CHAPTER

System Design Techniques

7

CHAPTER POINTS

- A deeper look into design methodologies, requirements, specifications, and system analysis.
- System modeling.
- Formal and informal methods for system specification.
- Dependability, security, and safety.

7.1 Introduction

In this chapter, we consider the techniques required to create complex embedded systems. Thus far, our design examples have been small, so important concepts can be conveyed simply. However, most embedded system designs are inherently complex, given that their functional specifications are rich, and they must obey multiple other requirements on cost, performance, and so on. We require methodologies to help guide our design decisions when designing large systems.

In Section 7.2, we look at design methodologies in more detail. Section 7.3 studies use cases and requirement analysis, which capture informal descriptions of what a system must do, and Section 7.4 considers techniques for more formally specifying system modeling and functionality. Section 7.5 proceeds to system analysis and architecture design. Section 7.6 focuses on safety and security as aspects of dependability.

7.2 Design methodologies

This section considers the complete **design methodology** (a **design process**) for embedded computing systems. We start with the rationale for design methodologies, and then we look at several different methodologies.

7.2.1 Why design methodologies?

The process is important because without it we can't reliably deliver the products we want to create. Thinking about the sequence of steps necessary to build something

Computers as Components. https://doi.org/10.1016/B978-0-323-85128-2.00007-4

may seem superfluous, but the fact is that everyone has their own design process, even if they don't articulate it. If you are designing embedded systems in your basement by yourself, having your own work habits is fine. However, when several people work together on a project, they need to agree on who will do things and how those things will be done. Being explicit about the process is important when people work together. Because many embedded computing systems are too complex to be designed and built by one person, we must consider design processes.

Product metrics

The obvious goal of a design process is to create a product that does something useful. Typical specifications for a product will include functionality (e.g., wearable health monitor), manufacturing cost (e.g., must have a retail price below $200), performance (e.g., must power up within 2 s), power consumption (e.g., must run for 12 h without recharge), and other properties. Of course, a design process has several important goals beyond function, performance, and power:

- *Time-to-market*. Customers always want new features. A product that comes out first can win the market, even while setting customer preferences for future generations of the product. The profitable market life for some products is 3—6 months. If you are 3 months late, you will never make money. In some categories, the competition is against the calendar, not just against competitors. Calculators, for example, are disproportionately sold just before school starts in the fall. If you miss your market window, you must wait a year for another sales season.
- *Design cost*. Many consumer products are very cost sensitive. Industrial buyers are also increasingly concerned about costs. The costs of designing the system are distinct from the manufacturing cost. The cost of engineers' salaries, computers used in design, and so on must be spread across the units sold. In some cases, only one or a few copies of an embedded system may be built; hence, design costs can dominate manufacturing costs. Design costs can also be important for high-volume consumer devices when time-to-market pressures cause teams to swell in size.
- *Quality*. Customers not only want their products fast and cheap, but they also want them to be right. A design methodology that cranks out shoddy products will eventually be forced out of the marketplace. Correctness, reliability, and usability must be explicitly addressed from the beginning of the design job, to obtain a high-quality product at the end.

Design processes evolve over time owing to external and internal forces. Customers change, requirements change, products change, and available components change. Internally, people learn how to do things better, people move on to other projects, others come in, and companies are bought and sold to merge and shape corporate cultures.

Software engineers have spent a great deal of time thinking about software design processes. Much of this thinking has been motivated by mainframe software such as databases. However, embedded applications have also inspired some important thinking about software design.

A good methodology is critical to building systems that work properly. Delivering buggy systems to customers always causes dissatisfaction. However, in some

applications, such as medical and automotive systems, bugs create serious safety problems that can endanger users' lives. We discuss quality in more detail in Section 7.6.1. As an introduction, the next three examples discuss software errors that affect space missions.

Example 7.1: Loss of the Mars Climate Observer

In September 1999, the Mars Climate Observer, an unmanned U.S. spacecraft designed to study Mars, was lost. It most likely exploded as it heated up in the atmosphere of Mars after approaching the planet too closely. This occurred because of a series of problems, according to an analysis by *IEEE Spectrum* and contributing editor James Oberg [Obe99]. From an embedded systems perspective, the first problem is best classified as a requirement problem. The contractors who built the spacecraft at Lockheed Martin calculated the values for the flight controllers at the Jet Propulsion Laboratory (JPL). JPL did not specify the physical units to be used, but it expected them to be in Newtons. The Lockheed Martin engineers returned values in units of pound force. This discrepancy resulted in trajectory adjustments being 4.45 times larger than they should have been. The error was not caught by a software configuration process, nor was it caught by manual inspections. Although there were concerns about the spacecraft's trajectory, errors in the calculation of the spacecraft's position were not caught in time.

Example 7.2: New Horizons Communications Blackout

New Horizons, a NASA probe, experienced an 81-min radio communications blackout while approaching its Jupiter fly-by [Klo15]. The blackout was traced to a timing bug in New Horizon's command sequence.

Example 7.3: LightSail File Overflow Bug

A file overflow bug caused the LightSail satellite to malfunction in orbit [Cha15]. A telemetry data file was allowed to grow beyond its allocated memory. The bug was not found in pre-flight testing because the tests were not run for a sufficient period. The bug occurred only after about 40 h of operation. In this case, after 8 days, cosmic rays caused the computer to reset, allowing the machine to be restarted. The machine was then restarted once a day to avoid activating the bug again.

7.2.2 Design methodologies for embedded computing

A design methodology (**design flow**) is a sequence of steps to be followed during design. Some steps can be performed by tools, such as compilers or CAD systems; other steps can be performed by hand. In this section, we look at the basic characteristics of design flows.

Design methodologies created for other domains, such as Web sites or financial transactions, may not provide all characteristics and assurances required for embedded computing systems. The **waterfall** model was an early software design methodology that moved from requirements to architecture, coding, testing, and maintenance. Only local interactions between adjacent stages were allowed. The waterfall model is regarded as not sufficiently flexible for large modern software systems. **Agile** models are often used to design consumer-oriented software; they do not generally afford the documentation and testing regimens required for real-time systems.

V model

The **V model** [ISO18B] was defined to support safety methodologies for automobiles, and has been widely adopted for real-time embedded systems. As shown in Fig. 7.1, this methodology uses a top-down design and bottom-up verification. The bottom-up verification phases correspond to the top-down design phases, so that the design is verified from smallest to largest units. In the spirit of hierarchical design flows, the V model is also applied to both hardware and software design processes within the overall system design. We discuss ISO 26262, the standard that introduced the V model, in Section 7.6.5.

Hierarchical design flows

Many complex embedded systems are built with smaller designs. The complete system may require the design of significant software components, field-programmable gate arrays (FPGAs), and so on, and these in turn may be built from smaller components that need to be designed. The design flow follows the levels of abstraction in the system, from complete system designs to individual components. The implementation phase of a flow is a complete flow from specification through testing. In such a large project, each flow will be handled by separate people or teams. The teams must rely on each other's results. The component teams take their requirements from the team handling the next higher level of abstraction, and the higher-level team relies on the quality of the design and testing performed by the component team. Good communication is vital in such large projects.

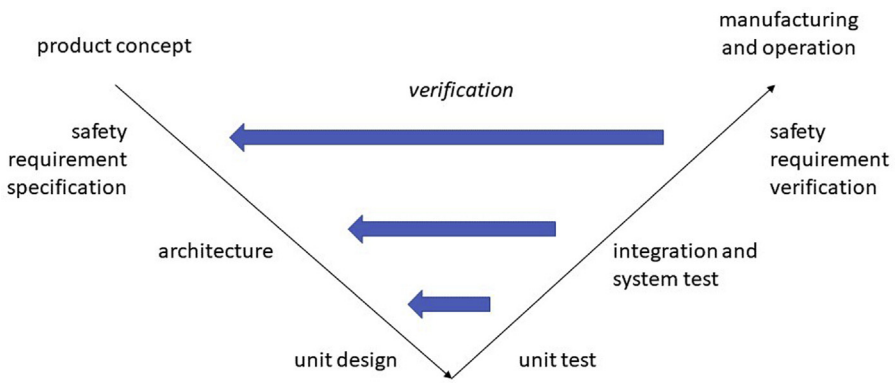


FIGURE 7.1

ISO 26262 V model [ISO18B].

7.3 Requirements analysis and specification

Before designing a system, we need to know what we are designing. We need to gather information from a variety of sources to determine the desired characteristics of the system. We also need to capture these characteristics in ways that can be used by the rest of the design team.

Requirements and specifications

A **requirement** is a description of a desired characteristic of the system: some aspect of its behavior, its responsiveness, cost, and so on. A **specification** is a complete set of requirements for a system. Requirements and specifications are, however, directed toward the outward behavior of the system, not its internal structure.

Functional and nonfunctional requirements

There are two types of requirements: **functional** and **nonfunctional.** A functional requirement states what the system must do, such as computing an FFT. A nonfunctional requirement can be any number of other attributes, including physical size, cost, power consumption, design time, reliability, and so on.

We often start with a more informal process to determine the basic requirements of the system. We then refined those characteristics to form a complete specification.

7.3.1 Requirements capture

How do you determine the requirements? If the product is a continuation of a series, then many of the requirements will be well understood. However, even with the most modest upgrade, talking to the customer is valuable. In a large company, marketing or sales departments may do most of the work of asking customers what they want, but a surprising number of companies have designers talk directly with customers. Direct customer contact gives the designer an unfiltered sample of what the customer needs. It also helps build empathy with the customer, which often pays off in cleaner, easier-to-use customer interfaces. Talking to the customer may also include conducting surveys, organizing focus groups, or asking selected customers to test a mockup or prototype.

Informal requirements capture

Although the final architecture must be complete and correct, informal methods of requirement analysis can help start the process. The **CRC card** methodology is a well-known and useful way to help analyze a system. The acronym *CRC* stands for these three major items that the methodology tries to identify:

- *Classes* define the logical groupings of data and functionality.
- *Responsibilities* describe what the classes do.
- *Collaborators* are the other classes with which a given class works.

The CRC card methodology has people write on index cards. In the United States, the standard size for index cards is 3×5 in. Hence, these cards are often called "3×5 cards." An example card is shown in Fig. 7.2; it has space to write down the class name, its responsibilities and collaborators, and other information. The essence of the CRC card methodology is to have people write on these cards, talk about them, and update them until they are satisfied with the descriptions.

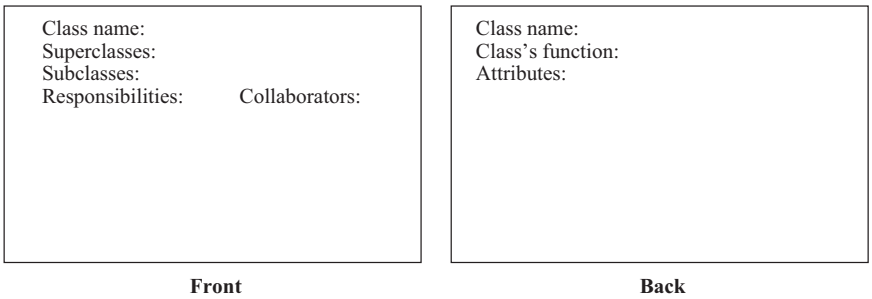


FIGURE 7.2

Layout of a CRC card.

This technique may seem like a primitive way to design computer systems. However, it has several important advantages. First, it is easy to get noncomputer people to create CRC cards. Getting the advice of domain experts (e.g., automobile designers for automotive electronics or human factors experts for appliance design) is vital to system design. The CRC card methodology is informal enough that it will not intimidate noncomputer specialists and will allow you to capture their input. Second, it aids computer specialists by encouraging them to work in groups to analyze scenarios. The walkthrough process used with CRC cards is especially useful in scoping out a design and determining which parts of a system are poorly understood. This informal technique is valuable for tool-based design and coding. If you still feel a need to use tools to help you practice the CRC methodology, software engineering tools are available that automate the creation of CRC cards.

Before going through the methodology, let's review the CRC concepts in a little more detail. We are familiar with classes; they encapsulate functionality. A class may represent a real-world object or it may describe an object that has been created solely to help architect a system. A class has both an internal state and a functional interface; the functional interface describes the class's capabilities. The responsibility set is an informal way of describing the functional interface. The responsibilities provide the class's interface, not its internal implementation. Unlike describing a class in a programming language, the responsibilities may be described informally in English (or in your favorite language). The collaborators of a class are simply the classes to which it talks: classes that use its capabilities or that it calls upon to help it do its work.

The class terminology is a little misleading when an object-oriented programmer looks at CRC cards. In the methodology, a class is used more like an object in an object-oriented programming language. The CRC card class is used to represent a real actor in the system. However, the CRC card class is easily transformed into a class definition in an object-oriented design.

CRC card analysis is performed by a team of people. It is possible to use it by yourself, but a lot of the benefit of the method comes from talking about developing classes with others. Before beginning the process, you should create many copies of

blank CRC cards using the basic format shown in Fig. 7.2. As you work in a group, you will write on these cards and will discard many just to rewrite them as the system evolves. The CRC card methodology is informal, but you should go through the following steps when using it to analyze a system:

1. *Develop an initial list of classes*. Write down the class name and perhaps a few words on what it does. A class may represent a real-world object or an architectural object. Identifying which category the class falls into, perhaps by putting a star next to the name of a real-world object, is helpful. Each person can be responsible for handling a part of the system, but team members should talk during this process to ensure that no classes are missed and that duplicate classes are not created.
2. *Write an initial list of responsibilities and collaborators*. The responsibilities list helps describe in more detail what the class does. The collaborators' list should be built from obvious relationships between the classes. Both the responsibilities and collaborators will be refined in later stages.
3. *Create some usage scenarios*. These scenarios describe what the system does. Scenarios begin with some type of outside stimulus, which is an important reason for identifying relevant real-world objects.
4. *Walk through the scenarios*. This is the heart of the methodology. During the walkthrough, each person on the team represents one or more classes. The scenario should be simulated by action. Team members may announce what their class does, ask other classes to perform operations, and so on. Moving around, for example, to show the transfer of data, team members may visualize the system's operation. During the walkthrough, all information created thus far is targeted for updating and refinement, including the classes, responsibilities, collaborators, and usage scenarios. Classes may be created, destroyed, or modified during this process. You will also probably find many holes in the scenario itself.
5. *Refine* the classes, responsibilities, and collaborators. Some of this will be done during the walkthrough, while adding a second pass after the scenarios. A longer perspective will help you make more global changes to the CRC cards.
6. *Add class relationships*. When CRC cards have been refined, subclass and superclass relationships should become clearer and can be added.

The CRC cards must then be used to help drive the implementation. In some cases, it may work best to use the CRC cards as direct source material for the implementors; this is particularly true if you can get the designers involved in the CRC card process. In other cases, you may want to write a more formal description in UML or other information captured during the CRC card analysis. This formal description is then used to build the design document for the system implementors. Example 7.5 illustrates the use of the CRC card methodology.

Example 7.4: CRC Card Analysis of an Elevator System

Let's perform a CRC card analysis of an elevator system. First, we need the following basic set of classes:

- *Real-world classes*: elevator car, passenger, floor control, car control, and car sensor.
- *Architectural classes*: car state, floor control reader, car control reader, car control sender, and scheduler.

For each class, we need the following initial set of responsibilities and collaborators. An asterisk is used to remind ourselves which classes represent real-world objects.

Class	Responsibilities	Collaborators
Elevator car*	Moves up and down	Car control, car sensor, car control sender
Passenger*	Pushes floor control and car control buttons	Floor control, car control
Floor control*	Transmits floor requests	Passenger, floor control reader
Car control*	Transmits car requests	Passenger, car control reader
Car sensor*	Senses car position	Scheduler
Car state	Records current position of car	Scheduler, car sensor
Floor control reader	Interface between floor control and rest of system	Floor control, scheduler
Car control reader	Interface between car control and rest of system	Car control, scheduler
Car control sender	Interface between scheduler and car	Scheduler, elevator car
Scheduler	Sends commands to cars based upon requests	Floor control reader, car control reader, car control sender, car state

Several usage scenarios define the basic operation of the elevator system as well as some unusual scenarios:

1. One passenger requests a car on a floor, gets in the car when it arrives, requests another floor, and gets out when the car reaches that floor.
2. One passenger requests a car on the floor, gets in the car when it arrives, and requests the floor that the car is currently on.
3. A second passenger requests a car, while another passenger is riding in the elevator.
4. Two people push the floor buttons on different floors at the same time.
5. Two people push car control buttons in different cars at the same time.

At this point, we must walk through the scenarios and make sure they are reasonable. Therefore, we should find a set of people and walk through these scenarios. Do the classes, responsibilities, collaborators, and scenarios make sense? How would you modify them to improve the system specifications?

7.3.2 From requirements to specification

Characteristics of specifications

A good specification should meet several tests [Dav90]:

- *Correctness*: The requirements should not mistakenly describe what the customer wants. Part of correctness is avoiding over-requiring; the requirements should not add unnecessary conditions.
- *Unambiguousness*: The requirements document should be clear and have only one plain language interpretation.
- *Completeness*: All requirements should be included.
- *Verifiability*: There should be a cost-effective way to ensure that each requirement is satisfied in the final product. For example, a requirement that the system package be "attractive" would be hard to verify without some agreed upon definition of attractiveness.
- *Consistency*: One requirement should not contradict another requirement.
- *Modifiability*: The requirements document should be structured so that it can be modified to meet changing requirements without losing consistency, verifiability, and so on.
- *Traceability*: Each requirement should be traceable in the following ways:
  - We should be able to trace backward from the requirements to know why each requirement exists.
  - We should be able to trace forward from documents created before the requirements (e.g., marketing memos) to understand how they relate to the final requirements.
  - We should be able to trace forward and understand how each requirement is satisfied in the implementation.
  - We should also be able to trace backward from the implementation to know which requirements they were intended to satisfy.

7.3.3 Validating the specification

The specification is generated early in the design process. The goal of validation is to ensure that the specification properly captures the user's needs.

Validating the specification is important for the simple reason that bugs in the requirements or specifications can be extremely expensive to fix later. Fig. 7.3 shows how the cost of fixing bugs grows over the course of the design process. We use the waterfall model as a simple example, but the same holds for any design flow. The longer a bug survives in the system, the more expensive it will be to fix it. A coding bug, if not found until after system deployment, will cost money to recall and reprogram, among other things. However, a bug introduced earlier in the flow and not discovered until the same point will accrue all costs and more. A bug was introduced in the specification and left until maintenance could force an entire redesign of the product, not just the distribution of a software update. Discovering bugs early is crucial because it prevents them from being released to customers, minimizes design costs, and reduces design time. Although some specification bugs will become apparent in the detailed design stages

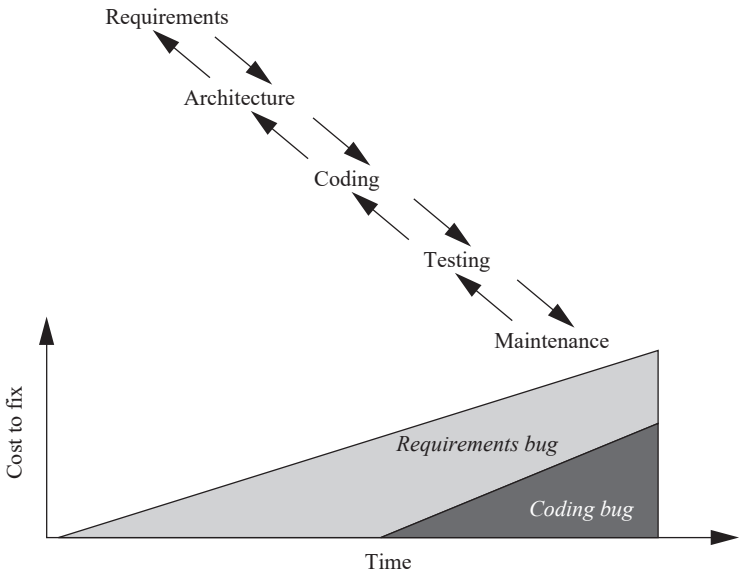


FIGURE 7.3

Long-lived bugs are more expensive to fix.

(e.g., as the consequences of certain requirements are better understood), it is possible and desirable to weed out as many bugs as possible during the generation of the requirements and specifications.

The goal of validating the specification is to ensure that it satisfies the criteria originally applied in Section 7.3.2: correctness, completeness, consistency, and so on. Validation is part of the larger requirements and specification process. Various techniques can be applied, while they are being created, to help you understand the requirements and specifications, whereas others can be applied to a draft with the results used to modify the specifications.

Prototyping

Prototypes are useful when dealing with end users. Rather than simply describing the system to them in broad technical terms, a prototype can let them see, hear, and touch at least some of the important aspects of the system. Of course, the prototype will not be fully functional, because the design work has not yet been done. However, user interfaces are well suited for prototyping and user testing. Canned or randomly generated data can be used to simulate the system's internal operation. A prototype can help the end user critique numerous functional and nonfunctional requirements, such as data displays, speed of operation, size, weight, and so forth. Certain programming languages, sometimes called **prototyping languages** or **specification languages**, are especially well suited to prototyping. Very high-level languages, such as MATLAB in the signal processing domain, may be able to perform functional attributes, such as the mathematical function to be performed, but not nonfunctional attributes, such as the speed of execution. **Preexisting systems** can also be used to

help end users articulate their needs. Specifying what someone does or doesn't like about an existing machine is much easier than having them talk about the new system in the abstract. In some cases, it may be possible to construct a prototype of the new system from the preexisting system.

Auditing tools may be useful in verifying consistency, completeness, and so forth. Working through **usage scenarios** often helps designers fill out the details of a specification and ensure its completeness and correctness.

Formal methods

In some cases, **formal methods** (that is, design techniques that make use of mathematical proofs) may be useful. Proofs may be done either manually or automatically. In some cases, proving that a particular condition can or cannot occur according to the specification is important. Automated proofs are particularly useful in certain types of complex systems that can be specified succinctly, but whose behavior over time is complex. For example, complex protocols have been successfully formally verified in this fashion.

7.4 System modeling

In this section, we look at some advanced techniques for specification and how they can be used. Section 7.4.1 looks at a model-based design, and Section 7.4.2 studies two UML dialects for modeling.

7.4.1 Model-based design

Model-based design

Model-based design [Kar03] has emerged as an important methodology for cyber-physical systems. This approach is more holistic than traditional design flows because it takes a unified view of both the physical plant and the embedded computing system.

Model-based design makes use of a set of tools that compile a description of cyber-physical systems into a set of implementations: software, computing platform, and physical plant components. Given the wide range of cyber-physical systems, we should not expect a single toolset to serve all applications. Instead, a set of tools is created to serve a particular class of designs.

Domain-specific modeling languages

Designers create their designs in a **domain-specific modeling language** (**DSML**). This language is designed to be used by a domain expert, such as an aircraft or automobile designer. The DSML captures both the functional and nonfunctional characteristics of the system. This description can be used in several ways:

- The cyber-physical system specification can be simulated, for example, by generating a Simulink model.
- The implementation design space can be explored under a range of design parameters.
- Given the choice of design parameters, an implementation can be synthesized.

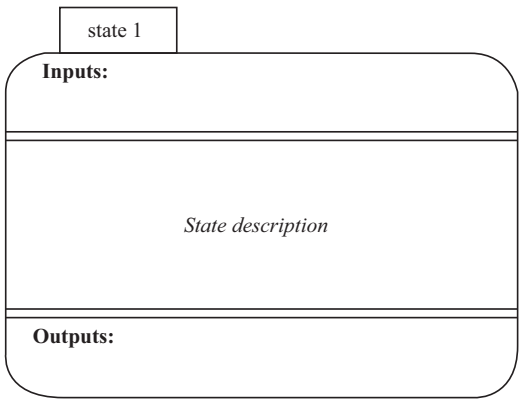
The integration of the tools ensures that the implementation is consistent with the high-level simulation and the original specifications.

Example 7.5 describes the specification of a real-world, safety-critical system used in aircraft. The specification techniques developed to ensure the correctness and safety of this system can also be used in many applications, particularly in systems where much of the complexity goes into the control structure.

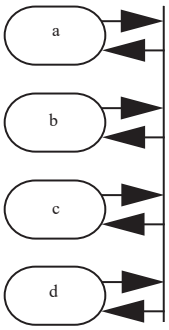
Example 7.5: Traffic Alert and Collision Avoidance System II Specification

TCAS II (Traffic Alert and Collision Avoidance System) is a collision avoidance system for aircraft. Based on a variety of information, a TCAS unit in an aircraft keeps track of the position of other nearby aircraft. If TCAS decides that a mid-air collision may be likely, it uses audio commands to suggest evasive action. For example, a prerecorded voice may warn "DESCEND! DESCEND!" if TCAS believes that an aircraft above poses a threat and there is room to maneuver below. TCAS makes sophisticated decisions in real time and is clearly safety-critical. On the one hand, it must detect as many potential collision events as possible within the limits of its sensors. On the other hand, it must generate as few false alarms as possible because the extreme maneuvers it recommends are themselves potentially dangerous.

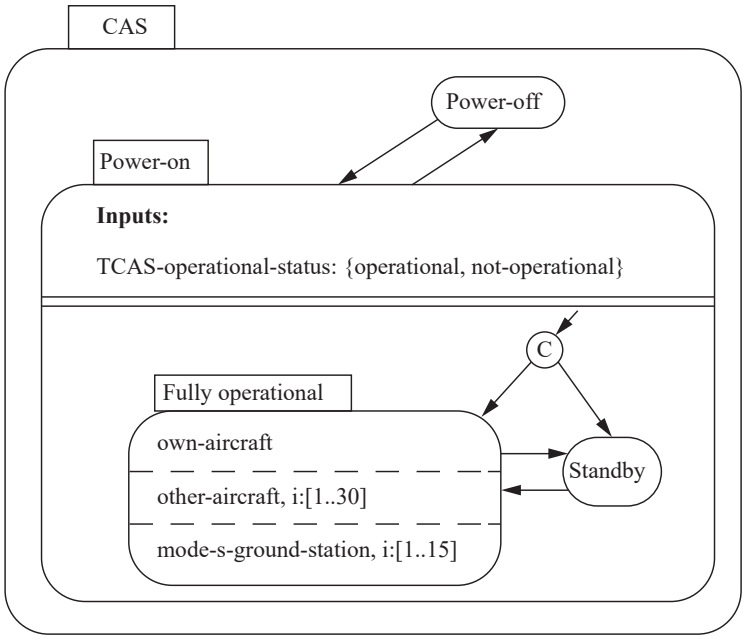
Leveson et al. [Lev94] developed a specification for the TCAS II system. We won't cover the entire specification here, but just enough to provide its flavor. The TCAS II specification was written in root system markup language. They use a modified version of state chart notation for specifying states, in which the inputs to and outputs from the state are made explicit. The basic state notation looks like this:



They also use a transition bus to show sets of states in which there are transitions between all (or almost all) states. In this example, there are transitions from a, b, c, or d to any of the other states:



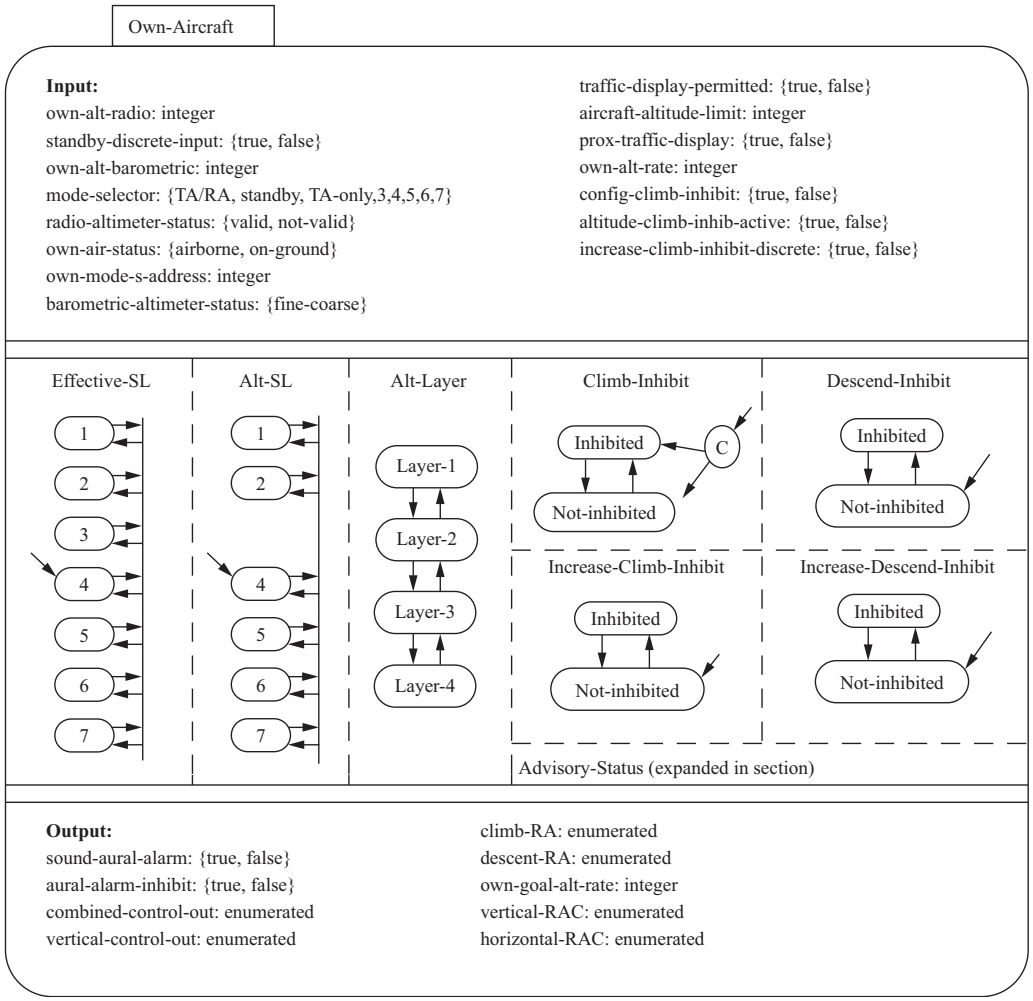
The top-level description of the collision avoidance system is relatively simple:



This diagram specifies that the system has *Power-off* and *Power-on* states. In the Power-on state, the system may be in *Standby* or *Fully operational* mode. In the Fully operational mode, three components operate in parallel, as specified by the AND state: the *own-aircraft* subsystem, a subsystem to keep track of up to 30 other aircraft, and a subsystem to keep track of up to 15 Mode S ground stations, which provide radar information.

The next diagram shows a specification of the *Own-Aircraft* AND state. Once again, the behavior of *Own-Aircraft* is an AND composition of several subbehaviors. The *Effective-SL* and *Alt-SL* states reflect two ways to control the sensitivity level (SL) of the system, with each state representing a different sensitivity level. Differing sensitivities are required depending on the distance from the ground and other factors. The *Alt-Layer* state divides the vertical airspace into layers, with this state keeping track of the current layer. *Climb-Inhibit* and *Descent-Inhibit* states are used to selectively inhibit climbs) which may be difficult at high

altitudes) or descents (clearly dangerous near the ground), respectively. Similarly, the *Increase-Climb-Inhibit* and *Increase-Descend-Inhibit* states can inhibit high-rate climbs and descents. Because the *Advisory-Status* state is complicated, its details are not shown here.



7.4.2 UML dialects for modeling

Several UML profiles have been defined for use in system modeling and real-time embedded computing system design. These dialects provide standard ways to talk

about systems and embedded concepts. We will use a simple engine control unit to illustrate some characteristics of these dialects.

SysML

The **Systems Modeling Language** (**SysML**) [OMG17] is based on UML and is designed as a broad-spectrum systems engineering language. SysML defines nine types of diagrams, some adopted from the base UML definition and others newly defined. Frames are optional parts of UML diagrams, but are required by SysML; each SysML diagram has a defined abbreviation. The sequence (**sd**), state machine (STM), package (pkg), and use case (uc) diagrams are all used in a manner consistent with UML.

SysML supports two types of block diagrams: the **block definition diagram** (**bdd**) defines the types of blocks used in the structure, and the **internal block diagram** (**ibd**) shows the connections between blocks. Fig. 7.4 shows a simple bbd for the engine control unit with three blocks or subsystems: engine sensors, the control law computation block, and engine actuators. Fig. 7.5 shows the associated ibd.

The **activity diagram** (**act**) combines object flows with object data flows. It is used to describe the behavior of a block. Fig. 7.6 shows a simple act. Control flow is described using state machine constructs, while an object for the engine temperature value shows the flow of information.

The **requirement diagram** (**req**) and **parametric diagram** (**par**) are new to SysML. Fig. 7.7 shows a simple req: one requirement is functional, and the other is

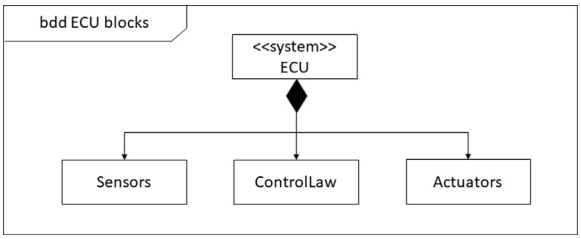


FIGURE 7.4

A SysML block definition diagram.

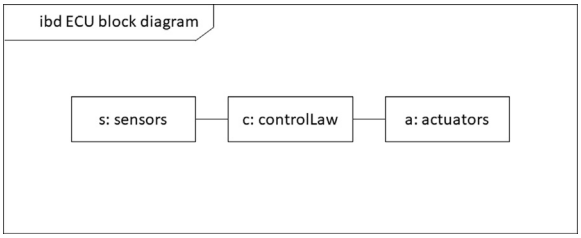


FIGURE 7.5

A SysML internal block diagram.

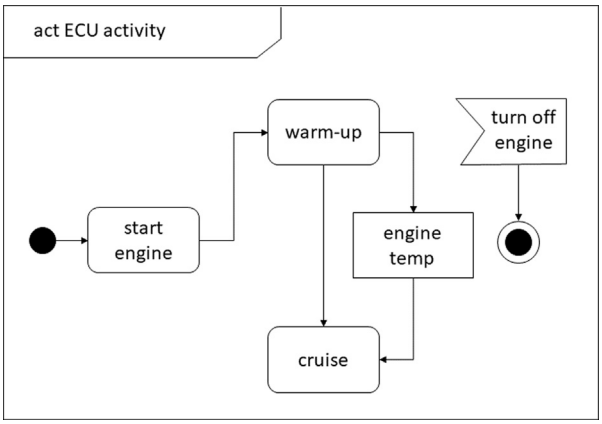


FIGURE 7.6

A SysML activity diagram.

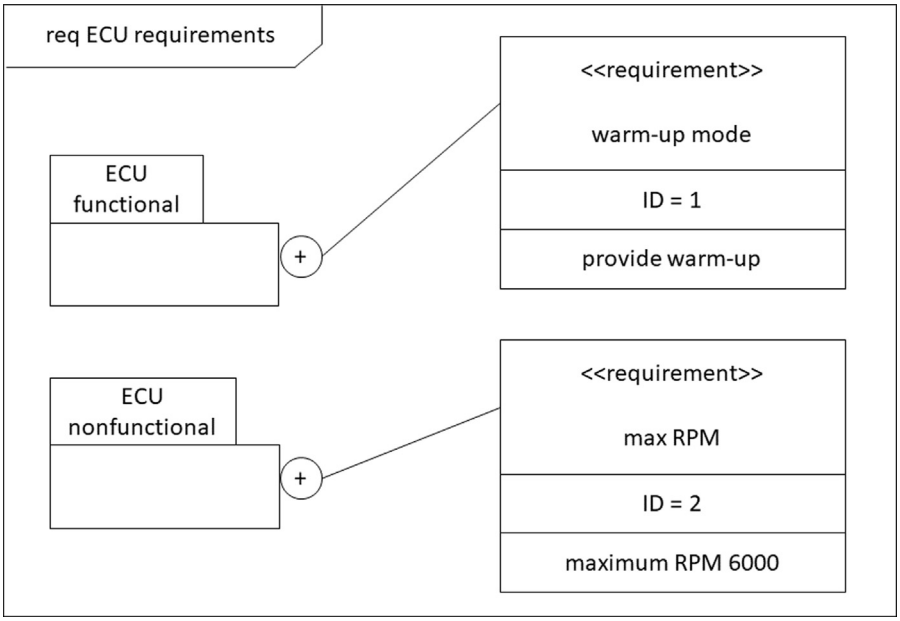


FIGURE 7.7

A SysML requirement diagram.

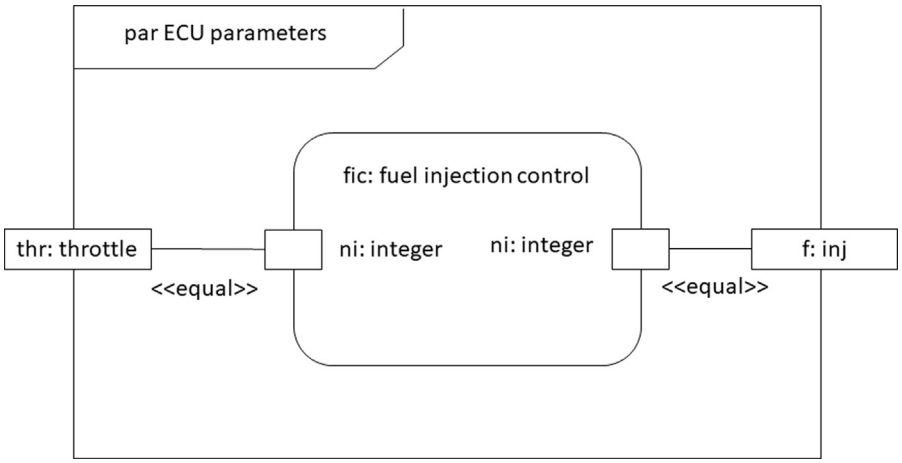


FIGURE 7.8

A SysML parametric diagram.

nonfunctional. A par is shown in Fig. 7.8. A block is used to define the relationship between the parameters that are represented as inputs and outputs.

SysML also supports tables to describe the relationships between elements.

MARTE

MARTE [OMG19] stands for *modeling and analysis of real-time and embedded systems.* It provides several types of diagrams specific to that design domain.

The **Non-Functional Properties Modeling** (**NFP**) module is used to capture the nonfunctional properties of the system. The **Time Modeling** (**Time**) module models both chronometric and sequential time. Fig. 7.9 shows a simple sequence diagram with an MARTE timing constraint. The constraint is given as a range of values. Two types of time can be described: an untimed causal model, a partially timed synchronous model, and a real-time physical model.

MARTE also supports the description of allocation decisions.

The Allocation Modeling (Alloc) module supports the allocation of functions and operations to resources. Fig.7.10 shows an object diagram with allocation constraints at two levels: from the functional blocks to the RTOS abstraction level and from there to the computing platform.

The **High-Level Application Modeling** (HLAM) profile provides for the description of both quantitative and qualitative features. The **Detailed Resource Modeling** (**DRM**) profile provides both **software resource modeling** (SRM) and **hardware resource modeling** (**HRM**).

The **Generic Quantitative Analysis Modeling** (**GQAM**) profile describes two types of evaluation: schedulability analysis for software task sets, and performance analysis, typically of a statistical nature. The MARTE schedulability analysis profile supports the analysis of both temporal constraints and schedulability, as well as sensitivity analysis of timing to system design parameters. Schedulability can describe a

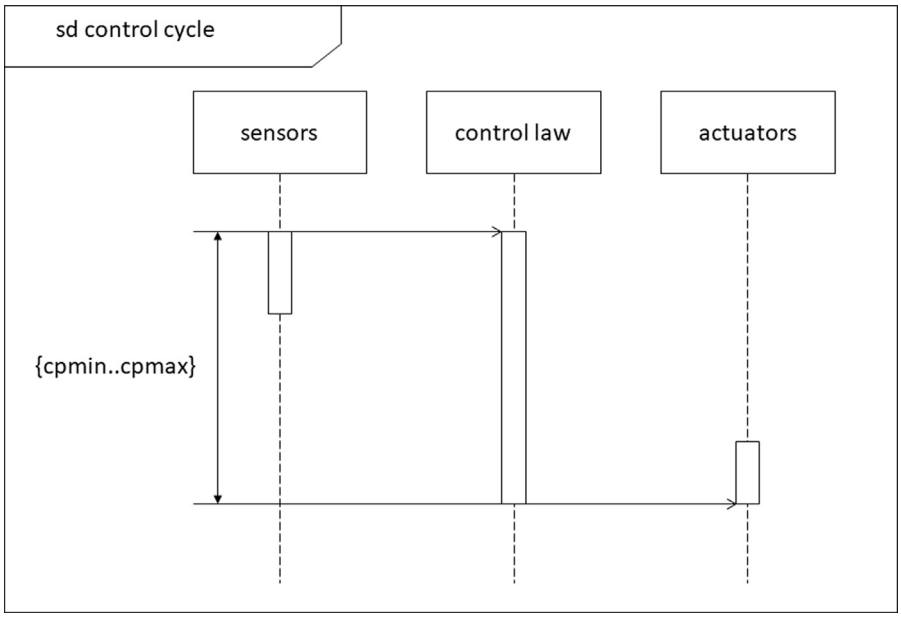


FIGURE 7.9

Sequence diagram with MARTE timing constraint.

workload, timed observers to capture constraints and timing results, and resources, such as RTOS schedulers. The **Performance Analysis Modeling** (**PAM**)_profile is used to describe best-effort and soft real-time systems.

7.5 System analysis and architecture design

In this section, we consider how to turn a specification into an architectural design. We already have several techniques for making specific decisions. Here, we look at how to get a handle on the overall system architecture: Section 7.5.1 looks at common design patterns for embedded computing systems, and Section 7.5.2 considers the transaction-level modeling of architectures.

7.5.1 Design patterns

A **design pattern** is a solution to a given type of recurring engineering problem. They capture best practices and help us identify structure in our designs. A design pattern for an embedded architecture might consider several elements: computing platform, allocation of operations, schedules for computations, and data movement.

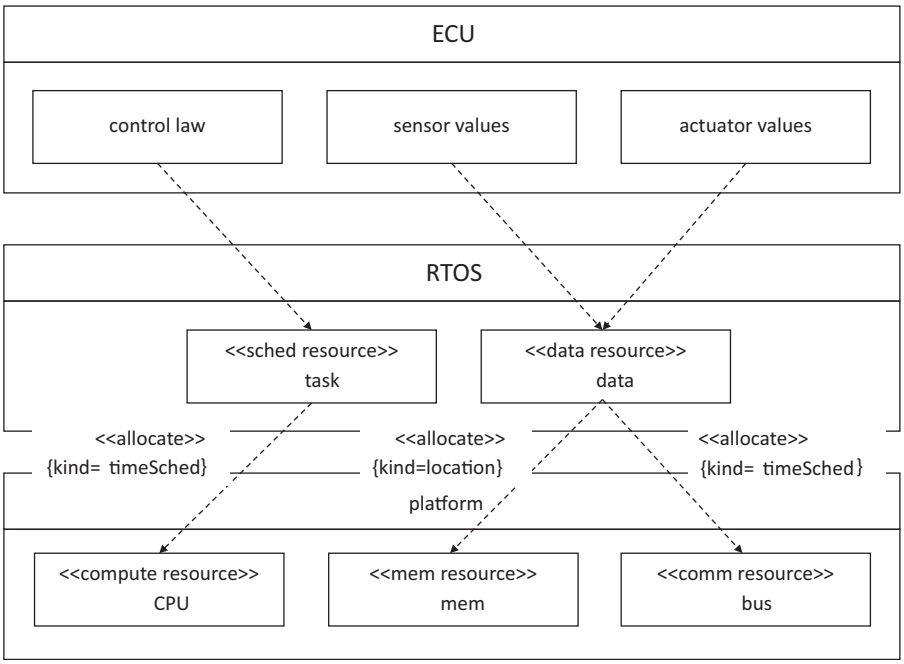


FIGURE 7.10

Object diagram MARTE allocation annotations.

The microcontroller provides one of the most basic design patterns in embedded computing. A single processor connects to memory and I/O devices over a bus. Threads run under the control of an RTOS. The single-chip nature of a microcontroller simplifies many design decisions: we choose the entire configuration of CPU, memory, and I/O as a unit; on-chip communication is generally less costly in time and energy than off-chip communication.

A heterogeneous multiprocessor system-on-chip (MPSoC) provides more sophisticated variation. The MPSoC includes several processors, some of which may be accelerators with limited programmability. The chip may include several internal communication networks.

A **networked control system** typically refers to a single-bus system. It shares the same structure as a microcontroller (CPU, memory, I/O, bus), but these components are embedded across multiple chips and may be spread over a significant physical space. The schedule of data movements in the network is an important aspect of the design process.

Several other design patterns for networked control have more complex topologies. For example, a unit topology places the processing for a particular operation within its physical housing. A functional architecture groups the processing of several related functions. A shared architecture combines several different functions into a

processing unit. A zone architecture groups computations based on their physical location. We explore these architectures and their uses in automobiles and airplanes in Chapter 9.

7.5.2 Transaction-level modeling

Refining a specification into an architecture requires the consideration of several factors: functionality, real-time performance, power consumption, and so on. Intermediate levels of abstraction help us manage the design process and competing constraints. A traditional level of abstraction in hardware design is **register transfer design**, a clock cycle-accurate model. We used threads as a model for software design. An intermediate model is a **transaction-level model** [Cai03]. A **transaction** is a communication between concurrent entities (either software threads or hardware units). Transactions can be modeled at several levels of detail, providing a path for design refinement:

- Untimed models (causal in MARTE terminology) preserve the partial ordering of transactions but do not provide information about execution time. Untimed models support functional debugging.
- Bus-accurate models (synchronous in MARTE terminology) provide refined timing models of the transactions, typically an estimate of the bus transaction determined from the bus specifications and estimates of software execution time.
- Cycle-accurate models (also synchronous in MARTE terminology) model communication, software, and hardware to clock cycle accuracy.

SystemC [IEE12] is a C++ library that supports the simulation and synthesis of concurrent processes, such as embedded software and hardware. The OCCN framework [Cop04] is a modeling and simulation environment built on top of SystemC.

7.6 Dependability, safety, and security

The quality of a product or service can be judged by how well it satisfies its intended function. A product can be of low quality for several reasons, such as if it were manufactured shoddily. Its components may have been improperly designed, its architecture was poorly conceived, and the product's requirements were poorly understood.

When we evaluate modern embedded computing systems, the traditional notions of quality, which are shaped in large part by consumer satisfaction, are no longer sufficient. **Dependability, safety**, and **security** are all vital aspects of a satisfactory system. These concepts are related but distinct. Dependability is a quantitative term that measures the length of time a system can operate without defects [Sie98]. Safety and security can be measured using dependability metrics. All terms are related to the general concept of quality.

The software testing techniques described in Section 5.10 constitute one way to help meet quality-related goals. However, the pursuit of dependability, security, and safety must extend throughout the design flow. For example, settling on the proper requirements and specifications is an important determinant of quality. If the system is too difficult to design, it will probably be difficult to keep it working properly. Customers may desire features that sound nice, but, in fact, don't add much to the overall usefulness of the system. In many cases, having too many features only makes the design more vulnerable to both design and implementation errors, as well as attacks from hostile sources.

In this section, we review these related concepts in more detail. We start with a discussion of quality assurance processes. Section 7.6.2 discusses verifying requirements and specifications. Section 7.6.3 introduces design reviews as a method for quality management. Section 7.6.4 introduces several safety-oriented methodologies. Section 7.6.5 considers methodologies for safety-critical systems.

7.6.1 Quality assurance techniques

Quality assurance refers to both informal and more rigorous forms of product reliability, safety, security, and other aspects of fitness for use. The International Standards Organization (ISO) has created a set of quality standards known as **ISO 9000.** ISO 9000 was created to apply to a broad range of industries, including but not limited to embedded hardware and software. A standard developed for a particular product, such as wooden construction beams, could specify criteria particular to that product, such as the load that a beam must be able to carry. However, a wide-ranging standard, such as ISO 9000, cannot specify the detailed standards for every industry. Consequently, ISO 9000 concentrates on the processes used to create the product or service. The processes used to satisfy ISO 9000 affect the entire organization, as well as the individual steps taken during design and manufacturing.

A detailed description of ISO 9000 is beyond the scope of this book; several books [Sch94, Jen95] describe ISO 9000's applicability to software development. We can, however, make the following observations about quality management based on ISO 9000:

- *Process is crucial*. Haphazard development leads to low-quality haphazard products. Knowing what steps are to be followed to create a high-quality product is essential to ensuring that all the necessary steps are followed.
- *Documentation is important*. Documentation has several roles: The creation of the documents describing processes helps those involved understand the processes; documentation helps internal quality monitoring groups to ensure that the required processes are being followed; and documentation helps outside groups (e.g., customers, auditors, and so on) understand the processes and how they are being implemented.
- *Communication is important*. Quality ultimately relies on people. Good documentation helps people understand the total quality process. The people in the

organization should understand not only their specific tasks but also how their jobs can affect overall system quality.

Many types of techniques can be used to verify system designs and ensure quality. Techniques can be either *manual* or *tool-based*. Manual techniques are surprisingly effective in practice. In Section 7.6.4, we discuss *design reviews*, which are simply meetings at which the design is discussed; such meetings are prosperous in identifying bugs. Many software testing techniques described in Chapter 5 can be applied manually by tracing through the program to determine the required tests. Tool-based verification helps considerably in managing large quantities of information that may be generated in a complex design. Test generation programs can automate much of the drudgery of creating test sets for programs. Tracking tools can help ensure that various steps have been performed. Design flow tools automate the process of running design data through other tools.

Metrics are important to the quality control process. To determine whether we have achieved high levels of quality, we must be able to measure aspects of the system and our design process. We can measure certain aspects of the system itself, such as the execution speed of programs or the coverage of test patterns. We can also measure aspects of the design process, such as the rate at which bugs are found.

Tool-driven and manual techniques must fit into an overall process. The details of that process will be determined by several factors, including the type of product being designed (e.g., video game, laser printer, or air traffic control system), the number of units to be manufactured, the time allowed for design, the existing practices in the company into which any new processes must be integrated, and many other factors. An important role of ISO 9000 is to help organizations study their total processes, not just particular segments that may appear to be important at a particular time.

One well-known way of measuring the quality of an organization's software development process is the **Capability Maturity Model (CMM),** developed by Carnegie Mellon University's Software Engineering Institute [SEI99]. The CMM provides a model for judging an organization. It defines the following five levels of maturity:

1. *Initial*. A poorly organized process with limited well-defined processes. The success of a project depends on the efforts of individuals, not the organization itself.
2. *Repeatable*. This level provides basic tracking mechanisms that allow management to understand cost, scheduling, and how well the systems under development meet their goals.
3. *Defined*. The management and engineering processes are documented and standardized. All projects make use of documented and approved standard methods.
4. *Managed*. This phase makes detailed measurements of the development process and product quality.
5. *Optimizing*. At the highest level, feedback from detailed measurements is used to continually improve an organization's processes.

The Software Engineering Institute has found limited organizations anywhere in the world that meet the highest level of continuous improvement and quite a few organizations that operate under the chaotic processes of the initial level. However, the CMM provides a benchmark by which organizations can judge themselves and use that information for improvement.

7.6.2 Design reviews

The **design review** [Fag76] is a critical component of any quality assurance process. The design review is a simple, low-cost way to catch bugs early in the design process. A design review is simply a meeting in which team members discuss a design and review how a component of the system works. Some bugs are caught simply by preparing for the meeting, as the designer is forced to think through the design in detail. Other bugs are caught by people attending the meeting who will notice problems that may not be caught by the unit's designer. By catching bugs early and not allowing them to propagate into the implementation, we reduce the time required to get a working system. We can also use the design review to improve the quality of the implementation and make future changes easier to implement.

Design review format

A design review is held to review a particular component of the system. A design review team has several types of members:

- The *designers* of the component being reviewed are, of course, central to the design process. They present their design to the rest of the team for review and analysis.
- The *review leader* coordinates the pre-meeting activities, the design review, and the post-meeting follow-up.
- The *review scribe* records the minutes of the meeting so that designers and others know which problems need to be fixed.
- The *review audience* studies the component. Audience members will naturally include other members of the project for which this component is being designed. Audience members from other projects often add valuable perspectives and may notice problems that team members have missed.

The design review process begins before the meeting itself. The design team prepares a set of documents (e.g., code listings, flowcharts, and specifications) that will be used to describe each component. These documents are distributed to other members of the team in advance of the meeting so that everyone has time to become familiar with the material. The review leader coordinates the meeting time, the distribution of handouts, and so forth.

During the meeting, the leader handles ensuring that the meeting runs smoothly while the scribe takes notes about what happens. The designers are responsible for presenting the component design. A top-down presentation often works well, beginning with the requirements and interface description, followed by the overall structure

of the component, the details, and then the testing strategy. The audience should look for all types of problems at every level of detail:

- Is the design team's view of the component's specification consistent with the overall system specification, or has the team misinterpreted something?
- Is the interface specification correct?
- Does the component's internal architecture work well?
- Are there coding errors in the component?
- Is the testing strategy adequate?

The notes taken by the scribe are used in the meeting follow-up. The design team should correct bugs and address the concerns raised at the meeting. While doing so, the team should keep notes describing what they did. The design review leader coordinates with the design team both to make sure that the changes are made and to distribute the change results to the audience. If the changes are straightforward, then a written report is adequate. If the errors found during the review caused major reworking of the component, a new design review meeting for the new implementation using as many of the original team members as possible may be useful.

7.6.3 Safety-oriented methodologies

We start with an example that describes the serious safety problems of one computer-controlled medical system. Medical equipment, like aviation electronics, is a safety-critical application; unfortunately, this medical equipment caused deaths before its design errors were properly understood. This example also allows us to use specification techniques to understand software design problems.

Example 7.6: The Therac-25 Medical Imaging System

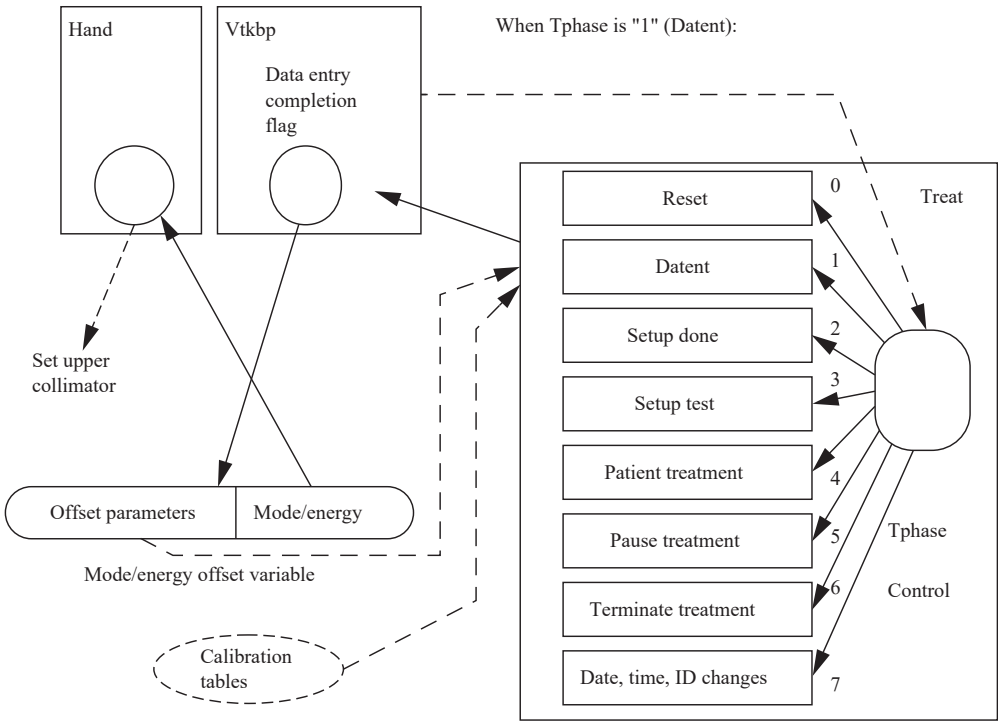
The Therac-25 medical imaging system caused what Leveson and Turner called the most serious computer-related accidents to date (at least nonmilitary and admitted) [Lev93]. During six known accidents, these machines delivered massive radiation overdoses, causing deaths and serious injuries. Leveson and Turner analyzed the Therac-25 system and the causes of these accidents.

Therac-25 was controlled by a programmed data processor (PDP)-11 minicomputer. The computer was responsible for controlling a radiation gun that delivered a dose of radiation to the patient. It also runs a terminal that presents the main user interface. The machine's software was developed by a single programmer in PDP-11 assembly language over several years. The software includes four major components: stored data, a scheduler, a set of tasks, and interrupt services. The three major critical tasks in the system are as follows:

- A treatment monitor controls and monitors the setup and delivery of the treatment in eight phases.
- A servo task controls the radiation gun, machine motions, and so on.
- A housekeeper task takes care of system status interlocks and limit checks. A limit check determines whether some system parameters have gone beyond preset limits.

The code was relatively crude: the software allowed several processes access to shared memory, there was no synchronization mechanism aside from shared variables, and the test-and-set for shared variables were not indivisible operations.

Let's examine the software problems responsible for one series of accidents. Leveson and Turner reverse-engineered a specification for the relevant software as follows:



- *Treat* is a treatment monitor task that is divided into eight subroutines (*Reset, Datent,* and others).
- *Tphase* is a variable that controls which of these subroutines are currently executing. *Treat* reschedules itself after the execution of each subroutine.
- The *Datent* subroutine communicates with the keyboard entry task via the data entry completion flag, which is a shared variable. *Datent* looks at this flag to determine when it should leave the data entry mode and go to the *Setup test* mode.
- The *Mode/energy offset* variable is a shared variable: The top byte holds offset parameters used by the *Datent* subroutine, and the low-order byte holds mode and energy offset used by the *Hand task*.

When the machine is run, the operator is forced to enter the mode and energy (there is one mode in which the energy is set to a default). However, the operator can later edit the mode and energy separately. The software's behavior is timing-dependent. If the keyboard handler sets the completion variable before the operator changes the *Mode/energy* data, the *Datent* task will not detect the change. Once *Treat* leaves *Datent*, it will not enter that subroutine again during treatment. However, the *Hand* task, which runs concurrently, will see the new *Mode/energy* information. Apparently, the software included no checks to detect incompatible data.

After the *Mode/energy* data are set, the software sends parameters to a digital/analog converter and then calls a *Magnet* subroutine to set the bending magnets. Setting the magnets takes about 8 s, and a subroutine, *Ptime*, is used to introduce a time delay. Owing to the way that *Datent*, *Magnet*, and *Ptime* are written, it is possible that changes to the parameters made by the user can be shown on the screen but will not be sensed by *Datent.* One accident occurred when the operator initially entered *Mode/energy*, went to the command line, changed *Mode/energy*, and returned to the command line within 8 s. The error therefore depended on the typing speed of the operator. Because operators become faster and more skillful with the machine over time, this error is likelier to occur with experienced operators.

Leveson and Turner emphasized that several poor design methodologies and flawed architectures were at the root of the bugs that led to the accidents:

- The designers performed a limited safety analysis. For example, low probabilities were assigned to certain errors with no apparent justification.
- Mechanical backups were not used to check the operation of the machine (e.g., testing beam energy), although such backups were employed in earlier models of the machine.
- Programmers created overly complex programs based on unreliable coding styles.

In summary, the designers of Therac-25 relied on system testing with insufficient module testing or formal analysis.

Several methodologies have been developed to specifically address the design of safety-critical systems. Some of these methodologies have been codified in standards that have been adopted by various industrial organizations. Safety-oriented methodologies supplement other aspects of methodology to apply techniques that have been demonstrated to improve the safety of the resulting systems.

Availability

Because dependability often depends on hidden characteristics that can only be modeled stochastically, **availability** is the term for the probability of a system's correct operation over time. A detailed discussion of dependability is beyond our scope here, but a common stochastic model for reliability is an exponential distribution:

$$R(t) = e^{-\lambda t} \qquad \text{(Eq. 7.1)}$$

The parameter, λ, is the number of failures per unit time.

Safety analysis phases

Safety-oriented methodologies include three phases:

- **Hazard analysis** determines the types of safety-related problems that may occur.
- **Risk assessment** analyzes the effects of hazards, such as the severity or likelihood of injury.
- **Risk mitigation** modifies the design to improve the system's response to identified hazards.

Several standards have been developed for the design of safety-critical systems. Many of these standards focus on applications, such as aviation or automotive. Standards may also cover different levels of abstraction, with some covering earlier phases of design, while others concentrate on coding.

ISO 26262

ISO 26262 [ISO18A] is a standard that governs functional safety management for automotive electrics and electronics. The standard describes how to assess hazards, identify methods to reduce risks, and track safety requirements through to the

delivered product. Hazard analysis and risk assessment result in an Automotive Safety Integrity Level (ASIL) [ISO18C]. Events are assigned a severity classification for the degree of injury expected, an exposure classification describing how likely a person is to be exposed to the event, and a controllability classification for how well people can react to control the hazard.

DO-178C

DO-178C [RTC11] recommends safety-related procedures for airborne software. Hazard analysis results in the assignment of each failure condition to a **software level**: *A* catastrophic, *B* hazardous, *C* major, *D* minor, or *E* no effect. The standard requires that the generated safety-related requirements be traceable in software implementation and testing.

SAE standards

The Society of Automotive Engineers (SAE) has several standards for automotive software: J2632 for coding practices for C code, J2516 for software development lifecycle, J2640 for software design requirements, and J2734 for software verification and validation.

Coding standards

Several coding standards have been developed to specify specific ways of writing software that increase software reliability. Coding standards inevitably target specific programming languages, but a consideration of some specific examples of coding standards helps us understand the role they play in reliability and quality assurance.

MISRA

The Motor Industry Software Reliability Association (MISRA) formulated a series of standards for software coding in automotive and other critical systems. MISRA C:2012 [MIS13] is an updated version of the coding standards for the C programming language while MISRA C++ [MIS08] is a set of standards for C++. Both standards provide directives and rules for how programs in these languages are to be written. They also place these guidelines in the context of overall development methodologie. The guidelines are to be used as part of a documented software process.

The MISRA C standard gives a set of general directives, some of which are general (e.g., traceability of code to requirements), and others are more specific (e.g., code should compile without errors). It also provides a set of more detailed rules. For example, a project should not contain unreachable or dead code. (Unreachable code can never be executed, whereas dead code can be executed, but has no effect.) As another example, all function types are required to be in prototype form. Early versions of C did not require the program to declare the types of a function's arguments or its return. Later versions of C created *function prototypes* that included type information. This rule requires that the prototype form always be used.

These standards are intended to be enforced with the help of tools. Relying on manual methods, such as design reviews, to enforce a large number of very detailed rules in these standards would be unwieldy. Commercial tools have been developed that specifically check for these rules and generate reports that can be used to document the design process.

CERT C and 17961

CERT C [Sea14] is a standard for coding in C-language programs. It does not directly target embedded computing. Its rules can be divided into 14 categories, including topics such as memory management, expressions, integers, floating point,

arrays, strings, and error handling. ISO/IEC TS 17961 [ISO13] is a standard for secure coding in C.

7.6.4 Security

Security

Safety-critical designs must also be secure. Devices with user accounts and Internet access provide obvious avenues for attacks. However, attacks may also come from more indirect sources.

Air gaps

The **air gap myth** is a continuing source of vulnerabilities. The notion that we can build a system with an *air gap* (no direct Internet connection) is naïve and unrealistic in modern embedded computing systems. A variety of devices and techniques can be used to carry infections onto embedded processors.

The next example discusses the first cyber-physical attack, Stuxnet. The attacked system was air-gapped, but was still successfully attacked.

Example 7.7: Stuxnet

The name "Stuxnet" was given to a series of attacks on Iranian nuclear processing facilities [McD13, Fal10, Fal11]. The facilities were not directly connected to the Internet, but were air-gapped. However, the facility's computers were infected by workers who used USB devices that had been infected while connected to outside machines; those workers were likely to use software tools carried on USB memory devices to conduct standard software operations.

The attacks targeted a particular type of programmable logic controller (PLC) used to control centrifuges that were part of the nuclear processing equipment. The PLCs were programmed using PCs. A PC infected with Stuxnet executed two dynamically linked libraries to attack the PLC software: one that identified PLC code for attack (a process known as **fingerprinting**) and another that modified the PLC's programming.

The PLC software was modified to improperly operate the centrifuges. The attack code had some knowledge of the structure of the nuclear processing equipment and which centrifuges to attack. Stuxnet used **replay** to hide its attacks: it first recorded the centrifuge's output during nonfaulty behavior, and then replayed that behavior while it maliciously operated the centrifuge. In addition, Stuxnet modified the PLC software to hide the existence of PLC modifications.

These attacks caused extensive damage to the nuclear processing facilities and seriously compromised their ability to operate.

Graff and van Wyck [Gra03] recommended several methodological steps to help improve the security of a program. Assessing threats and the risks posed by those threats is an important early step in program design. The **attack surface** of a program is the set of program locations and use cases in which it can be attacked. Some applications have naturally small attack surfaces, whereas others inherently expose much larger attack surfaces. As we will see in Chapter 9, remote telematics interfaces to cars provide large attack surfaces. Once risks are identified, several methods can be used to mitigate them, such as avoiding using the application in certain circumstances, performing checks to limit risks, and so forth. Finding unusual metaphors for a program's operations may help identify potential weaknesses.

7.7 Summary

System design takes a comprehensive view of the application and the system under design. To ensure that we design an acceptable system, we must understand the application and its requirements. Numerous techniques, such as object-oriented design, can be used to create useful architecture from the system's original requirements. Along the way, by measuring our design processes, we can gain a clearer understanding of where bugs are introduced, how to fix them, and how to avoid introducing them in the future.

What we learned

- Design methodologies and flows can be organized in many different ways.
- A variety of methods can be used to elicit requirements.
- System modeling can capture both the functional and nonfunctional aspects of a system.
- Dependability, safety, and security are related attributes of embedded systems and concerns during design.

Further reading

Pressman [Pre97] provides a thorough introduction to software engineering. Davis [Dav90] provides a good survey of software requirements. Beizer [Bei84] surveys system-level testing techniques. Leveson [Lev86] provides a good introduction to software safety. Schmauch [Sch94] and Jenner [Jen95] both described ISO 9000 for software development. A tutorial edited by Chow [Cho85] includes many important early papers on software quality assurance. Cusumano [Cus91] provides a fascinating account of software factories in both the United States and Japan.

Questions

Q7-1 Provide realistic examples of how a requirements document may be:

a. ambiguous
b. incorrect
c. incomplete
d. unverifiable

Q7-2 How can poor specifications lead to poor quality code—do aspects of a poorly constructed specification necessarily lead to bad software?

Q7-3 What are the main phases of a design review?

Q7-4 What is an example dependability measure?

Q7-5 What are example types of functional, safety-related bugs were found in Therac-25?

Q7-6 What are example types of nonfunctional, safety-related bugs were found in Therac-25?

Q7-7 What are example types of methodological, safety-related problems were found in Therac-25?

Q7-8 How could a replay attack be used against an automobile?

Lab exercises

L7-1 Draw a diagram showing the developmental steps of one of the projects you recently designed.

L7-2 Find a detailed description of a system of interest to you. Write your own description of what it does and how it works.