment may lead to problems in using the system. Decisions made during the original procurement may be forgotten when system changes are proposed. This may lead to unforeseen consequences arising from the implementation of the changes.

An important difference between systems and software engineering is the involvement of a range of professionals throughout the lifetime of the system. These include engineers who may be involved in hardware and software design, system end-users, managers who are concerned with organizational issues, and experts in the system's application domain. For example, engineering the insulin pump system introduced in Chapter 1 requires experts in electronics, mechanical engineering, software, and medicine.

For very large systems, an even wider range of expertise may be required. Figure 19.2 illustrates the technical disciplines that may be involved in the procurement and development of a new system for air traffic management. Architects and civil engineers are involved because new air traffic management systems usually have to be installed in a new building. Electrical and mechanical engineers are involved to specify and maintain the power and air conditioning. Electronic engineers are concerned with computers, radars, and other equipment. Ergonomists

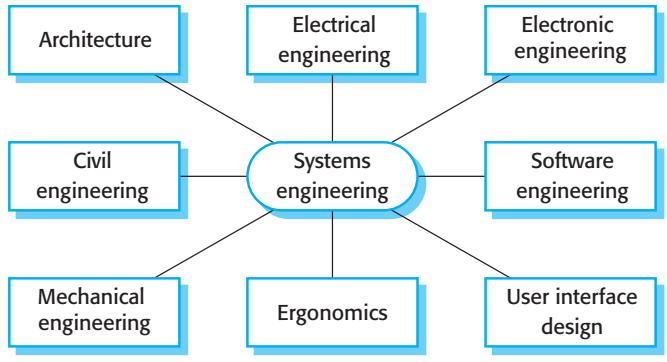


Figure 19.2 Professional disciplines involved in ATC systems engineering

design the controller workstations and software engineers, and user interface designers are responsible for the software in the system.

The involvement of a range of professional disciplines is essential because of the different types of components in complex systems. However, differences and misunderstandings between disciplines can lead to inappropriate design decisions. These poor decisions can delay the system's development or make it less suitable for its intended purpose. There are three reasons why there may be misunderstandings or other differences between engineers with different backgrounds:

1. Different professional disciplines often use the same words, but these words do not always mean the same thing. Consequently, misunderstandings are common in discussions between engineers from different backgrounds. If these are not discovered and resolved during system development, they can lead to errors in delivered systems. For example, an electronic engineer may know a bit about C programming but may not understand that a method in Java is like a function in C.

2. Each discipline makes assumptions about what other disciplines can or cannot do. These assumptions are often based on an inadequate understanding of what is possible. For example, an electronic engineer may decide that all signal processing (a computationally intensive task) should be done by software to simplify the hardware design. However, this may mean significantly greater software effort to ensure that the system processor can cope with the amount of computation that is resolved.

3. Disciplines try to protect their professional boundaries and may argue for certain design decisions because these decisions will call for their professional expertise. Therefore, a software engineer may argue for a software-based door locking system in a building, although a mechanical, key-based system may be more reliable.

My experience is that interdisciplinary working can be successful only if enough time is available for these issues to be discussed and resolved. This requires regular face-to-face discussions and a flexible approach from everyone involved in the systems engineering process.

19.1 Sociotechnical systems

The term *system* is universally used. We talk about computer systems, operating systems, payment systems, the education system, the system of government, and so on. These are all obviously quite different uses of the word "system," although they share the essential characteristic that, somehow, the system is more than simply the sum of its parts.

Abstract systems, such as the system of government, are outside the scope of this book. I focus here on systems that include computers and software and that have some specific purpose such as to enable communication, support navigation, or maintain medical records. A useful working definition of these types of system is as follows:

A system is a purposeful collection of interrelated components of different kinds that work together to deliver a set of services to the system owner and its users.

This general definition can cover a very wide range of systems. For example, a simple system, such as a laser pointer, delivers an indication service. It may include a few hardware components with a tiny control program in read-only memory (ROM). By contrast, an air traffic control system includes thousands of hardware and software components as well as human users who make decisions based on information from that computer system. It delivers a range of services, including providing information to pilots, maintaining safe separation of planes, utilizing airspace, and so on.

In all complex systems, the properties and behavior of the system components are inextricably intermingled. The successful functioning of each system component depends on the functioning of other components. Software can only operate if the processor is operational. The processor can only carry out computations if the software system defining these computations has been successfully installed.

Large-scale systems are often "systems of systems." That is, they are made up of several separate systems. For example, a police command and control system may include a geographical information system to provide details of the location of incidents. The same geographical information system may be used in systems for transport logistics and emergency command and control. Engineering systems of systems is an increasingly important topic in software engineering that I cover in Chapter 20.

Large-scale systems are, with a few exceptions, sociotechnical systems, which I explained in Chapter 10. That is, they do not just include software and hardware but also people, processes, and organizational policies. Sociotechnical systems are enterprise systems that are intended to help deliver a business purpose. This purpose might be to increase sales, reduce material used in manufacturing, collect taxes, maintain a safe airspace, and so on. Because they are embedded in an organizational environment, the procurement, development, and use of these systems are influenced by the organization's policies and procedures, as well as by its working culture. The users of the system are people who are influenced by the way the organization is managed and by their interactions with other people inside and outside of the organization.

The close relationships between sociotechnical systems and the organizations that use these systems means that it is often difficult to establish system boundaries.

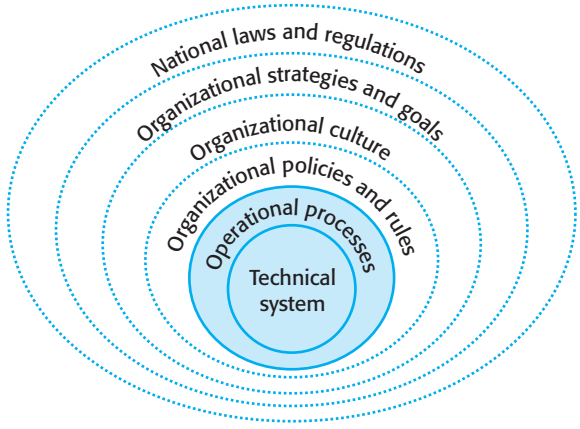


Figure 19.3 Layered structure of sociotechnical systems

Different people within the organization will see the boundaries of the system in different ways. This is significant because establishing what is and what is not in the scope of the system is important when defining the system requirements.

Figure 19.3 illustrates this problem. The diagram shows a sociotechnical system as a set of layers, where each layer contributes, in some way, to the functioning of the system. At the core is a software-intensive technical system and its operational processes (shaded in Figure 19.3). Most people would agree that these are both parts of the system. However, the system's behavior is influenced by a range of sociotechnical factors outside of the core. Should the system boundary simply be drawn around the core, or should it include other organizational levels?

Whether or not these broader sociotechnical considerations should be considered to be part of a system depends on the organization and its policies and rules. If organizational rules and policies can be changed, then some people might argue they should be part of the system. However, it is more difficult to change organizational culture and even more challenging to change strategy and goals. Only governments can change laws to accommodate a system. Moreover, different stakeholders may have different opinions on where the system boundaries should be drawn. There are no simple answers to these questions, but they have to be discussed and negotiated during the system design process.

Generally, large sociotechnical systems are used in organizations. When you are designing and developing sociotechnical systems, you need to understand, as far as possible, the organizational environment in which they will be used. If you don't, the systems may not meet business needs. Users and their managers may reject the system or fail to use it to its full potential.

Figure 19.4 shows the key elements in an organization that may affect the requirements, design, and operation of a sociotechnical system. A new system may lead to changes in some or all of these elements:

1. *Process changes* A new system may mean that people have to change the way that they work. If so, training will certainly be required. If changes are significant, or if they involve people losing their jobs, there is a danger that the users will resist the introduction of the system.

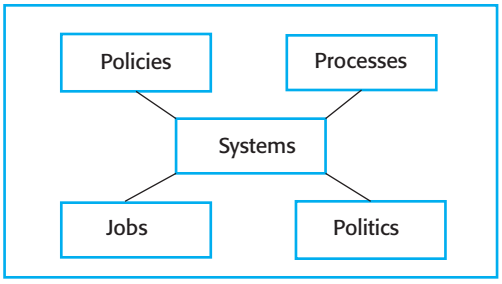


Figure 19.4
Organizational elements

2. *Job changes* New systems may deskill the users in an environment or cause them to change the way they work. If so, users may actively resist the introduction of the system into the organization. Professional staff, such as doctors or teachers, may resist system designs that require them to change their normal way of working. The people involved may feel that their professional expertise is being eroded and that their status in the organization is being reduced by the system.
3. *Organizational policies* The proposed system may not be completely consistent with organizational policies (e.g., on privacy). This may require system changes, policy changes, or process changes to bring the system and policies into line.
4. *Organizational politics* The system may change the political power structure in an organization. For example, if an organization is dependent on a complex system, those who control access to that system have a great deal of political power. Alternatively, if an organization reorganizes itself into a different structure, this may affect the requirements and use of the system.

Sociotechnical systems are complex systems, which means that it is practically impossible to have a complete understanding, in advance, of their behavior. This complexity leads to three important characteristics of sociotechnical systems:

1. They have emergent properties that are properties of the system as a whole, rather than associated with individual parts of the system. Emergent properties depend on both the system components and the relationships between them. Some of these relationships only come into existence when the system is integrated from its components, so the emergent properties can only be evaluated at that time. Security and dependability are examples of important emergent system properties.
2. They are nondeterministic, so that when presented with a specific input, they may not always produce the same output. The system's behavior depends on the human operators, and people do not always react in the same way. Furthermore, use of the system may create new relationships between the system components and hence change its emergent behavior.
3. The system's success criteria are subjective rather than objective. The extent to which the system supports organizational objectives does not just depend on the system itself. It also depends on the stability of these objectives, the relationships

Property	Description
Reliability	System reliability depends on component reliability, but unexpected interactions can cause new types of failure and therefore affect the reliability of the system.
Repairability	This property reflects how easy it is to fix a problem with the system once it has been discovered. It depends on being able to diagnose the problem, access the components that are faulty, and modify or replace these components.
Security	The security of the system (its ability to resist attack) is a complex property that cannot be easily measured. Attacks may be devised that were not anticipated by the system designers and so may defeat built-in safeguards.
Usability	This property reflects how easy it is to use the system. It depends on the technical system components, its operators, and its operating environment.
Volume	The volume of a system (the total space occupied) depends on how the component assemblies are arranged and connected.

Figure 19.5 Examples of emergent properties

and conflicts between organizational objectives, and how people in the organization interpret these objectives. New management may reinterpret the organizational objectives that a system was designed to support so that a "successful" system may then be seen as no longer fit for its intended purpose.

Sociotechnical considerations are often critical in determining whether or not a system has successfully met its objectives. Unfortunately, taking these into account is very difficult for engineers who have little experience of social or cultural studies. To help understand the effects of systems on organizations, various sociotechnical systems methodologies have been proposed. My paper on sociotechnical systems design discusses the advantages and disadvantages of these sociotechnical design methodologies (Baxter and Sommerville 2011).

19.1.1 Emergent properties

The complex relationships between the components in a system mean that a system is more than simply the sum of its parts. It has properties that are properties of the system as a whole. These "emergent properties" (Checkland 1981) cannot be attributed to any specific part of the system. Rather, they only emerge once the system components have been integrated. Some emergent properties, such as weight, can be derived directly from the subsystem properties. More often, however, they emerge from a combination of subsystem properties and subsystem relationships. The system property cannot be calculated directly from the properties of the individual system components. Examples of emergent properties are shown in Figure 19.5.

There are two types of emergent properties:

1. *Functional emergent properties,* when the purpose of a system only emerges after its components are integrated. For example, a bicycle has the functional property of being a transportation device once it has been assembled from its components.

2. *Non-functional emergent properties,* which relate to the behavior of the system in its operational environment. Reliability, performance, safety, and security are examples of these properties. These system characteristics are critical for computer-based systems, as failure to achieve a minimum defined level in these properties usually makes the system unusable. Some users may not need some of the system functions, so the system may be acceptable without them. However, a system that is unreliable or too slow is likely to be rejected by all its users.

Emergent properties, such as reliability, depend on both the properties of individual components and their interactions or relationships. For example, the reliability of a sociotechnical system is influenced by three things:

1. *Hardware reliability* What is the probability of hardware components failing, and how long does it take to repair a failed component?
2. *Software reliability* How likely is it that a software component will produce an incorrect output? Software failure is unlike hardware failure in that software does not wear out. Failures are often transient. The system carries on working after an incorrect result has been produced.
3. *Operator reliability* How likely is it that the operator of a system will make an error and provide an incorrect input? How likely is it that the software will fail to detect this error and propagate the mistake?

Hardware, software, and operator reliability are not independent but affect each other in unpredictable ways. Figure 19.6 shows how failures at one level can be propagated to other levels in the system. Say a hardware component in a system starts to go wrong. Hardware failure can sometimes generate spurious signals that are outside the range of inputs expected by the software. The software can then behave unpredictably and produce unexpected outputs. These may confuse and consequently cause stress in the system operator.

We know that people are more likely to make mistakes when they feel stressed. So a hardware failure may be the trigger for operator errors. These mistakes can, in turn, lead to unexpected software behavior, resulting in additional demands on the processor. This could overload the hardware, causing more failures and so on. Thus, an initial, relatively minor, failure, can rapidly develop into a serious problem that could lead to a complete shutdown of the system.

The reliability of a system depends on the context in which that system is used. However, the system's environment cannot be completely specified, and it is often impossible for the system designers to limit the environment for operational systems. Different systems operating within an environment may react to problems in unpredictable ways, thus affecting the reliability of all of these systems.

For example, say a system is designed to operate at normal room temperature. To allow for variations and exceptional conditions, the electronic components of a system are designed to operate within a certain range of temperatures, say, from 0 degrees to 40 degrees Celsius. Outside this temperature range, the components will

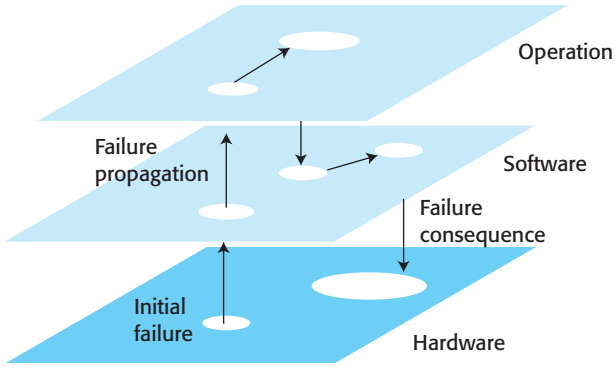


Figure 19.6 Failure propagation

behave in an unpredictable way. Now assume that this system is installed close to an air conditioner. If this air conditioner fails and vents hot gas over the electronics, then the system may overheat. The components, and hence the whole system may then fail.

If this system had been installed elsewhere in that environment, this problem would not have occurred. When the air conditioner worked properly, there were no problems. However, because of the physical closeness of these machines, an unanticipated relationship existed between them that led to system failure.

Like reliability, emergent properties such as performance or usability are hard to assess but can be measured after the system is operational. Properties such as safety and security, however, are not directly measurable. Here, you are not simply concerned with attributes that relate to the behavior of the system but also with unwanted or unacceptable behavior.

A secure system is one that does not allow unauthorized access to its data. Unfortunately, it is clearly impossible to predict all possible modes of access and explicitly forbid them. Therefore, it may only be possible to assess these "shall not" properties after the system is operational. That is, you only know that a system is insecure when someone manages to penetrate the system.

19.1.2 Non-determinism

A deterministic system is one that is absolutely predictable. If we ignore issues of concurrency, software systems that run on reliable hardware are deterministic. When they are presented with a sequence of inputs they will always produce the same sequence of outputs. Of course, there is no such thing as completely reliable hardware, but hardware is usually reliable enough to think of hardware systems as deterministic.

People, on the other hand, are non-deterministic. When presented with exactly the same input (say a request to complete a task), their responses will depend on their emotional and physical state, the person making the request, other people in the environment, and whatever else they are doing. Sometimes they will be happy to do the work, and, at other times, they will refuse; sometimes they will perform a task well, and sometimes they will do it badly.

Sociotechnical systems are nondeterministic partly because they include people and partly because changes to the hardware, software, and data in these systems are

so frequent. The interactions between these changes are complex, and so the behavior of the system is unpredictable. Users do not know when and why changes have been made, so they see the system as nondeterministic.

For example, say a system is presented with a set of 20 test inputs. It processes these inputs and the results are recorded. At some later time, the same 20 test inputs are processed, and the results are compared to the previous stored results. Five of them are different. Does this mean that there have been five failures? Or are the differences simply reasonable variations in the system's behavior? You can only find this out by looking at the results in more depth and making judgments about the way the system has handled each input.

Non-determinism is often seen as a bad thing, and it is felt that designers should try to avoid nondeterministic behavior wherever possible. In fact, in sociotechnical systems, non-determinism has important benefits. It means that the behavior of a system is not fixed for all time but can change depending on the system's environment. For example, operators may observe that a system is showing signs of failure. Instead of using the system normally, they can change their behavior to diagnose and recover from the detected problems.

19.1.3 Success criteria

Generally, complex sociotechnical systems are developed to tackle "wicked problems" (Rittel and Webber 1973). A wicked problem is a problem that is so complex and that involves so many related entities that there is no definitive problem specification. Different stakeholders see the problem in different ways, and no one has a full understanding of the problem as a whole. The true nature of the problem may only emerge as a solution is developed.

An extreme example of a wicked problem is emergency planning to deal with the aftermath of an earthquake. No one can accurately predict where the epicenter of an earthquake will be, what time it will occur, or what effect it will have on the local environment. It is impossible to specify in detail how to deal with the problem. System designers have to make assumptions, but understanding what is required emerges only after the earthquake has happened.

This makes it difficult to define the success criteria for a system. How do you decide if a new system contributes to the business goals of the company that paid for the system? The judgment of success is not usually made against the original reasons for procuring and developing the system. Rather, it is based on whether or not the system is effective at the time it is deployed. As the business environment can change very quickly, the business goals may have changed significantly during the development of the system.

The situation is even more complex when there are multiple conflicting goals that are interpreted differently by different stakeholders. For instance, the system on which the Mentcare system is based was designed to support two separate business goals:

1. To improve the quality of care for sufferers from mental illness.
2. To improve the cost-effectiveness of treatments by providing managers with detailed reports of care provided and the costs of that care.

Unfortunately, these proved to be conflicting goals because the information that was needed to satisfy the reporting goal meant that doctors and nurses had to provide additional information, over and above the health records that they normally maintained. This reduced the quality of care for patients as it meant that clinical staff had less time to talk with them. From a doctor's perspective, this system was not an improvement on the previous manual system, but from a manager's perspective, it was.

Thus, any success criteria that are established in the early stages of the systems engineering process have to be regularly reconsidered during system development and use. You cannot evaluate these criteria objectively as they depend on the system's effect on its environment and its users. A system may apparently meet its requirements as originally specified but be practically useless because of changes in the environment where it is used.

19.2 Conceptual design

Once an idea for a system has been suggested, conceptual design is the very first thing that you do in the systems engineering process. In the conceptual design phase, you take that initial idea, investigate its feasibility, and develop it to create an overall vision of a system that could be developed. You then have to describe the envisaged system so that nonexperts, such as system users, senior company decision makers, or politicians, can understand what you are proposing.

There is an obvious overlap between conceptual design and requirements engineering. As part of the conceptual design process, you have to imagine how the proposed system will be used. This may involve discussions with potential users and other stakeholders, focus groups, and observations of how existing systems are used. The goal of these activities is to understand how users work, what is important to them, and what practical constraints on the system there might be.

The importance of establishing a vision of a proposed system is rarely mentioned in the software design and requirements literature. However, this vision has been part of the systems engineering process for military systems for many years. Fairley et al. (Fairley, Thayer, and Bjorke 1994) discuss the idea of concept analysis and the documentation of the results of concept analysis in a "Concept of Operations" (ConOps) document. This idea of developing a ConOps document is now widely used for large-scale systems, and you can find many examples of ConOps documents on the web.

Unfortunately, as is so often the case with military and government systems, good ideas can become mired in bureaucracy and inflexible standards. This is exactly what happened with ConOps, and a ConOps document standard was proposed (IEEE, 2007). As Mostashari et al. say (Mostashari et al. 2012), this tends to lead to long and unreadable documents, which do not serve their intended purpose. They propose a more agile approach to the development of a ConOps document with a shorter and more flexible document as the output of the process.

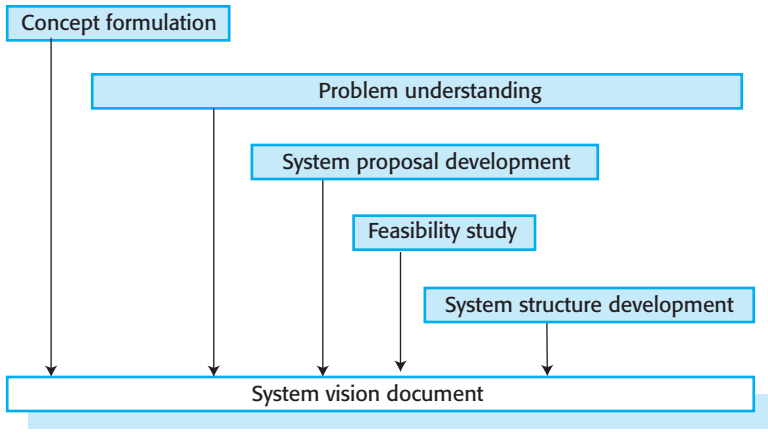


Figure 19.7 Conceptual design activities

I don't like the term *Concept of Operations* partly because of its military connotations and partly because I think that a conceptual design document is not just about system operation. It should also present the system engineer's understanding of why the system is being developed, an explanation of why the design proposals are appropriate, and, sometimes, an initial organization for the system. As Fairley says, "It should be organized to tell a story," that is, written so that people without a technical background can understand the proposals that are being made.

Figure 19.7 shows activities that may be part of the conceptual design process. Conceptual design should always be a team process that involves people from different backgrounds. I was part of the conceptual design team for the digital learning environment, introduced in Chapter 1. For the digital learning environment, the design team included teachers, education researchers, software engineers, system administrators, and system managers.

Concept formulation is the first stage of the process where you try to refine an initial statement of needs and work out what type of system would be best to meet the needs of system stakeholders. Initially, we were tasked with proposing an intranet for information sharing across schools that was easier to use than the current system. However, after discussions with teachers, we discovered that this was not really what was required. The existing system was awkward to use, but people had found workarounds. What was really required was a flexible digital learning environment that could be adapted by adding subject and age-specific tools and content that are freely available on the Internet.

We discovered this because the concept formulation activity overlapped with the activity of problem understanding. To understand a problem, you need to discuss with users and other stakeholders how they do their work. You need to find out what is important to them, what are the barriers that stop them from doing what they want to do, and their ideas of what changes are required. You need to be open-minded (it is their problem, not yours) and to be prepared to change your ideas when the reality does not match your initial vision.

In the system proposal development stage, the conceptual design team set out their ideas for alternative systems and these are the basis for a feasibility study to decide which of the ideas are worth further development. In a feasibility study, you should look at comparable systems that have been developed elsewhere and technological issues (e.g., use of mobile devices) that may affect use of the system. Then you need to assess whether or not the system could be implemented using current hardware and software technologies.

I have found that an additional useful activity is to develop an outline structure or architecture for the system. This activity is helpful both for making a feasibility assessment and for providing a basis for more detailed requirements engineering and architectural design. Furthermore, as the majority of systems are now assembled from existing systems and components, an initial architecture means that the key parts of the system have been identified and can be procured separately. This approach is often better than procuring a system as a monolithic unit from a single supplier.

For the digital learning environment, we decided on a layered service architecture (shown in Figure 1.8). All components in the system should be considered to be replaceable services. In this way, users can replace a standard service with their preferred alternative and so adapt the system to the ages and interests of the students learning with the system.

All of these activities generate information that is used to develop the system vision document. This is a critical document that senior decision makers use to decide whether or not further development of the system should go ahead. It is also used to develop further documents such as a risk analysis and budget estimate, which are also important inputs to the decision-making process.

Managers use the system vision document to understand the system; a procurement team uses it to define a tender document; and requirements engineers use it as a basis for refining the system requirements. Because these different people need different levels of detail, I suggest that the document should be structured into two parts:

1. A short summary for senior decision makers that presents the key points of the problem and the proposed system. It should be written so that readers can immediately see how the system will be used and the benefits that it will provide.
2. A number of appendices that develop the ideas in more detail and that can be used in the system procurement and requirements engineering activities.

It is challenging to write a summary of the system vision inasmuch as the readers are busy people who are unlikely to have a technical background. I have found that using user stories is very effective, providing a tangible vision of system use that nontechnical people can relate to. Stories should be short and personalized and should be a feasible description of the use of the system, as shown in Figure 19.8. There is another example of a user story from the same system in Chapter 4 (Figure 4.9).

Digital art

Jill is an S2 pupil at a secondary school in Dundee. She has a smartphone of her own, and the family has a shared Samsung tablet and a Dell laptop computer. At school, Jill signs on to the school computer and is presented with a personalized Glow+ environment, which includes a range of services, some chosen by her teachers and some she has chosen herself from the Glow app library.

She is working on a Celtic art project, and she uses Google to research a range of art sites. She sketches out some designs on paper and then uses the camera on her phone to photograph what she has done; she uploads this using the school wifi to her personal Glow+ space. Her homework is to complete the design and write a short commentary on her ideas.

At home, she uses the family tablet to sign on to Glow+, and she then uses an artwork app to process her photograph and to extend the work, add color, and so on. She finishes this part of the work, and to complete it she moves to her home laptop to type up her commentary. She uploads the finished work to Glow+ and sends a message to her art teacher that it is available for review. Her teacher looks at the project in a free period before Jill's next art class using a school tablet, and, in class, she discusses the work with Jill.

After the discussion, the teacher and Jill decide that the work should be shared, and so they publish it to the school web pages that show examples of students' work. In addition, the work is included in Jill's e-portfolio—her record of schoolwork from age 3 to 18.

Figure 19.8 A user story used in a system vision document

User stories are effective because, as already noted, readers can relate to them; in addition, they can show the capabilities of the proposed system in an easily accessible way. Of course, these are only part of a system vision, and the summary must also include a high-level description of the basic assumptions made and the ways in which the system will deliver value to the organization.

19.3 System procurement

System procurement or system acquisition is a process whose outcome is a decision to buy one or more systems from system suppliers. At this stage, decisions are made on the scope of a system that is to be purchased, system budgets and timescales, and high-level system requirements. Using this information, further decisions are then made on whether to procure a system, the type of system required, and the supplier or suppliers of the system. The drivers for these decisions are:

1. *The replacement of other organizational systems* If the organization has a mixture of systems that cannot work together or that are expensive to maintain, then procuring a replacement system, with additional capabilities, may lead to significant business benefits.
2. *The need to comply with external regulations* Increasingly, businesses are regulated and have to demonstrate compliance with externally defined regulations (e.g., Sarbanes–Oxley accounting regulations in the United States). Compliance may require the replacement of noncompliant systems or the provision of new systems specifically to monitor compliance.

3. *External competition* If a business needs to compete more effectively or maintain a competitive position, managers may decide to buy new systems to improve business efficiency or effectiveness. For military systems, the need to improve capability in the face of new threats is an important reason for procuring new systems.
4. *Business reorganization* Businesses and other organizations frequently restructure with the intention of improving efficiency and/or customer service. Reorganizations lead to changes in business processes that require new systems support.
5. *Available budget* The budget that is available is an obvious factor in determining the scope of new systems that can be procured.

In addition, new government systems are often procured to reflect political changes and political policies. For example, politicians may decide to buy new surveillance systems, which they claim will counter terrorism. Buying such systems shows voters that they are taking action.

Large complex systems are usually engineered using a mixture of off-the-shelf and specially built components. They are often integrated with existing legacy systems and organizational databases. When legacy systems and off-the-shelf systems are used, new custom software may be needed to integrate these components. The new software manages the component interfaces so that these components can interoperate. The need to develop this "glueware" is one reason why the savings from using off-the-shelf components are sometimes not as great as anticipated.

Three types of systems or system components may have to be procured:

1. Off-the-shelf applications that may be used without change and that need only minimal configuration for use.
2. Configurable application or ERP systems that have to be modified or adapted for use either by modifying the code or by using inbuilt configuration features, such as process definitions and rules.
3. Custom systems that have to be specially designed and implemented for use.

Each of these components tends to follow a different procurement process. Figure 19.9 illustrates the main features of the procurement process for these types of system. Key issues that affect procurement processes are:

1. Organizations often have an approved and recommended set of application software that has been checked by the IT department. It is usually possible to buy or acquire open-source software from this set directly without the need for detailed justification. For example, in the iLearn system, we recommended that Wordpress should be made available for student and staff blogs. If microphones are needed, off-the-shelf hardware can be bought. There are no detailed requirements, and the users adapt to the features of the chosen application.
2. Off-the-shelf components do not usually match requirements exactly, unless the requirements have been written with these components in mind. Therefore, choosing

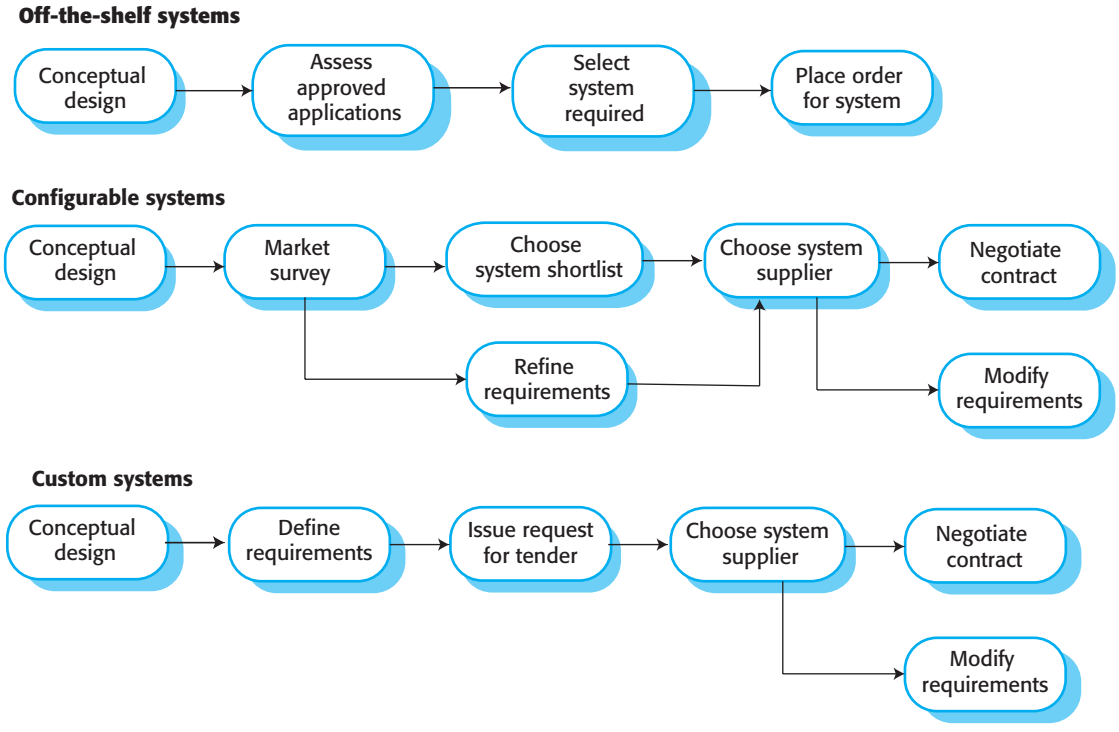


Figure 19.9 System procurement processes

a system means that you have to find the closest match between the system requirements and the facilities offered by off-the-shelf systems. ERP and other large-scale application systems usually fall into this category. You may then have to modify the requirements to fit in with the system assumptions. This can have knock-on effects on other subsystems. You also usually have an extensive configuration process to tailor and adapt the application or ERP system to the buyer's working environment.

3. When a system is to be built specially, the specification of requirements is part of the contract for the system being acquired. It is therefore a legal as well as a technical document. The requirements document is critical, and procurement processes of this type usually take a considerable amount of time.

4. For public sector systems in particular, there are detailed rules and regulations that affect the procurement of systems. For example, in the European Union, all public sector systems over a certain price must be open to tender by any supplier in Europe. This requires detailed tender documents to be drawn up and the tender to be advertised across Europe for a fixed period of time. Not only does this rule slow down the procurement process, it also tends to inhibit agile development. It forces the system buyer to develop requirements so that all companies have enough information to bid for the system contract.

5. For application systems that require change or for custom systems, there is usually a contract negotiation period when the customer and supplier negotiate the terms and conditions for development of the system. Once a system has been

selected, you may negotiate with the supplier on costs, license conditions, possible changes to the system, and other contractual issues. For custom systems, negotiations are likely to involve payment schedules, reporting, acceptance criteria, requirements change requests, and costs of system changes. During this process, requirements changes may be agreed that will reduce the overall costs and avoid some development problems.

Complex sociotechnical systems are rarely developed "in house" by the buyer of the system. Rather, external systems companies are invited to bid for the systems engineering contract. The customer's business is not systems engineering, so its employees do not have the skills needed to develop the systems themselves. For complex hardware/software systems, it may be necessary to use a group of suppliers, each with a different type of expertise.

For large systems, such as an air traffic management system, a group of suppliers may form a consortium to bid for a contract. The consortium should include all of the capabilities required for this type of system. For an ATC system, this would include computer hardware suppliers, software companies, peripheral suppliers, and suppliers of specialist equipment such as radar systems.

Customers do not usually wish to negotiate with multiple suppliers, so the contract is usually awarded to a principal contractor, who coordinates the project. The principal contractor coordinates the development of different subsystems by subcontractors. The subcontractors design and build parts of the system to a specification that is negotiated with the principal contractor and the customer. Once completed, the principal contractor integrates these components and delivers them to the customer.

Decisions made at the procurement stage of the systems engineering process are critical for later stages in that process. Poor procurement decisions often lead to problems such as late delivery of a system and development of systems that are unsuited to their operational environment. If the wrong system or the wrong supplier is chosen, then the technical processes of system and software engineering become more complex.

For example, I studied a system "failure" where a decision was made to choose an ERP system because this would "standardize" operations across the organization. These operations were very diverse, and it turned out there were good reasons for this. Standardization was practically impossible. The ERP system could not be adapted to cope with this diversity. It was ultimately abandoned after incurring costs of around £10 million.

Decisions and choices made during system procurement have a profound effect on the security and dependability of a system. For example, if a decision is made to procure an off-the-shelf system, then the organization has to accept that they have no influence over the security and dependability requirements of this system. System security depends on decisions made by system vendors. In addition, off-the-shelf systems may have known security weaknesses or may require complex configuration. Configuration errors, where entry points to the system are not properly secured, are a significant source of security problems.

On the other hand, a decision to procure a custom system means that a lot of effort must be devoted to understanding and defining security and dependability requirements. If a company has limited experience in this area, this is quite a difficult thing to do. If the

required level of dependability as well as acceptable system performance is to be achieved, then the development time may have to be extended and the budget increased.

Many bad procurement decisions stem from political rather than technical causes. Senior management may wish to have more control and so demand that a single system is used across an organization. Suppliers may be chosen because they have a long-standing relationship with a company rather than because they offer the best technology. Managers may wish to maintain compatibility with existing systems because they feel threatened by new technologies. As I discuss in Chapter 20, people who do not understand the required system are often responsible for procurement decisions. Engineering issues do not necessarily play a major part in their decision-making process.

19.4 System development

System development is a complex process in which the elements that are part of the system are developed or purchased and then integrated to create the final system. The system requirements are the bridge between the conceptual design and the development processes. During conceptual design, business and high-level functional and non-functional system requirements are defined. You can think of this as the start of development, hence the overlapping processes shown in Figure 19.1. Once contracts for the system elements have been agreed, more detailed requirements engineering takes place.

Figure 19.10 is a model of the systems development process. Systems engineering processes usually follow a "waterfall" process model similar to the one that I discussed in Chapter 2. Although the waterfall model is inappropriate for most types of software development, higher-level systems engineering processes are plan-driven processes that still follow this model.

Plan-driven processes are used in systems engineering because different elements of the system are independently developed. Different contractors are working concurrently on separate subsystems. Therefore, the interfaces to these elements have to be designed before development begins. For systems that include hardware and other equipment, changes during development can be very expensive or, sometimes, practically impossible. It is essential therefore, that the system requirements are fully understood before hardware development or building work begins.

One of the most confusing aspects of systems engineering is that companies use different terminology for each stage of the process. Sometimes, requirements engineering is part of the development process, and sometimes it is a separate activity. However, after conceptual design, there are seven fundamental development activities:

1. *Requirements engineering* is the process of refining, analyzing, and documenting the high-level and business requirements identified in the conceptual design. I have covered the most important requirements engineering activities in Chapter 4.

2. *Architectural design* overlaps significantly with the requirements engineering process. The process involves establishing the overall architecture of the system,

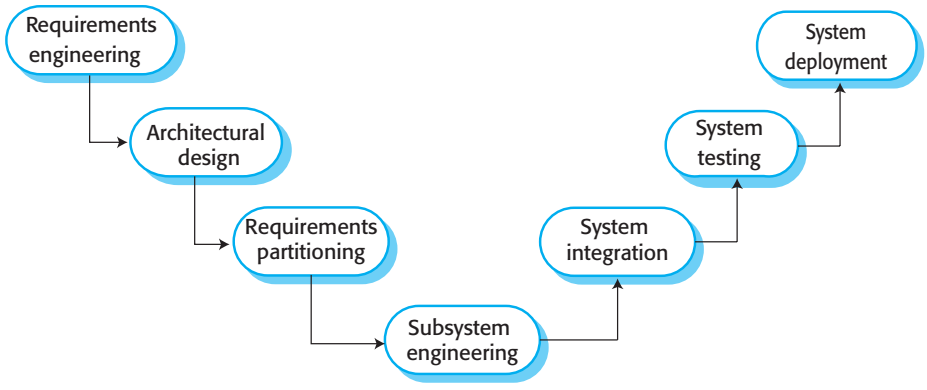


Figure 19.10 The systems development process

identifying the different system components, and understanding the relationships between them.

3. *Requirements partitioning* is concerned with deciding which subsystems (identified in the system architecture) are responsible for implementing the system requirements. Requirements may have to be allocated to hardware, software, or operational processes and prioritized for implementation. Ideally, you should allocate requirements to individual subsystems so that the implementation of a critical requirement does not need subsystem collaboration. However, this is not always possible. At this stage you also decide on the operational processes and on how these are used in the requirements implementation.

4. *Subsystem engineering* involves developing the software components of the system, configuring off-the-shelf hardware and software, designing, if necessary, special-purpose hardware, defining the operational processes for the system, and re-designing essential business processes.

5. *System integration* is the process of putting together system elements to create a new system. Only then do the emergent system properties become apparent.

6. *System testing* is an extended activity where the whole system is tested and problems are exposed. The subsystem engineering and system integration phases are reentered to repair these problems, tune the performance of the system, and implement new requirements. System testing may involve both testing by the system developer and acceptance/user testing by the organization that has procured the system.

7. *System deployment* is the process of making the system available to its users, transferring data from existing systems, and establishing communications with other systems in the environment. The process culminates with a "go live," after which users start to use the system to support their work.

Although the overall process is plan-driven, the processes of requirements development and system design are inextricably linked. The requirements and the high-level

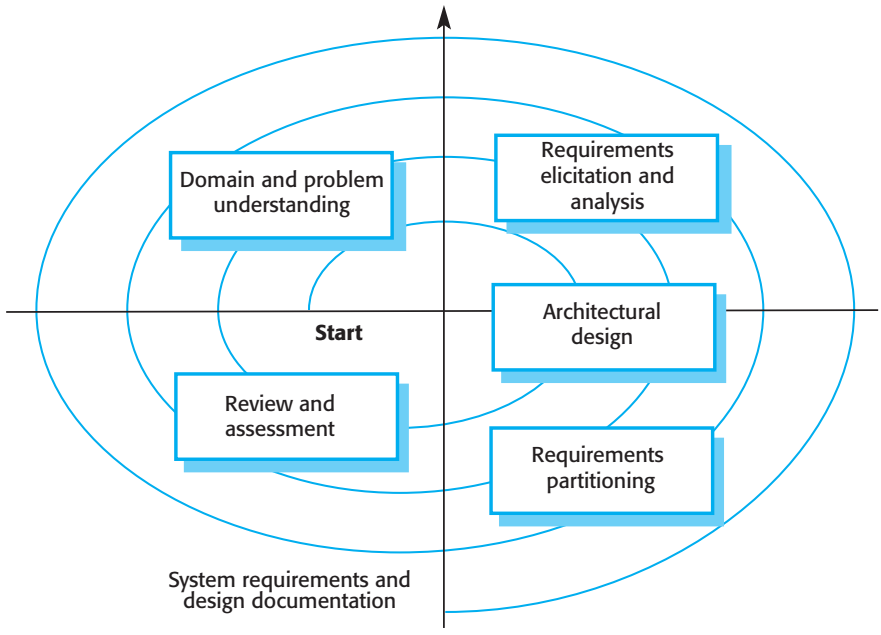


Figure 19.11 Requirements and design spiral

design are developed concurrently. Constraints posed by existing systems may limit design choices, and these choices may be specified in the requirements. You may have to do some initial design to structure and organize the requirements engineering process. As the design process continues, you may discover problems with existing requirements and new requirements may emerge. Consequently, you can think of these linked processes as a spiral, as shown in Figure 19.11.

The spiral reflects the reality that requirements affect design decisions and vice versa, and so it makes sense to interleave these processes. Starting in the center, each round of the spiral may add detail to the requirements and the design. As subsystems are identified in the architecture, decisions are made on the responsibilities of these subsystems for providing the system requirements. Some rounds of the spiral may focus on requirements, others on design. Sometimes new knowledge collected during the requirements and design process means that the problem statement itself has to be changed.

For almost all systems, many possible designs meet the requirements. These cover a range of solutions that combine hardware, software, and human operations. The solution that you choose for further development may be the most appropriate technical solution that meets the requirements. However, wider organizational and political considerations may influence the choice of solution. For example, a government client may prefer to use national rather than foreign suppliers for its system, even if national products are technically inferior.

These influences usually take effect in the review and assessment phase of the spiral model where designs and requirements may be accepted or rejected. The process ends when a review decides that the requirements and high-level design are sufficiently detailed for subsystems to be specified and designed.

Subsystem engineering involves designing and building the system's hardware and software components. For some types of systems, such as spacecraft, all hardware and software components may be designed and built during the development process. However, in most systems, some components are bought rather than developed. It is usually much cheaper to buy existing products than to develop special-purpose components. However, if you buy large off-the-shelf systems, such as ERP systems, there is a significant cost in configuring these systems for use in their operational environment.

Subsystems are usually developed in parallel. When problems that cut across subsystem boundaries are encountered, a system modification request must be made. Where systems involve extensive hardware engineering, making modifications after manufacturing has started is usually very expensive. Often "workarounds" that compensate for the problem must be found. These workarounds usually involve software changes to implement new requirements.

During systems integration, you take the independently developed subsystems and put them together to make up a complete system. This integration can be achieved using a "big bang" approach, where all the subsystems are integrated at the same time. However, for technical and managerial reasons, an incremental integration process where subsystems are integrated one at a time is the best approach:

1. It is usually impossible to schedule the development of all the subsystems so that they are all finished at the same time.
2. Incremental integration reduces the cost of error location. If many subsystems are simultaneously integrated, an error that arises during testing may be in any of these subsystems. When a single subsystem is integrated with an already working system, errors that occur are probably in the newly integrated subsystem or in the interactions between the existing subsystems and the new subsystem.

As an increasing number of systems are built by integrating off-the-shelf hardware and software application systems, the distinction between implementation and integration is becoming blurred. In some cases, there is no need to develop new hardware or software. Essentially, systems integration is the implementation phase of the system.

During and after the integration process, the system is tested. This testing should focus on testing the interfaces between components and the behavior of the system as a whole. Inevitably, testing also reveals problems with individual subsystems that have to be repaired. Testing takes a long time, and a common problem in system development is that the testing team may run out of either budget or time. This problem can lead to the delivery of error-prone systems that need be repaired after they have been deployed.

Subsystem faults that are a consequence of invalid assumptions about other subsystems are often exposed during system integration. This may lead to disputes between the contractors responsible for implementing different subsystems. When problems are discovered in subsystem interaction, the contractors may argue about which subsystem is faulty. Negotiations on how to solve the problems can take weeks or months.

The final stage of the system development process is system delivery and deployment. The software is installed on the hardware and is readied for operation. This may

involve more system configuration to reflect the local environment where it is used, the transfer of data from existing systems, and the preparation of user documentation and training. At this stage, you may also have to reconfigure other systems in the environment to ensure that the new system interoperates with them.

Although system deployment is straightforward in principle, it is often more difficult than anticipated. The user environment may be different from that anticipated by the system developers. Adapting the system to make it work in an unexpected environment can be difficult. The existing system data may require extensive clean-up, and parts of it may involve more effort than expected. The interfaces to other systems may not be properly documented. You may find that the planned operational processes have to be changed because they are not compatible with the operational processes for other systems. User training is often difficult to arrange, with the consequence that, initially at least, users are unable to access the capabilities of the system. System deployment can therefore take much longer and cost much more than anticipated.

19.5 System operation and evolution

Operational processes are the processes that are involved in using the system as intended by its designers. For example, operators of an air traffic control system follow specific processes when aircraft enter and leave airspace, when they have to change height or speed, when an emergency occurs, and so on. For new systems, these operational processes have to be defined and documented during the system development process. Operators may have to be trained and other work processes adapted to make effective use of the new system. Undetected problems may arise at this stage because the system specification may contain errors or omissions. While the system may perform to specification, its functions may not meet the real operational needs. Consequently, the operators may not use the system as its designers intended.

Although the designers of operational processes may have based their process designs on extensive user studies, there is always a period of "domestication" (Stewart and Williams 2005) when users adapt to the new system and work out practical processes of how to use it. While user interface design is important, studies have shown that, given time, users can adapt to complex interfaces. As they become experienced, they prefer ways of using the system quickly rather than easily. This means that when designing systems, you should not simply cater for inexperienced users but you should design the user interface to be adaptable for experienced users.

Some people think that system operators are a source of problems in a system and that we should move toward automated systems where operator involvement is minimized. In my opinion, there are two problems with this approach:

1. It is likely to increase the technical complexity of the system because it has to be designed to cope with all anticipated failure modes. This increases the costs and

time required to build the system. Provision also has to be made to bring in people to deal with unanticipated failures.

2. People are adaptable and can cope with problems and unexpected situations. Thus, you do not have to anticipate everything that could possibly go wrong when you are specifying and designing the system.

People have a unique capability of being able to respond effectively to the unexpected, even when they have never had direct experience of these unexpected events or system states. Therefore, when things go wrong, the system operators can often recover the situation by finding workarounds and using the system in nonstandard ways. Operators also use their local knowledge to adapt and improve processes. Normally, the actual operational processes are different from those anticipated by the system designers.

Consequently, you should design operational processes to be flexible and adaptable. The operational processes should not be too constraining; they should not require operations to be done in a particular order; and the system software should not rely on a specific process being followed. Operators usually improve the process because they know what does and does not work in a real situation.

A problem that may only emerge after the system goes into operation is the operation of the new system alongside existing systems. There may be physical problems of incompatibility, or it may be difficult to transfer data from one system to another. More subtle problems might arise because different systems have different user interfaces. Introducing a new system may increase the operator error rate, as the operators use user interface commands for the wrong system.

19.5.1 System evolution

Large, complex systems usually have a long lifetime. Complex hardware/software systems may remain in use for more than 20 years, even though both the original hardware and software technologies used are obsolete. There are several reasons for this longevity, as shown in Figure 19.12.

Over their lifetime, large complex systems change and evolve to correct errors in the original system requirements and to implement new requirements that have emerged. The system's computers are likely to be replaced with new, faster machines. The organization that uses the system may reorganize itself and hence use the system in a different way. The external environment of the system may change, forcing changes to the system. Hence, evolution is a process that runs alongside normal system operational processes. System evolution involves reentering the development process to make changes and extensions to the system's hardware, software, and operational processes.

System evolution, like software evolution (discussed in Chapter 9), is inherently costly for several reasons:

1. Proposed changes have to be analyzed very carefully from a business and a technical perspective. Changes have to contribute to the goals of the system and should not simply be technically motivated.

Factor	Rationale
Investment cost	The costs of a systems engineering project may be tens or even hundreds of millions of dollars. These costs can only be justified if the system can deliver value to an organization for many years.
Loss of expertise	As businesses change and restructure to focus on their core activities, they often lose engineering expertise. This may mean that they lack the ability to specify the requirements for a new system.
Replacement cost	The cost of replacing a large system is very high. Replacing an existing system can be justified only if this leads to significant cost savings over the existing system.
Return on investment	If a fixed budget is available for systems engineering, spending on new systems in some other area of the business may lead to a higher return on investment than replacing an existing system.
Risks of change	Systems are an inherent part of business operations, and the risks of replacing existing systems with new systems cannot be justified. The danger with a new system is that things can go wrong in the hardware, software, and operational processes. The potential costs of these problems for the business may be so high that they cannot take the risk of system replacement.
System dependencies	Systems are interdependent and replacing one of these systems may lead to extensive changes in other systems.

Figure 19.12 Factors that influence system lifetimes

2. Because subsystems are never completely independent, changes to one subsystem may have side-effects that adversely affect the performance or behavior of other subsystems. Consequent changes to these subsystems may therefore be needed.
3. The reasons for original design decisions are often unrecorded. Those responsible for the system evolution have to work out why particular design decisions were made.
4. As systems age, their structure becomes corrupted by change, so the costs of making further changes increases.

Systems that have been in use for many years are often reliant on obsolete hardware and software technology. These "legacy systems" (discussed in Chapter 9) are sociotechnical computer-based systems that have been developed using technology that is now obsolete. However, they don't just include legacy hardware and software. They also rely on legacy processes and procedures—old ways of doing things that are difficult to change because they rely on legacy software. Changes to one part of the system inevitably involve changes to other components.

Changes made to a system during system evolution are often a source of problems and vulnerabilities. If the people implementing the changes are different from those who developed the system, they may be unaware that a design decision was taken for dependability and security reasons. Therefore, they may change the system and lose some safeguards that were deliberately implemented when the system was built. Furthermore, as testing is so expensive, complete retesting may be impossible after every system change. Consequently, testing may not discover the adverse side-effects of changes that introduce or expose faults in other system components.

KEY POINTS

- Systems engineering is concerned with all aspects of specifying, buying, designing, and testing complex sociotechnical systems.
- Sociotechnical systems include computer hardware, software, and people, and are situated within an organization. They are designed to support organizational or business goals and objectives.
- The emergent properties of a system are characteristics of the system as a whole rather than of its component parts. They include properties such as performance, reliability, usability, safety, and security.
- The fundamental systems engineering processes are conceptual systems design, system procurement, system development, and system operation.
- Conceptual systems design is a key activity where high-level system requirements and a vision of the operational system is developed.
- System procurement covers all of the activities involved in deciding what system to buy and who should supply that system. Different procurement processes are used for off-the-shelf application systems, configurable COTS systems, and custom systems.
- System development processes include requirements specification, design, construction, integration, and testing.
- When a system is put into use, the operational processes and the system itself inevitably change to reflect changes to the business requirements and the system's environment.

FURTHER READING

"Airport 95: Automated Baggage System." An excellent, readable case study of what can go wrong with a systems engineering project and how software tends to get the blame for wider systems failures. (*ACM Software Engineering Notes*, 21, March 1996). http://doi.acm.org/10.1145/227531.227544

"Fundamentals of Systems Engineering." This is the introductory chapter in NASA's systems engineering handbook. It presents an overview of the systems engineering process for space systems. Although these are mostly technical systems, there are sociotechnical issues to be considered. Dependability is obviously critically important. (In *NASA Systems Engineering Handbook*, NASA-SP 2007-6105, 2007). http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20080008301_2008008500.pdf

The LSCITS Socio-technical Systems Handbook. This handbook introduces sociotechnical systems in an accessible way and provides access to more detailed papers on sociotechnical topics. (Various authors, 2012). http://archive.cs.st-andrews.ac.uk/STSE-Handbook

Architecting systems: Concepts, Principles and Practice. This is a refreshingly different book on systems engineering that does not have the hardware focus of many "traditional" systems engineering books.

The author, who is an experienced systems engineer, draws on examples from a wide range of systems and recognizes the importance of sociotechnical as well as technical issues. (H. Sillitto, College Publications, 2014).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/systems-engineering/

EXERCISES

19.1. Give two examples of government functions that are supported by complex sociotechnical systems and explain why, in the foreseeable future, these functions cannot be completely automated.

19.2. Explain briefly why the involvement of a range of professional disciplines is essential in systems engineering.

19.3. Complex sociotechnical systems lead to three important characteristics. What are they? Explain each in brief.

19.4. What is a "wicked problem"? Explain why the development of a national medical records system should be considered a "wicked problem."

19.5. A multimedia virtual museum system offering virtual experiences of ancient Greece is to be developed for a consortium of European museums. The system should provide users with the facility to view 3-D models of ancient Greece through a standard web browser and should also support an immersive virtual reality experience. Develop a conceptual design for such a system, highlighting its key characteristics and essential high-level requirements.

19.6. Explain why you need to be flexible and adapt system requirements when procuring large off-the-shelf software systems, such as ERP systems. Search the web for discussions of the failures of such systems and explain, from a sociotechnical perspective, why these failures occurred. A possible starting point is: http://blog.360cloudsolutions.com/blog/bid/94028/Top-Six-ERP-Implementation-Failures

19.7. Why is system integration a particularly critical part of the systems development process? Suggest three sociotechnical issues that may cause difficulties in the system integration process.

19.8. Why is system evolution inherently costly?

19.9. What are the arguments for and against considering system engineering as a profession in its own right, like electrical engineering or software engineering?

19.10. You are an engineer involved in the development of a financial system. During installation, you discover that this system will make a significant number of people redundant. The people in the environment deny you access to essential information to complete the system installation. To what extent should you, as a systems engineer, become involved in this situation? Is it your professional responsibility to complete the installation as contracted? Should you simply abandon the work until the procuring organization has sorted out the problem?

REFERENCES

Baxter, G., and I. Sommerville. 2011. "Socio-Technical Systems: From Design Methods to Systems Engineering." *Interacting with Computers* 23 (1): 4–17. doi:10.1016/j.intcom.2010.07.003.

Checkland, P. 1981. *Systems Thinking, Systems Practice*. Chichester, UK: John Wiley & Sons.

Fairley, R. E., R. H. Thayer, and P. Bjorke. 1994. "The Concept of Operations: The Bridge from Operational Requirements to Technical Specifications." In *1st Int. Conf. on Requirements Engineering,* 40–7. Colorado Springs, CO. doi:10.1109/ICRE.1994.292405.

IEEE. 2007. "IEEE Guide for Information Technology. System Definition—Concept of Operations (ConOps) Document." *Electronics*. Vol. 1998. doi:10.1109/IEEESTD.1998.89424. http://ieeexplore.ieee.org/servlet/opac?punumber=6166

Mostashari, A., S. A. McComb, D. M. Kennedy, R. Cloutier, and P. Korfiatis. 2012. "Developing a Stakeholder-Assisted Agile CONOPS Development Process." *Systems Engineering* 15 (1): 1–13. doi:10.1002/sys.20190.

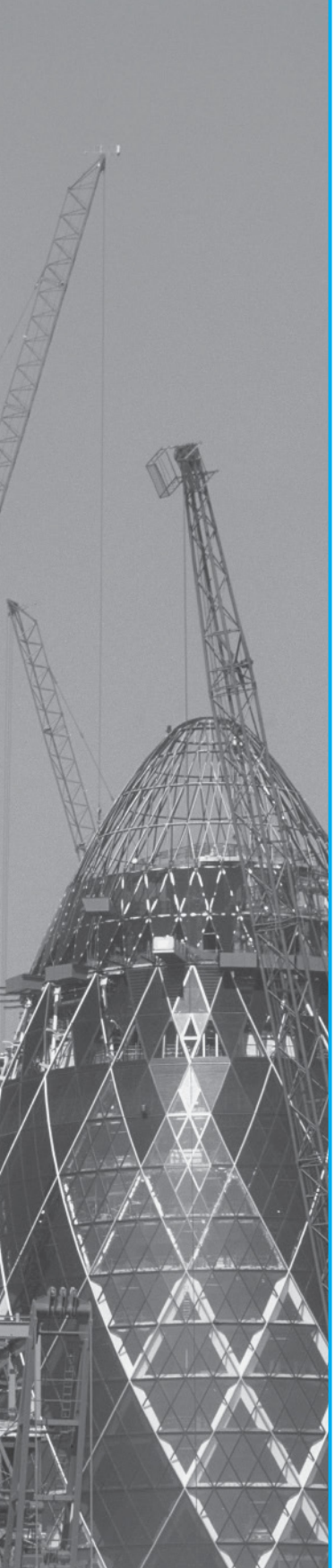
Rittel, H., and M. Webber. 1973. "Dilemmas in a General Theory of Planning." *Policy Sciences* 4: 155–169. doi:10.1007/BF01405730.

Stevens, R., P. Brook, K. Jackson, and S. Arnold. 1998. *Systems Engineering: Coping with Complexity.* London: Prentice-Hall.

Stewart, J., and R. Williams. 2005. "The Wrong Trousers? Beyond the Design Fallacy: Social Learning and the User." In *User Involvement in Innovation Processes. Strategies and Limitations from a Socio-Technical Perspective,* edited by H. Rohrache, 39–71. Berlin: Profil-Verlag.

Thayer, R. H. 2002. "Software System Engineering: A Tutorial." *IEEE Computer* 35 (4): 68–73. doi:10.1109/MC.2002.993773.

White, S., M. Alford, J. Holtzman, S. Kuehl, B. McCay, D. Oliver, D. Owens, C. Tully, and A. Willey. 1993. "Systems Engineering of Computer-Based Systems." *IEEE Computer* 26 (11): 54–65. doi:10.1109/ECBS.1994.331687.



20
Systems of systems

Objectives

The objectives of this chapter are to introduce the idea of a system of systems and to discuss the challenges of building complex systems of software systems. When you have read this chapter, you will:

- understand what is meant by a system of systems and how it differs from an individual system;
- understand systems of systems classification and the differences between different types of systems of systems;
- understand why conventional methods of software engineering that are based on reductionism are inadequate for developing systems of systems;
- have been introduced to the systems of systems engineering process and architectural patterns for systems of systems.

Contents

We need software engineering because we create large and complex software systems. The discipline emerged in the 1960s because the first attempts to build large software systems mostly went wrong. Creating software was much more expensive than expected, took longer than planned, and the software itself was often unreliable. To address these problems, we have developed a range of software engineering techniques and technologies, which have been remarkably successful. We can now build systems that are much larger, more complex, much more reliable, and more effective than the software systems of the 1970s.

However, we have not "solved" the problems of large system engineering. Software project failures are still common. For example, there have been serious problems and delays in the implementation of government health care systems in both the United States and the UK. The root cause of these problems is, as it was in the 1960s, that we are trying to build systems that are larger and more complex than before. We are attempting to build these "mega-systems" using methods and technology that were never designed for this purpose. As I discuss later in the chapter, I believe that current software engineering technology cannot scale up to cope with the complexity that is inherent in many of the systems now being proposed.

The increase in size of software systems since the introduction of software engineering has been remarkable. Today's large systems may be a hundred or even a thousand times larger than the "large" systems of the 1960s. Northrop and her colleagues (Northrop et al. 2006) suggested in 2006 that we would shortly see the development of systems with a billion lines of code. Almost 10 years after this prediction, I suspect such systems are already in use.

Of course, we do not start with nothing and then write a billion lines of code. As I discussed in Chapter 15, the real success story of software engineering has been software reuse. It is only because we have developed ways of reusing software across applications and systems that large-scale development is possible. Very large-scale systems now and in the future will be built by integrating existing systems from different providers to create systems of systems (SoS).

What do we mean when we talk about a system of systems? As Hitchens says (Hitchins 2009), from a general systems perspective, there is no difference between a system and a system of systems. Both have emergent properties and can be composed from subsystems. However, from a software engineering perspective, I think there is a useful distinction between these terms. This distinction is sociotechnical rather than technical:

> *A system of systems is a system that contains two or more independently managed elements.*

This means that there is no single manager for all of the parts of the system of systems and that different parts of a system are subject to different management and control policies and rules. As we shall see, distributed management and control has a profound effect on the overall complexity of the system.

This definition of systems of systems says nothing about the size of systems of systems. A relatively small system that includes services from different providers is

a system of systems. Some of the problems of SoS engineering apply to such small systems, but the real challenges emerge when the constituent systems are themselves large-scale systems.

Much of the work in the area of systems of systems has come from the defense community. As the capability of software systems increased in the late 20th century, it became possible to coordinate and control previously independent military systems, such as naval and ground-based air and ship defense systems. The system might include tens or hundreds of separate elements, with software systems keeping track of these elements and providing controllers with information that allows them to be deployed most effectively.

This type of system of systems is outside the scope of a software engineering book. Instead, I focus here on systems of systems where the system elements are software systems rather than hardware such as aircraft, military vehicles, or radars. Systems of software systems are created by integrating separate software systems, and, at the time of writing, most software SoS include a relatively small number of separate systems. Each constituent system is usually a complex system in its own right. However, it is predicted that, over the next few years, the size of software SoS is likely to grow significantly as more and more systems are integrated to make use of the capabilities that they offer.

Examples of systems of systems of software systems are:

1. A cloud management system that handles local private cloud management and management of servers on public clouds such as Amazon and Microsoft.
2. An online banking system that handles loan requests and that connects to a credit reference system provided by credit reference agencies to check the credit of applicants.
3. An emergency information system that integrates information from police, ambulance, fire, and coast guard services about the assets available to deal with civil emergencies such as flooding and large-scale accidents.
4. The digital learning environment (iLearn) that I introduced in Chapter 1. This system provides a range of learning support by integrating separate software systems such as Microsoft Office 365, virtual learning environments such as Moodle, simulation modeling tools, and content such as newspaper archives.

Maier (Maier 1998) identified five essential characteristics of systems of systems:

1. *Operational independence of elements* Parts of the system are not simply components but can operate as useful systems in their own right. The systems within the SoS evolve independently.
2. *Managerial independence of elements* Parts of the system are "owned" and managed by different organizations or by different parts of a larger organization. Therefore different rules and policies apply to the management and evolution of

these systems. As I have suggested, this is the key factor that distinguishes a system of systems from a system.

3. *Evolutionary development* SoS are not developed in a single project but evolve over time from their constituent systems.

4. *Emergence* SoS have emergent characteristics that only become apparent after the SoS has been created. Of course, as I have discussed in Chapter 19, emergence is a characteristic of all systems, but it is particularly important in SoS.

5. *Geographical distribution of elements* The elements of a SoS are often geographically distributed across different organizations. This is important technically because it means that an externally-managed network is an integral part of the SoS. It is also important managerially as it increases the difficulties of communication between those involved in making system management decisions and adds to the difficulties of maintaining system security.[†]

I would like to add two further characteristics to Maier's list that are particularly relevant to systems of software systems:

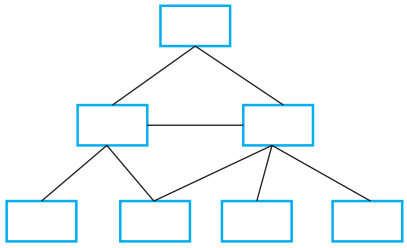
1. *Data intensive* A software SoS typically relies on and manages a very large volume of data. In terms of size, this may be tens or even hundreds of times larger than the code of the constituent systems itself.

2. *Heterogeneity* The different systems in a software SoS are unlikely to have been developed using the same programming languages and design methods. This is a consequence of the very rapid pace of evolution of software technologies. Companies frequently update their development methods and tools as new, improved versions become available. In a 20-year lifetime of a large SoS, technologies may change four or five times.

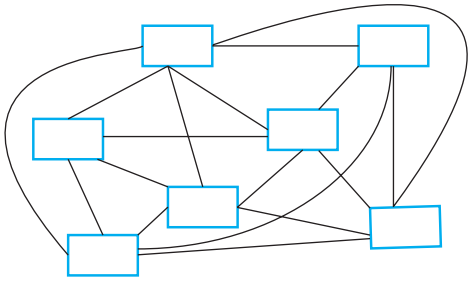
As I discuss in Section 20.1, these characteristics mean that SoS can be much more complex than systems with a single owner and manager. I believe that our current software engineering methods and techniques cannot scale to cope with this complexity. Consequently, problems with the very large and complex systems that we are now developing are inevitable. We need a completely new set of abstractions, methods, and technologies for software systems of systems engineering.

This need has been recognized independently by a number of different authorities. In the UK, a report published in 2004 (Royal Academy of Engineering 2004) led to the establishment of a national research and training initiative in large-scale complex IT systems (Sommerville et al. 2012). In the United States, the Software Engineering Institute reported on Ultra-Large Scale Systems in 2006 (Northrop et al. 2006). From the systems engineering community, Stevens (Stevens 2010) discusses the problems of constructing "mega-systems" in transport, health care, and defense.

[†]Maier, M. W. 1998. "Architecting Principles for Systems-of-Systems." Systems Engineering 1 (4): 267–284. doi:10.1002/(SICI)1520-6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D.



System (a)



System (b)

Figure 20.1 Simple and complex systems

20.1 System complexity

I suggested in the introduction that the engineering problems that arise when constructing systems of software systems are due to the inherent complexity of these systems. In this section, I explain the basis of system complexity and discuss the different types of complexity that arise in software SoS.

All systems are composed of parts (elements) with relationships between these elements of the system. For example, the parts of a program may be objects, and the parts of each object may be constants, variables, and methods. Examples of relationships include "calls" (method A calls method B), "inherits-from" (object X inherits the methods and attributes of object Y), and "part of" (method A is part of object X).

The complexity of any system depends on the number and types of relationships between system elements. Figure 20.1 shows examples of two systems. System (a) is a relatively simple system with only a small number of relationships between its elements. By contrast, System (b), with the same number of elements, is a more complex system because it has many more element–element relationships.

The type of relationship also influences the overall complexity of a system. Static relationships are relationships that are planned and analyzable from static depictions of the system. Therefore, the "uses" relationship in a software system is a static relationship. From either the software source code or a UML model of a system, you can work out how any one software component uses other components.

Dynamic relationships are relationships that exist in an executing system. The "calls" relationship is a dynamic relationship because, in any system with if-statements, you cannot tell whether or not one method will call another method. It depends on the runtime inputs to the system. Dynamic relationships are more complex to analyze as you need to know the system inputs and data used as well as the source code of the system.

As well as system complexity, we also have to consider the complexity of the processes used to develop and maintain the system once it has gone into use. Figure 20.2 illustrates these processes and their relationship with the developed system.

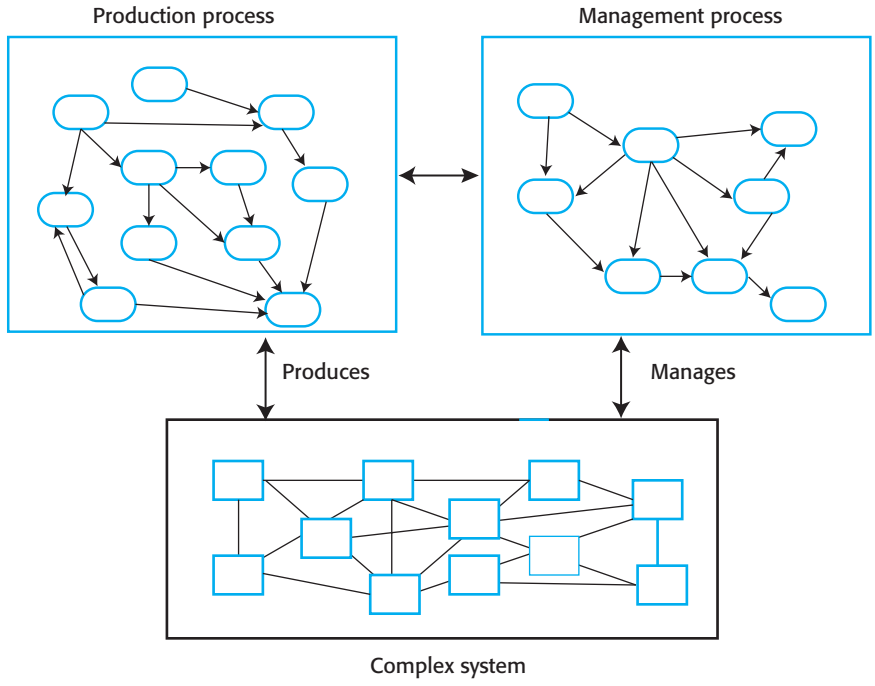


Figure 20.2 Production and management processes

As systems grow in size, they need more complex production and management processes. Complex processes are themselves complex systems. They are difficult to understand and may have undesirable emergent properties. They are more time consuming than simpler processes, and they require more documentation and coordination between the people and the organizations involved in the system development. The complexity of the production process is one of the main reasons why projects go wrong, with software delivered late and overbudget. Therefore, large systems are always at risk of cost and time overruns.

Complexity is important for software engineering because it is the main influence on the understandability and the changeability of a system. The more complex a system, the more difficult it is to understand and analyze. Given that complexity is a function of the number of relationships between elements of a system, it is inevitable that large systems are more complex than small systems. As complexity increases, there are more and more relationships between elements of the system and an increased likelihood that changing one part of a system will have undesirable effects elsewhere.

Several different types of complexity are relevant to sociotechnical systems:

1. The *technical complexity* of the system is derived from the relationships between the different components of the system itself.
2. The *managerial complexity* of the system is derived from the complexity of the relationships between the system and its managers (i.e., what can managers change in the system) and the relationships between the managers of different parts of the system.

3. The *governance complexity* of a system depends on the relationships between the laws, regulations, and policies that affect the system and the relationships between the decision-making processes in the organizations responsible for the system. As different parts of the system may be in different organizations and in different countries, different laws, rules, and policies may apply to each system within the SoS.

Governance and managerial complexity are related, but they are not the same thing. Managerial complexity is an operational issue—what can and can't actually be done with the system. Governance complexity is associated with the higher level of decision-making processes in organizations that affect the system. These decision-making processes are constrained by national and international laws and regulations.

For example, say a company decides to allow its staff to access its systems using their own mobile devices rather than company-issued laptops. The decision to allow this is a governance decision because it changes the policy of the company. As a result of this decision, management of the system becomes more complex as managers have to ensure that the mobile devices are configured properly so that company data is secure. The technical complexity of the system also increases as there is no longer a single implementation platform. Software may have to be modified to work on laptops, tablets and phones.

As well as technical complexity, the characteristics of systems of systems may also lead to significantly increased managerial and governance complexity. Figure 20.3 summarizes how the different SoS characteristics primarily contribute to different types of complexity:

1. *Operational independence* The constituent systems in the SoS are subject to different policies and rules (governance complexity) and ways of managing the system (managerial complexity).
2. *Managerial independence* The constituent systems in the SoS are managed by different people in different ways. They have to coordinate to ensure that management changes are consistent (managerial complexity). Special software may be needed to support consistent management and evolution (technical complexity).
3. Evolutionary development contributes to the technical complexity of a SoS because different parts of the system are likely to be built using different technologies.
4. Emergence is a consequence of complexity. The more complex a system, the more likely it is that it will have undesirable emergent properties. These properties increase the technical complexity of the system as software has to be developed or changed to compensate for them.
5. Geographical distribution increases the technical, managerial, and governance complexity in a SoS. Technical complexity is increased because software is required to coordinate and synchronize remote systems; managerial complexity is increased because it is more difficult for managers in different countries to coordinate their actions; governance complexity is increased because different

SoS characteristic	Technical complexity	Managerial complexity	Governance complexity
Operational independence		X	X
Managerial independence	X	X	
Evolutionary development	X		
Emergence	X		
Geographical distribution	X	X	X
Data-intensive	X		X
Heterogeneity	X		

Figure 20.3 SoS characteristics and system complexity

parts of the systems may be located in different jurisdictions and so are subject to different laws and regulations.

6. Data-intensive systems are technically complex because of the relationships between the data items. The technical complexity is also likely to be increased to cope with data errors and incompleteness. Governance complexity may be increased because of different laws governing the use of data.

7. The heterogeneity of a system contributes to its technical complexity because of the difficulties of ensuring that different technologies used in different parts of the system are compatible.

Large-scale systems of systems are now unimaginably complex entities that cannot be understood or analyzed as a whole. As I discuss in Section 20.3, the large number of interactions between the parts and the dynamic nature of these interactions means that conventional engineering approaches do not work well for complex systems. It is complexity that is the root cause of problems in projects to develop large software-intensive systems, not poor management or technical failings.

20.2 Systems of systems classification

Earlier, I suggested that the distinguishing feature of a system of systems was that two or more of its elements were independently managed. Different people with different priorities have the authority to take day-to-day operational decisions about changes to the system. As their work is not necessarily aligned, conflicts can arise that require a significant amount of time and effort to resolve. Systems of systems, therefore, always have some degree of managerial complexity.

However, this broad definition of SoS covers a very wide range of system types. It includes systems that are owned by a single organization but are managed by different

parts of that organization. It also includes systems whose constituent systems are owned and managed by different organizations that may, at times, compete with each other. Maier (Maier 1998) devised a classification scheme for SoS based on their governance and management complexity:

1. *Directed systems.* Directed SoS are owned by a single organization and are developed by integrating systems that are also owned by that organization. The system elements may be independently managed by parts of the organization. However, there is an ultimate governing body within the organization that can set priorities for system management. It can resolve disputes between the managers of different elements of the system. Directed systems therefore have some managerial complexity but no governance complexity. A military command-and-control system that integrates information from airborne and ground-based systems is an example of a directed SoS.
2. *Collaborative systems.* Collaborative SoS are systems with no central authority to set management priorities and resolve disputes. Typically, elements of the system are owned and governed by different organizations. However, all of the organizations involved recognize the mutual benefits of joint governance of the system. They therefore usually set up a voluntary governance body that makes decisions about the system. Collaborative systems have both managerial complexity and a limited degree of governance complexity. An integrated public transport information system is an example of a collaborative system of systems. Bus, rail, and air transport providers agree to link their systems to provide passengers with up-to-date information.
3. *Virtual systems.* Virtual systems have no central governance, and the participants may not agree on the overall purpose of the system. Participant systems may enter or leave the SoS. Interoperability is not guaranteed but depends on published interfaces that may change. These systems have a very high degree of both managerial and governance complexity. An example of a virtual SoS is an automated high-speed algorithmic trading system. These systems from different companies automatically buy and sell stock from each other, with trades taking place in fractions of a second.

Unfortunately, I think that the names that Maier has used do not really reflect the distinctions between these different types of systems. As Maier himself says, there is always some collaboration in the management of the system elements. So, "collaborative systems" is not really a good name. The term *directed systems* implies top-down authority. However, even within a single organization, the need to maintain good working relationships between the people involved means that governance is agreed to rather than imposed.

In "virtual" SoS, there may be no formal mechanisms for collaboration, but the system has some mutual benefit for all participants. Therefore, they are likely to collaborate informally to ensure that the system can continue to operate. Furthermore, Maier's use of the term *virtual* could be confusing because "virtual" has now come to mean "implemented by software," as in virtual machines and virtual reality.

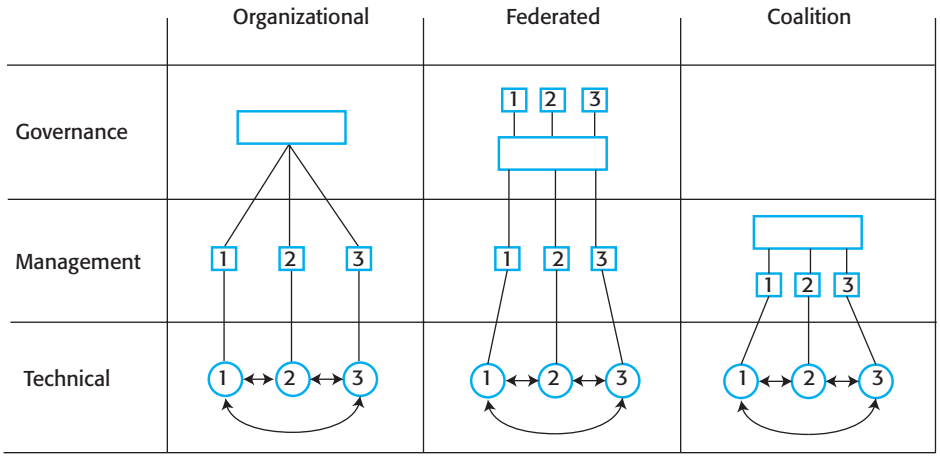


Figure 20.4 SoS collaboration

Figure 20.4 illustrates the collaboration in these different types of system. Rather than use Maier's names, I have used what I hope are more descriptive terms:

1. *Organizational systems of systems* are SoS where the governance and management of the system lies within the same organization or company. These correspond to Maier's "directed SoS." Collaboration between system owners is managed by the organization. The SoS may be geographically distributed, with different parts of the system subject to different national laws and regulations. In Figure 20.4, Systems 1, 2, and 3 are independently managed, but the governance of these systems is centralized.
2. *Federated systems* are SoS where the governance of the SoS depends on a voluntary participative body in which all of the system owners are represented. In Figure 20.4, this is shown by the owners of Systems 1, 2, and 3 participating in a single governance body. The system owners agree to collaborate and believe that decisions made by the governance body are binding. They implement these decisions in their individual management policies, although implementations may differ because of national laws, regulations, and culture.
3. *System of system coalitions* are SoS with no formal governance mechanisms but where the organizations involved informally collaborate and manage their own systems to maintain the system as a whole. For example, if one system provides a data feed to others, the managers of that system will not change the format of the data without notice. Figure 20.4 shows that there is no governance at the organizational level but that informal collaboration exists at the management level.

This governance-based classification scheme provides a means of identifying the governance requirements for a SoS. By classifying a system according to this model, you can check if the appropriate governance structures exist and if these are the ones you really need. Setting up these structures across organizations is a political process and inevitably takes a long time. It is therefore helpful to understand the governance

problem early in the process and take actions to ensure that appropriate governance is in place. It may be the case that you need to adopt a governance model that moves a system from one class to another. Moving the governance model to the left in Figure 20.4 usually reduces complexity.

As I have suggested, the school digital learning environment (iLearn) is a system of systems. As well as the digital learning system itself, it is connected to school administration systems and to network management systems. These network management systems are used for Internet filtering, which stops students from accessing undesirable material on the Internet.

iLearn is a relatively simple technical system, but it has a high level of governance complexity. This complexity arises because of the way that education is funded and managed. In many countries pre-university education is funded and organized at a local level rather than at a national level. States, cities, or counties are responsible for schools in their area and have autonomy in deciding school funding and policies. Each local authority maintains its own school administration system and network management system.

In Scotland, there are 32 local authorities with responsibility for education in their area. School administration is outsourced to one of three providers and iLearn must connect to their systems. However, each local authority has its own network management policies with separate network management systems involved.

The development of a digital learning system is a national initiative, but to create a digital learning environment, it has to be integrated with network management and school administration systems. It is therefore a system of systems with administration and network management systems, as well as the systems within iLearn such as Office 365 and Wordpress. There is no common governance process across authorities, so, according to the classification scheme, this is a coalition of systems. In practice, this means that it cannot be guaranteed that students in different places can access the same tools and content, because of different Internet filtering policies.

When we produced the conceptual model for the system, we made a strong recommendation that common policies should be established across local authorities on administrative information provision and Internet filtering. In essence, we suggested that the system should be a federated system rather than a coalition of systems. This suggestion requires a new governance body to be established to agree on common policies and standards for the system.

20.3 Reductionism and complex systems

I have already suggested that our current software engineering methods and technologies cannot cope with the complexity that is inherent in modern systems of systems. Of course, this idea is not new: Progress in all engineering disciplines has always been driven by challenging and difficult problems. New methods and tools are developed in response to failures and difficulties with existing approaches.

In software engineering, we have seen the incredibly rapid development of the discipline to help manage the increasing size and complexity of software systems. This effort has been very successful indeed. We can now build systems that are orders of magnitude larger and more complex than those of the 1960s and 1970s.

As with other engineering disciplines, the approach that has been the basis of complexity management in software engineering is called *reductionism*. Reductionism is a philosophical position based on the assumptions that any system is made up of parts or subsystems. It assumes that the behavior and properties of the system as a whole can be understood and predicted by understanding the individual parts and the relationships between these parts. Therefore, to design a system, the parts making up that system are identified, constructed separately, and then assembled into the complete system. Systems can be thought of as hierarchies, with the important relationships between parent and child nodes in the hierarchy.

Reductionism has been and continues to be the fundamental underpinning approach to all kinds of engineering. We can identify common abstractions across the same types of system and design and build these separately. They can then be integrated to create the required system. For example, the abstractions in an automobile might be a body shell, a drive train, an engine, a fuel system, and so on. There are a relatively small number of relationships between these abstractions, so it is possible to specify interfaces and design and build each part of the system separately.

The same reductionist approach has been the basis of software engineering for almost 50 years. Top-down design, where you start with a very high-level model of a system and break this down to its components is a reductionist approach. This is the basis of all software design methods, such as object-oriented design. Programming languages include abstractions, such as procedures and objects that directly reflect reductionist system decomposition.

Agile methods, although they may appear quite different from top-down systems design, are also reductionist. They rely on being able to decompose a system into parts, implement these parts separately, and then integrate these to create the system. The only real difference between agile methods and top-down design is that the system is decomposed into components incrementally rather than all at once.

Reductionist methods are most successful when there are relatively few relationships or interactions between the parts of a system and it is possible to model these relationships in a scientific way. This is generally true for mechanical and electrical systems where there are physical linkages between the system components. It is less true for electronic systems and certainly not the case for software systems, where there may be many more static and dynamic relationships between system components.

The distinctions between software and hardware components was recognized in the 1970s. Design methods emphasized the importance of limiting and controlling the relationships between the parts of a system. These methods suggested that components should be tightly integrated with loose coupling between these components. Tight integration meant that most of the relationships were internal to a component, and loose coupling meant that there were relatively few component–component

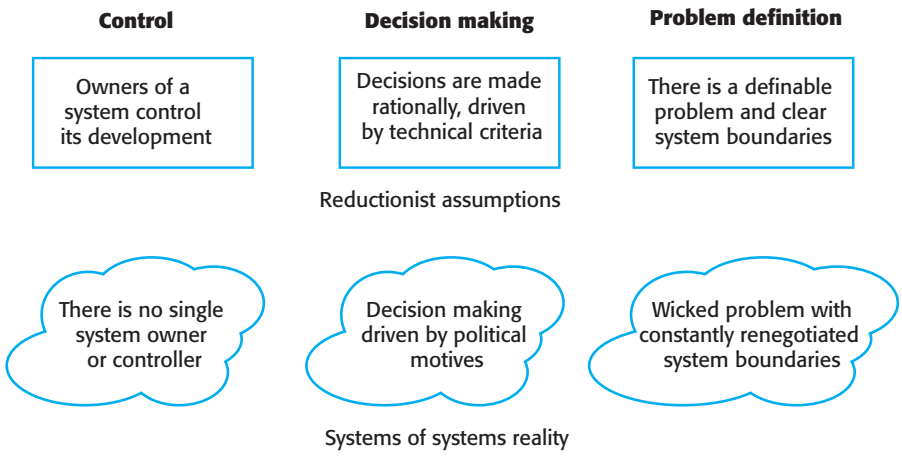


Figure 20.5 Reductionist assumptions and complex system reality

relationships. The need for tight integration (data and operations) and loose coupling was the driver for the development of object-oriented software engineering.

Unfortunately, controlling the number and types of relationship is practically impossible in large systems, especially systems of systems. Reductionism does not work well when there are many relationships in a system and when these relationships are difficult to understand and analyze. Therefore, any type of large system development is likely to run into difficulties.

The reasons for these potential difficulties are that the fundamental assumptions inherent to reductionism are inapplicable for large and complex systems (Sommerville et al. 2012). These assumptions are shown in Figure 20.5 and apply in three areas:

1. *System ownership and control* Reductionism assumes that there is a controlling authority for a system that can resolve disputes and make high-level technical decisions that will apply across the system. As we have seen, because there are multiple bodies involved in their governance, this is simply not true for systems of systems.

2. *Rational decision making* Reductionism assumes that interactions between components can be objectively assessed by, for example, mathematical modeling. These assessments are the driver for system decision making. Therefore, if one particular design of a vehicle, say, offers the best fuel economy without a reduction in power, then a reductionist approach assumes that this will be the design chosen.

3. *Defined system boundaries* Reductionism assumes that the boundaries of a system can be agreed to and defined. This is often straightforward: There may be a physical shell defining the system as in a car, a bridge has to cross a given stretch of water, and so on. Complex systems are often developed to address wicked problems (Rittel and Webber 1973). For such problems, deciding on what is part of the system and what is outside it is usually a subjective judgment, with frequent disagreements between the stakeholders involved.

These reductionist assumptions break down for all complex systems, but when these systems are software-intensive, the difficulties are compounded:

1. Relationships in software systems are not governed by physical laws. We cannot produce mathematical models of software systems that will predict their behavior and attributes. We therefore have no scientific basis for decision making. Political factors are usually the driver of decision making for large and complex software systems.
2. Software has no physical limitations; hence there are no limits on where the boundaries of a system should be drawn. Different stakeholders will argue for the boundaries to be placed in such a way that is best for them. Furthermore, it is much easier to change software requirements than hardware requirements. The boundaries and the scope of a system are likely to change during its development.
3. Linking software systems from different owners is relatively easy; hence we are more likely to try and create a SoS where there is no single governing body. The management and evolution of the different systems involved cannot be completely controlled.

For these reasons, I believe that the problems and difficulties that are commonplace in large software systems engineering are inevitable. Failures of large government projects such as the health automation projects in the UK and the United States are a consequence of complexity rather than technical or project management failures.

Reductionist approaches such as object-oriented development have been very successful in improving our ability to engineer many types of software system. They will continue to be useful and effective in developing small and medium-sized systems whose complexity can be controlled and which may be parts of a software SoS. However, because of the fundamental assumptions underlying reductionism, "improving" these methods will not lead to an improvement in our ability to engineer complex systems of systems. Rather, we need new abstractions, methods, and tools that recognize the technical, human, social, and political complexities of SoS engineering. I believe that these new methods will be probabilistic and statistical and that tools will rely on system simulation to support decision making. Developing these new approaches is a major challenge for software and systems engineering in the 21st century.

20.4 Systems of systems engineering

Systems of systems engineering is the process of integrating existing systems to create new functionality and capabilities. Systems of systems are not designed in a top-down way. Rather, they are created when an organization recognizes that they can add value to existing systems by integrating these into a SoS. For example, a city government might wish to reduce air pollution at particular hot-spots in the city. To do so, it might integrate its traffic management system with a national real-time pollution monitoring systems. This then allows for the traffic management system to alter its strategy to reduce pollution by changing traffic light sequences, speed limits and so on.

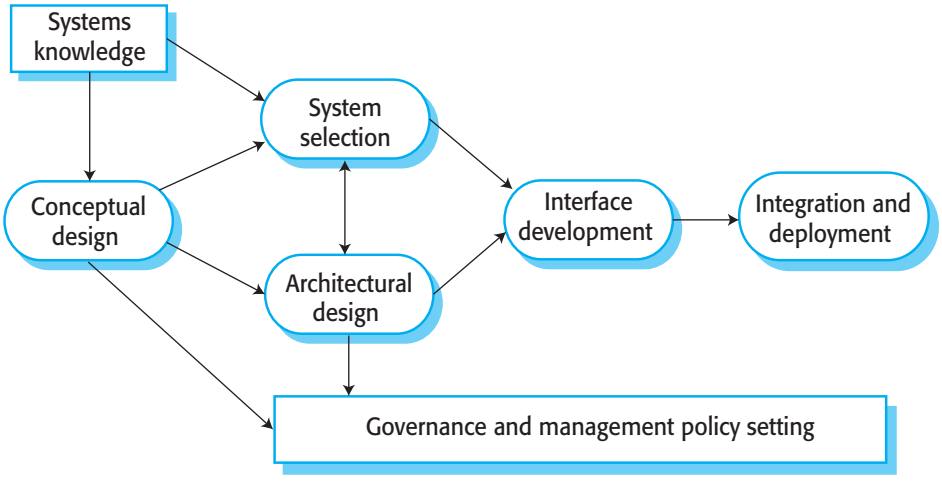


Figure 20.6 An SoS engineering process

The problems of software SoS engineering have much in common with the problems of integrating large-scale application systems that I discussed in Chapter 15 (Boehm and Abts 1999). To recap, these were:

1. Lack of control over system functionality and performance.
2. Differing and incompatible assumptions made by the developers of the different systems.
3. Different evolution strategies and timetables for the different systems.
4. Lack of support from system owners when problems arise.

Much of the effort in building systems of software systems comes from addressing these problems. It involves deciding on the system architecture, developing software interfaces that reconcile differences between the participating systems, and making the system resilient to unforeseen changes that may occur.

Software systems of systems are large and complex entities, and the processes used for their development vary widely depending on the type of systems involved, the application domain, and the needs of the organizations involved in developing the SoS. However, as shown in Figure 20.6, five general activities are involved in SoS development processes:

1. *Conceptual design* I introduced the idea of conceptual design in Chapter 19, which covers systems engineering. Conceptual design is the activity of creating a high-level vision for a system, defining essential requirements, and identifying constraints on the overall system. In SoS engineering, an important input to the conceptual design process is knowledge of the existing systems that may participate in the SoS.
2. *System selection* During this activity, a set of systems for inclusion in the SoS is chosen. This process is comparable to the process of choosing application

systems for reuse, covered in Chapter 15. You need to assess and evaluate existing systems to choose the capabilities that you need. When you are selecting application systems, the selection criteria are largely commercial; that is, which systems offer the most suitable functionality at a price you are prepared to pay?

However, political imperatives and issues of system governance and management are often the key factors that influence what systems are included in a SoS. For example, some systems may be excluded from consideration because an organization does not wish to collaborate with a competitor. In other cases, organizations that are contributing to a federation of systems may have systems in place and insist that these are used, even though they are not necessarily the best systems.

3. *Architectural design* In parallel with system selection, an overall architecture for the SoS has to be developed. Architectural design is a major topic in its own right that I cover in Section 20.5.
4. *Interface development* The different systems involved in a SoS usually have incompatible interfaces. Therefore, a major part of the software engineering effort in developing a SoS is to develop interfaces so that constituent systems can interoperate. This may also involve the development of a unified user interface so that SoS operators do not have to deal with multiple user interfaces as they use the different systems in the SoS.
5. *Integration and deployment* This stage involves making the different systems involved in the SoS work together and interoperate through the developed interfaces. System deployment means putting the system into place in the organizations concerned and making it operational.

In parallel with these technical activities, there needs to be a high-level activity concerned with establishing policies for the governance of the system of systems and defining management guidelines to implement these policies. Where there are several organizations involved, this process can be prolonged and difficult. It may involve organizations changing their own policies and processes. It is therefore important to start governance discussions at an early stage in the SoS development process.

20.4.1 Interface development

The constituent systems in a SoS are usually developed independently for some specific purpose. Their user interface is tailored to that original purpose. These systems may or may not have application programming interfaces (APIs) that allow other systems to interface directly to them. Therefore, when these systems are integrated into a SoS, software interfaces have to be developed, which allows the constituent systems in the SoS to interoperate.

In general, the aim in SoS development is for systems to be able to communicate directly with each other without user intervention. If these systems already offer a service-based interface, as discussed in Chapter 18, then this communication can be implemented using this approach. Interface development involves describing how to

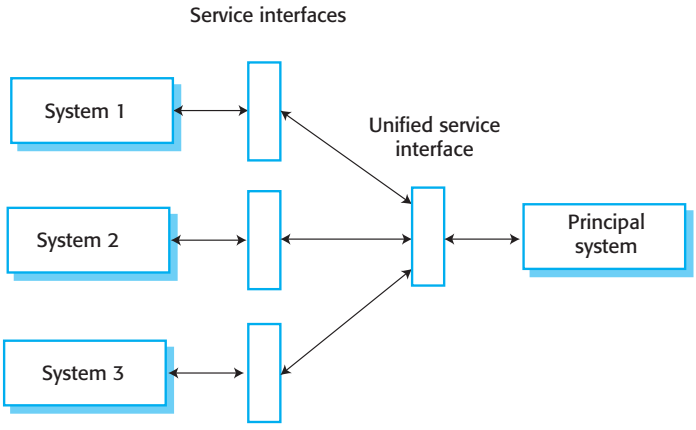


Figure 20.7 Systems with service interfaces

use the interfaces to access the functionality of each system. The systems involved can communicate directly with each other. System coalitions, where all of the systems involved are peers, are likely to use this type of direct interaction as it does not require prearranged agreements on system communication protocols.

More commonly, however, the constituent systems in a SoS either have their own specialized API or only allow their functionality to be accessed through their user interfaces. You therefore have to develop software that reconciles the differences between these interfaces. It is best to implement these interfaces as service-based interfaces, as shown in Figure 20.7 (Sillitto 2010).

To develop service-based interfaces, you have to examine the functionality of existing systems and define a set of services to reflect that functionality. The interface then provides these services. The services are implemented either by calls to the underlying system API or by mimicking user interaction with the system. One of the systems in the SoS is usually a principal or coordinating system that manages the interactions between the constituent systems. The principal system acts as a service broker, directing service calls between the different systems in the SoS. Each system therefore does not need to know which other system is providing a called service.

User interfaces for each system in a SoS are likely to be different. The principal system must have some overall user interfaces that handle user authentication and provide access to the features of the underlying system. However, it is usually expensive and time consuming to implement a unified user interface to replace the individual interfaces of the underlying systems.

A unified user interface (UI) makes it easier for new users to learn to use the SoS and reduces the likelihood of user error. However, whether or not unified UI development is cost-effective depends on a number of factors:

1. *The interaction assumptions of the systems in the SoS* Some systems may have a process-driven model of interaction where the system controls the interface and prompts the user for inputs. Others may give control to the user, so that the user chooses the sequence of interactions with the system. It is practically impossible to unify different interaction models.

2. *The mode of use of the SoS* In many cases, SoS are used in such a way that most of the interactions of users at a site are with one of the constituent systems. They use other systems only when additional information is required. For example, air traffic controllers may normally use a radar system for flight information and only access a flight plan database when additional information is required. A unified interface is a bad idea in these situations because it would slow down interaction with the most commonly used system. However, if the operators interact with all of the constituent systems, then a unified UI may be the best way forward.

3. *The "openness" of the SoS* If the SoS is open, so that new systems may be added to it when it is in use, then unified UI development is impractical. It is impossible to anticipate what the UI of new systems will be. Openness also applies to the organizations using the SoS. If new organizations can become involved, then they may have existing equipment and their own preferences for user interaction. They may therefore prefer not to have a unified UI.

In practice, the limiting factor in UI unification is likely to be the budget and time available for UI development. UI development is one of the most expensive systems engineering activities. In many cases, there is simply not enough project budget available to pay for the creation of a unified SoS user interface.

20.4.2 Integration and deployment

System integration and deployment are usually separate activities. A system is integrated from its components by an integration and testing team, validated, and then released for deployment. The components are managed so that changes are controlled and the integration team can be confident that the required version is included in the system. However, for SoS, such an approach may not be possible. Some of the component systems may already be deployed and in use, and the integration team cannot control changes to these systems.

For SoS, therefore, it makes sense to consider integration and deployment to be part of the same process. This approach reflects one of the design guidelines that I discuss in the following section, which is that an incomplete system of systems should be usable and provide useful functionality. The integration process should begin with systems that are already deployed, with new systems added to the SoS to provide coherent additions to the functionality of the overall system.

It often makes sense to plan the deployment of the SoS to reflect this, so that SoS deployment takes place in a number of stages. For example, Figure 20.8 illustrates a three-stage deployment process for the iLearn digital learning environment:

1. The initial deployment provides authentication, basic learning functionality, and integration with school administration systems.

2. Stage 2 of the deployment adds an integrated storage system and a set of more specialized tools to support subject-specific learning. These tools might include

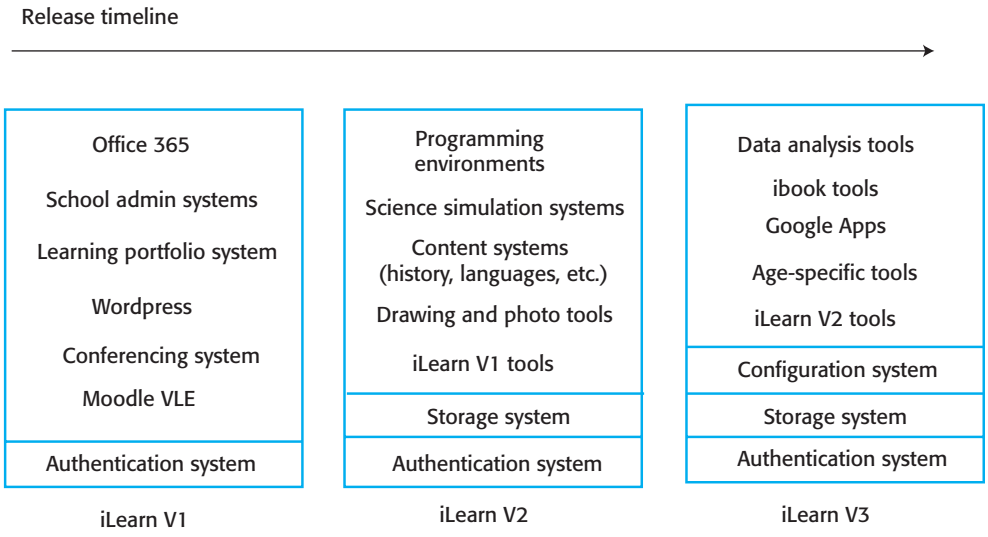


Figure 20.8 Release sequence For the iLearn SoS

archives for history, simulation systems for science, and programming environments for computing.

3. Stage 3 adds features for user configuration and the ability of users to add new systems to the iLearn environment. This stage allows different versions of the system to be created for different age groups, further specialized tools, and alternatives to the standard tools to be included.

As in any large systems engineering project, the most time-consuming and expensive part of system integration is system testing. Testing systems of systems is difficult and expensive for three reasons:

1. There may not be a detailed requirements specification that can be used as a basis for system testing. It may not be cost-effective to develop a SoS requirements document because the details of the system functionality are defined by the systems that are included.
2. The constituent systems may change in the course of the testing process, so tests may not be repeatable.
3. If problems are discovered, it may not be possible to fix the problems by requiring one or more of the constituent systems to be changed. Rather, some intermediate software may have to be introduced to solve the problem.

To help address some of these problems, I believe that SoS testing should take on board some of the testing techniques developed in agile methods:

1. Agile methods do not rely on having a complete system specification for system acceptance testing. Rather, stakeholders are closely engaged with the testing process

and have the authority to decide when the overall system is acceptable. For SoS, a range of stakeholders should be involved in the testing process if possible, and they can comment on whether or not the system is ready for deployment.

2. Agile methods make extensive use of automated testing. This makes it much easier to rerun tests to discover if unexpected system changes have caused problems for the SoS as a whole.

Depending on the type of system, you may have to plan the installation of equipment and user training as part of the deployment process. If the system is being installed in a new environment, equipment installation is straightforward. However, if it is intended to replace an existing system, there may be problems in installing new equipment if it is not compatible with the equipment that is in use. There may not be the physical space for the new equipment to be installed alongside the working system. There may be insufficient electrical power, or users may not have time to be involved because they are busy using the current system. These nontechnical issues can delay the deployment process and slow down the adoption and use of the SoS.

20.5 Systems of systems architecture

Perhaps the most crucial activity of the systems of systems engineering process is architectural design. Architectural design involves selecting the systems to be included in the SoS, assessing how these systems will interoperate, and designing mechanisms that facilitate interaction. Key decisions on data management, redundancy, and communications are made. In essence, the SoS architect is responsible for realizing the vision set out in the conceptual design of the system. For organizational and federated systems, in particular, decisions made at this stage are crucial to the performance, resilience, and maintainability of the system of systems.

Maier (Maier 1998) discusses four general principles for the architecting of complex systems of systems:

1. Design systems so that they can deliver value if they are incomplete. Where a system is composed of several other systems, it should not just be useful if all of its components are working properly. Rather, there should be several "stable intermediate forms" so that a partial system works and can do useful things.

2. Be realistic about what can be controlled. The best performance from a SoS may be achieved when an individual or group exerts control over the overall system and its constituents. If there is no control, then delivering value from the SoS is difficult. However, attempts to overcontrol the SoS are likely to lead to resistance from the individual system owners and consequent delays in system deployment and evolution.

3. Focus on the system interfaces. To build a successful system of systems, you have to design interfaces so that the system elements can interoperate. It is

important that these interfaces are not too restrictive so that the system elements can evolve and continue to be useful participants in the SoS.

4. Provide collaboration incentives. When the system elements are independently owned and managed, it is important each system owner have incentives to continue to participate in the system. These may be financial incentives (pay per use or reduced operational costs), access incentives (you share your data and I'll share mine), or community incentives (participate in a SoS and you get a say in the community).

Sillitto (Sillitto 2010) has added to these principles and suggests additional important design guidelines. These include the following:

1. Design a SoS as node and web architecture. Nodes are sociotechnical systems that include data, software, hardware, infrastructure (technical components), and organizational policies, people, processes, and training (sociotechnical). The web is not just the communications infrastructure between nodes, but it also provides a mechanism for informal and formal social communications between the people managing and running the systems at each node.
2. Specify behavior as services exchanged between nodes. The development of service-oriented architectures now provides a standard mechanism for system operability. If a system does not already provide a service interface, then this interface should be implemented as part of the SoS development process.
3. Understand and manage system vulnerabilities. In any SoS, there will be unexpected failures and undesirable behavior. It is critically important to try to understand vulnerabilities and design the system to be resilient to such failures.

The key message that emerges from both Maier's and Sillitto's work is that SoS architects have to take a broad perspective. They need to look at the system as a whole, taking into account both technical and sociotechnical considerations. Sometimes the best solution to a problem is not more software but changes to the rules and policies that govern the operation of the system.

Architectural frameworks such as MODAF (MOD 2008) and TOGAF (TOGAF is a registered trademark of The Open Group 2011) have been suggested as a means of supporting the architectural design of systems of systems. Architectural frameworks were originally developed to support enterprise systems architectures, which are portfolios of separate systems. Enterprise systems may be organizational systems of systems, or they may have a simpler management structure so that the system portfolio can be managed as a whole. Architectural frameworks are intended for the development of organizational systems of systems where there is a single governance authority for the entire SoS.

An architectural framework recognizes that a single model of an architecture does not present all of the information needed for architectural and business analysis. Rather, frameworks propose a number of architectural views that should be created and maintained to describe and document enterprise systems. Frameworks have much in common and tend to reflect the language and history of the organizations

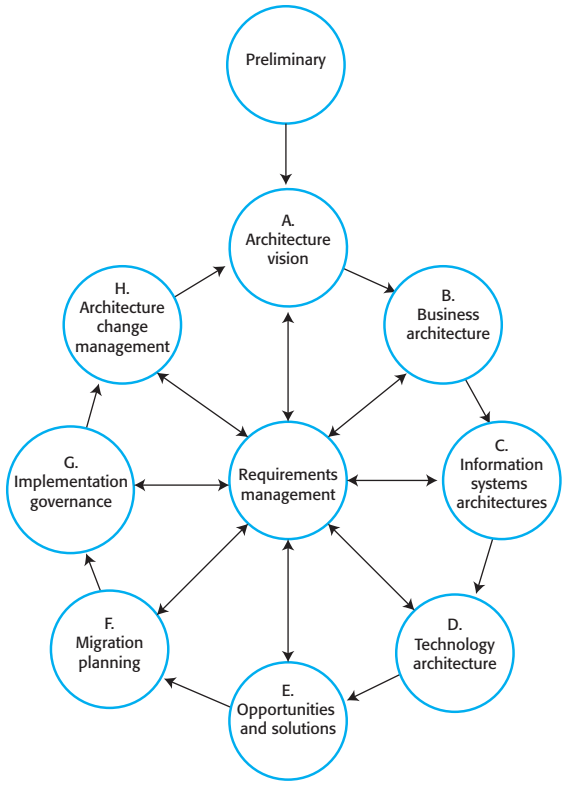


Figure 20.9 The TOGAF architecture development method (TOGAF® Version 9.1, © 1999–2011. The Open Group.)

involved. For example, MODAF and DODAF are comparable frameworks from the UK Ministry of Defence (MOD) and the U.S. Department of Defense (DOD).

The TOGAF framework has been developed by the Open Group as an open standard and is intended to support the design of a business architecture, a data architecture, an application architecture, and a technology architecture for an enterprise. At its heart is the Architecture Development Method (ADM), which consists of a number of discrete phases. These are shown in Figure 20.9, taken from the TOGAF reference documentation (Open Group 2011).

All architectural frameworks involve the production and management of a large set of architectural models. Each of the activities shown in Figure 20.8 leads to the production of system models. However, this is problematic for two reasons:

1. Initial model development takes a long time and involves extensive negotiations between system stakeholders. This slows the development of the overall system.
2. It is time-consuming and expensive to maintain model consistency as changes are made to the organization and the constituent systems in a SoS.

Architecture frameworks are fundamentally reductionist, and they largely ignore sociotechnical and political issues. While they do recognize that problems are difficult to define and are open-ended, they assume a degree of control and governance

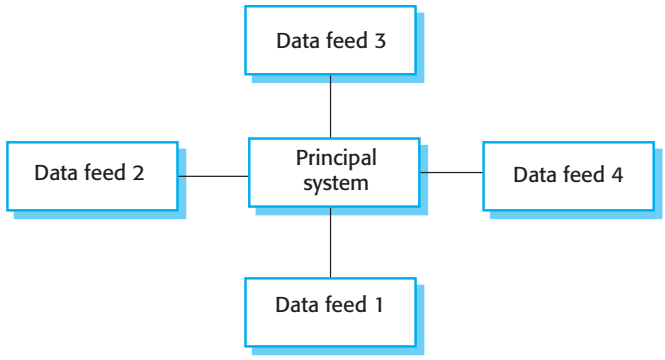


Figure 20.10 Systems as data feeds

that is impossible to achieve in many systems of systems. They are a useful checklist to remind architects of things to think about in the architectural design process. However, I think that the overhead involved in model management and the reductionist approach taken by frameworks limits their usefulness in SoS architectural design.

20.5.1 Architectural patterns for systems of systems

I have described architectural patterns for different types of system in Chapters 6, 17, and 21. In short, an architectural pattern is a stylized architecture that can be recognized across a range of different systems. Architectural patterns are a useful way of stimulating discussions about the most appropriate architecture for a system and for documenting and explaining the architectures used. This section covers a number of "typical" patterns in systems of software systems. As with all architectural patterns, real systems are usually based on more than one of these patterns.

The notion of architectural patterns for systems of systems is still at an early stage of development. Kawalsky (Kawalsky et al. 2013) discusses the value of architectural patterns in understanding and supporting SoS design, with a focus on patterns for command and control systems. I find that patterns are effective in illustrating SoS organization, without the need for detailed domain knowledge.

Systems as data-feeds

In this architectural pattern (Figure 20.10), there is a principal system that requires data of different types. This data is available from other systems, and the principal system queries these systems to get the data required. Generally, the systems that provide data do not interact with each other. This pattern is often observed in organizational or federated systems where some governance mechanisms are in place.

For example, to license a vehicle in the UK, you need to have both valid insurance and a roadworthiness certificate. When you interact with the vehicle licensing system, it interacts with two other systems to check that these documents are valid. These systems are:

1. An *"insured vehicles" system*, which is a federated system run by car insurance companies that maintains information about all current car insurance policies.

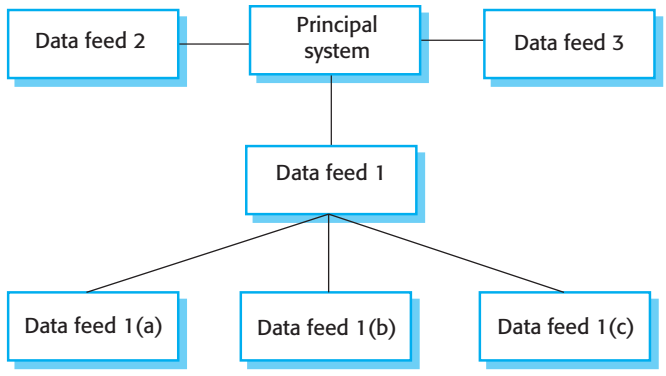


Figure 20.11 Systems as data feeds with a unifying interface

2. An "*MOT certificate" system*, which is used to record all roadworthiness certificates issued by testing agencies licensed by the government.

The "systems as data feeds" architecture is an appropriate architecture to use when it is possible to identify entities in a unique way and create relatively simple queries about these entities. In the licensing system, vehicles can be uniquely identified by their registration number. In other systems, it may be possible to identify entities such as pollution monitors by their GPS coordinates.

A variant of the "systems as data feeds" architecture arises when a number of systems provide data that are similar but not identical. Therefore, the architecture has to include an intermediate layer as shown in Figure 20.11. The role of this intermediate layer is to translate the general query from the principal system into the specific query required by the individual information system.

For example, the iLearn environment interacts with school administration systems from three different providers. All of these systems provide the same information about students (names, personal information, etc.) but have different interfaces. The databases have different organizations, and the format of the data returned differs from one system to another. The unifying interface here detects where the user of the system is based and, using this regional information, knows which administrative system should be accessed. It then converts a standard query into the appropriate query for that system.

Problems that can arise in systems that use this pattern are primarily interface problems when the data feeds are unavailable or are slow to respond. It is important to ensure that timeouts are included in the system so that a failure of a data feed does not compromise the response time of the system as a whole. Governance mechanisms should be in place to ensure that the format of provided data is not changed without the agreement of all system owners.

Systems in a container

Systems in a container are systems of systems where one of the systems acts as a virtual container and provides a set of common services such as an authentication and a storage service. Conceptually, other systems are then placed into this container

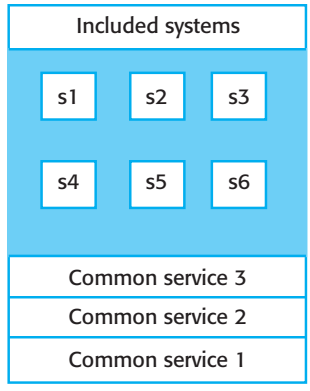


Figure 20.12 Systems in a container

to make their functionality accessible to system users. Figure 20.12 illustrates a container system with three common services and six included systems. The systems that are included may be selected from an approved list of systems and need not be aware that they are included in the container. This pattern of SoS is most often observed in federated systems or system coalitions.

The iLearn environment is a system in a container. There are common services that support authentication, storage of user data, and system configuration. Other functionality comes from choosing existing systems such as a newspaper archive or a virtual learning environment and integrating these into the container.

Of course, you don't place systems into a real container to implement these systems of systems. Rather, for each approved system, there is a separate interface that allows it to be integrated with the common services. This interface manages the translation of the common services provided by the container and the requirements of the integrated system. It may also be possible to include systems that are not approved. However, these will not have access to the common services provided by the container.

Figure 20.13 illustrates this integration. This graphic is a simplified version of iLearn that provides three common services:

1. An authentication service that provides a single sign-in to all approved systems. Users do not have to maintain separate credentials for these systems.
2. A storage service for user data. This service can be seamlessly transferred to and from approved systems.
3. A configuration service that is used to include or remove systems from the container.

This example shows a version of iLearn for Physics. As well as an office productivity system (Office 365) and a VLE (Moodle), this system includes simulation and data analysis systems. Other systems—YouTube and a science encyclopedia—are also part of this system. However, these are not "approved," and so no container interface is available. Users must log on to these systems separately and organize their own data transfers.

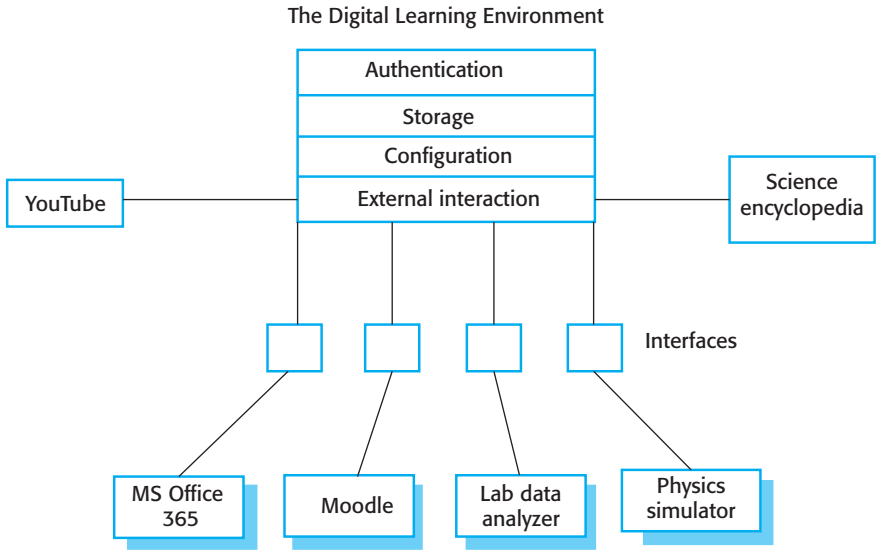


Figure 20.13 The DLE as a container system

There are two problems with this type of SoS architecture:

1. A separate interface must be developed for each approved system so that common services can be used with these systems. This means that only a relatively small number of approved systems can be supported.
2. The owners of the container system have no influence on the functionality and behavior of the included systems. Systems may stop working, or they may be withdrawn at any time.

However, the main benefit of this architecture is that it allows for incremental development. An early version of the container system can be based on "unapproved" systems. Interfaces to these can be developed in later versions so that they are more closely integrated with the container services.

Trading systems

Trading systems are systems of systems where there is no single principal system but processing may take place in any of the constituent systems. The systems involved trade information among themselves. There may be one-to-one or one-to-many interactions between these systems. Each system publishes its own interface, but there may not be any interface standards that are followed by all systems. This system is shown in Figure 20.14. Trading systems may be federated systems or system coalitions.

An example of a trading SoS is a system of systems for algorithmic trading of stocks and shares. Brokers all have their own separate systems that can automatically buy and sell stock from other systems. They set prices and negotiate individually with these systems. Another example of a trading system is a travel aggregator that shows price comparisons and allows travel to be booked directly by a user.

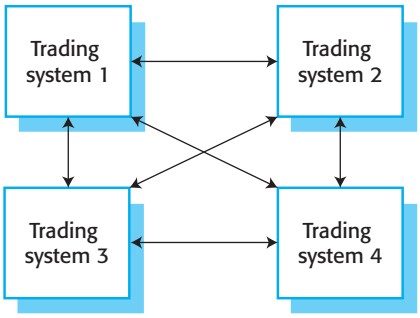


Figure 20.14 A trading system of systems

Trading systems may be developed for any type of marketplace, with the information exchanged being information about the goods being traded and their prices. Although trading systems are systems in their own right and could conceivably be used for individual trading, they are most useful in an automated trading context where the systems negotiate directly with each other.

The major problem with this type of system is that there is no governance mechanism, so any of the systems involved may change at any time. Because these changes may contradict the assumptions made by other systems, trading cannot continue. Sometimes the owners of the systems in the coalition wish to be able to continue trading with other systems and so may make informal arrangements to ensure that changes to one system do not make trading impossible. In other cases, such as a travel aggregator, an airline may deliberately change its system so that it is unavailable and so force bookings to be made directly with it.

KEY POINTS

- Systems of systems are systems where two or more of the constituent systems are independently managed and governed.
- Three types of complexity are important for systems of systems—technical complexity, managerial complexity, and governance complexity.
- System governance can be used as the basis for a classification scheme for SoS. This leads to three classes of SoS, namely, organizational systems, federated systems, and system coalitions.
- Reductionism as an engineering method breaks down because of the inherent complexity of systems of systems. Reductionism assumes clear system boundaries, rational decision making, and well-defined problems. None of these are true for systems of systems.
- The key stages of the SoS development process are conceptual design, system selection, architectural design, interface development, and integration and deployment. Governance and management policies must be designed in parallel with these activities.

- Architectural patterns for systems of systems are a means of describing and discussing typical architectures for SoS. Important patterns are systems as data feeds, systems in a container, and trading systems.

FURTHER READING

"Architecting Principles for Systems of Systems." A now-classic paper on systems of systems that introduces a classification scheme for SoS, discusses its value, and proposes a number of architectural principles for SoS design. (M. Maier, *Systems Engineering*, 1 (4), 1998).

Ultra-large Scale Systems: The Software Challenge of the Future This book, produced for the U.S. Department of Defense in 2006, introduces the notion of ultra-large-scale systems, which are systems of systems with hundreds of nodes. It discusses the issues and challenges in developing such systems. (L. Northrop et al., Software Engineering Institute, 2006). http://www.sei.cmu.edu/library/assets/ULS_Book20062.pdf

"Large-scale Complex IT Systems." This paper discusses the problems of large-scale complex IT systems that are systems of systems and expands on the ideas here on the breakdown of reductionism. It proposes a number of research challenges in the area of SoS. (I. Sommerville et al., Communications of the ACM, 55 (7), July 2012). http://dx.doi.org/ 10.1145/2209249.2209268

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/systems-engineering/

EXERCISES

20.1. Explain why managerial and operational independence are the key distinguishing characteristics of systems of systems when compared to other large, complex systems.

20.2. Briefly explain any four essential characteristics of systems of systems.

20.3. The classification of SoS presented in Section 20.2 suggests a governance-based classification scheme. Giving reasons for your answer, identify the classifications for the following systems of systems:

(a) A health care system that provides unified access to all patient health records from hospitals, clinics, and primary care.
(b) The World Wide Web
(c) A government system that provides access to a range of welfare services such as pensions, disability benefits, and unemployment benefits.

Are there any problems with the suggested classification for any of these systems?

20.4. Explain what is meant by reductionism and why it is effective as a basis for many kinds of engineering.

20.5. Define systems of systems engineering. List the problems of software SoS engineering that are also common to problems of integrating large-scale application systems.

20.6. How beneficial is a unified user interface in the interface design of SoS? What are the factors on which the cost-effectiveness of a unified user interface is dependent?

20.7. Sillitto suggests that communications between nodes in a SoS are not just technical but should also include informal sociotechnical communications between the people involved in the system. Using the iLearn SoS as an example, suggest where these informal communications may be important to improve the effectiveness of the system.

20.8. Suggest the closest-fit architectural pattern for the systems of systems introduced in Exercise 20.3.

20.9. The trading system pattern assumes that there is no central authority involved. However, in areas such as equity trading, trading systems must follow regulatory rules. Suggest how this pattern might be modified to allow a regulator to check that these rules have been followed. This should not involve all trades going through a central node.

20.10. You work for a software company that has developed a system that provides information about consumers and that is used within a SoS by a number of other retail businesses. They pay you for the services used. Discuss the ethics of changing the system interfaces without notice to coerce users into paying higher charges. Consider this question from the point of view of the company's employees, customers, and shareholders.

REFERENCES

Boehm, B., and C. Abts. 1999. "COTS Integration: Plug and Pray?" *Computer* 32 (1): 135–138. doi:10.1109/2.738311.

Hitchins, D. 2009. "System of Systems—The Ultimate Tautology." http://www.hitchins.net/profs-stuff/profs-blog/system-of-systems---the.html

Kawalsky, R., D. Joannou, Y. Tian, and A. Fayoumi. 2013. “Using Architecture Patterns to Architect and Analyze Systems of Systems.” In *Conference on Systems Engineering Research (CSER 13)*, 283–292. doi:10.1016/j.procs.2013.01.030.

Maier, M. W. 1998. “Architecting Principles for Systems-of-Systems.” *Systems Engineering* 1 (4): 267–284. doi:10.1002/(SICI)1520–6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D.

MOD, UK. 2008. “MOD Architecture Framework.” https://www.gov.uk/mod-architecture-framework

Northrop, Linda, R. P. Gabriel, M. Klein, and D. Schmidt. 2006. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Pittsburgh: Software Engineering Institute. http://www.sei.cmu.edu/library/assets/ULS_Book20062.pdf

Open Group. 2011. “Open Group Standard TOGAF Version 9.1.” http://pubs.opengroup.org/architecture/togaf91-doc/arch/

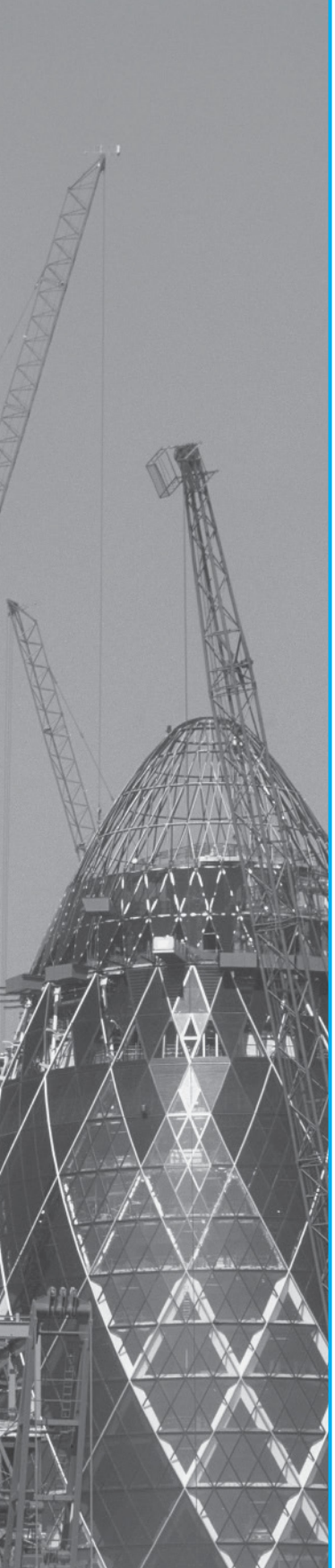
Rittel, H., and M. Webber. 1973. “Dilemmas in a General Theory of Planning.” *Policy Sciences* 4: 155–169. doi:10.1007/BF01405730.

Royal Academy of Engineering. 2004. “Challenges of Complex IT Projects.” London. http://www.bcs.org/upload/pdf/complexity.pdf

Sillitto, H. 2010. “Design Principles for Ultra-Large-Scale Systems.” In *Proceedings of the 20th International Council for Systems Engineering International Symposium*. Chicago.

Sommerville, I., D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Kwiatkowska, J. McDermid, and R. Paige. 2012. “Large-Scale Complex IT Systems.” *Comm. ACM* 55 (7): 71–77. doi:10.1145/2209249.2209268.

Stevens, R. 2010. *Engineering Mega-Systems: The Challenge of Systems Engineering in the Information Age*. Boca Raton, FL: CRC Press.



21

Real-time software engineering

Objectives

The objective of this chapter is to introduce some of the characteristic features of embedded real-time software engineering. When you have read this chapter, you will:

- understand the concept of embedded software, which is used to control systems that react to external events in their environment;
- have been introduced to a design process for real-time systems, where the software systems are organized as a set of cooperating processes;
- understand three architectural patterns that are commonly used in embedded real-time systems design;
- understand the organization of real-time operating systems and the role that they play in an embedded, real-time system.

Contents

Computers are used to control a wide range of systems from simple domestic machines, through games controllers, to entire manufacturing plants. These computers interact directly with hardware devices. Their software must react to events generated by the hardware and often issue control signals in response to these events. These signals result in an action, such as the initiation of a phone call, the movement of a character on the screen, the opening of a valve, or the display of the system status. The software in these systems is embedded in system hardware, often in read-only memory. It responds, in real time, to events from the system's environment. By real time, I mean that the software system has a deadline for responding to external events. If this deadline is missed, then the overall hardware–software system will not operate correctly.

Embedded software is very important economically because almost every electrical device now includes software. There are therefore many more embedded software systems than other types of software systems. Ebert and Jones (Ebert and Jones 2009) estimated that there were about 30 embedded microprocessor systems per person in developed countries. This figure was increasing between 10% and 20% per year. This suggests that, by 2020, there will be more than 100 embedded systems per person.

Responsiveness in real time is the critical difference between embedded systems and other software systems, such as information systems, web-based systems, or personal software systems, whose main purpose is data processing. For non–real-time systems, the correctness of a system can be defined by specifying how system inputs map to corresponding outputs that should be produced by the system. In response to an input, a corresponding output should be generated by the system and, often, some data should be stored. For example, if you choose a create command in a patient information system, then the correct system response is to create a new patient record in a database and to confirm that this has been done. Within reasonable limits, it does not matter how long this takes.

However, in a real-time system, the correctness depends both on the response to an input and the time taken to generate that response. If the system takes too long to respond, then the required response may be ineffective. For example, if embedded software controlling a car's braking system is too slow, then an accident may occur because it is impossible to stop the car in time.

Therefore, time is fundamental in the definition of a real-time software system:

> *A real-time software system is a system whose correct operation depends on both the results produced by the system and the time at which these results are produced. A "soft real-time system" is a system whose operation is degraded if results are not produced according to the specified timing requirements. If results are not produced according to the timing specification in a "hard real-time system," this is considered to be a system failure.*

Timely response is an important factor in all embedded systems, but not all embedded systems require a very fast response. For example, the insulin pump software that I have used as an example in several chapters of this book is an embedded system. However, while the system needs to check the glucose level at periodic intervals, it does not need to

respond very quickly to external events. The wilderness weather station software is also an embedded system, but, again, it does not require a fast response to external events.

As well as the need for real-time response, there are other important differences between embedded systems and other types of software system:

1. Embedded systems generally run continuously and do not terminate. They start when the hardware is switched on, and execute until the hardware is switched off. Techniques for reliable software engineering, as discussed in Chapter 11, may therefore have to be used to ensure continuous operation. The real-time system may include update mechanisms that support dynamic reconfiguration so that the system can be updated while it is in service.
2. Interactions with the system's environment are unpredictable. In interactive systems, the pace of the interaction is controlled by the system. By limiting user options, the events and commands to be processed are known in advance. By contrast, real-time embedded systems must be able to respond to expected and unexpected events at any time. This leads to a design for real-time systems based on concurrency, with several processes executing in parallel.
3. Physical limitations may affect the design of a system. Examples of limitations include restrictions on the power available to the system and the physical space taken up by the hardware. These limitations may generate requirements for the embedded software, such as the need to conserve power and so prolong battery life. Size and weight limitations may mean that the software has to take over some hardware functions because of the need to limit the number of chips used in the system.
4. Direct hardware interaction may be necessary. In interactive systems and information systems, a layer of software (the device drivers) hides the hardware from the operating system. This is possible because you can only connect a few types of device to these systems, such as keyboards, mice, and displays. By contrast, embedded systems may have to interact with a wide range of hardware devices that do not have separate device drivers.
5. Issues of safety and reliability may dominate the system design. Many embedded systems control devices whose failure may have high human or economic costs. Therefore, dependability is critical, and the system design has to ensure safety-critical behavior at all times. This often leads to a conservative approach to design where tried and tested techniques are used instead of newer techniques that may introduce new failure modes.

Real-time embedded systems can be thought of as reactive systems; that is, they must react to events in their environment (Berry 1989; Lee 2002). Response times are often governed by the laws of physics rather than chosen for human convenience. This is in contrast to other types of software where the system controls the speed of the interaction. For example, the word processor that I am using to write this book can check spelling and grammar, and there are no practical limits on the time taken to do so.

21.1 Embedded system design

During the design process for embedded software, software designers have to consider in detail the design and performance of the system hardware. Part of the system design process may involve deciding which system capabilities are to be implemented in software and which in hardware. For many real-time systems that are embedded in consumer products, such as the systems in cell phones, the costs and power consumption of the hardware are critical. Specific processors designed to support embedded systems may be used. For some systems, special-purpose hardware may have to be designed and built.

A top-down software design process, in which the design starts with an abstract model that is decomposed and developed in a series of stages, is impractical for most real-time systems. Low-level decisions on hardware, support software, and system timing must be considered early in the process. These limit the flexibility of system designers. Additional software functionality, such as battery and power management, may have to be included in the system.

Given that embedded systems are reactive systems that react to events in their environment, the most general approach to embedded, real-time software design is based on a stimulus-response model. A stimulus is an event occurring in the software system's environment that causes the system to react in some way; a response is a signal or message that the software sends to its environment.

You can define the behavior of a real-time system by listing the stimuli received by the system, the associated responses, and the time at which the response must be produced. For example, Figure 21.1 shows possible stimuli and system responses for a burglar alarm system (discussed in Section 21.2.1).

Stimuli fall into two classes:

1. *Periodic stimuli* These occur at predictable time intervals. For example, the system may examine a sensor every 50 milliseconds and take action (respond) depending on that sensor value (the stimulus).
2. *Aperiodic stimuli* These occur irregularly and unpredictably and are usually signaled, using the computer's interrupt mechanism. An example of such a stimulus would be an interrupt indicating that an I/O transfer was complete and that data was available in a buffer.

Stimuli come from sensors in the system's environment, and responses are sent to actuators, as shown in Figure 21.2. These actuators control equipment, such as a pump, which then makes changes to the system's environment. The actuators themselves may also generate stimuli. The stimuli from actuators often indicate that some problem with the actuator has occurred, which must be handled by the system.

A general design guideline for real-time systems is to have separate control processes for each type of sensor and actuator (Figure 21.3). For each type of sensor, there may be a sensor management process that handles data collection from these sensors. Data-processing processes compute the required responses for the stimuli received by the system. Actuator control processes are associated with each actuator

Stimulus	Response
Clear alarms	Switch off all active alarms; switch off all lights that have been switched on.
Console panic button positive	Initiate alarm; turn on lights around console; call police.
Power supply failure	Call service technician.
Sensor failure	Call service technician.
Single sensor positive	Initiate alarm; turn on lights around site of positive sensor.
Two or more sensors positive	Initiate alarm; turn on lights around sites of positive sensors; call police with location of suspected break-in.
Voltage drop of between 10% and 20%	Switch to battery backup; run power supply test.
Voltage drop of more than 20%	Switch to battery backup; initiate alarm; call police, run power supply test.

Figure 21.1 Stimuli and responses for a burglar alarm system

and manage the operation of that actuator. This model allows data to be collected quickly from the sensor (before it is overwritten by the next input) and enables processing and the associated actuator response to be carried out later.

A real-time system has to respond to stimuli that occur at different times. You therefore have to organize the system architecture so that, as soon as a stimulus is received, control is transferred to the correct handler. This is impractical in sequential programs. Consequently, real-time software systems are normally designed as a set of concurrent, cooperating processes. To support the management of these processes, the execution platform on which the real-time system executes may include a real-time operating system (discussed in Section 21.4). The functions provided by this operating system are accessed through the runtime support system for the real-time programming language that is used.

There is no standard embedded system design process. Rather, different processes are used that depend on the type of system, available hardware, and the organization that is developing the system. The following activities may be included in a real-time software design process:

1. *Platform selection* In this activity, you choose an execution platform for the system, that is, the hardware and the real-time operating system to be used. Factors that influence these choices include the timing constraints on the system, limitations on power available, the experience of the development team, and the price target for the delivered system.
2. *Stimuli/response identification* This involves identifying the stimuli that the system must process and the associated response or responses for each stimulus.

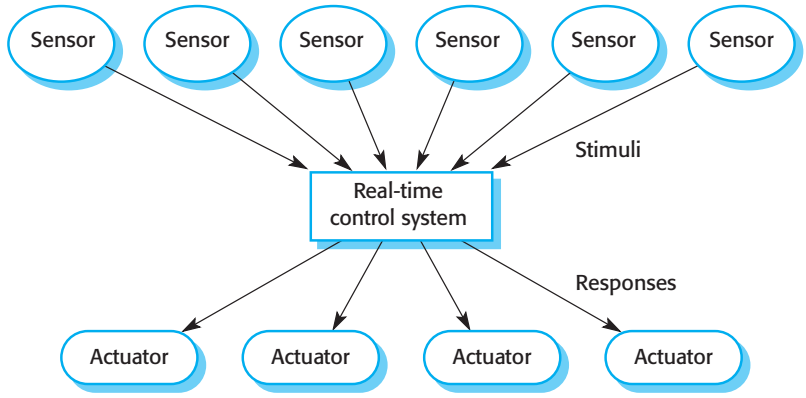


Figure 21.2 A general model of an embedded real-time system

3. *Timing analysis* For each stimulus and associated response, you identify the timing constraints that apply to both stimulus and response processing. These constraints are used to establish the deadlines for the processes in the system.
4. *Process design* Process design involves aggregating the stimulus and response processing into a number of concurrent processes. A good starting point for designing the process architecture is the architectural patterns that I describe in Section 20.2. You then optimize the process architecture to reflect the specific requirements that you have to implement.
5. *Algorithm design* For each stimulus and response, you design algorithms to carry out the required computations. Algorithm designs may have to be developed relatively early in the design process to indicate the amount of processing required and the time needed to complete that processing. This is especially important for computationally intensive tasks, such as signal processing.
6. *Data design* You specify the information that is exchanged by processes and the events that coordinate information exchange, and design data structures to manage this information exchange. Several concurrent processes may share these data structures.
7. *Process scheduling* You design a scheduling system that will ensure that processes are started in time to meet their deadlines.

The specific activities and the activity sequence in a real-time system design process depend on the type of system being developed, its novelty, and its environment.

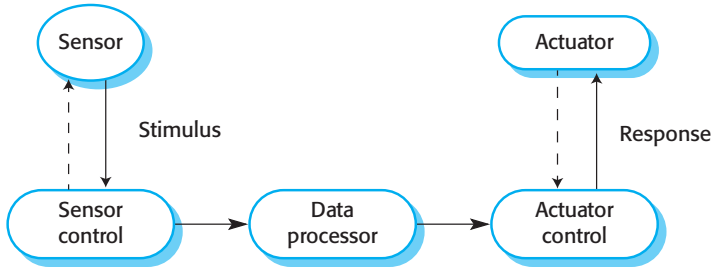


Figure 21.3 Sensor and actuator processes

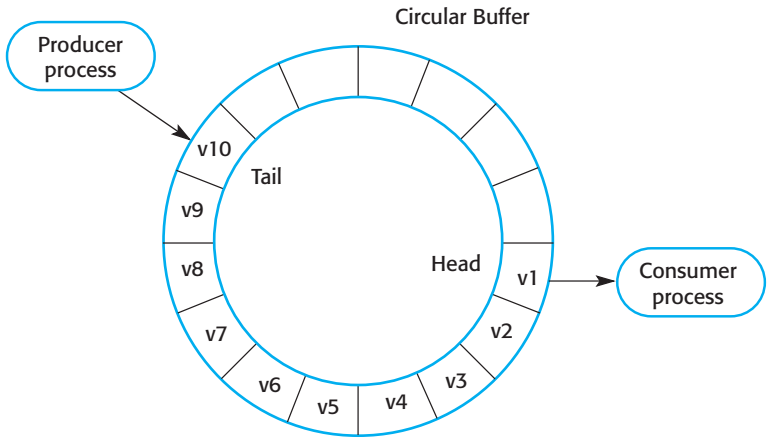


Figure 21.4 Producer/consumer processes sharing a circular buffer

In some cases, for new systems, you may be able to follow a fairly abstract approach where you start with the stimuli and associated processing, and decide on the hardware and execution platforms late in the process. In other cases, the choice of hardware and operating system is made before the software design starts. You then have to design the software to take account of the constraints imposed by the system hardware.

Processes in a real-time system have to be coordinated and share information. Process coordination mechanisms ensure mutual exclusion to shared resources. When one process is modifying a shared resource, other processes should not be able to change that resource. Mechanisms for ensuring mutual exclusion include semaphores, monitors, and critical regions. These process synchronization mechanisms are described in most operating system books (Silberschaltz, Galvin, and Gagne 2013; Stallings 2014).

When designing the information exchange between processes, you have to take into account that these processes may be running at different speeds. One process is producing information, and the other process is consuming that information. If the producer is running faster than the consumer, new information could overwrite a previously read information item before the consumer process has read the original information. If the consumer process is running faster than the producer process, the same item could be read twice.

To avoid this problem, you should implement information exchange using a shared buffer and use mutual exclusion mechanisms to control access to that buffer. This means that information can't be overwritten before it has been read and that information cannot be read twice. Figure 21.4 illustrates the organization of a shared buffer. This is usually implemented as a circular queue, using a list data structure. Mismatches in speed between the producer and consumer processes can be accommodated without having to delay process execution.

The producer process always enters data in the buffer location at the end of the queue (represented as v10 in Figure 21.4). The consumer process always retrieves information from the head of the queue (represented as v1 in Figure 21.4). After the consumer process has retrieved the information, the tail of the queue is adjusted to point at the next item (v2). After the producer process has added information, the tail of the queue is adjusted to point at the next free slot in the queue.

Obviously, it is important to ensure that the producer and consumer process do not attempt to access the same item at the same time (i.e., when Head = Tail). If they do, the value of the item is unpredictable. The system also has to ensure that the producer process does not add items to a full buffer and that the consumer process does not try to take items from an empty buffer.

To do this, you implement the circular buffer as a process with Get and Put operations to access the buffer. The Put operation is called by the producer process and the Get operation by the consumer process. Synchronization primitives, such as semaphores or critical regions, are used to ensure that the operation of Get and Put are synchronized, so that they don't access the same location simultaneously. If the buffer is full, the Put process has to wait until a slot is free; if the buffer is empty, the Get process has to wait until an entry has been made.

Once you have chosen the execution platform for the system, designed a process architecture, and decided on a scheduling policy, you have to check that the system will meet its timing requirements. You can perform this check through static analysis of the system using knowledge of the timing behavior of components, or through simulation. This analysis may reveal that the system will not perform adequately. The process architecture, the scheduling policy, the execution platform, or all of these may then have to be redesigned to improve the performance of the system.

Timing constraints or other requirements may sometimes mean that it is best to implement some system functions, such as signal processing, in hardware. Modern hardware components, such as FPGAs (field-programmable gate arrays), are flexible and can be adapted to different functions. Hardware components deliver much better performance than the equivalent software. System processing bottlenecks can be identified and replaced by hardware, thus avoiding expensive software optimization.

21.1.1 Real-time system modeling

The events that a real-time system must react to often cause the system to move from one state to another. For this reason, state models, which I introduced in Chapter 5, are used to describe real-time systems. A state model of a system assumes that, at any time, the system is in one of a number of possible states. When a stimulus is received, this may cause a transition to a different state. For example, a system controlling a valve may move from a state "Valve open" to a state "Valve closed" when an operator command (the stimulus) is received.

State models are an integral part of real-time system design methods. The UML supports the development of state models based on Statecharts (Harel 1987, 1988). Statecharts are formal state machine models that support hierarchical states, so that groups of states can be considered as a single entity. Douglass discusses the use of the UML in real-time systems development (Douglass 1999).

I have already illustrated this approach to system modeling in Chapter 5 where I used an example of a model of a simple microwave oven. Figure 21.5 is another example of a state model that shows the operation of a fuel delivery software system embedded in a petrol (gas) pump. The rounded rectangles represent system states, and the arrows represent stimuli that force a transition from one state to another.

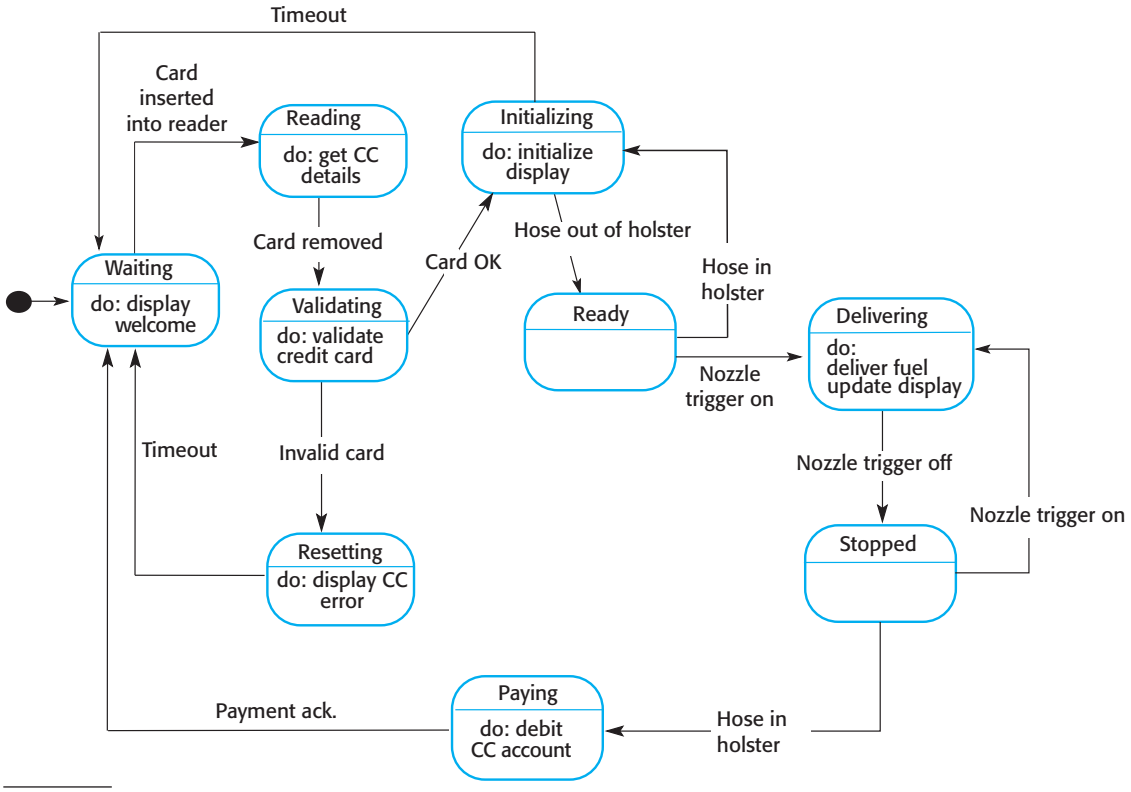


Figure 21.5 State machine model of a petrol (gas) pump

The names chosen in the state machine diagram are descriptive. The associated information indicates actions taken by the system actuators or information that is displayed. Notice that this system never terminates but idles in a waiting state when the pump is not operating.

The fuel delivery system is designed to allow unattended operation, with the following sequence of actions:

1. The buyer inserts a credit card into a card reader built into the pump. This causes a transition to a Reading state where the card details are read and the buyer is then asked to remove the card.
2. Removal of the card triggers a transition to a Validating state where the card is validated.
3. If the card is valid, the system initializes the pump and, when the fuel hose is removed from its holster, transitions to the Delivering state, where is ready to deliver fuel. Activating the trigger on the nozzle causes fuel to be pumped; this stops when the trigger is released (for simplicity, I have ignored the pressure switch that is designed to stop fuel spillage).

Real-time Java

The Java programming language has been modified to make it suitable for real-time systems development. These modifications include asynchronous communications, the addition of time, including absolute and relative time, a new thread model where threads cannot be interrupted by garbage collection, and a new memory management model that avoids the unpredictable delays that can result from garbage collection.

http://software-engineering-book.com/web/real-time-java/

4. After the fuel delivery is complete and the buyer has replaced the hose in its holster, the system moves to a Paying state where the user's account is debited.
5. After payment, the pump software returns to the Waiting state.

State models are used in model-driven engineering, which I discussed in Chapter 5, to define the operation of a system. They can be transformed automatically or semiautomatically to an executable program.

21.1.2 Real-time programming

Programming languages for real-time systems development have to include facilities to access system hardware, and it should be possible to predict the timing of particular operations in these languages. Hard real-time systems, running on limited hardware, are still sometimes programmed in assembly language so that tight deadlines can be met. Systems programming languages, such as C, which allow efficient code to be generated, are widely used.

The advantage of using a systems programming language like C is that it allows the development of efficient programs. However, these languages do not include constructs to support concurrency or the management of shared resources. Concurrency and resource management are implemented through calls to primitives provided by the real-time operating system for mutual exclusion. Because the compiler cannot check these calls, programming errors are more likely. Programs are also often more difficult to understand because the language does not include real-time features. As well as understanding the program, the reader also has to know how real-time support is provided using system calls.

Because real-time systems must meet their timing constraints, you may not be able to use object-oriented development for hard real-time systems. Object-oriented development involves hiding data representations and accessing attribute values through operations defined with the object. There is a significant performance overhead in object-oriented systems because extra code is required to mediate access to attributes and handle calls to operations. The consequent loss of performance may make it impossible to meet real-time deadlines.

A version of Java has been developed for embedded systems development (Burns and Wellings 2009; Bruno and Bollella 2009). This language includes a modified thread mechanism, which allows threads to be specified that will not be interrupted

by the language garbage collection mechanism. Asynchronous event handling and timing specification has also been included. However, at the time of writing, this specification has mostly been used on platforms that have significant processor and memory capacity (e.g., a cell phone) rather than simpler embedded systems, with more limited resources. These systems are still usually implemented in C.

21.2 Architectural patterns for real-time software

Architectural patterns are abstract, stylized descriptions of good design practice. They capture knowledge about the organization of system architectures, when these architectures should be used, and their advantages and disadvantages. You use an architectural pattern to understand an architecture and as starting point for creating your own, specific architectural design.

The difference between real-time and interactive software means that there are distinct architectural patterns for real-time embedded systems. Real-time systems' patterns are process-oriented rather than object- or component-oriented. In this section, I discuss three real-time architectural patterns that are commonly used:

1. *Observe and React* This pattern is used when a set of sensors are routinely monitored and displayed. When the sensors show that some event has occurred (e.g., an incoming call on a cell phone), the system reacts by initiating a process to handle that event.
2. *Environmental Control* This pattern is used when a system includes sensors, which provide information about the environment and actuators that can change the environment. In response to environmental changes detected by the sensor, control signals are sent to the system actuators.
3. *Process Pipeline* This pattern is used when data has to be transformed from one representation to another before it can be processed. The transformation is implemented as a sequence of processing steps, which may be carried out concurrently. This allows for very fast data processing, because a separate core or processor can execute each transformation.

These patterns can of course be combined, and you will often see more than one of them in a single system. For example, when the Environmental Control pattern is used, it is very common for the actuators to be monitored using the Observe and React pattern. In the event of an actuator failure, the system may react by displaying a warning message, shutting down the actuator, switching in a backup system, and so forth.

The patterns that I cover are architectural patterns that describe the overall structure of an embedded system. Douglass (Douglass 2002) describes lower-level, real-time design patterns that support more detailed design decision making. These patterns include design patterns for execution control, communications, resource allocation, and safety and reliability.

Name	Observe and React
Description	The input values of a set of sensors of the same types are collected and analyzed. These values are displayed in some way. If the sensor values indicate that some exceptional condition has arisen, then actions are initiated to draw the operator's attention to that value and, if necessary, take actions in response to the exceptional value.
Stimuli	Values from sensors attached to the system.
Responses	Outputs to display, alarm triggers, signals to reacting systems.
Processes	Observer, Analysis, Display, Alarm, Reactor.
Used in	Monitoring systems, alarm systems.

Figure 21.6 The Observe and React pattern

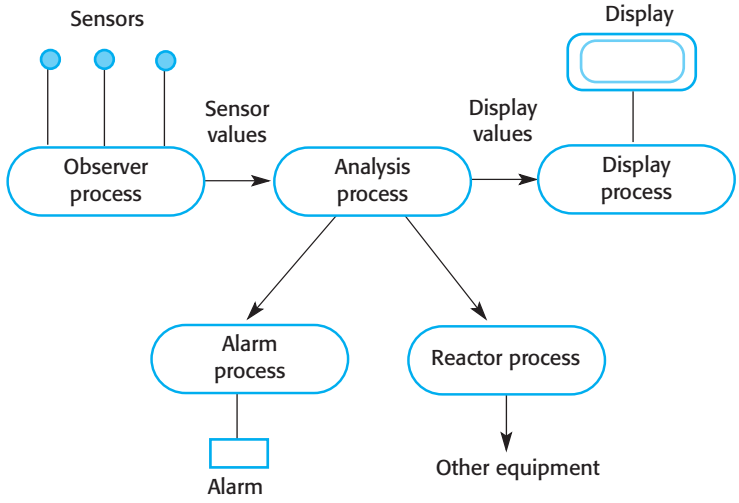


Figure 21.7 The Observe and React process structure

These architectural patterns should be the starting point for an embedded systems design; however, they are not design templates. If you use them as such, you will probably end up with an inefficient process architecture. You have to optimize the process structure to ensure that you do not have too many processes. You also should ensure that there is a clear correspondence between the processes and the sensors and actuators in the system.

21.2.1 Observe and react

Monitoring systems are an important class of embedded real-time systems. A monitoring system examines its environment through a set of sensors and usually displays the state of the environment in some way. This could be on a built-in screen, on special-purpose instrument displays, or on a remote display. If the system detects some exceptional event or sensor state, the monitoring system takes some action.

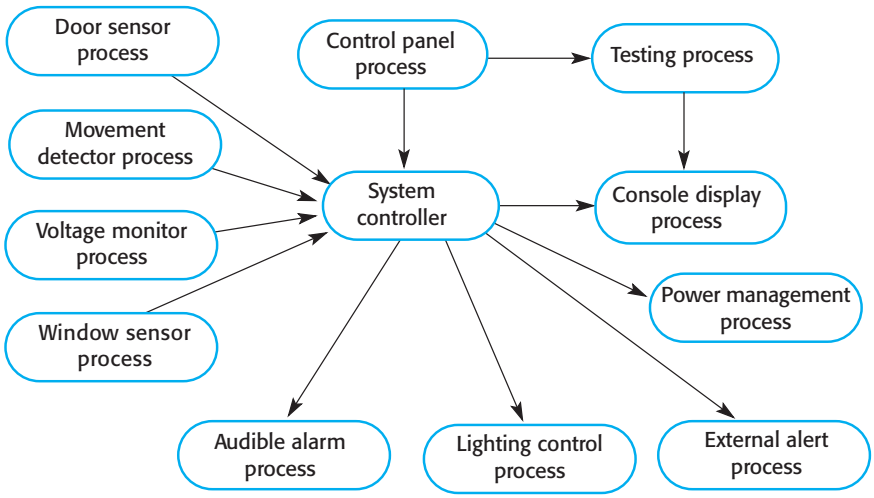


Figure 21.8 The process structure of a burglar alarm system

This often involves raising an alarm to draw an operator's attention to the event. Sometimes the system may initiate some other preventative action, such as shutting down the system to preserve it from damage.

The Observe and React pattern (Figures 21.6 and 21.7) is commonly used in monitoring systems. The values of sensors are observed, and the system initiates actions that depend on these sensor values. Monitoring systems may be composed of several instantiations of the Observe and React pattern, one for each type of sensor in the system. Depending on the system requirements, you may then optimize the design by combining processes (e.g., you may use a single display process to display the information from all of the different types of sensor).

As an example of the use of this pattern, consider the design of a burglar alarm system to be installed in an office building:

> *A software system is to be implemented as part of a burglar alarm system for commercial buildings. This uses several different types of sensors. These sensors include movement detectors in individual rooms, door sensors that detect corridor doors opening, and window sensors on ground-floor windows that can detect when a window has been opened.*
>
> *When a sensor detects the presence of an intruder, the system automatically calls the local police and, using a voice synthesizer, reports the location of the alarm. It switches on lights in the rooms around the active sensor and sets off an audible alarm. The sensor system is normally powered by mains power but is equipped with a battery backup. Power loss is detected using a separate power circuit monitor that monitors the mains voltage. If a voltage drop is detected, the system assumes that intruders have interrupted the power supply, so an alarm is raised.*

A process architecture for the alarm system is shown in Figure 21.8. The arrows represent signals sent from one process to another. This system is a "soft" real-time system that does not have stringent timing requirements. The sensors only need to detect

Name	Environmental Control
Description	The system analyzes information from a set of sensors that collect data from the system's environment. Further information may also be collected on the state of the actuators that are connected to the system. Based on the data from the sensors and actuators, control signals are sent to the actuators, which then cause changes to the system's environment. Information about the sensor values and the state of the actuators may be displayed.
Stimuli	Values from sensors attached to the system and the state of the system actuators.
Responses	Control signals to actuators display information.
Processes	Monitor, Control, Display, Actuator driver, Actuator monitor.
Used in	Control systems.

Figure 21.9 The Environmental Control pattern

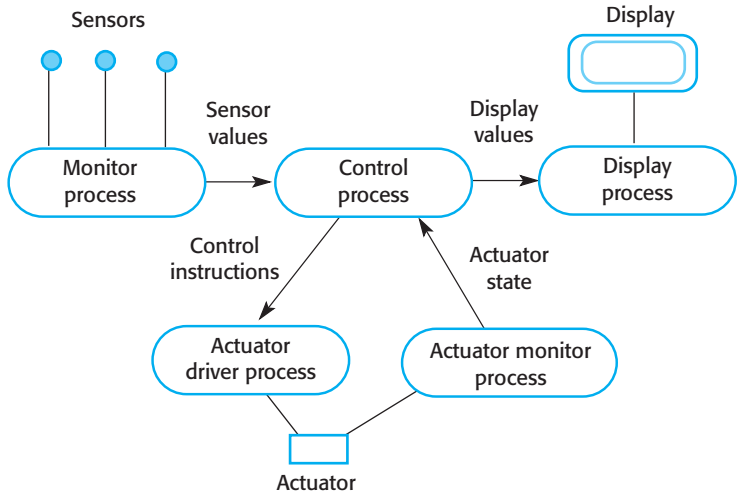


Figure 21.10 The Environmental Control process structure

the presence of people rather than high-speed events, so they only need to be polled 2 or 3 times per second. I cover the timing requirements for this system in Section 21.3.

I have already introduced the stimuli and responses in this alarm system in Figure 21.1. These responses are used as a starting point for the system design. The Observe and React pattern is used in this design. There are observer processes associated with each type of sensor and reactor processes for each type of reaction. A single analysis process checks the data from all of the sensors. The display processes in the pattern are combined into a single display process.

21.2.2 Environmental Control

The most widespread use of real-time embedded software is in control systems. In these systems, the software controls the operation of equipment, based on stimuli

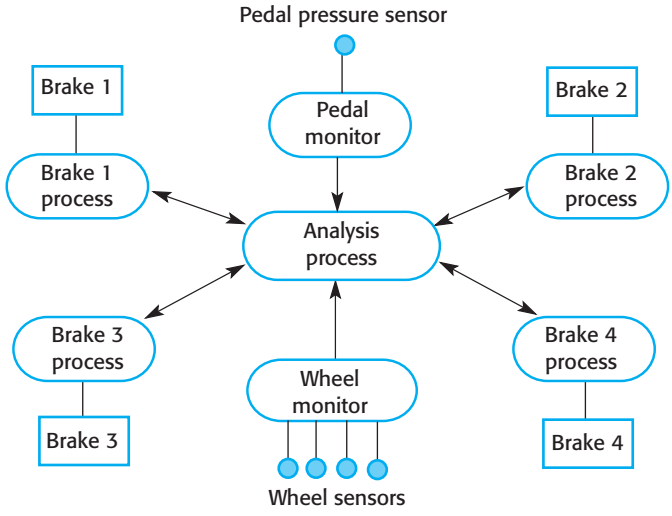


Figure 21.11 Control system architecture for an anti-skid braking system

from the equipment's environment. For example, an anti-skid braking system in a car monitors the car's wheels and brake system (the system's environment). It looks for signs that the wheels are skidding when brake pressure is applied. If this is the case, the system adjusts the brake pressure to stop the wheels locking and reduce the likelihood of a skid.

Control systems may make use of the Environmental Control pattern, which is a general control pattern that includes sensor and actuator processes. This pattern is described in Figure 21.9, with the process architecture shown in Figure 21.10. A variant of this pattern leaves out the display process. This variant is used in situations where user intervention is not required or where the rate of control is so high that a display would not be meaningful.

This pattern can be the basis for a control system design with an instantiation of the Environmental Control pattern for each actuator (or actuator type) being controlled. You then optimize the design to reduce the number of processes. For example, you may combine actuator monitoring and actuator control processes, or you may have a single monitoring and control process for several actuators. The optimizations that you choose depend on the timing requirements. You may need to monitor sensors more frequently than you send control signals, in which case it may be impractical to combine control and monitoring processes. There may also be direct feedback between the actuator control and the actuator monitoring process. This allows fine-grain control decisions to be made by the actuator control process.

You can see how this pattern is used in Figure 21.11, which shows an example of a controller for a car braking system. The starting point for the design is associating an instance of the pattern with each actuator type in the system. In this case, there are four actuators, with each controlling the brake on one wheel. The individual sensor processes are combined into a single wheel-monitoring process that monitors the sensors on all

Name	Process Pipeline
Description	A pipeline of processes is set up with data moving in sequence from one end of the pipeline to another. The processes are often linked by synchronized buffers to allow the producer and consumer processes to run at different speeds. The culmination of a pipeline may be display or data storage, or the pipeline may terminate in an actuator.
Stimuli	Input values from the environment or some other process
Responses	Output values to the environment or a shared buffer
Processes	Producer, Buffer, Consumer
Used in	Data acquisition systems, multi-media systems

Figure 21.12 The Process Pipeline pattern

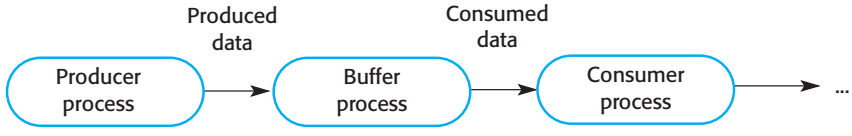


Figure 21.13 Process Pipeline process structure

wheels. This monitors the state of each wheel to check if the wheel is turning or locked. A separate process monitors the pressure on the brake pedal exerted by the car driver.

The system includes an anti-skid feature, which is triggered if the sensors indicate that a wheel is locked when the brake has been applied. This means that there is insufficient friction between the road and the tire; in other words, the car is skidding. If the wheel is locked, the driver cannot steer that wheel. To counteract this effect, the system sends a rapid sequence of on/off signals to the brake on that wheel, which allows the wheel to turn and control to be regained.

The Wheel monitor process monitors whether or not each wheel is turning. If a wheel is skidding (not turning), it informs the Analysis process. This then signals the processes associated with the wheels that are skidding to initiate anti-skid braking.

21.2.3 Process pipeline

Many real-time systems are concerned with collecting analog data from the system's environment. They then digitize that data for analysis and processing by the system. The system may also convert digital data to analog data, which it then sends to its environment. For example, a software radio accepts incoming packets of digital data representing the radio transmission and transforms the data into a sound signal that people can listen to.

The data processing involved in many of these systems has to be carried out very quickly. Otherwise, incoming data may be lost and outgoing signals may be broken up because essential information is missing. The Process Pipeline pattern makes this rapid processing possible by breaking down the required data processing into a sequence of separate transformations. Each of these transformations is implemented

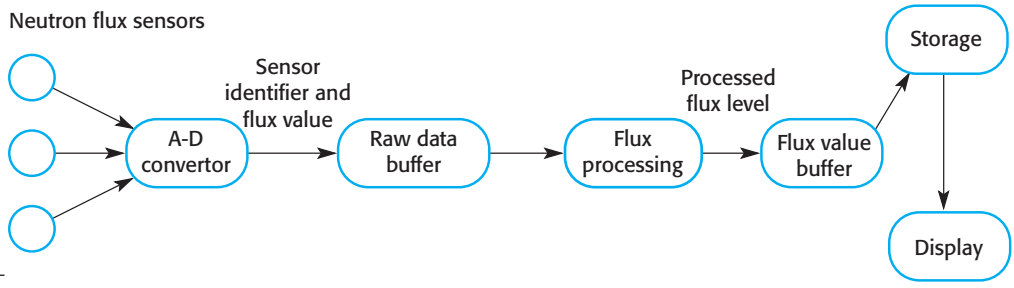


Figure 21.14 Neutron flux data acquisition

by an independent process. This architecture is efficient for systems that use multiple processors or multicore processors. Each process in the pipeline can be associated with a separate processor or core, so that the processing steps can be carried out in parallel.

Figure 21.12 is a brief description of the data pipeline pattern, and Figure 21.13 shows the process architecture for this pattern. Notice that the processes involved produce and consume information. The processes exchange information using synchronized buffers, as I explained in Section 21.1. Producer and consumer processes can thereby operate at different speeds without data losses.

An example of a system that may use a process pipeline is a high-speed data acquisition system. Data acquisition systems collect data from sensors for subsequent processing and analysis. These systems are used in situations where the sensors are collecting large volumes of data from the system's environment and it isn't possible or necessary to process that data in real time. Rather, it is collected and stored for later analysis. Data acquisition systems are often used in scientific experiments and process control systems where physical processes, such as chemical reactions, are very rapid. In these systems, the sensors may be generating data very quickly, and the data acquisition system has to ensure that a sensor reading is collected before the sensor value changes.

Figure 21.14 is a simplified model of a data acquisition system that might be part of the control software in a nuclear reactor. This system collects data from sensors monitoring the neutron flux (the density of neutrons) in the reactor. The sensor data is placed in a buffer from which it is extracted and processed. The average flux level is displayed on an operator's display and stored for future processing.

21.3 Timing analysis

As I discussed in the introduction to this chapter, the correctness of a real-time system depends not just on the correctness of its outputs but also on the time at which these outputs were produced. Therefore, timing analysis is an important activity in the embedded, real-time software development process. In such an analysis, you calculate how often each process in the system must be executed to ensure that all inputs

are processed and all system responses are produced in a timely way. The results of the timing analysis are used to decide how frequently each process should execute and how these processes should be scheduled by the real-time operating system.

Timing analysis for real-time systems is particularly difficult when the system has to deal with a mixture of periodic and aperiodic stimuli and responses. Because aperiodic stimuli are unpredictable, you have to make assumptions about the probability of these stimuli occurring and therefore requiring service at any particular time. These assumptions may be incorrect, and system performance after delivery may not be adequate. Cooling's book (Cooling 2003) discusses techniques for real-time system performance analysis that takes aperiodic events into account.

As computers have become faster, it has become possible in many systems to design using only periodic stimuli. When processors were slow, aperiodic stimuli had to be used to ensure that critical events were processed before their deadline, as delays in processing usually involved some loss to the system. For example, the failure of a power supply in an embedded system may mean that the system has to shut down attached equipment in a controlled way, within a very short time (say 50 milliseconds). This could be implemented as a "power fail" interrupt. However, it can also be implemented using a periodic process that runs frequently and checks the power. As long as the time between process invocations is short, there is still time to perform a controlled shutdown of the system before the lack of power causes damage. For this reason, I only discuss timing issues for periodic processes.

When you are analyzing the timing requirements of embedded real-time systems and designing systems to meet these requirements, you have to consider three key factors:

1. *Deadlines* The times by which stimuli must be processed and some response produced by the system. If the system does not meet a deadline, then, if it is a hard real-time system, this is a system failure; in a soft real-time system, it results in degraded system service.
2. *Frequency* The number of times per second that a process must execute so that you are confident that it can always meet its deadlines.
3. *Execution time* The time required to process a stimulus and produce a response. Execution time is not always the same because of the conditional execution of code, delays waiting for other processes, and so on. Therefore, you may have to consider both the average execution time of a process and the worst-case execution time for that process. The worst-case execution time is the maximum time that the process takes to execute. In a hard real-time system, you may have to make assumptions based on the worst-case execution time to ensure that deadlines are not missed. In soft real-time systems, you can base your calculations on the average execution time.

To continue the example of a power supply failure, let's calculate the worst-case execution time for a process that switches equipment power from mains

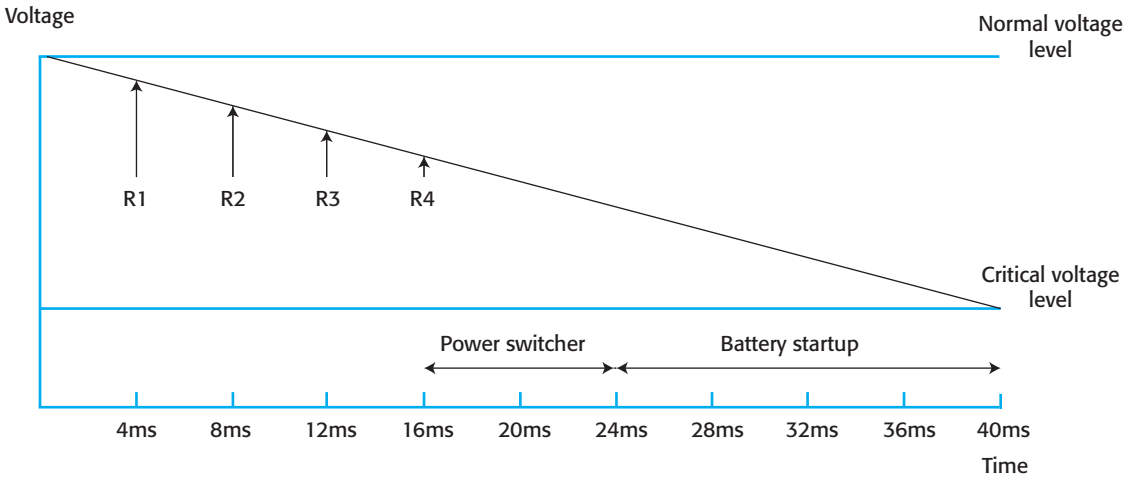


Figure 21.15
Power failure timing analysis

power to a battery backup. Figure 21.15 presents a timeline showing the events in the system:

1. Assume that, after a mains power failure event, it takes 50 milliseconds (ms) for the supplied voltage to drop to a level where the equipment may be damaged. The battery backup must therefore be activated and in operation within 50 ms. Usually, you allow for a margin of error, so you should set a shorter deadline of 40 ms because of physical variations in the equipment. This means that all equipment must be running on the battery backup power supply within 40 ms.
2. However, the battery backup system cannot be instantaneously activated. It takes 16 ms from starting the backup power supply to the supply being fully operational. This means that the time available to detect the power failure and start the battery backup system is 24 ms.
3. There is a process that is scheduled to run 250 times per second, that is, every 4 ms. This process assumes that there is a power supply problem if a significant drop in voltage occurs between readings and is sustained for three readings. This time is allowed so that temporary fluctuations do not cause a switch to the battery backup system.
4. In the above timeline, the power fails immediately after a reading has been taken. Therefore, reading R1 is the start reading for the power fail check. The voltage continues to drop for readings R2–R4, so a power failure is assumed. This is the worst possible case, where a power failure event occurs immediately after a sensor check, so 16 ms have elapsed since that event.
5. At this stage, the process that switches to the battery backup is started. Because the battery backup takes 16 ms to become operational, the worst-case execution time for this process is 8 ms, so that the 40 ms deadline can be reached.

Stimulus/Response	Timing requirements
Audible alarm	The audible alarm should be switched on within half a second of an alarm being raised by a sensor.
Communications	The call to the police should be started within 2 seconds of an alarm being raised by a sensor.
Door alarm	Each door alarm should be polled twice per second.
Lights switch	The lights should be switched on within half a second of an alarm being raised by a sensor.
Movement detector	Each movement detector should be polled twice per second.
Power failure	The switch to backup power must be completed within a deadline of 50 ms.
Voice synthesizer	A synthesized message should be available within 2 seconds of an alarm being raised by a sensor.
Window alarm	Each window alarm should be polled twice per second.

Figure 21.16 Timing requirements for the burglar alarm system

The starting point for timing analysis in a real-time system is the timing requirements, which should set out the deadlines for each required response in the system. Figure 21.16 shows possible timing requirements for the office building burglar alarm system discussed in Section 21.2.1. To simplify this example, let us ignore stimuli generated by system testing procedures and external signals to reset the system in the event of a false alarm. This means there are only two types of stimulus processed by the system:

1. Power failure is detected by observing a voltage drop of more than 20%. The required response is to switch the circuit to backup power by signaling an electronic power-switching device that switches the mains power to battery backup.
2. Intruder alarm is a stimulus generated by one of the system sensors. The response to this stimulus is to compute the room number of the active sensor, set up a call to the police, initiate the voice synthesizer to manage the call, and switch on the audible intruder alarm and building lights in the area.

As shown in Figure 21.16, you should list the timing constraints for each class of sensor separately, even when (as in this case) they are the same. By considering them separately, you leave scope for future change and make it easier to compute the number of times the controlling process has to be executed each second.

Allocating the system functions to concurrent processes is the next design stage. Four types of sensors must be polled periodically, each with an associated process: the voltage sensor, door sensors, window sensors, and movement detectors. Normally, the processes associated with the sensor will execute very quickly as all

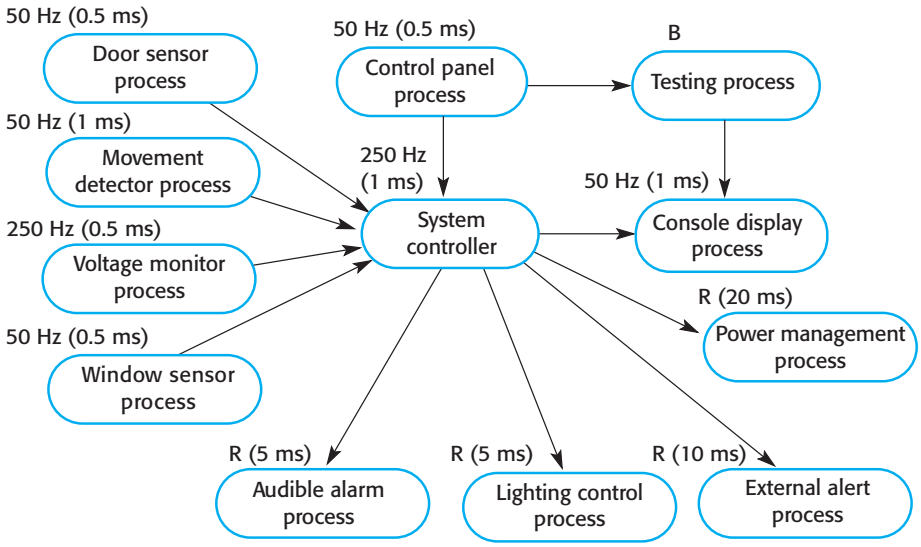


Figure 21.17
Alarm process timing

they are doing is checking whether or not a sensor has changed its status (e.g., from off to on). It is reasonable to assume that the execution time to check and assess the state of one sensor is less than 1 millisecond.

To ensure that you meet the deadlines defined by the timing requirements, you then have to decide how frequently the related processes have to run and how many sensors should be examined during each execution of the process. There are obvious trade-offs here between frequency and execution time:

1. The deadline for detecting a change of state is 0.25 second, which means that each sensor has to be checked 4 times per second. If you examine one sensor during each process execution, then if there are N sensors of a particular type, you must schedule the process 4N times per second to ensure that all sensors are checked within the deadline.
2. If you examine four sensors, say, during each process execution, then the execution time is increased to about 4 ms, but you need only run the process N times/second to meet the timing requirement.

In this case, because the system requirements define actions when two or more sensors are positive, the best strategy is to examine sensors in groups, with groups based on the physical proximity of the sensors. If an intruder has entered the building, then it will probably be adjacent sensors that are positive.

When you have completed the timing analysis, you may then annotate the process model with information about frequency of execution and their expected execution time (see Figure 21.17). Here, periodic processes are annotated with their frequency, processes that are started in response to a stimulus are annotated with R, and the testing process is a background process, annotated with B. This background process

only runs when processor time is available. In general, it is simpler to design a system so that there are a small number of process frequencies. The execution times represent the required worst-case execution times of the processes.

The final step in the design process is to design a scheduling system that will ensure that a process will always be scheduled to meet its deadlines. You can only do this if you know the scheduling approaches that are supported by the real-time operating system (OS) used (Burns and Wellings 2009). The scheduler in the real-time OS allocates a process to a processor for a given amount of time. The time can be fixed, or it may vary depending on the priority of the process.

In allocating process priorities, you have to consider the deadlines of each process so that processes with short deadlines receive processor time to meet these deadlines. For example, the voltage monitor process in the burglar alarm needs to be scheduled so that voltage drops can be detected and a switch made to backup power before the system fails. This should therefore have a higher priority than the processes that check sensor values, as these have fairly relaxed deadlines compared to their expected execution time.

21.4 Real-time operating systems

The execution platform for most application systems is an operating system that manages shared resources and provides features such as a file system and runtime process management. However, the extensive functionality in a conventional operating system takes up a great deal of space and slows down the operation of programs. Furthermore, the process management features in the system may not be designed to allow fine-grain control over the scheduling of processes.

For these reasons, standard operating systems, such as Linux and Windows, are not normally used as the execution platform for real-time systems. Very simple embedded systems may be implemented as "bare metal" systems. The systems provide their own execution support and so include system startup and shutdown, process and resource management, and process scheduling. More commonly, however, embedded applications are built on top of a real-time operating system (RTOS), which is an efficient operating system that offers the features needed by real-time systems. Examples of RTOS are Windows Embedded Compact, VxWorks, and RTLinux.

A real-time operating system manages processes and resource allocation for a real-time system. It starts and stops processes so that stimuli can be handled, and it allocates memory and processor resources. The components of an RTOS (Figure 21.18) depend on the size and complexity of the real-time system being developed. For all except the simplest systems, they usually include:

1. A real-time clock, which provides the information required to schedule processes periodically.
2. If interrupts are supported, an interrupt handler, which manages aperiodic requests for service.

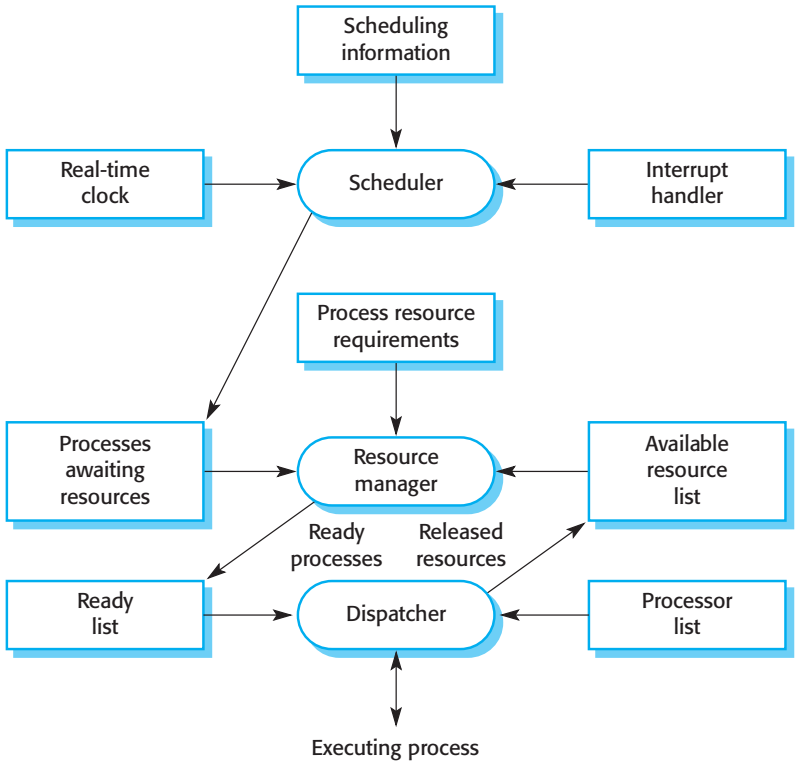


Figure 21.18 Components of a real-time operating system

3. A scheduler, which is responsible for examining the processes that can be executed and for choosing one of these processes for execution.
4. A resource manager, which allocates appropriate memory and processor resources to processes that have been scheduled for execution.
5. A dispatcher, which is responsible for starting the execution of processes.

Real-time operating systems for large systems, such as process control or telecommunication systems, may have additional facilities, namely, disk storage management, fault management facilities that detect and report system faults, and a configuration manager that supports the dynamic reconfiguration of real-time applications.

21.4.1 Process management

Real-time systems have to handle external events quickly and, in some cases, meet deadlines for processing these events. The event-handling processes must therefore be scheduled for execution in time to detect the event. They must also be allocated sufficient processor resources to meet their deadline. The process manager in an RTOS is responsible for choosing processes for execution, allocating processor and memory resources, and starting and stopping process execution on a processor.

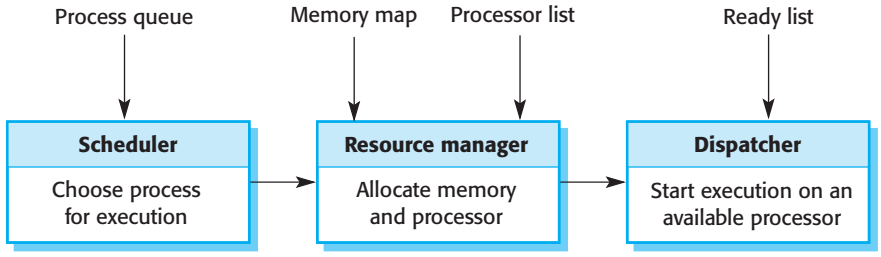


Figure 21.19 RTOS actions required to start a process

The process manager has to manage processes with different priorities. For some stimuli, such as those associated with certain exceptional events, it is essential that their processing should be completed within the specified time limits. Other processes may be safely delayed if a more critical process requires service. Consequently, the RTOS has to be able to manage at least two priority levels for system processes:

1. *Clock level* This level of priority is allocated to periodic processes.
2. *Interrupt level* This is the highest priority level. It is allocated to processes that need a very fast response. One of these processes will be the real-time clock process. This process is not required if interrupts are not supported in the system.

A further priority level may be allocated to background processes (such as a self-checking process) that do not need to meet real-time deadlines. These processes are scheduled for execution when processor capacity is available.

Periodic processes must be executed at specified time intervals for data acquisition and actuator control. In most real-time systems, there will be several types of periodic process. Using the timing requirements specified in the application program, the RTOS arranges the execution of periodic processes so that they can all meet their deadlines.

The actions taken by the operating system for periodic process management are shown in Figure 21.19. The scheduler examines the list of periodic processes and selects a process to be executed. The choice depends on the process priority, the process periods, the expected execution times, and the deadlines of the ready processes. Sometimes two processes with different deadlines should be executed at the same clock tick. In such a situation, one process must be delayed. Normally, the system will choose to delay the process with the longest deadline.

Processes that have to respond quickly to asynchronous events may be interrupt-driven. The computer's interrupt mechanism causes control to transfer to a predetermined memory location. This location contains an instruction to jump to a simple and fast interrupt service routine. The service routine disables further interrupts to avoid being interrupted itself. It then discovers the cause of the interrupt and initiates, with a high priority, a process to handle the stimulus causing the interrupt. In some high-speed data acquisition systems, the interrupt handler saves the data that the interrupt signaled was available in a buffer for later processing. Interrupts are then enabled again, and control is returned to the operating system.

At any one time several processes, all with different priorities, could be executed. The process scheduler implements system-scheduling policies that determine the order of process execution. There are two commonly used scheduling strategies:

1. *Nonpreemptive scheduling* After a process has been scheduled for execution, it runs to completion or until it is blocked for some reason, such as waiting for input. This can cause problems if there are processes with different priorities and a high-priority process has to wait for a low-priority process to finish.
2. *Preemptive scheduling* The execution of an executing process may be stopped if a higher-priority process requires service. The higher-priority process preempts the execution of the lower-priority process and is allocated to a processor.

Within these strategies, different scheduling algorithms have been developed. These include round-robin scheduling, where each process is executed in turn; rate monotonic scheduling, where the process with the shortest period (highest frequency) is given priority; and shortest deadline first scheduling, where the process in the queue with the shortest deadline is scheduled (Burns and Wellings 2009).

Information about the process to be executed is passed to the resource manager. The resource manager allocates memory and, in a multiprocessor system, also adds a processor to this process. The process is then placed on the "ready list," a list of processes that are ready for execution. When a processor finishes executing a process and becomes available, the dispatcher is invoked. It scans the ready list to find a process that can be executed on the available processor and starts its execution.

KEY POINTS

- An embedded software system is part of a hardware/software system that reacts to events in its environment. The software is "embedded" in the hardware. Embedded systems are normally real-time systems.
- A real-time system is a software system that must respond to events in real time. System correctness does not just depend on the results it produces, but also on the time when these results are produced.
- Real-time systems are usually implemented as a set of communicating processes that react to stimuli to produce responses.
- State models are an important design representation for embedded real-time systems. They are used to show how the system reacts to its environment as events trigger changes of state in the system.
- Several standard patterns can be observed in different types of embedded system. These include a pattern for monitoring the system's environment for adverse events, a pattern for actuator control, and a data-processing pattern.

- Designers of real-time systems have to do a timing analysis, which is driven by the deadlines for processing and responding to stimuli. They have to decide how often each process in the system should run and the expected and worst-case execution time for processes.
- A real-time operating system is responsible for process and resource management. It always includes a scheduler, which is the component responsible for deciding which process should be scheduled for execution.

FURTHER READING

Real-time Systems and Programming Language: Ada, Real-time Java and C/Real-time POSIX, 4th ed. An excellent and comprehensive text that provides broad coverage of all aspects of real-time systems. (A. Burns and A. Wellings, Addison-Wesley, 2009).

"Trends in Embedded Software Engineering." This article suggests that model-driven development (as discussed in Chapter 5 of this book) will become an important approach to embedded systems development. This is part of a special issue on embedded systems, and other articles, such as the one by Ebert and Jones, are also useful reading. (*IEEE Software,* 26 (3), May–June 2009). http://dx.doi.org/10.1109/MS.2009.80

Real-time systems: Design Principles for Distributed Embedded Applications, 2nd ed. This is a comprehensive textbook on modern real-time systems that may be distributed and mobile systems. The author focuses on hard real-time systems and covers important topics such as Internet connectivity and power management. (H. Kopetz, Springer, 2013).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/systems-engineering/

EXERCISES

21.1. Explain why responsiveness in real time is the critical difference between embedded systems and other software systems.

21.2. Identify possible stimuli and the expected responses for an embedded system that controls a home refrigerator or a domestic washing machine.

21.3. Using the state-based approach to modeling, as discussed in Section 21.1.1, model the operation of the embedded software for a voicemail system that is included in a landline phone.

Train protection system

- The system acquires information on the speed limit of a segment from a trackside transmitter, which continually broadcasts the segment identifier and its speed limit. The same transmitter also broadcasts information on the status of the signal controlling that track segment. The time required to broadcast track segment and signal information is 50 ms.
- The train can receive information from the trackside transmitter when it is within 10 m of a transmitter.
- The maximum train speed is 180 kph.
- Sensors on the train provide information about the current train speed (updated every 250 ms) and the train brake status (updated every 100 ms).
- If the train speed exceeds the current segment speed limit by more than 5 kph, a warning is sounded in the driver's cabin. If the train speed exceeds the current segment speed limit by more than 10 kph, the train's brakes are automatically applied until the speed falls to the segment speed limit. Train brakes should be applied within 100 ms of the time when the excessive train speed has been detected.
- If the train enters a track segment that is signaled with a red light, the train protection system applies the train brakes and reduces the speed to zero. Train brakes should be applied within 100 ms of the time when the red light signal is received.
- The system continually updates a status display in the driver's cabin.

Figure 21.20 Requirements for a train protection system

This should display the number of recorded messages on an LED display and should allow the user to dial-in and listen to the recorded messages.

21.4. What are the commonly used architectural patterns in real-time systems and when are they used?

21.5. Show how the Environmental Control pattern could be used as the basis of the design of a system to control the temperature in a greenhouse. The temperature should be between 10 and 30 degrees Celsius. If it falls below 10 degrees, the heating system should be switched on; if it goes above 30, the windows should be automatically opened.

21.6. Design a process architecture for an environmental monitoring system that collects data from a set of air quality sensors situated around a city. There are 5000 sensors organized into 100 neighborhoods. Each sensor must be interrogated four times per second. When more than 30% of the sensors in a particular neighborhood indicate that the air quality is below an acceptable level, local warning lights are activated. All sensors return the readings to a central computer, which generates reports every 15 minutes on the air quality in the city.

21.7. A train protection system automatically applies the brakes of a train if the speed limit for a segment of track is exceeded or if the train enters a track segment that is currently signaled with a red light (i.e., the segment should not be entered). Details are shown in Figure 21.20. Identify the stimuli that must be processed by the on-board train control system and the associated responses to these stimuli.

21.8. Suggest a possible process architecture for this system.

21.9. If a periodic process in the on-board train protection system is used to collect data from the trackside transmitter, how often must it be scheduled to ensure that the system is guaranteed to collect information from the transmitter? Explain how you arrived at your answer.

21.10. With the help of examples, define what a real-time operating system is. Explain how it is different from a conventional operating system. What are the components included in real-time operating systems and what are their responsibilities?

REFERENCES

Berry, G. 1989. "Real-Time Programming: Special-Purpose or General-Purpose Languages." In *Information Processing*, edited by G. Ritter, 89:11–17. Amsterdam: Elsevier Science Publishers.

Bruno, E. J., and G. Bollella. 2009. *Real-Time Java Programming: With Java RTS*. Boston: Prentice-Hall.

Burns, A., and A. Wellings. 2009. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Boston: Addison-Wesley.

Cooling, J. 2003. *Software Engineering for Real-Time Systems*. Harlow, UK: Addison-Wesley.

Douglass, B. P. 1999. *Real-Time UML: Developing Efficient Objects for Embedded Systems, 2nd ed.* Boston: Addison-Wesley.

———. 2002. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Boston: Addison-Wesley.

Ebert, C., and C. Jones. 2009. "Embedded Software: Facts, Figures and Future." *IEEE Computer* 26 (3): 42–52. doi:10.1109/MC.2009.118.

Harel, D. 1987. "Statecharts: A Visual Formalism for Complex Systems." *Sci. Comput. Programming* 8 (3): 231–274. doi:10.1016/0167-6423(87)90035-9.

———. 1988. "On Visual Formalisms." *Comm. ACM* 31 (5): 514–530. doi:10.1145/42411.42414.

Lee, E A. 2002. "Embedded Software." In *Advances in Computers*, edited by M. Zelkowitz. Vol. 56. London: Academic Press.

Silberschaltz, A., P. B. Galvin, and G. Gagne. 2013. *Operating System Concepts, 9th ed.* New York: John Wiley & Sons.

Stallings, W. 2014. *Operating Systems: Internals and Design Principles, 8th ed.* Boston: Prentice-Hall.

This page intentionally left blank