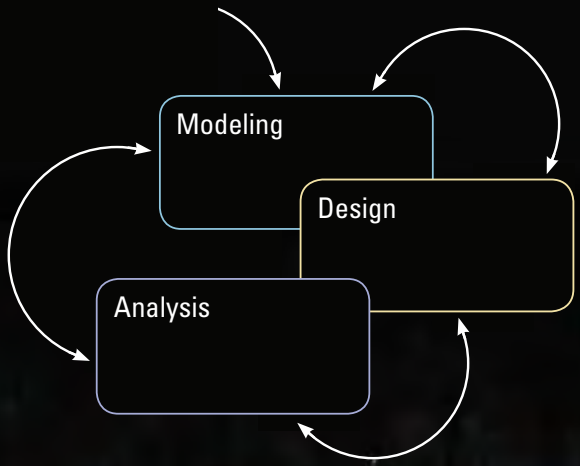
Edward Ashford Lee and
Sanjit Arunkumar Seshia

INTRODUCTION TO EMBEDDED SYSTEMS

A CYBER-PHYSICAL SYSTEMS APPROACH

Second Edition



Copyright © 2017
Edward Ashford Lee & Sanjit Arunkumar Seshia

This work is licensed under a Creative Commons
Attribution-NonCommercial-NoDerivatives 4.0 International License.

Second Edition, Version 2.2

ISBN: 978-0-262-53381-2

Please cite this book as:

E. A. Lee and S. A. Seshia,
Introduction to Embedded Systems - A Cyber-Physical Systems Approach,
Second Edition, MIT Press, 2017.

This book is dedicated to our families.

Contents

Preface

What This Book Is About

The most visible use of computers and software is processing information for human consumption. We use them to write books (like this one), search for information on the web, communicate via email, and keep track of financial data. The vast majority of computers in use, however, are much less visible. They run the engine, brakes, seatbelts, airbag, and audio system in your car. They digitally encode your voice and construct a radio signal to send it from your cell phone to a base station. They control your microwave oven, refrigerator, and dishwasher. They run printers ranging from desktop inkjet printers to large industrial high-volume printers. They command robots on a factory floor, power generation in a power plant, processes in a chemical plant, and traffic lights in a city. They search for microbes in biological samples, construct images of the inside of a human body, and measure vital signs. They process radio signals from space looking for supernovae and for extraterrestrial intelligence. They bring toys to life, enabling them to react to human touch and to sounds. They control aircraft and trains. These less visible computers are called **embedded systems**, and the software they run is called **embedded software**.

Despite this widespread prevalence of embedded systems, computer science has, throughout its relatively short history, focused primarily on information processing. Only recently have embedded systems received much attention from researchers. And only recently has

the community recognized that the engineering techniques required to design and analyze these systems are distinct. Although embedded systems have been in use since the 1970s, for most of their history they were seen simply as small computers. The principal engineering problem was understood to be one of coping with limited resources (limited processing power, limited energy sources, small memories, etc.). As such, the engineering challenge was to optimize the designs. Since all designs benefit from optimization, the discipline was not distinct from anything else in computer science. It just had to be more aggressive about applying the same optimization techniques.

Recently, the community has come to understand that the principal challenges in embedded systems stem from their interaction with physical processes, and not from their limited resources. The term cyber-physical systems (CPS) was coined by Helen Gill at the National Science Foundation in the U.S. to refer to the integration of computation with physical processes. In CPS, embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa. The design of such systems, therefore, requires understanding the joint dynamics of computers, software, networks, and physical processes. It is this study of *joint* dynamics that sets this discipline apart.

When studying CPS, certain key problems emerge that are rare in so-called general-purpose computing. For example, in general-purpose software, the time it takes to perform a task is an issue of *performance*, not *correctness*. It is not incorrect to take longer to perform a task. It is merely less convenient and therefore less valuable. In CPS, the time it takes to perform a task may be critical to correct functioning of the system. In the physical world, as opposed to the cyber world, the passage of time is inexorable.

In CPS, moreover, many things happen at once. Physical processes are compositions of many things going on at once, unlike software processes, which are deeply rooted in sequential steps. Abelson and Sussman (1996) describe computer science as "procedural epistemology," knowledge through procedure. In the physical world, by contrast, processes are rarely procedural. Physical processes are compositions of many parallel processes. Measuring and controlling the dynamics of these processes by orchestrating actions that influence the processes are the main tasks of embedded systems. Consequently, concurrency is intrinsic in CPS. Many of the technical challenges in designing and analyzing embedded software stem from the need to bridge an inherently sequential semantics with an intrinsically concurrent physical world.

Why We Wrote This Book

The mechanisms by which software interacts with the physical world are changing rapidly. Today, the trend is towards "smart" sensors and actuators, which carry microprocessors, network interfaces, and software that enables remote access to the sensor data and remote activation of the actuator. Called variously the Internet of Things (IoT), Industry 4.0, the Industrial Internet, Machine-to-Machine (M2M), the Internet of Everything, the Smarter Planet, TSensors (Trillion Sensors), or The Fog (like The Cloud, but closer to the ground), the vision is of a technology that deeply connects our physical world with our information world. In the IoT world, the interfaces between these worlds are inspired by and derived from information technology, particularly web technology.

IoT interfaces are convenient, but not yet suitable for tight interactions between the two worlds, particularly for real-time control and safety-critical systems. Tight interactions still require technically intricate, low-level design. Embedded software designers are forced to struggle with interrupt controllers, memory architectures, assembly-level programming (to exploit specialized instructions or to precisely control timing), device driver design, network interfaces, and scheduling strategies, rather than focusing on specifying desired behavior.

The sheer mass and complexity of these technologies (at both the high level and the low level) tempts us to focus an introductory course on mastering them. But a better introductory course would focus on how to model and design the joint dynamics of software, networks, and physical processes. Such a course would present the technologies only as today's (rather primitive) means of accomplishing those joint dynamics. This book is our attempt at a textbook for such a course.

Most texts on embedded systems focus on the collection of technologies needed to get computers to interact with physical systems (Barr and Massa, 2006; Berger, 2002; Burns and Wellings, 2001; Kamal, 2008; Noergaard, 2005; Parab et al., 2007; Simon, 2006; Valvano, 2007; Wolf, 2000). Others focus on adaptations of computer-science techniques (like programming languages, operating systems, networking, etc.) to deal with technical problems in embedded systems (Buttazzo, 2005a; Edwards, 2000; Pottie and Kaiser, 2005). While these implementation technologies are (today) necessary for system designers to get embedded systems working, they do not form the intellectual core of the discipline. The intellectual core is instead in models and abstractions that conjoin computation and physical dynamics.

A few textbooks offer efforts in this direction. Jantsch (2003) focuses on concurrent models of computation, Marwedel (2011) focuses on models of software and hardware behavior, and Sriram and Bhattacharyya (2009) focus on dataflow models of signal processing behavior and their mapping onto programmable DSPs. Alur (2015) focuses on formal modeling, specification, and verification of cyber-physical systems. These are excellent textbooks that cover certain topics in depth. Models of concurrency (such as dataflow) and abstract models of software (such as Statecharts) provide a better starting point than imperative programming languages (like C), interrupts and threads, and architectural annoyances that a designer must work around (like caches). These texts, however, do not address all the needs of an introductory course. They are either too specialized or too advanced or both. This book is our attempt to provide an introductory text that follows the spirit of focusing on models and their relationship to realizations of systems.

The major theme of this book is on models and their relationship to realizations of systems. The models we study are primarily about dynamics, the evolution of a system state in time. We do not address structural models, which represent static information about the construction of a system, although these too are important to embedded system design.

Working with models has a major advantage. Models can have formal properties. We can say definitive things about models. For example, we can assert that a model is determinate, meaning that given the same inputs it will always produce the same outputs. No such absolute assertion is possible with any physical realization of a system. If our model is a good abstraction of the physical system (here, "good abstraction" means that it omits only inessential details), then the definitive assertion about the model gives us confidence in the physical realization of the system. Such confidence is hugely valuable, particularly for embedded systems where malfunctions can threaten human lives. Studying models of systems gives us insight into how those systems will behave in the physical world.

Our focus is on the interplay of software and hardware with the physical environment in which they operate. This requires explicit modeling of the temporal dynamics of software and networks and explicit specification of concurrency properties intrinsic to the application. The fact that the implementation technologies have not yet caught up with this perspective should not cause us to teach the wrong engineering approach. We should teach design and modeling as it should be, and enrich this with a *critical* presentation of how it is. Embedded systems technologies today, therefore, should not be presented dispassionately as a collection of facts and tricks, as they are in many of the above cited books, but rather as stepping stones towards a sound design practice. The focus should be on what that sound design practice is, and on how today's technologies both impede and achieve it.

Stankovic et al. (2005) support this view, stating that "existing technology for RTES [real-time embedded systems] design does not effectively support the development of reliable and robust embedded systems." They cite a need to "raise the level of programming abstraction." We argue that raising the level of abstraction is insufficient. We also have to fundamentally change the abstractions that are used. Timing properties of software, for example, cannot be effectively introduced at higher levels of abstraction if they are entirely absent from the lower levels of abstraction on which these are built.

We require robust and predictable designs with repeatable temporal dynamics (Lee, 2009a). We must do this by building abstractions that appropriately reflect the realities of cyber-physical systems. The result will be CPS designs that can be much more sophisticated, including more adaptive control logic, evolvability over time, and improved safety and reliability, all without suffering from the brittleness of today's designs, where small changes have big consequences.

In addition to dealing with temporal dynamics, CPS designs invariably face challenging concurrency issues. Because software is so deeply rooted in sequential abstractions, concurrency mechanisms such as interrupts and multitasking, using semaphores and mutual exclusion, loom large. We therefore devote considerable effort in this book to developing a critical understanding of threads, message passing, deadlock avoidance, race conditions, and data determinism.

Note about This Edition

This is the second edition of the textbook. In addition to several bug fixes and improvements to presentation and wording, it includes two new chapters. Chapter 7 covers sensors and actuators with an emphasis on modeling. Chapter 17 covers the basics of security and privacy for embedded systems.

What Is Missing

Even with the new additions, this version of the book is not complete. It is arguable, in fact, that complete coverage of embedded systems in the context of CPS is impossible. Specific topics that we cover in the undergraduate Embedded Systems course at Berkeley (see http://LeeSeshia.org) and hope to include in future versions of this book include networking, fault tolerance, simulation techniques, control theory, and hardware/software codesign.

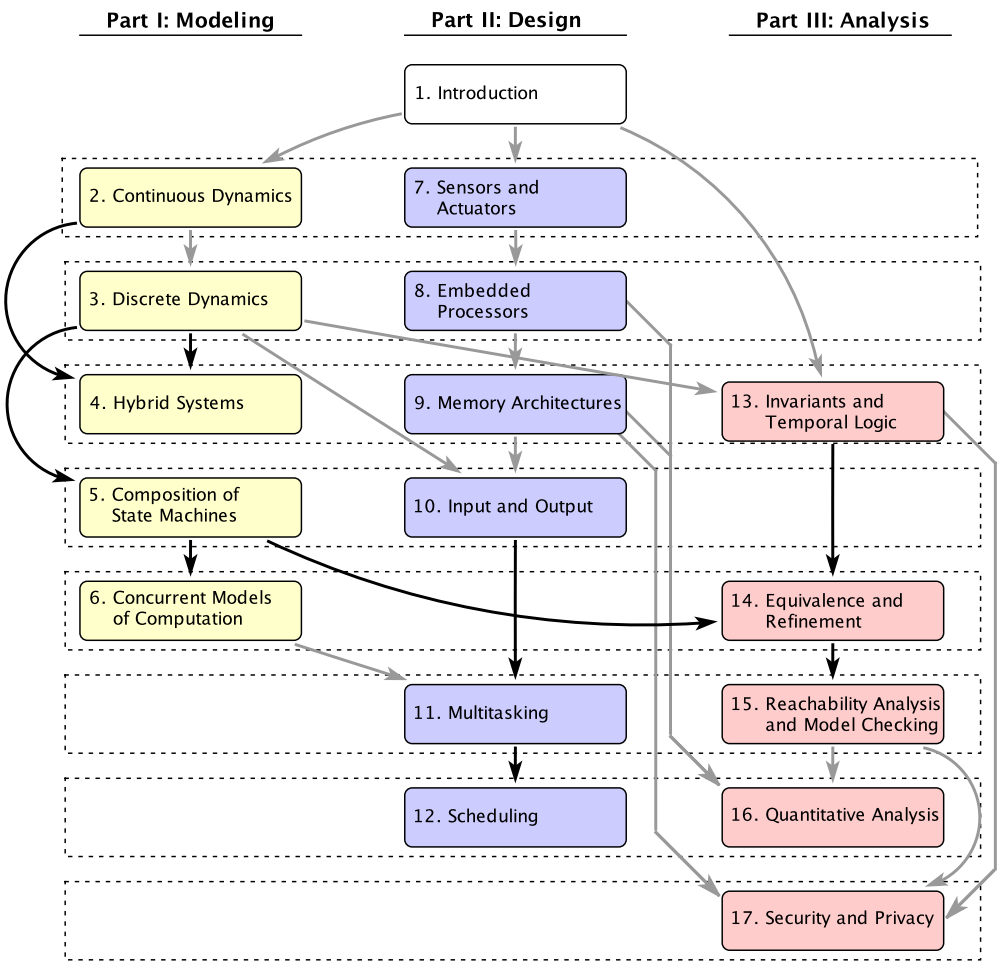


Figure 1: Map of the book with strong and weak dependencies between chapters. Strong dependencies between chapters are shown with arrows in black. Weak dependencies are shown in grey. When there is a weak dependency from chapter i to chapter j, then j may mostly be read without reading i, at most requiring skipping some examples or specialized analysis techniques.

How to Use This Book

This book is divided into three major parts, focused on modeling, design, and analysis, as shown in Figure 1. The three parts of the book are relatively independent of one another and are largely meant to be read concurrently. A systematic reading of the text can be accomplished in eight segments, shown with dashed outlines. Most segments include two chapters, so complete coverage of the text is possible in a 15 week semester, allowing two weeks for most modules.

The appendices provide background material that is well covered in other textbooks, but which can be quite helpful in reading this text. Appendix A reviews the notation of sets and functions. This notation enables a higher level of precision than is common in the study of embedded systems. Appendix B reviews basic results in the theory of computability and complexity. This facilitates a deeper understanding of the challenges in modeling and analysis of systems. Note that Appendix B relies on the formalism of state machines covered in Chapter 3, and hence should be read after reading Chapter 3.

In recognition of recent advances in technology that are fundamentally changing the technical publishing industry, this book is published in a non-traditional way. At least the present version is available free in the form of PDF file designed specifically for reading on tablet computers. It can be obtained from the website http://LeeSeshia.org. The layout is optimized for medium-sized screens, particularly laptop computers and the iPad and other tablets. Extensive use of hyperlinks and color enhance the online reading experience.

We attempted to adapt the book to e-book formats, which, in theory, enable reading on various sized screens, attempting to take best advantage of the available screen. However, like HTML documents, e-book formats use a reflow technology, where page layout is recomputed on the fly. The results are highly dependent on the screen size and prove ludicrous on many screens and suboptimal on all. As a consequence, we have opted for controlling the layout, and we do not recommend attempting to read the book on an smartphone.

Although the electronic form is convenient, we recognize that there is real value in a tangible manifestation on paper, something you can thumb through, something that can live on a bookshelf to remind you of its existence. This edition is published by MIT Press, who has assured us that they will keep the book affordable.

Two disadvantages of print media compared to electronic media are the lack of hyperlinks and the lack of text search. We have attempted to compensate for those limitations by providing page number references in the margins whenever a term is used that is defined elsewhere. The term that is defined elsewhere is underlined with a discrete light gray line. In addition, we have provided an extensive index, with more than 2,000 entries.

There are typographic conventions worth noting. When a term is being defined, it will appear in **bold face**, and the corresponding index entry will also be in bold face. Hyperlinks are shown in blue in the electronic version. The notation used in diagrams, such as those for finite-state machines, is intended to be familiar, but not to conform with any particular programming or modeling language.

Intended Audience

This book is intended for students at the advanced undergraduate level or introductory graduate level, and for practicing engineers and computer scientists who wish to understand the engineering principles of embedded systems. We assume that the reader has some exposure to machine structures (e.g., should know what an ALU is), computer programming (we use C throughout the text), basic discrete mathematics and algorithms, and at least an appreciation for signals and systems (what it means to sample a continuous-time signal, for example).

Reporting Errors

If you find errors or typos in this book, or if you have suggestions for improvements or other comments, please send email to:

authors@leeseshia.org

Please include the version number of the book, whether it is the electronic or the hard-copy distribution, and the relevant page numbers. Thank you!

Acknowledgments

The authors gratefully acknowledge contributions and helpful suggestions from Murat Arcak, Dai Bui, Janette Cardoso, Gage Eads, Stephen Edwards, Suhaib Fahmy, Shanna-Shaye Forbes, Daniel Holcomb, Jeff C. Jensen, Garvit Juniwal, Hokeun Kim, Jonathan Kotker, Wenchao Li, Isaac Liu, Slobodan Matic, Mayeul Marcadella, Le Ngoc Minh, Christian Motika, Chris Myers, Steve Neuendorffer, David Olsen, Minxue Pan, Hiren Patel, Jan Reineke, Rhonda Righter, Alberto Sangiovanni-Vincentelli, Chris Shaver, Shih-Kai Su (together with students in CSE 522, lectured by Dr. Georgios E. Fainekos at Arizona State University), Stavros Tripakis, Pravin Varaiya, Reinhard von Hanxleden, Armin Wasicek, Kevin Weekly, Maarten Wiggers, Qi Zhu, and the students in UC Berkeley's EECS 149 class over the past years, particularly Ned Bass and Dan Lynch. The authors are especially grateful to Elaine Cheong, who carefully read most chapters and offered helpful editorial suggestions. We also acknowledge the bug fixes and suggestions sent in by several readers which has helped us improve the book since its initial publication. We give special thanks to our families for their patience and support, particularly to Helen, Katalina, and Rhonda (from Edward), and Amma, Appa, Ashwin, Bharathi, Shriya, and Viraj (from Sanjit).

This book is almost entirely constructed using open-source software. The typesetting is done using LaTeX, and many of the figures are created using Ptolemy II. See:

http://ptolemy.org

Further Reading

Many textbooks on embedded systems have appeared in recent years. These books approach the subject in surprisingly diverse ways, often reflecting the perspective of a more established discipline that has migrated into embedded systems, such as VLSI design, control systems, signal processing, robotics, real-time systems, or software engineering. Some of these books complement the present one nicely. We strongly recommend them to the reader who wishes to broaden his or her understanding of the subject.

Specifically, Patterson and Hennessy (1996), although not focused on embedded processors, is the canonical reference for computer architecture, and a must-read for anyone interested embedded processor architectures. Sriram and Bhattacharyya (2009) focus on signal processing applications, such as wireless communications and digital media, and give particularly thorough coverage to dataflow programming methodologies. Wolf (2000) gives an excellent overview of hardware design techniques and microprocessor architectures and their implications for embedded software design. Mishra and Dutt (2005) give a view of embedded architectures based on architecture description languages (ADLs). Oshana (2006) specializes in DSP processors from Texas Instruments, giving an overview of architectural approaches and a sense of assembly-level programming.

Focused more on software, Buttazzo (2005a) is an excellent overview of scheduling techniques for real-time software. Liu (2000) gives one of the best treatments yet of techniques for handling sporadic real-time events in software. Edwards (2000) gives a good overview of domain-specific higher-level programming languages used in some embedded system designs. Pottie and Kaiser (2005) give a good overview of networking technologies, particularly wireless, for embedded systems. Koopman (2010) focuses on design process for embedded software, including requirements management, project management, testing plans, and security plans. Alur (2015) provides an excellent, in-depth treatment of formal modeling and verification of cyber-physical systems.

No single textbook can comprehensively cover the breadth of technologies available to the embedded systems engineer. We have found useful information in many of the books that focus primarily on today's design techniques (Barr and Massa, 2006; Berger, 2002; Burns and Wellings, 2001; Gajski et al., 2009; Kamal, 2008; Noergaard, 2005; Parab et al., 2007; Simon, 2006; Schaumont, 2010; Vahid and Givargis, 2010).

Notes for Instructors

At Berkeley, we use this text for an advanced undergraduate course called *Introduction to Embedded Systems*. A great deal of material for lectures and labs can be found via the main web page for this text:

http://leeseshia.org

In addition, a solutions manual and other instructional material are available to qualified instructors at bona fide teaching institutions. See

http://chess.eecs.berkeley.edu/instructors/

or contact authors@leeseshia.org.

1 Introduction

A **cyber-physical system** (**CPS**) is an integration of computation with physical processes whose behavior is defined by *both* cyber and physical parts of the system. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa. As an intellectual challenge, CPS is about the *intersection*, not the union, of the physical and the cyber. It is not sufficient to separately understand the physical components and the computational components. We must instead understand their interaction.

In this chapter, we use a few CPS applications to outline the engineering principles of such systems and the processes by which they are designed.

1.1 Applications

CPS applications arguably have the potential to eclipse the 20th century information technology (IT) revolution. Consider the following examples.

Example 1.1: Heart surgery often requires stopping the heart, performing the surgery, and then restarting the heart. Such surgery is extremely risky and carries many detrimental side effects. A number of research teams have been working on an alternative where a surgeon can operate on a beating heart rather than stopping the heart. There are two key ideas that make this possible. First, surgical tools can be robotically controlled so that they move with the motion of the heart (Kremen, 2008). A surgeon can therefore use a tool to apply constant pressure to a point on the heart while the heart continues to beat. Second, a stereoscopic video system can present to the surgeon a video illusion of a still heart (Rice, 2008). To the surgeon, it looks as if the heart has been stopped, while in reality, the heart continues to beat. To realize such a surgical system requires extensive modeling of the heart, the tools, the computational hardware, and the software. It requires careful design of the software that ensures precise timing and safe fallback behaviors to handle malfunctions. And it requires detailed analysis of the models and the designs to provide high confidence.

Example 1.2: Consider a city where traffic lights and cars cooperate to ensure efficient flow of traffic. In particular, imagine never having to stop at a red light unless there is actual cross traffic. Such a system could be realized with expensive infrastructure that detects cars on the road. But a better approach might be to have the cars themselves cooperate. They track their position and communicate to cooperatively use shared resources such as intersections. Making such a system reliable, of course, is essential to its viability. Failures could be disastrous.

Example 1.3: Imagine an airplane that refuses to crash. While preventing all possible causes of a crash is not possible, a well-designed flight control system can prevent certain causes. The systems that do this are good examples of cyber-physical systems.

In traditional aircraft, a pilot controls the aircraft through mechanical and hydraulic linkages between controls in the cockpit and movable surfaces on the wings and tail of the aircraft. In a **fly-by-wire** aircraft, the pilot commands are mediated by a flight computer and sent electronically over a network to actuators in the wings and tail. Fly-by-wire aircraft are much lighter than traditional aircraft, and therefore more fuel efficient. They have also proven to be more reliable. Virtually all new aircraft designs are fly-by-wire systems.

In a fly-by-wire aircraft, since a computer mediates the commands from the pilot, the computer can modify the commands. Many modern flight control systems modify pilot commands in certain circumstances. For example, commercial airplanes made by Airbus use a technique called **flight envelope protection** to prevent an airplane from going outside its safe operating range. They can prevent a pilot from causing a stall, for example.

The concept of flight envelope protection could be extended to help prevent certain other causes of crashes. For example, the **soft walls** system proposed by Lee (2001), if implemented, would track the location of the aircraft on which it is installed and prevent it from flying into obstacles such as mountains and buildings. In Lee's proposal, as an aircraft approaches the boundary of an obstacle, the fly-by-wire flight control system creates a virtual pushing force that forces the aircraft away. The pilot feels as if the aircraft has hit a soft wall that diverts it. There are many challenges, both technical and non-technical, to designing and deploying such a system. See Lee (2003) for a discussion of some of these issues.

Although the soft walls system of the previous example is rather futuristic, there are modest versions in automotive safety that have been deployed or are in advanced stages of research and development. For example, many cars today detect inadvertent lane changes and warn the driver. Consider the much more challenging problem of automatically correcting the driver's actions. This is clearly much harder than just warning the driver.

How can you ensure that the system will react and take over only when needed, and only exactly to the extent to which intervention is needed?

It is easy to imagine many other applications, such as systems that assist the elderly; telesurgery systems that allow a surgeon to perform an operation at a remote location; and home appliances that cooperate to smooth demand for electricity on the power grid. Moreover, it is easy to envision using CPS to improve many existing systems, such as robotic manufacturing systems; electric power generation and distribution; process control in chemical factories; distributed computer games; transportation of manufactured goods; heating, cooling, and lighting in buildings; people movers such as elevators; and bridges that monitor their own state of health. The impact of such improvements on safety, energy consumption, and the economy is potentially enormous.

Many of the above examples will be deployed using a structure like that sketched in Figure 1.1. There are three main parts in this sketch. First, the **physical plant** is the "physical" part of a cyber-physical system. It is simply that part of the system that is not realized with computers or digital networks. It can include mechanical parts, biological or chemical processes, or human operators. Second, there are one or more computational **platform**s, which consist of sensors, actuators, one or more computers, and (possibly) one or more operating systems. Third, there is a **network fabric**, which provides the mechanisms for the computers to communicate. Together, the platforms and the network fabric form the "cyber" part of the cyber-physical system.

Figure 1.1 shows two networked platforms each with its own sensors and/or actuators. The action taken by the actuators affects the data provided by the sensors through the physical plant. In the figure, Platform 2 controls the physical plant via Actuator 1. It measures the processes in the physical plant using Sensor 2. The box labeled Computation 2 implements a **control law**, which determines based on the sensor data what commands to issue to the actuator. Such a loop is called a **feedback control** loop. Platform 1 makes additional measurements using Sensor 1, and sends messages to Platform 2 via the network fabric. Computation 3 realizes an additional control law, which is merged with that of Computation 2, possibly preempting it.

Example 1.4: Consider a high-speed printing press for a print-on-demand service. This might be structured similarly to Figure 1.1, but with many more platforms, sensors, and actuators. The actuators may control motors that drive paper through the press and ink onto the paper. The control laws may include a strategy

About the Term "Cyber-Physical Systems"

The term "cyber-physical systems" emerged in 2006, coined by Helen Gill at the National Science Foundation in the US. We may be tempted to associate the term "**cyberspace**" with CPS, but the roots of the term CPS are older and deeper. It would be more accurate to view the terms "cyberspace" and "cyber-physical systems" as stemming from the same root, "**cybernetics**," rather than viewing one as being derived from the other.

The term "cybernetics" was coined by Norbert Wiener (Wiener, 1948), an American mathematician who had a huge impact on the development of control systems theory. During World War II, Wiener pioneered technology for the automatic aiming and firing of anti-aircraft guns. Although the mechanisms he used did not involve digital computers, the principles involved are similar to those used today in a huge variety of computer-based feedback control systems. Wiener derived the term from the Greek $\kappa\upsilon\beta\varepsilon\rho\nu\eta\tau\eta\varsigma$ (kybernetes), meaning helmsman, governor, pilot, or rudder. The metaphor is apt for control systems. Wiener described his vision of cybernetics as the conjunction of control and communication. His notion of control was deeply rooted in closed-loop feedback, where the control logic is driven by measurements of physical processes, and in turn drives the physical processes. Even though Wiener did not use digital computers, the control logic is effectively a computation, and therefore cybernetics is the conjunction of physical processes, computation, and communication. Wiener could not have anticipated the powerful effects of digital computation and networks. The fact that the term "cyber-physical systems" may be ambiguously interpreted as the conjunction of cyberspace with physical processes, therefore, helps to underscore the enormous impact that CPS will have. CPS leverages an information technology that far outstrips even the wildest dreams of Wiener's era.

The term CPS relates to the currently popular terms Internet of Things (IoT), Industry 4.0, the Industrial Internet, Machine-to-Machine (M2M), the Internet of Everything, TSensors (trillion sensors), and the Fog (like the Cloud, but closer to the ground). All of these reflect a vision of a technology that deeply connects our physical world with our information world. In our view, the term CPS is more foundational and durable than all of these, because it does not directly reference either implementation approaches (e.g., the "Internet" in IoT) nor particular applications (e.g., "Industry" in Industry 4.0). It focuses instead on the fundamental intellectual problem of conjoining the engineering traditions of the cyber and the physical worlds.

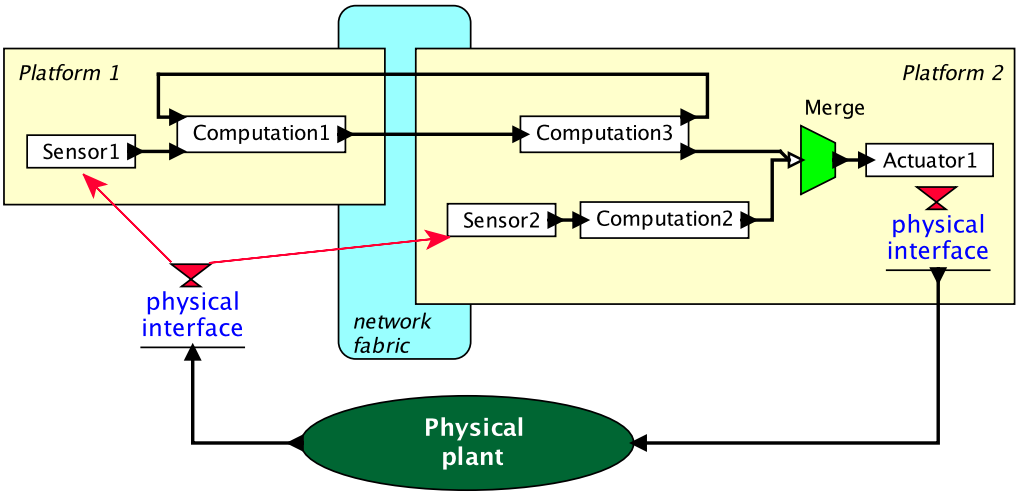


Figure 1.1: Example structure of a cyber-physical system.

for compensating for paper stretch, which will typically depend on the type of paper, the temperature, and the humidity. A networked structure like that in Figure 1.1 might be used to induce rapid shutdown to prevent damage to the equipment in case of paper jams. Such shutdowns need to be tightly orchestrated across the entire system to prevent disasters. Similar situations are found in high-end instrumentation systems and in energy production and distribution (Eidson et al., 2009).

1.2 Motivating Example

In this section, we describe a motivating example of a cyber-physical system. Our goal is to use this example to illustrate the importance of the breadth of topics covered in this text. The specific application is the Stanford testbed of autonomous rotorcraft for multi agent control (**STARMAC**), developed by Claire Tomlin and colleagues as a cooperative effort at Stanford and Berkeley (Hoffmann et al., 2004). The STARMAC is a small **quadrotor** aircraft; it is shown in flight in Figure 1.2. Its primary purpose is to serve as a testbed for

Figure 1.2: The STARMAC quadrotor aircraft in flight (reproduced with permission).

experimenting with multi-vehicle autonomous control techniques. The objective is to be able to have multiple vehicles cooperate on a common task.

There are considerable challenges in making such a system work. First, controlling the vehicle is not trivial. The main actuators are the four rotors, which produce a variable amount of downward thrust. By balancing the thrust from the four rotors, the vehicle can take off, land, turn, and even flip in the air. How do we determine what thrust to apply? Sophisticated control algorithms are required.

Second, the weight of the vehicle is a major consideration. The heavier it is, the more stored energy it needs to carry, which of course makes it even heavier. The heavier it is, the more thrust it needs to fly, which implies bigger and more powerful motors and rotors. The design crosses a major threshold when the vehicle is heavy enough that the rotors become dangerous to humans. Even with a relatively light vehicle, safety is a considerable concern, and the system needs to be designed with fault handling.

Third, the vehicle needs to operate in a context, interacting with its environment. It might, for example, be under the continuous control of a watchful human who operates it by remote control. Or it might be expected to operate autonomously, to take off, perform some mission, return, and land. Autonomous operation is enormously complex and challeng-

ing because it cannot benefit from the watchful human. Autonomous operation demands more sophisticated sensors. The vehicle needs to keep track of where it is (it needs to perform **localization**). It needs to sense obstacles, and it needs to know where the ground is. With good design, it is even possible for such vehicles to autonomously land on the pitching deck of a ship. The vehicle also needs to continuously monitor its own health, to detect malfunctions and react to them so as to contain the damage.

It is not hard to imagine many other applications that share features with the quadrotor problem. The problem of landing a quadrotor vehicle on the deck of a pitching ship is similar to the problem of operating on a beating heart (see Example 1.1). It requires detailed modeling of the dynamics of the environment (the ship, the heart), and a clear understanding of the interaction between the dynamics of the embedded system (the quadrotor, the robot) and its environment.

The rest of this chapter will explain the various parts of this book, using the quadrotor example to illustrate how the various parts contribute to the design of such a system.

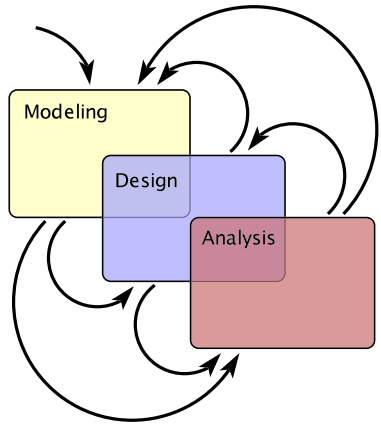


Figure 1.3: Creating embedded systems requires an iterative process of modeling, design, and analysis.

1.3 The Design Process

The goal of this book is to understand how to go about designing and implementing cyber-physical systems. Figure 1.3 shows the three major parts of the process, **modeling**, **design**, and **analysis**. Modeling is the process of gaining a deeper understanding of a system through imitation. Models imitate the system and reflect properties of the system. Models specify **what** a system does. Design is the structured creation of artifacts. It specifies **how** a system does what it does. Analysis is the process of gaining a deeper understanding of a system through dissection. It specifies **why** a system does what it does (or fails to do what a model says it should do).

As suggested in Figure 1.3, these three parts of the process overlap, and the design process iteratively moves among the three parts. Normally, the process will begin with modeling, where the goal is to understand the problem and to develop solution strategies.

> **Example 1.5:** For the quadrotor problem of Section 1.2, we might begin by constructing models that translate commands from a human to move vertically or laterally into commands to the four motors to produce thrust. A model will reveal that if the thrust is not the same on the four rotors, then the vehicle will tilt and move laterally.
>
> Such a model might use techniques like those in Chapter 2 (Continuous Dynamics), constructing differential equations to describe the dynamics of the vehicle. It would then use techniques like those in Chapter 3 (Discrete Dynamics) to build state machines that model the modes of operation such as takeoff, landing, hovering, and lateral flight. It could then use the techniques of Chapter 4 (Hybrid Systems) to blend these two types of models, creating hybrid system models of the system to study the transitions between modes of operation. The techniques of Chapters 5 (Composition of State Machines) and 6 (Concurrent Models of Computation) would then provide mechanisms for composing models of multiple vehicles, models of the interactions between a vehicle and its environment, and models of the interactions of components within a vehicle.

The process may progress quickly to the design phase, where we begin selecting components and putting them together (motors, batteries, sensors, microprocessors, memory

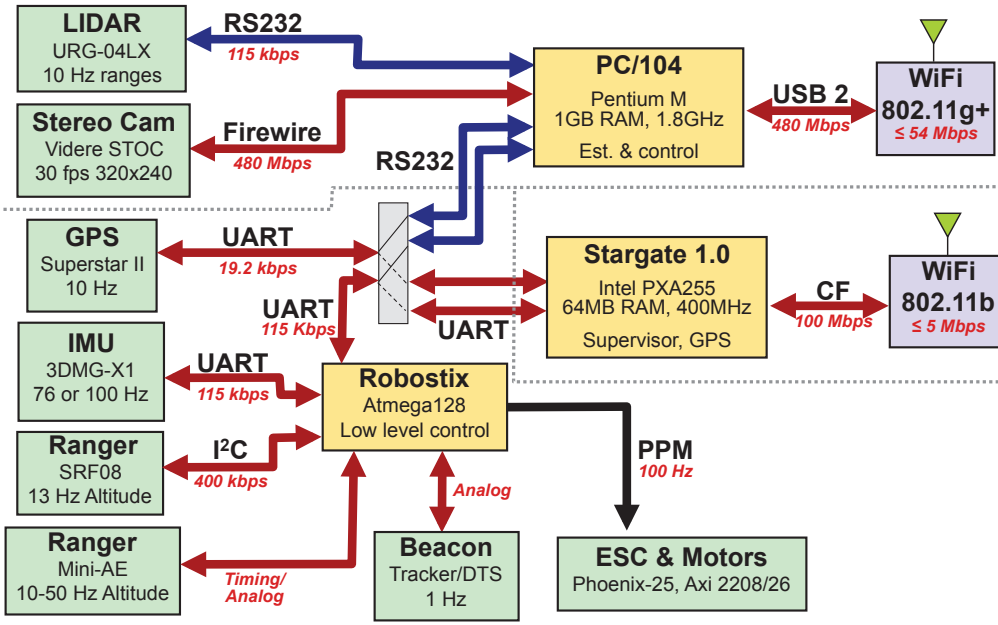


Figure 1.4: The STARMAC architecture (reproduced with permission).

systems, operating systems, wireless networks, etc.). An initial prototype may reveal flaws in the models, causing a return to the modeling phase and revision of the models.

Example 1.6: The hardware architecture of the first generation STARMAC quadrotor is shown in Figure 1.4. At the left and bottom of the figure are a number of sensors used by the vehicle to determine where it is (localization) and what is around it. In the middle are three boxes showing three distinct microprocessors. The Robostix is an Atmel AVR 8-bit microcontroller that runs with no operating system and performs the low-level control algorithms to keep the craft flying. The other two processors perform higher-level tasks with the help of an operating system. Both processors include wireless links that can be used by cooperating vehicles and ground controllers.

Chapter 7 (Sensors and Actuators) considers sensors and actuators, including the IMU and rangers shown in Figure 1.4. Chapter 8 (Embedded Processors) considers processor architectures, offering some basis for comparing the relative advantages of one architecture or another. Chapter 9 (Memory Architectures) considers the design of memory systems, emphasizing the impact that they can have on overall system behavior. Chapter 10 (Input and Output) considers the interfacing of processors with sensors and actuators. Chapters 11 (Multitasking) and 12 (Scheduling) focus on software architecture, with particular emphasis on how to orchestrate multiple real-time tasks.

In a healthy design process, analysis figures prominently early in the process. Analysis will be applied to the models and to the designs. The models may be analyzed for safety conditions, for example to ensure an invariant that asserts that if the vehicle is within one meter of the ground, then its vertical speed is no greater than 0.1 meter/sec. The designs may be analyzed for the timing behavior of software, for example to determine how long it takes the system to respond to an emergency shutdown command. Certain analysis problems will involve details of both models and designs. For the quadrotor example, it is important to understand how the system will behave if network connectivity is lost and it becomes impossible to communicate with the vehicle. How can the vehicle detect that communication has been lost? This will require accurate modeling of the network and the software.

Example 1.7: For the quadrotor problem, we use the techniques of Chapter 13 (Invariants and Temporal Logic) to specify key safety requirements for operation of the vehicles. We would then use the techniques of Chapters 14 (Equivalence and Refinement) and 15 (Reachability Analysis and Model Checking) to verify that these safety properties are satisfied by implementations of the software. The techniques of Chapter 16 (Quantitative Analysis) would be used to determine whether real-time constraints are met by the software. Finally, the techniques of Chapter 17 would be used to ensure that malicious parties cannot take control of the quadrotor and that any confidential data it may be gathering is not leaked to an adversary.

Corresponding to a design process structured as in Figure 1.3, this book is divided into three major parts, focused on modeling, design, and analysis (see Figure 1 on page xv). We now describe the approach taken in the three parts.

1.3.1 Modeling

The modeling part of the book, which is the first part, focuses on models of dynamic behavior. It begins with a light coverage of the big subject of modeling of physical dynamics in Chapter 2, specifically focusing on continuous dynamics in time. It then talks about discrete dynamics in Chapter 3, using state machines as the principal formalism. It then combines the two, continuous and discrete dynamics, with a discussion of hybrid systems in Chapter 4. Chapter 5 (Composition of State Machines) focuses on concurrent composition of state machines, emphasizing that the semantics of composition is a critical issue with which designers must grapple. Chapter 6 (Concurrent Models of Computation) gives an overview of concurrent models of computation, including many of those used in design tools that practitioners frequently leverage, such as Simulink and LabVIEW.

In the modeling part of the book, we define a **system** to be simply a combination of parts that is considered as a whole. A **physical system** is one realized in matter, in contrast to a conceptual or **logical system** such as software and algorithms. The **dynamics** of a system is its evolution in time: how its state changes. A **model** of a physical system is a description of certain aspects of the system that is intended to yield insight into properties of the system. In this text, models have mathematical properties that enable systematic analysis. The model imitates properties of the system, and hence yields insight into that system.

A model is itself a system. It is important to avoid confusing a model and the system that it models. These are two distinct artifacts. A model of a system is said to have high **fidelity** if it accurately describes properties of the system. It is said to **abstract** the system if it omits details. Models of physical systems inevitably *do* omit details, so they are always abstractions of the system. A major goal of this text is to develop an understanding of how to use models, of how to leverage their strengths and respect their weaknesses.

A cyber-physical system (CPS) is a system composed of physical subsystems together with computing and networking. Models of cyber-physical systems normally include all three parts. The models will typically need to represent both dynamics and **static properties** (those that do not change during the operation of the system). It is important to note that a model of a cyber-physical system need not have both discrete and continuous parts. It is possible for a purely discrete (or purely continuous) model to have high fidelity for the properties of interest.

Each of the modeling techniques described in this part of the book is an enormous subject, much bigger than one chapter, or even one book. In fact, such models are the focus of

many branches of engineering, physics, chemistry, and biology. Our approach is aimed at engineers. We assume some background in mathematical modeling of dynamics (calculus courses that give some examples from physics are sufficient), and then focus on how to compose diverse models. This will form the core of the cyber-physical system problem, since joint modeling of the cyber side, which is logical and conceptual, with the physical side, which is embodied in matter, is the core of the problem. We therefore make no attempt to be comprehensive, but rather pick a few modeling techniques that are widely used by engineers and well understood, review them, and then compose them to form a cyber-physical whole.

1.3.2 Design

The second part of the book has a very different flavor, reflecting the intrinsic heterogeneity of the subject. This part focuses on the design of embedded systems, with emphasis on the role they play *within* a CPS. Chapter 7 (Sensors and Actuators) considers sensors and actuators, with emphasis on how to model them so that their role in overall system dynamics is understood. Chapter 8 (Embedded Processors) discusses processor architectures, with emphasis on specialized properties most suited to embedded systems. Chapter 9 (Memory Architectures) describes memory architectures, including abstractions such as memory models in programming languages, physical properties such as memory technologies, and architectural properties such as memory hierarchy (caches, scratchpads, etc.). The emphasis is on how memory architecture affects dynamics. Chapter 10 (Input and Output) is about the interface between the software world and the physical world. It discusses input/output mechanisms in software and computer architectures, and the digital/analog interface, including sampling. Chapter 11 (Multitasking) introduces the notions that underlie operating systems, with particular emphasis on multitasking. The emphasis is on the pitfalls of using low-level mechanisms such as threads, with a hope of convincing the reader that there is real value in using the modeling techniques covered in the first part of the book. Those modeling techniques help designers build confidence in system designs. Chapter 12 (Scheduling) introduces real-time scheduling, covering many of the classic results in the area.

In all chapters in the design part, we particularly focus on the mechanisms that provide concurrency and control over timing, because these issues loom large in the design of cyber-physical systems. When deployed in a product, embedded processors typically have a dedicated function. They control an automotive engine or measure ice thickness in the Arctic. They are not asked to perform arbitrary functions with user-defined soft-

ware. Consequently, the processors, memory architectures, I/O mechanisms, and operating systems can be more specialized. Making them more specialized can bring enormous benefits. For example, they may consume far less energy, and consequently be usable with small batteries for long periods of time. Or they may include specialized hardware to perform operations that would be costly to perform on general-purpose hardware, such as image analysis. Our goal in this part is to enable the reader to *critically* evaluate the numerous available technology offerings.

One of the goals in this part of the book is to teach students to implement systems while *thinking across traditional abstraction layers* — e.g., hardware *and* software, computation *and* physical processes. While such cross-layer thinking is valuable in implementing systems in general, it is particularly essential in embedded systems given their heterogeneous nature. For example, a programmer implementing a control algorithm expressed in terms of real-valued quantities must have a solid understanding of computer arithmetic (e.g., of fixed-point numbers) in order to create a reliable implementation. Similarly, an implementor of automotive software that must satisfy real-time constraints must be aware of processor features – such as pipelines and caches – that can affect the execution time of tasks and hence the real-time behavior of the system. Likewise, an implementor of interrupt-driven or multi-threaded software must understand the atomic operations provided by the underlying software-hardware platform and use appropriate synchronization constructs to ensure correctness. Rather than doing an exhaustive survey of different implementation methods and platforms, this part of the book seeks to give the reader an appreciation for such cross-layer topics, and uses homework exercises to facilitate a deeper understanding of them.

1.3.3 Analysis

Every system must be designed to meet certain requirements. For embedded systems, which are often intended for use in safety-critical, everyday applications, it is essential to certify that the system meets its requirements. Such system requirements are also called **properties** or **specification**s. The need for specifications is aptly captured by the following quotation, paraphrased from Young et al. (1985):

> "A design without specifications cannot be right or wrong, it can only be surprising!"

The analysis part of the book focuses on precise specifications of properties, on techniques for comparing specifications, and on techniques for analyzing specifications and the resulting designs. Reflecting the emphasis on dynamics in the text, Chapter 13 (Invariants and Temporal Logic) focuses on temporal logics, which provide precise descriptions of dynamic properties of systems. These descriptions are treated as models. Chapter 14 (Equivalence and Refinement) focuses on the relationships between models. Is one model an abstraction of another? Is it equivalent in some sense? Specifically, that chapter introduces type systems as a way of comparing static properties of models, and language containment and simulation relations as a way of comparing dynamic properties of models. Chapter 15 (Reachability Analysis and Model Checking) focuses on techniques for analyzing the large number of possible dynamic behaviors that a model may exhibit, with particular emphasis on model checking as a technique for exploring such behaviors. Chapter 16 (Quantitative Analysis) is about analyzing quantitative properties of embedded software, such as finding bounds on resources consumed by programs. It focuses particularly on execution time analysis, with some introduction to other quantitative properties such as energy and memory usage. Chapter 17 (Security and Privacy) introduces the basics of security and privacy for embedded systems design, including cryptographic primitives, protocol security, software security, secure information flow, side channels, and sensor security.

In present engineering practice, it is common to have system requirements stated in a natural language such as English. It is important to precisely state requirements to avoid ambiguities inherent in natural languages. The goal of this part of the book is to help replace descriptive techniques with *formal* ones, which we believe are less error prone.

Importantly, formal specifications also enable the use of automatic techniques for formal verification of both models and implementations. The analysis part of the book introduces readers to the basics of formal verification, including notions of equivalence and refinement checking, as well as reachability analysis and model checking. In discussing these verification methods, we attempt to give users of verification tools an appreciation of what is "under the hood" so that they may derive the most benefit from them. This *user's view* is supported by examples discussing, for example, how model checking can be applied to find subtle errors in concurrent software, or how reachability analysis can be used in computing a control strategy for a robot to achieve a particular task.

1.4 Summary

Cyber-physical systems are heterogeneous blends by nature. They combine computation, communication, and physical dynamics. They are harder to model, harder to design, and harder to analyze than homogeneous systems. This chapter gives an overview of the engineering principles addressed in this book for modeling, designing, and analyzing such systems.

Part I

Modeling Dynamic Behaviors

This part of this text studies modeling of embedded systems, with emphasis on joint modeling of software and physical dynamics. We begin in Chapter 2 with a discussion of established techniques for modeling the dynamics of physical systems, with emphasis on their continuous behaviors. In Chapter 3, we discuss techniques for modeling discrete behaviors, which reflect better the behavior of software. In Chapter 4, we bring these two classes of models together and show how discrete and continuous behaviors are jointly modeled by hybrid systems. Chapters 5 and 6 are devoted to reconciling the inherently concurrent nature of the physical world with the inherently sequential world of software. Chapter 5 shows how state machine models, which are fundamentally sequential, can be composed concurrently. That chapter specifically introduces the notion of synchronous composition. Chapter 6 shows that synchronous composition is but one of the ways to achieve concurrent composition.

2 Continuous Dynamics

This chapter reviews a few of the many modeling techniques for studying dynamics of a physical system. We begin by studying mechanical parts that move (this problem is known as **classical mechanics**). The techniques used to study the dynamics of such parts extend broadly to many other physical systems, including circuits, chemical processes, and biological processes. But mechanical parts are easiest for most people to visualize, so they make our example concrete. Motion of mechanical parts can often be modeled using **differential equations**, or equivalently, **integral equations**. Such models really only work well for "smooth" motion (a concept that we can make more precise using notions of linearity, time invariance, and continuity). For motions that are not smooth, such as those modeling collisions of mechanical parts, we can use modal models that represent

distinct modes of operation with abrupt (conceptually instantaneous) transitions between modes. Collisions of mechanical objects can be usefully modeled as discrete, instantaneous events. The problem of jointly modeling smooth motion and such discrete events is known as hybrid systems modeling and is studied in Chapter 4. Such combinations of discrete and continuous behaviors bring us one step closer to joint modeling of cyber and physical processes.

We begin with simple equations of motion, which provide a model of a system in the form of **ordinary differential equations** (**ODEs**). We then show how these ODEs can be represented in actor models, which include the class of models in popular modeling languages such as LabVIEW (from National Instruments) and Simulink (from The MathWorks, Inc.). We then consider properties of such models such as linearity, time invariance, and stability, and consider consequences of these properties when manipulating models. We develop a simple example of a feedback control system that stabilizes an unstable system. Controllers for such systems are often realized using software, so such systems can serve as a canonical example of a cyber-physical system. The properties of the overall system emerge from properties of the cyber and physical parts.

2.1 Newtonian Mechanics

In this section, we give a brief working review of some principles of classical mechanics. This is intended to be just enough to be able to construct interesting models, but is by no means comprehensive. The interested reader is referred to many excellent texts on classical mechanics, including Goldstein (1980); Landau and Lifshitz (1976); Marion and Thornton (1995).

Motion in space of physical objects can be represented with **six degrees of freedom**, illustrated in Figure 2.1. Three of these represent position in three dimensional space, and three represent orientation in space. We assume three axes, x, y, and z, where by convention x is drawn increasing to the right, y is drawn increasing upwards, and z is drawn increasing out of the page. **Roll** θ_x is an angle of rotation around the x axis, where by convention an angle of 0 radians represents horizontally flat along the z axis (i.e., the angle is given relative to the z axis). **Yaw** θ_y is the rotation around the y axis, where by convention 0 radians represents pointing directly to the right (i.e., the angle is given relative to the x axis). **Pitch** θ_z is rotation around the z axis, where by convention 0 radians represents pointing horizontally (i.e., the angle is given relative to the x axis).

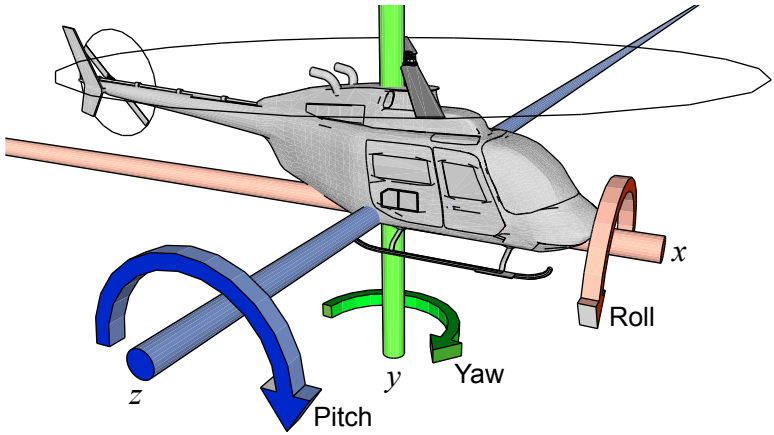


Figure 2.1: Modeling position with six degrees of freedom requires including pitch, roll, and yaw, in addition to position.

The position of an object in space, therefore, is represented by six functions of the form $f: \mathbb{R} \to \mathbb{R}$, where the domain represents time and the codomain represents either distance along an axis or angle relative to an axis.[1] Functions of this form are known as **continuous-time signals**.[2] These are often collected into vector-valued functions $\mathbf{x}: \mathbb{R} \to \mathbb{R}^3$ and $\theta: \mathbb{R} \to \mathbb{R}^3$, where $\mathbf{x}$ represents position, and θ represents orientation.

Changes in position or orientation are governed by **Newton's second law**, relating force with acceleration. Acceleration is the second derivative of position. Our first equation handles the position information,

$$\mathbf{F}(t) = M\ddot{\mathbf{x}}(t), \tag{2.1}$$

where $\mathbf{F}$ is the force vector in three directions, M is the mass of the object, and $\ddot{\mathbf{x}}$ is the second derivative of $\mathbf{x}$ with respect to time (i.e., the acceleration). Velocity is the integral

[1] If the notation is unfamiliar, see Appendix A.

[2] The domain of a continuous-time signal may be restricted to a connected subset of $\mathbb{R}$, such as $\mathbb{R}_+$, the non-negative reals, or $[0, 1]$, the interval between zero and one, inclusive. The codomain may be an arbitrary set, though when representing physical quantities, real numbers are most useful.

of acceleration, given by

$$\forall\, t > 0, \quad \dot{\mathbf{x}}(t) \;=\; \dot{\mathbf{x}}(0) + \int_0^t \ddot{\mathbf{x}}(\tau) d\tau$$

where $\dot{\mathbf{x}}(0)$ is the initial velocity in three directions. Using (2.1), this becomes

$$\forall\, t > 0, \quad \dot{\mathbf{x}}(t) = \dot{\mathbf{x}}(0) + \frac{1}{M} \int_0^t \mathbf{F}(\tau) d\tau,$$

Position is the integral of velocity,

$$\begin{aligned} \mathbf{x}(t) &= \mathbf{x}(0) + \int_0^t \dot{\mathbf{x}}(\tau) d\tau \\ &= \mathbf{x}(0) + t\dot{\mathbf{x}}(0) + \frac{1}{M} \int_0^t \int_0^\tau \mathbf{F}(\alpha) d\alpha d\tau, \end{aligned}$$

where $\mathbf{x}(0)$ is the initial position. Using these equations, if you know the initial position and initial velocity of an object and the forces on the object in all three directions as a function of time, you can determine the acceleration, velocity, and position of the object at any time.

The versions of these equations of motion that affect orientation use **torque**, the rotational version of force. It is again a three-element vector as a function of time, representing the net rotational force on an object. It can be related to angular velocity in a manner similar to equation (2.1),

$$\mathbf{T}(t) = \frac{d}{dt}\left(\mathbf{I}(t)\dot{\theta}(t)\right), \tag{2.2}$$

where $\mathbf{T}$ is the torque vector in three axes and $\mathbf{I}(t)$ is the **moment of inertia tensor** of the object. The moment of inertia is a 3×3 matrix that depends on the geometry and orientation of the object. Intuitively, it represents the reluctance that an object has to spin around any axis as a function of its orientation along the three axes. If the object is spherical, for example, this reluctance is the same around all axes, so it reduces to a constant scalar I (or equivalently, to a diagonal matrix $\mathbf{I}$ with equal diagonal elements I). The equation then looks much more like (2.1),

$$\mathbf{T}(t) = I\ddot{\theta}(t). \tag{2.3}$$

To be explicit about the three dimensions, we might write (2.2) as

$$\begin{bmatrix} T_x(t) \\ T_y(t) \\ T_z(t) \end{bmatrix} = \frac{d}{dt}\left(\begin{bmatrix} I_{xx}(t) & I_{xy}(t) & I_{xz}(t) \\ I_{yx}(t) & I_{yy}(t) & I_{yz}(t) \\ I_{zx}(t) & I_{zy}(t) & I_{zz}(t) \end{bmatrix} \begin{bmatrix} \dot{\theta}_x(t) \\ \dot{\theta}_y(t) \\ \dot{\theta}_z(t) \end{bmatrix} \right) .$$

Here, for example, $T_y(t)$ is the net torque around the y axis (which would cause changes in yaw), $I_{yx}(t)$ is the inertia that determines how acceleration around the x axis is related to torque around the y axis.

Rotational velocity is the integral of acceleration,

$$\dot{\theta}(t) = \dot{\theta}(0) + \int_0^t \ddot{\theta}(\tau) d\tau,$$

where $\dot{\theta}(0)$ is the initial rotational velocity in three axes. For a spherical object, using (2.3), this becomes

$$\dot{\theta}(t) = \dot{\theta}(0) + \frac{1}{I}\int_0^t \mathbf{T}(\tau) d\tau.$$

Orientation is the integral of rotational velocity,

$$\begin{aligned} \theta(t) &= \theta(0) + \int_0^t \dot{\theta}(\tau) d\tau \\ &= \theta(0) + t\dot{\theta}(0) + \frac{1}{I}\int_0^t \int_0^\tau \mathbf{T}(\alpha) d\alpha d\tau \end{aligned}$$

where $\theta(0)$ is the initial orientation. Using these equations, if you know the initial orientation and initial rotational velocity of an object and the torques on the object in all three axes as a function of time, you can determine the rotational acceleration, velocity, and orientation of the object at any time.

Often, as we have done for a spherical object, we can simplify by reducing the number of dimensions that are considered. In general, such a simplification is called a **model-order reduction**. For example, if an object is a moving vehicle on a flat surface, there may be little reason to consider the y axis movement or the pitch or roll of the object.

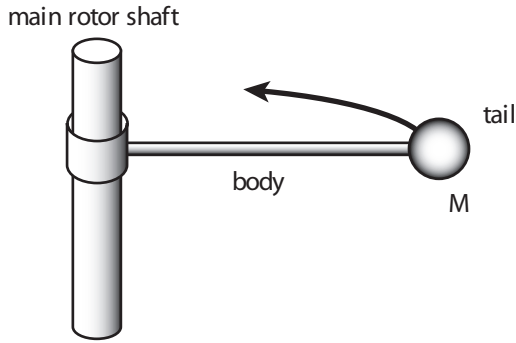


Figure 2.2: Simplified model of a helicopter.

Example 2.1: Consider a simple control problem that admits such reduction of dimensionality. A helicopter has two rotors, one above, which provides lift, and one on the tail. Without the rotor on the tail, the body of the helicopter would spin. The rotor on the tail counteracts that spin. Specifically, the force produced by the tail rotor must counter the torque produced by the main rotor. Here we consider this role of the tail rotor independently from all other motion of the helicopter.

A simplified model of the helicopter is shown in Figure 2.2. Here, we assume that the helicopter position is fixed at the origin, so there is no need to consider equations describing position. Moreover, we assume that the helicopter remains vertical, so pitch and roll are fixed at zero. These assumptions are not as unrealistic as they may seem since we can define the coordinate system to be fixed to the helicopter.

With these assumptions, the **moment of inertia** reduces to a scalar that represents a torque that resists changes in yaw. The changes in yaw will be due to **Newton's third law**, the **action-reaction law**, which states that every action has an equal and opposite reaction. This will tend to cause the helicopter to rotate in the opposite direction from the rotor rotation. The tail rotor has the job of countering that torque to keep the body of the helicopter from spinning.

We model the simplified helicopter by a system that takes as input a continuous-time signal T_y, the torque around the y axis (which causes changes in yaw). This

torque is the sum of the torque caused by the main rotor and that caused by the tail rotor. When these are perfectly balanced, that sum is zero. The output of our system will be the angular velocity $\dot{\theta}_y$ around the y axis. The dimensionally-reduced version of (2.2) can be written as

$$\ddot{\theta}_y(t) = T_y(t)/I_{yy}.$$

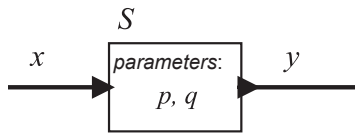
Integrating both sides, we get the output $\dot{\theta}$ as a function of the input T_y,

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) + \frac{1}{I_{yy}} \int_0^t T_y(\tau) d\tau. \tag{2.4}$$

The critical observation about this example is that if we were to choose to model the helicopter by, say, letting $\mathbf{x}: \mathbb{R} \to \mathbb{R}^3$ represent the absolute position in space of the tail of the helicopter, we would end up with a far more complicated model. Designing the control system would also be much more difficult.

2.2 Actor Models

In the previous section, a model of a physical system is given by a differential or an integral equation that relates input signals (force or torque) to output signals (position, orientation, velocity, or rotational velocity). Such a physical system can be viewed as a component in a larger system. In particular, a **continuous-time system** (one that operates on continuous-time signals) may be modeled by a box with an input **port** and an output port as follows:



where the input signal x and the output signal y are functions of the form

$$x: \mathbb{R} \to \mathbb{R}, \quad y: \mathbb{R} \to \mathbb{R}.$$

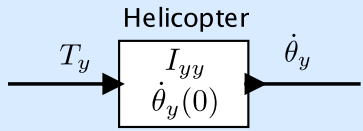
Here the domain represents **time** and the codomain represents the value of the signal at a particular time. The domain $\mathbb{R}$ may be replaced by $\mathbb{R}_+$, the non-negative reals, if we wish to explicitly model a system that comes into existence and starts operating at a particular point in time.

The model of the system is a function of the form

$$S: X \to Y, \tag{2.5}$$

where $X = Y = \mathbb{R}^{\mathbb{R}}$, the set of functions that map the reals into the reals, like x and y above.[3] The function S may depend on parameters of the system, in which case the parameters may be optionally shown in the box, and may be optionally included in the function notation. For example, in the above figure, if there are parameters p and q, we might write the system function as $S_{p,q}$ or even $S(p, q)$, keeping in mind that both notations represent functions of the form in 2.5. A box like that above, where the inputs are functions and the outputs are functions, is called an **actor**.

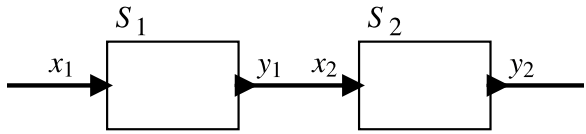
Example 2.2: The actor model for the helicopter of example 2.1 can be depicted as follows:



The input and output are both continuous-time functions. The parameters of the actor are the initial angular velocity $\dot{\theta}_y(0)$ and the moment of inertia I_{yy}. The function of the actor is defined by (2.4).

Actor models are composable. In particular, given two actors S_1 and S_2, we can form a **cascade composition** as follows:

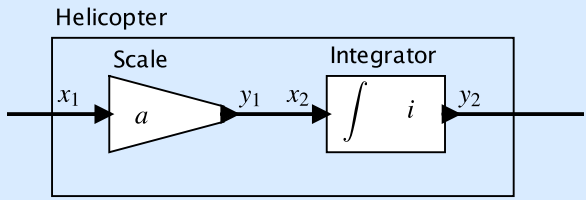
[3] As explained in Appendix A, the notation $\mathbb{R}^{\mathbb{R}}$ (which can also be written $(\mathbb{R} \to \mathbb{R})$) represents the set of all functions with domain $\mathbb{R}$ and codomain $\mathbb{R}$.



In the diagram, the "wire" between the output of S_1 and the input of S_2 means precisely that $y_1 = x_2$, or more pedantically,

$$\forall\, t \in \mathbb{R}, \quad y_1(t) = x_2(t).$$

Example 2.3: The actor model for the helicopter can be represented as a cascade composition of two actors as follows:



The left actor represents a Scale actor parameterized by the constant a defined by

$$\forall\, t \in \mathbb{R}, \quad y_1(t) = ax_1(t). \tag{2.6}$$

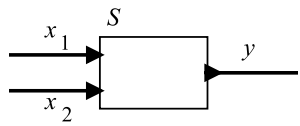
More compactly, we can write $y_1 = ax_1$, where it is understood that the product of a scalar a and a function x_1 is interpreted as in (2.6). The right actor represents an integrator parameterized by the initial value i defined by

$$\forall\, t \in \mathbb{R}, \quad y_2(t) = i + \int_0^t x_2(\tau)d\tau.$$

If we give the parameter values $a = 1/I_{yy}$ and $i = \dot{\theta}_y(0)$, we see that this system represents (2.4) where the input $x_1 = T_y$ is torque and the output $y_2 = \dot{\theta}_y$ is angular velocity.

In the above figure, we have customized the **icons**, which are the boxes representing the actors. These particular actors (scaler and integrator) are particularly useful building blocks for building up models of physical dynamics, so assigning them recognizable visual notations is useful.

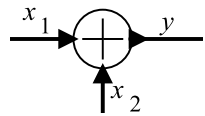
We can have actors that have multiple input signals and/or multiple output signals. These are represented similarly, as in the following example, which has two input signals and one output signal:



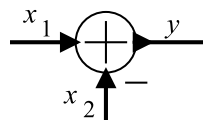
A particularly useful building block with this form is a signal **adder**, defined by

$$\forall\, t \in \mathbb{R}, \quad y(t) = x_1(t) + x_2(t).$$

This will often be represented by a custom icon as follows:



Sometimes, one of the inputs will be subtracted rather than added, in which case the icon is further customized with minus sign near that input, as below:



This actor represents a function $S\colon (\mathbb{R} \to \mathbb{R})^2 \to (\mathbb{R} \to \mathbb{R})$ given by

$$\forall\, t \in \mathbb{R},\ \forall\, x_1, x_2 \in (\mathbb{R} \to \mathbb{R}), \quad (S(x_1, x_2))(t) = y(t) = x_1(t) - x_2(t).$$

Notice the careful notation. $S(x_1, x_2)$ is a function in $\mathbb{R}^{\mathbb{R}}$. Hence, it can be evaluated at a $t \in \mathbb{R}$.

In the rest of this chapter, we will not make a distinction between a system and its actor model, unless the distinction is essential to the argument. We will assume that the actor model captures everything of interest about the system. This is an admittedly bold assumption. Generally the properties of the actor model are only approximate descriptions of the actual system.

2.3 Properties of Systems

In this section, we consider a number of properties that actors and the systems they compose may have, including causality, memorylessness, linearity, time invariance, and stability.

2.3.1 Causal Systems

Intuitively, a system is **causal** if its output depends only on current and past inputs. Making this notion precise is a bit tricky, however. We do this by first giving a notation for "current and past inputs." Consider a continuous-time signal $x : \mathbb{R} \to A$, for some set A. Let $x|_{t \leq \tau}$ represent a function called the **restriction in time** that is only defined for times $t \leq \tau$, and where it is defined, $x|_{t \leq \tau}(t) = x(t)$. Hence if x is an input to a system, then $x|_{t \leq \tau}$ is the "current and past inputs" at time τ.

Consider a continuous-time system $S : X \to Y$, where $X = A^{\mathbb{R}}$ and $Y = B^{\mathbb{R}}$ for some sets A and B. This system is causal if for all $x_1, x_2 \in X$ and $\tau \in \mathbb{R}$,

$$x_1|_{t \leq \tau} = x_2|_{t \leq \tau} \Rightarrow S(x_1)|_{t \leq \tau} = S(x_2)|_{t \leq \tau}$$

That is, the system is causal if for two possible inputs x_1 and x_2 that are identical up to (and including) time τ, the outputs are identical up to (and including) time τ. All systems we have considered so far are causal.

A system is **strictly causal** if for all $x_1, x_2 \in X$ and $\tau \in \mathbb{R}$,

$$x_1|_{t < \tau} = x_2|_{t < \tau} \Rightarrow S(x_1)|_{t \leq \tau} = S(x_2)|_{t \leq \tau}$$

That is, the system is strictly causal if for two possible inputs x_1 and x_2 that are identical up to (and *not* including) time τ, the outputs are identical up to (and including) time τ. The output at time t of a strictly causal system does not depend on its input at time t.

It only depends on past inputs. A strictly causal system, of course, is also causal. The Integrator actor is strictly causal. The adder is not strictly causal, but it is causal. Strictly causal actors are useful for constructing feedback systems.

2.3.2 Memoryless Systems

Intuitively, a system has memory if the output depends not only on the current inputs, but also on past inputs (or future inputs, if the system is not causal). Consider a continuous-time system $S\colon X \to Y$, where $X = A^{\mathbb{R}}$ and $Y = B^{\mathbb{R}}$ for some sets A and B. Formally, this system is **memoryless** if there exists a function $f\colon A \to B$ such that for all $x \in X$,

$$(S(x))(t) = f(x(t))$$

for all $t \in \mathbb{R}$. That is, the output $(S(x))(t)$ at time t depends only on the input $x(t)$ at time t.

The Integrator considered above is not memoryless, but the adder is. Exercise 2 shows that if a system is strictly causal and memoryless then its output is constant for all inputs.

2.3.3 Linearity and Time Invariance

Systems that are linear and time invariant (LTI) have particularly nice mathematical properties. Much of the theory of control systems depends on these properties. These properties form the main body of courses on signals and systems, and are beyond the scope of this text. But we will occasionally exploit simple versions of the properties, so it is useful to determine when a system is LTI.

A system $S\colon X \to Y$, where X and Y are sets of signals, is linear if it satisfies the **superposition** property:

$$\forall\, x_1, x_2 \in X \text{ and } \forall\, a, b \in \mathbb{R}, \quad S(ax_1 + bx_2) = aS(x_1) + bS(x_2).$$

It is easy to see that the helicopter system defined in Example 2.1 is linear if and only if the initial angular velocity $\dot{\theta}_y(0) = 0$ (see Exercise 3).

More generally, it is easy to see that an integrator as defined in Example 2.3 is linear if and only if the initial value $i = 0$, that the Scale actor is always linear, and that the cascade of any two linear actors is linear. We can trivially extend the definition of linearity to actors with more than one input or output signal and then determine that the adder is also linear.

To define time invariance, we first define a specialized continuous-time actor called a **delay**. Let $D_\tau : X \to Y$, where X and Y are sets of continuous-time signals, be defined by

$$\forall\, x \in X \text{ and } \forall\, t \in \mathbb{R}, \quad (D_\tau(x))(t) = x(t - \tau). \tag{2.7}$$

Here, τ is a parameter of the delay actor. A system $S : X \to Y$ is time invariant if

$$\forall\, x \in X \text{ and } \forall\, \tau \in \mathbb{R}, \quad S(D_\tau(x)) = D_\tau(S(x)).$$

The helicopter system defined in Example 2.1 and (2.4) is not time invariant. A minor variant, however, is time invariant:

$$\dot{\theta}_y(t) = \frac{1}{I_{yy}} \int_{-\infty}^{t} T_y(\tau) d\tau.$$

This version does not allow for an initial angular rotation.

A **linear time-invariant system** (**LTI**) is a system that is both linear and time invariant. A major objective in modeling physical dynamics is to choose an LTI model whenever possible. If a reasonable approximation results in an LTI model, it is worth making this approximation. It is not always easy to determine whether the approximation is reasonable, or to find models for which the approximation is reasonable. It is often easy to construct models that are more complicated than they need to be (see Exercise 4).

2.3.4 Stability

A system is said to be **bounded-input bounded-output stable** (**BIBO stable** or just **stable**) if the output signal is bounded for all input signals that are bounded.

Consider a continuous-time system with input w and output v. The input is bounded if there is a real number $A < \infty$ such that $|w(t)| \leq A$ for all $t \in \mathbb{R}$. The output is bounded if there is a real number $B < \infty$ such that $|v(t)| \leq B$ for all $t \in \mathbb{R}$. The system is stable if for any input bounded by some A, there is some bound B on the output.

Example 2.4: It is now easy to see that the helicopter system developed in Example 2.1 is unstable. Let the input be $T_y = u$, where u is the **unit step**, given

by

$$\forall\, t \in \mathbb{R}, \quad u(t) = \begin{cases} 0, & t < 0 \\ 1, & t \geq 0 \end{cases} . \tag{2.8}$$

This means that prior to time zero, there is no torque applied to the system, and starting at time zero, we apply a torque of unit magnitude. This input is clearly bounded. It never exceeds one in magnitude. However, the output grows without bound. In practice, a helicopter uses a feedback system to determine how much torque to apply at the tail rotor to keep the body of the helicopter straight. We study how to do that next.

2.4 Feedback Control

A system with **feedback** has directed cycles, where an output from an actor is fed back to affect an input of the same actor. An example of such a system is shown in Figure 2.3. Most control systems use feedback. They make measurements of an **error** (e in the figure), which is a discrepancy between desired behavior (ψ in the figure) and actual behavior ($\dot{\theta}_y$ in the figure), and use that measurement to correct the behavior. The error measurement is feedback, and the corresponding correction signal (T_y in the figure) should compensate to reduce future error. Note that the correction signal normally can only affect *future* errors, so a feedback system must normally include at least one strictly causal actor (the Helicopter in the figure) in every directed cycle.

Feedback control is a sophisticated topic, easily occupying multiple texts and complete courses. Here, we only barely touch on the subject, just enough to motivate the interactions between software and physical systems. Feedback control systems are often implemented using embedded software, and the overall physical dynamics is a composition of the software and physical dynamics. More detail can be found in Chapters 12-14 of Lee and Varaiya (2011).

Example 2.5: Recall that the helicopter model of Example 2.1 is not stable. We can stabilize it with a simple feedback control system, as shown in Figure 2.3. The input ψ to this system is a continuous-time system specifying the desired

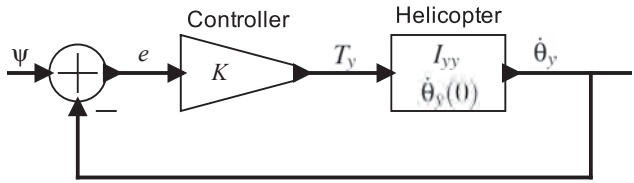


Figure 2.3: Proportional control system that stabilizes the helicopter.

angular velocity. The **error signal** e represents the difference between the actual and the desired angular velocity. In the figure, the controller simply scales the error signal by a constant K, providing a control input to the helicopter. We use (2.4) to write

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) + \frac{1}{I_{yy}} \int_0^t T_y(\tau) d\tau \tag{2.9}$$

$$= \dot{\theta}_y(0) + \frac{K}{I_{yy}} \int_0^t (\psi(\tau) - \dot{\theta}_y(\tau)) d\tau, \tag{2.10}$$

where we have used the facts (from the figure),

$$e(t) = \psi(t) - \dot{\theta}_y(t), \quad \text{and}$$

$$T_y(t) = Ke(t).$$

Equation (2.10) has $\dot{\theta}_y(t)$ on both sides, and therefore is not trivial to solve. The easiest solution technique uses Laplace transforms (see Lee and Varaiya (2011) Chapter 14). However, for our purposes here, we can use a more brute-force technique from calculus. To make this as simple as possible, we assume that $\psi(t) = 0$ for all t; i.e., we wish to control the helicopter simply to keep it from rotating at all. The desired angular velocity is zero. In this case, (2.10) simplifies to

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) - \frac{K}{I_{yy}} \int_0^t \dot{\theta}_y(\tau) d\tau. \tag{2.11}$$

Using the fact from calculus that, for $t \geq 0$,

$$\int_0^t ae^{a\tau} d\tau = e^{at}u(t) - 1,$$

where u is given by (2.8), we can infer that the solution to (2.11) is

$$\dot{\theta}_y(t) = \dot{\theta}_y(0)e^{-Kt/I_{yy}}u(t). \tag{2.12}$$

(Note that although it is easy to verify that this solution is correct, deriving the solution is not so easy. For this purpose, Laplace transforms provide a far better mechanism.)

We can see from (2.12) that the angular velocity approaches the desired angular velocity (zero) as t gets large as long as K is positive. For larger K, it will approach more quickly. For negative K, the system is unstable, and angular velocity will grow without bound.

The previous example illustrates a **proportional control** feedback loop. It is called this because the control signal is proportional to the error. We assumed a desired signal of zero. It is equally easy to assume that the helicopter is initially at rest (the angular velocity is zero) and then determine the behavior for a particular non-zero desired signal, as we do in the following example.

Example 2.6: Assume that the helicopter is **initially at rest**, meaning that

$$\dot{\theta}(0) = 0,$$

and that the desired signal is

$$\psi(t) = au(t)$$

for some constant a. That is, we wish to control the helicopter to get it to rotate at a fixed rate.

We use (2.4) to write

$$\begin{aligned}
\dot{\theta}_y(t) &= \frac{1}{I_{yy}} \int_0^t T_y(\tau) d\tau \\
&= \frac{K}{I_{yy}} \int_0^t (\psi(\tau) - \dot{\theta}_y(\tau)) d\tau \\
&= \frac{K}{I_{yy}} \int_0^t a d\tau - \frac{K}{I_{yy}} \int_0^t \dot{\theta}_y(\tau) d\tau \\
&= \frac{Kat}{I_{yy}} - \frac{K}{I_{yy}} \int_0^t \dot{\theta}_y(\tau) d\tau.
\end{aligned}$$

Using the same (black magic) technique of inferring and then verifying the solution, we can see that the solution is

$$\dot{\theta}_y(t) = au(t)(1 - e^{-Kt/I_{yy}}). \tag{2.13}$$

Again, the angular velocity approaches the desired angular velocity as t gets large as long as K is positive. For larger K, it will approach more quickly. For negative K, the system is unstable, and angular velocity will grow without bound.

Note that the first term in the above solution is exactly the desired angular velocity. The second term is an error called the **tracking error**, that for this example asymptotically approaches zero.

The above example is somewhat unrealistic because we cannot independently control the *net* torque of the helicopter. In particular, the net torque T_y is the sum of the torque T_t due to the top rotor and the torque T_r due to the tail rotor,

$$\forall\, t \in \mathbb{R}, \quad T_y(t) = T_t(t) + T_r(t)\ .$$

T_t will be determined by the rotation required to maintain or achieve a desired altitude, quite independent of the rotation of the helicopter. Thus, we will actually need to design a control system that controls T_r and stabilizes the helicopter for any T_t (or, more precisely,

any T_t within operating parameters). In the next example, we study how this changes the performance of the control system.

Example 2.7: In Figure 2.4(a), we have modified the helicopter model so that it has two inputs, T_t and T_r, the torque due to the top rotor and tail rotor respectively. The feedback control system is now controlling only T_r, and T_t is treated as an external (uncontrolled) input signal. How well will this control system behave?

Again, a full treatment of the subject is beyond the scope of this text, but we will study a specific example. Suppose that the torque due to the top rotor is given by

$$T_t = bu(t)$$

for some constant b. That is, at time zero, the top rotor starts spinning a constant velocity, and then holds that velocity. Suppose further that the helicopter is initially at rest. We can use the results of Example 2.6 to find the behavior of the system.

First, we transform the model into the equivalent model shown in Figure 2.4(b). This transformation simply relies on the algebraic fact that for any real numbers a_1, a_2, K,

$$Ka_1 + a_2 = K(a_1 + a_2/K).$$

We further transform the model to get the equivalent model shown in Figure 2.4(c), which has used the fact that addition is commutative. In Figure 2.4(c), we see that the portion of the model enclosed in the box is exactly the same as the control system analyzed in Example 2.6, shown in Figure 2.3. Thus, the same analysis as in Example 2.6 still applies. Suppose that desired angular rotation is

$$\psi(t) = 0.$$

Then the input to the original control system will be

$$x(t) = \psi(t) + T_t(t)/K = (b/K)u(t).$$

From (2.13), we see that the solution is

$$\dot{\theta}_y(t) = (b/K)u(t)(1 - e^{-Kt/I_{yy}}). \tag{2.14}$$

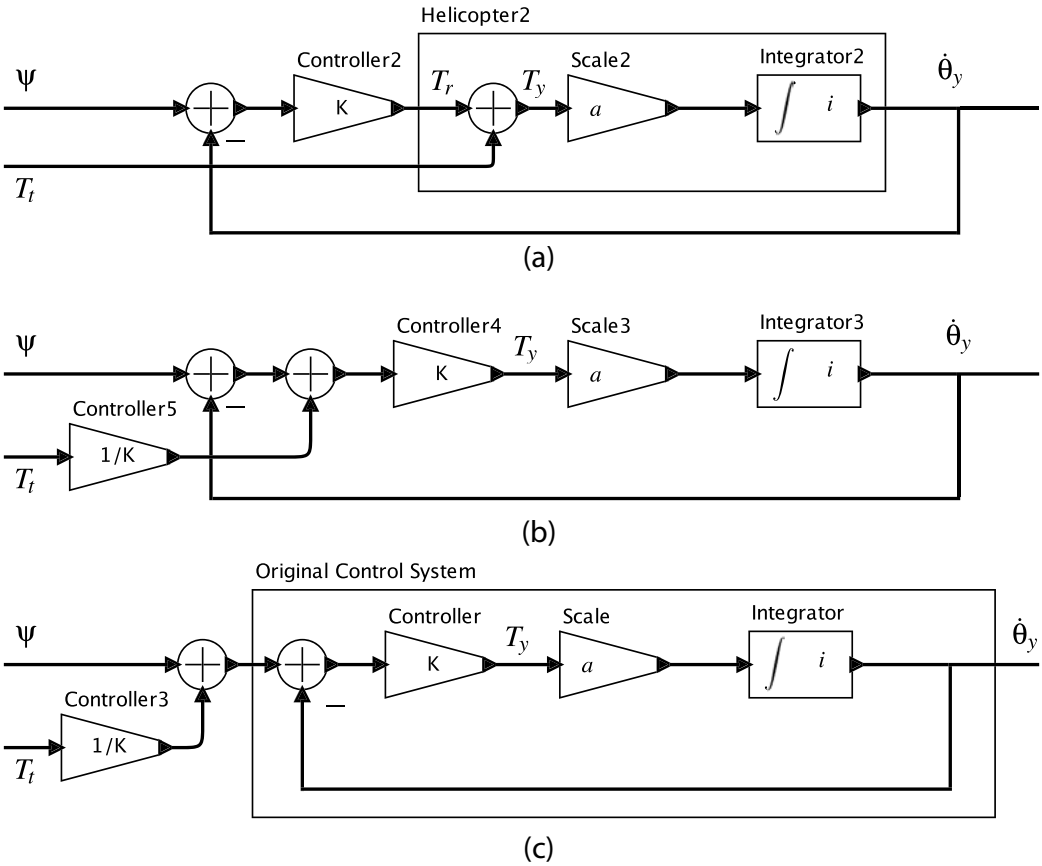


Figure 2.4: (a) Helicopter model with separately controlled torques for the top and tail rotors. (b) Transformation to an equivalent model (assuming $K > 0$). (c) Further transformation to an equivalent model that we can use to understand the behavior of the controller.

The desired angular rotation is zero, but the control system asymptotically approaches a non-zero angular rotation of b/K. This tracking error can be made arbitrarily small by increasing the control system feedback gain K, but with this controller design, it cannot be made to go to zero. An alternative controller design that yields an asymptotic tracking error of zero is studied in Exercise 7.

2.5 Summary

This chapter has described two distinct modeling techniques that describe physical dynamics. The first is ordinary differential equations, a venerable toolkit for engineers, and the second is actor models, a newer technique driven by software modeling and simulation tools. The two are closely related. This chapter has emphasized the relationship between these models, and the relationship of those models to the systems being modeled. These relationships, however, are quite a deep subject that we have barely touched upon. Our objective is to focus the attention of the reader on the fact that we may use multiple models for a system, and that models are distinct from the systems being modeled. The fidelity of a model (how well it approximates the system being modeled) is a strong factor in the success or failure of any engineering effort.

Exercises

1. A **tuning fork**, shown in Figure 2.5, consists of a metal finger (called a **tine**) that is displaced by striking it with a hammer. After being displaced, it vibrates. If the tine has no friction, it will vibrate forever. We can denote the displacement of the tine after being struck at time zero as a function $y: \mathbb{R}_+ \to \mathbb{R}$. If we assume that the initial displacement introduced by the hammer is one unit, then using our knowledge of physics we can determine that for all $t \in \mathbb{R}_+$, the displacement satisfies the differential equation

$$\ddot{y}(t) = -\omega_0^2 y(t)$$

where ω_0^2 is a constant that depends on the mass and stiffness of the tine, and where $\ddot{y}(t)$ denotes the second derivative with respect to time of y. It is easy to verify that y given by

$$\forall\, t \in \mathbb{R}_+, \quad y(t) = \cos(\omega_0 t)$$

is a solution to the differential equation (just take its second derivative). Thus, the displacement of the tuning fork is sinusoidal. If we choose materials for the tuning fork so that $\omega_0 = 2\pi \times 440$ radians/second, then the tuning fork will produce the tone of A-440 on the musical scale.

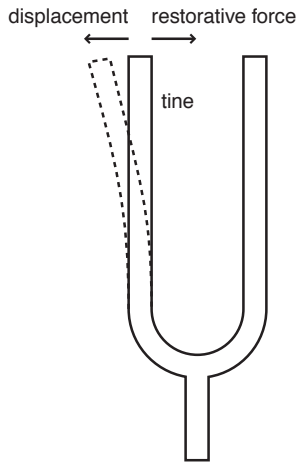


Figure 2.5: A tuning fork.

(a) Is $y(t) = \cos(\omega_0 t)$ the only solution? If not, give some others.

(b) Assuming the solution is $y(t) = \cos(\omega_0 t)$, what is the initial displacement?

(c) Construct a model of the tuning fork that produces y as an output using generic actors like Integrator, adder, scaler, or similarly simple actors. Treat the initial displacement as a parameter. Carefully label your diagram.

2. Show that if a system $S\colon A^{\mathbb{R}} \to B^{\mathbb{R}}$ is strictly causal and memoryless then its output is constant. Constant means that the output $(S(x))(t)$ at time t does not depend on t.

3. This exercise studies linearity.

(a) Show that the helicopter model defined in Example 2.1 is linear if and only if the initial angular velocity $\dot{\theta}_y(0) = 0$.

(b) Show that the cascade of any two linear actors is linear.

(c) Augment the definition of linearity so that it applies to actors with two input signals and one output signal. Show that the adder actor is linear.

4. Consider the helicopter of Example 2.1, but with a slightly different definition of the input and output. Suppose that, as in the example, the input is $T_y\colon \mathbb{R} \to \mathbb{R}$, as in the example, but the output is the position of the tail relative to the main rotor shaft. Specifically, let the x-y plane be the plane orthogonal to the rotor shaft, and let the position of the tail at time t be given by a tuple $((x(t), y(t))$. Is this model LTI? Is it BIBO stable?

5. Consider a rotating robot where you can control the angular velocity around a fixed axis.

(a) Model this as a system where the input is angular velocity $\dot{\theta}$ and the output is angle θ. Give your model as an equation relating the input and output as functions of time.

(b) Is this model BIBO stable?

(c) Design a proportional controller to set the robot onto a desired angle. That is, assume that the initial angle is $\theta(0) = 0$, and let the desired angle be $\psi(t) = au(t)$, where u is the unit step function. Find the actual angle as a function of time and the proportional controller feedback gain K. What is your output at $t = 0$? What does it approach as t gets large?

6. A DC motor produces a torque that is proportional to the current through the windings of the motor. Neglecting friction, the net torque on the motor, therefore, is this torque minus the torque applied by whatever load is connected to the motor. Newton's second law (the rotational version) gives

$$k_T i(t) - x(t) = I \frac{d}{dt}\omega(t), \tag{2.15}$$

where k_T is the motor torque constant, $i(t)$ is the current at time t, $x(t)$ is the torque applied by the load at time t, I is the moment of inertia of the motor, and $\omega(t)$ is the angular velocity of the motor.

   (a) Assuming the motor is initially at rest, rewrite (2.15) as an integral equation.

   (b) Assuming that both x and i are inputs and ω is an output, construct an actor model (a block diagram) that models this motor. You should use only primitive actors such as integrators and basic arithmetic actors such as scale and adder.

   (c) In reality, the input to a DC motor is not a current, but is rather a voltage. If we assume that the inductance of the motor windings is negligible, then the relationship between voltage and current is given by

$$v(t) = Ri(t) + k_b\omega(t),$$

   where R is the resistance of the motor windings and k_b is a constant called the motor back electromagnetic force constant. The second term appears because a rotating motor also functions as an electrical generator, where the voltage generated is proportional to the angular velocity.

   Modify your actor model so that the inputs are v and x rather than i and x.

7. (a) Using your favorite continuous-time modeling software (such as LabVIEW, Simulink, or Ptolemy II), construct a model of the helicopter control system shown in Figure 2.4. Choose some reasonable parameters and plot the actual angular velocity as a function of time, assuming that the desired angular velocity is zero, $\psi(t) = 0$, and that the top-rotor torque is non-zero, $T_t(t) = bu(t)$. Give your plot for several values of K and discuss how the behavior varies with K.

   (b) Modify the model of part (a) to replace the Controller of Figure 2.4 (the simple scale-by-K actor) with the alternative controller shown in Figure 2.6. This

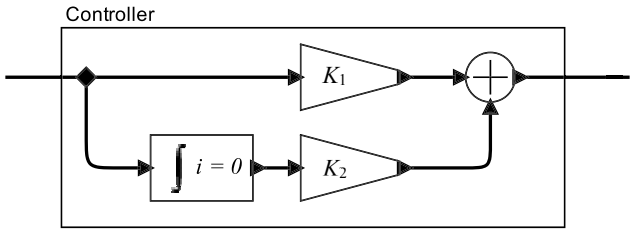


Figure 2.6: A PI controller for the helicopter.

alternative controller is called a **proportional-integrator (PI) controller**. It has two parameter K_1 and K_2. Experiment with the values of these parameters, give some plots of the behavior with the same inputs as in part (a), and discuss the behavior of this controller in contrast to the one of part (a).

3 Discrete Dynamics

Models of embedded systems include both **discrete** and **continuous** components. Loosely speaking, continuous components evolve smoothly, while discrete components evolve abruptly. The previous chapter considered continuous components, and showed that the physical dynamics of the system can often be modeled with ordinary differential or integral equations, or equivalently with actor models that mirror these equations. Discrete components, on the other hand, are not conveniently modeled by ODEs. In this chapter, we study how state machines can be used to model discrete dynamics. In the next chapter, we will show how these state machines can be combined with models of continuous dynamics to get hybrid system models.

3.1 Discrete Systems

A **discrete system** operates in a sequence of discrete steps and is said to have **discrete dynamics**. Some systems are inherently discrete.

Example 3.1: Consider a system that counts the number of cars that enter and leave a parking garage in order to keep track of how many cars are in the garage at any time. It could be modeled as shown in Figure 3.1. We ignore for now how to design the sensors that detect the entry or departure of cars. We simply assume that the ArrivalDetector actor produces an event when a car arrives, and the DepartureDetector actor produces an event when a car departs. The Counter

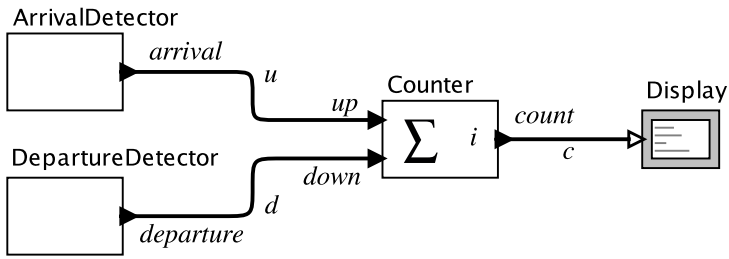


Figure 3.1: Model of a system that keeps track of the number of cars in a parking garage.

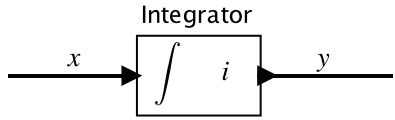


Figure 3.2: Icon for the Integrator actor used in the previous chapter.

actor keeps a running count, starting from an initial value i. Each time the count changes, it produces an output event that updates a display.

In the above example, each entry or departure is modeled as a **discrete event**. A discrete event occurs at an instant of time rather than over time. The Counter actor in Figure 3.1 is analogous to the Integrator actor used in the previous chapter, shown here in Figure 3.2. Like the Counter actor, the Integrator accumulates input values. However, it does so very differently. The input of an Integrator is a function of the form $x: \mathbb{R} \to \mathbb{R}$ or $x: \mathbb{R}_+ \to \mathbb{R}$, a continuous-time signal. The signal u going into the *up* input port of the Counter, on the other hand, is a function of the form

$$u: \mathbb{R} \to \{absent, present\}.$$

This means that at any time $t \in \mathbb{R}$, the input $u(t)$ is either *absent*, meaning that there is no event at that time, or *present*, meaning that there is. A signal of this form is known as a **pure signal**. It carries no value, but instead provides all its information by being either present or absent at any given time. The signal d in Figure 3.1 is also a pure signal.

Assume our Counter operates as follows. When an event is present at the *up* input port, it increments its count and produces on the output the new value of the count. When an event is present at the *down* input, it decrements its count and produces on the output the new value of the count.[1] At all other times (when both inputs are absent), it produces no output (the *count* output is absent). Hence, the signal c in Figure 3.1 can be modeled by a function of the form

$$c: \mathbb{R} \to \{absent\} \cup \mathbb{Z}\ .$$

(See Appendix A for notation.) This signal is not pure, but like u and d, it is either absent or present. Unlike u and d, when it is present, it has a value (an integer).

[1] It would be wise to design this system with a fault handler that does something reasonable if the count drops below zero, but we ignore this for now.

Assume further that the inputs are absent most of the time, or more technically, that the inputs are discrete (see the sidebar on page 45). Then the Counter reacts in sequence to each of a sequence of input events. This is very different from the Integrator, which reacts continuously to a continuum of inputs.

The input to the Counter is a pair of discrete signals that at certain times have an event (are present), and at other times have no event (are absent). The output also is a discrete signal that, when an input is present, has a value that is a natural number, and at other times is absent.[2] Clearly, there is no need for this Counter to do anything when the input is absent. It only needs to operate when inputs are present. Hence, it has discrete dynamics.

[2] As shown in Exercise 8, the fact that input signals are discrete does not necessarily imply that the output signal is discrete. However, for this application, there are physical limitations on the rates at which cars can arrive and depart that ensure that these signals are discrete. So it is safe to assume that they are discrete.

Probing Further: Discrete Signals

Discrete signals consist of a sequence of instantaneous events in time. Here, we make this intuitive concept precise.

Consider a signal of the form $e: \mathbb{R} \to \{absent\} \cup X$, where X is any set of values. This signal is a **discrete signal** if, intuitively, it is absent most of the time and we can count, in order, the times at which it is present (not absent). Each time it is present, we have a discrete event.

This ability to count the events in order is important. For example, if e is present at all rational numbers t, then we do not call this signal discrete. The times at which it is present cannot be counted in order. It is not, intuitively, a sequence of instantaneous events in time (it is a *set* of instantaneous events in time, but not a *sequence*).

To define this formally, let $T \subseteq \mathbb{R}$ be the set of times where e is present. Specifically,

$$T = \{t \in \mathbb{R} \ : \ e(t) \neq absent\}.$$

Then e is discrete if there exists a one-to-one function $f: T \to \mathbb{N}$ that is **order preserving**. Order preserving simply means that for all $t_1, t_2 \in T$ where $t_1 \leq t_2$, we have that $f(t_1) \leq f(t_2)$. The existence of such a one-to-one function ensures that we can count off the events *in temporal order*. Some properties of discrete signals are studied in Exercise 8.

The dynamics of a discrete system can be described as a sequence of steps that we call **reaction**s, each of which we assume to be instantaneous. Reactions of a discrete system are triggered by the environment in which the discrete system operates. In the case of the example of Figure 3.1, reactions of the Counter actor are triggered when one or more input events are present. That is, in this example, reactions are **event triggered**. When both inputs to the Counter are absent, no reaction occurs.

Probing Further: Modeling Actors as Functions

As in Section 2.2, the Integrator actor of Figure 3.2 can be modeled by a function of the form

$$I_i : \mathbb{R}^{\mathbb{R}_+} \to \mathbb{R}^{\mathbb{R}_+},$$

which can also be written

$$I_i : (\mathbb{R}_+ \to \mathbb{R}) \to (\mathbb{R}_+ \to \mathbb{R}).$$

(See Appendix A if the notation is unfamiliar.) In the figure,

$$y = I_i(x) \ ,$$

where i is the initial value of the integration and x and y are continuous-time signals. For example, if $i = 0$ and for all $t \in \mathbb{R}_+$, $x(t) = 1$, then

$$y(t) = i + \int_0^t x(\tau) d\tau = t \ .$$

Similarly, the Counter in Figure 3.1 can be modeled by a function of the form

$$C_i : (\mathbb{R}_+ \to \{absent, present\})^P \to (\mathbb{R}_+ \to \{absent\} \cup \mathbb{Z}),$$

where $\mathbb{Z}$ is the integers and P is the set of input ports, $P = \{up, down\}$. Recall that the notation A^B denotes the set of all functions from B to A. Hence, the input to the function C is a function whose domain is P that for each port $p \in P$ yields a function in $(\mathbb{R}_+ \to \{absent, present\})$. That latter function, in turn, for each time $t \in \mathbb{R}_+$ yields either *absent* or *present*.

A particular reaction will observe the values of the inputs at a particular time t and calculate output values for that same time t. Suppose an actor has input ports $P = \{p_1, \cdots, p_N\}$, where p_i is the name of the i-th input port. Assume further that for each input port $p \in P$, a set V_p denotes the values that may be received on port p when the input is present. V_p is called the **type** of port p. At a reaction we treat each $p \in P$ as a variable that takes on a value $p \in V_p \cup \{absent\}$. A **valuation** of the inputs P is an assignment of a value in V_p to each variable $p \in P$ or an assertion that p is absent.

If port p receives a pure signal, then $V_p = \{present\}$, a singleton set (set with only one element). The only possible value when the signal is not absent is *present*. Hence, at a reaction, the variable p will have a value in the set $\{present, absent\}$.

Example 3.2: For the garage counter, the set of input ports is $P = \{up, down\}$. Both receive pure signals, so the types are $V_{up} = V_{down} = \{present\}$. If a car is arriving at time t and none is departing, then at that reaction, $up = present$ and $down = absent$. If a car is arriving and another is departing at the same time, then $up = down = present$. If neither is true, then both are *absent*.

Outputs are similarly designated. Consider a discrete system with output ports $Q = \{q_1, \cdots, q_M\}$ with types $V_{q_1}, \cdots, V_{q_M}$. At each reaction, the system assigns a value $q \in V_q \cup \{absent\}$ to each $q \in Q$, producing a valuation of the outputs. In this chapter, we will assume that the output is *absent* at times t where a reaction does not occur. Thus, outputs of a discrete system are discrete signals. Chapter 4 describes systems whose outputs are not constrained to be discrete (see also box on page 58).

Example 3.3: The Counter actor of Figure 3.1 has one output port named *count*, so $Q = \{count\}$. Its type is $V_{count} = \mathbb{Z}$. At a reaction, *count* is assigned the count of cars in the garage.

3.2 The Notion of State

Intuitively, the **state** of a system is its condition at a particular point in time. In general, the state affects how the system reacts to inputs. Formally, we define the state to be an encoding of everything about the past that has an effect on the system's reaction to current or future inputs. The state is a summary of the past.

Consider the Integrator actor shown in Figure 3.2. This actor has state, which in this case happens to have the same value as the output at any time t. The state of the actor at a time t is the value of the integral of the input signal up to time t. In order to know how the subsystem will react to inputs at and beyond time t, we have to know what this value is at time t. We do not need to know anything more about the past inputs. Their effect on the future is entirely captured by the current value at t. The icon in Figure 3.2 includes i, an initial state value, which is needed to get things started at some starting time.

An Integrator operates in a time continuum. It integrates a continuous-time input signal, generating as output at each time the cumulative area under the curve given by the input plus the initial state. Its state at any given time is that accumulated area plus the initial state. The Counter actor in the previous section also has state, and that state is also an accumulation of past input values, but it operates discretely.

The state $y(t)$ of the Integrator at time t is a real number. Hence, we say that the **state space** of the Integrator is $States = \mathbb{R}$. For the Counter used in Figure 3.1, the state $s(t)$ at time t is an integer, so $States \subset \mathbb{Z}$. A practical parking garage has a finite and non-negative number M of spaces, so the state space for the Counter actor used in this way will be

$$States = \{0, 1, 2, \cdots, M\} .$$

(This assumes the garage does not let in more cars than there are spaces.) The state space for the Integrator is infinite (uncountably infinite, in fact). The state space for the garage counter is finite. Discrete models with finite state spaces are called finite-state machines (FSMs). There are powerful analysis techniques available for such models, so we consider them next.

3.3 Finite-State Machines

A **state machine** is a model of a system with discrete dynamics that at each reaction maps valuations of the inputs to valuations of the outputs, where the map may depend on

its current state. A **finite-state machine** (**FSM**) is a state machine where the set *States* of possible states is finite.

If the number of states is reasonably small, then FSMs can be conveniently drawn using a graphical notation like that in Figure 3.3. Here, each state is represented by a bubble, so for this diagram, the set of states is given by

$$States = \{\mathsf{State1}, \mathsf{State2}, \mathsf{State3}\}.$$

At the beginning of each sequence of reactions, there is an **initial state**, State1, indicated in the diagram by a dangling arrow into it.

3.3.1 Transitions

Transitions between states govern the discrete dynamics of the state machine and the mapping of input valuations to output valuations. A transition is represented as a curved arrow, as shown in Figure 3.3, going from one state to another. A transition may also start and end at the same state, as illustrated with State3 in the figure. In this case, the transition is called a **self transition**.

In Figure 3.3, the transition from State1 to State2 is labeled with "guard / action." The **guard** determines whether the transition may be taken on a reaction. The **action** specifies what outputs are produced on each reaction.

A guard is a **predicate** (a boolean-valued expression) that evaluates to *true* when the transition should be taken, changing the state from that at the beginning of the transition to that at the end. When a guard evaluates to *true* we say that the transition is **enabled**. An action is an assignment of values (or *absent*) to the output ports. Any output port not

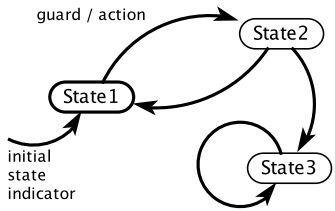


Figure 3.3: Visual notation for a finite state machine.

mentioned in a transition that is taken is implicitly *absent*. If no action at all is given, then all outputs are implicitly *absent*.

Example 3.4: Figure 3.4 shows an FSM model for the garage counter. The inputs and outputs are shown using the notation *name* : *type*. The set of states is $States = \{0, 1, 2, \cdots, M\}$. The transition from state 0 to 1 has a guard written as $up \wedge \neg down$. This is a predicate that evaluates to true when *up* is present and *down* is absent. If at a reaction the current state is 0 and this guard evaluates to true, then the transition will be taken and the next state will be 1. Moreover, the action indicates that the output should be assigned the value 1. The output port *count* is not explicitly named because there is only one output port, and hence there is no ambiguity.

If the guard expression on the transition from 0 to 1 had been simply *up*, then this could evaluate to true when *down* is also present, which would incorrectly count cars when a car was arriving at the same time that another was departing.

If p_1 and p_2 are pure inputs to a discrete system, then the following are examples of valid guards:

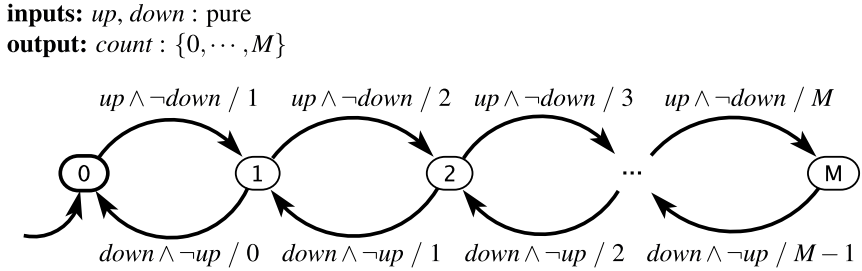


Figure 3.4: FSM model for the garage counter of Figure 3.1.

true	Transition is always enabled.
p_1	Transition is enabled if p_1 is *present*.
$\neg p_1$	Transition is enabled if p_1 is *absent*.
$p_1 \wedge p_2$	Transition is enabled if both p_1 and p_2 are *present*.
$p_1 \vee p_2$	Transition is enabled if either p_1 or p_2 is *present*.
$p_1 \wedge \neg p_2$	Transition is enabled if p_1 is *present* and p_2 is *absent*.

These are standard logical operators where *present* is taken as a synonym for *true* and *absent* as a synonym for *false*. The symbol $\neg$ represents logical **negation**. The operator $\wedge$ is logical **conjunction** (logical AND), and $\vee$ is logical **disjunction** (logical OR).

Suppose that in addition the discrete system has a third input port p_3 with type $V_{p_3} = \mathbb{N}$. Then the following are examples of valid guards:

p_3	Transition is enabled if p_3 is *present* (not *absent*).
$p_3 = 1$	Transition is enabled if p_3 is *present* and has value 1.
$p_3 = 1 \wedge p_1$	Transition is enabled if p_3 has value 1 and p_1 is *present*.
$p_3 > 5$	Transition is enabled if p_3 is *present* with value greater than 5.

Example 3.5: A major use of energy worldwide is in heating, ventilation, and air conditioning (**HVAC**) systems. Accurate models of temperature dynamics and temperature control systems can significantly improve energy conservation. Such modeling begins with a modest **thermostat**, which regulates temperature to maintain a **setpoint**, or target temperature. The word "thermostat" comes from Greek words for "hot" and "to make stand."

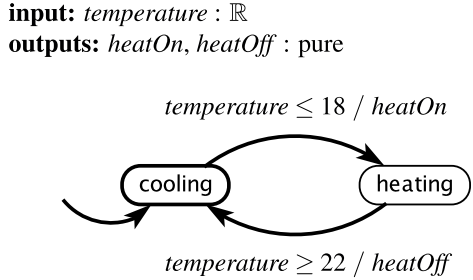


Figure 3.5: A model of a thermostat with hysteresis.

Consider a thermostat modeled by an FSM with $States = \{\text{heating}, \text{cooling}\}$ as shown in Figure 3.5. Suppose the setpoint is 20 degrees Celsius. If the heater is on, then the thermostat allows the temperature to rise past the setpoint to 22 degrees. If the heater is off, then it allows the temperature to drop past the setpoint to 18 degrees. This strategy is called hysteresis (see box on page 53). It avoids **chattering**, where the heater would turn on and off rapidly when the temperature is close to the setpoint temperature.

There is a single input *temperature* with type $\mathbb{R}$ and two pure outputs *heatOn* and *heatOff*. These outputs will be *present* only when a change in the status of the heater is needed (i.e., when it is on and needs to be turned off, or when it is off and needs to be turned on).

The FSM in Figure 3.5 could be event triggered, like the garage counter, in which case it will react whenever a *temperature* input is provided. Alternatively, it could be **time triggered**, meaning that it reacts at regular time intervals. The definition of the FSM does not change in these two cases. It is up to the environment in which an FSM operates when it should react.

On a transition, the **action** (which is the portion after the slash) specifies the resulting valuation on the output ports when a transition is taken. If q_1 and q_2 are pure outputs and q_3 has type $\mathbb{N}$, then the following are examples of valid actions:

q_1	q_1 is present and q_2 and q_3 are *absent*.
q_1, q_2	q_1 and q_2 are both *present* and q_3 is *absent*.
$q_3 := 1$	q_1 and q_2 are *absent* and q_3 is *present* with value 1.
$q_3 := 1, q_1$	q_1 is *present*, q_2 is *absent*, and q_3 is *present* with value 1.
	(nothing) q_1, q_2, and q_3 are all *absent*.

Any output port that is not mentioned in a transition that is taken is implicitly *absent*. When assigning a value to an output port, we use the notation $name := value$ to distinguish the **assignment** from a predicate, which would be written $name = value$. As in Figure 3.4, if there is only one output, then the assignment need not mention the port name.

3.3.2 When a Reaction Occurs

Nothing in the definition of a state machine constrains *when* it reacts. The environment determines when the machine reacts. Chapters 5 and 6 describe a variety of mechanisms

and give a precise meaning to terms like event triggered and time triggered. For now, however, we just focus on what the machine does when it reacts.

When the environment determines that a state machine should react, the inputs will have a valuation. The state machine will assign a valuation to the output ports and (possibly)

Probing Further: Hysteresis

The thermostat in Example 3.5 exhibits a particular form of state-dependent behavior called **hysteresis**. Hysteresis is used to prevent chattering. A system with hysteresis has memory, but in addition has a useful property called **time-scale invariance**. In Example 3.5, the input signal as a function of time is a signal of the form

$$\mathit{temperature} : \mathbb{R} \to \{\mathit{absent}\} \cup \mathbb{R} \, .$$

Hence, $\mathit{temperature}(t)$ is the temperature reading at time t, or *absent* if there is no temperature reading at that time. The output as a function of time has the form

$$\mathit{heatOn}, \mathit{heatOff} : \mathbb{R} \to \{\mathit{absent}, \mathit{present}\} \, .$$

Suppose that instead of *temperature* the input is given by

$$\mathit{temperature}'(t) = \mathit{temperature}(\alpha \cdot t)$$

for some $\alpha > 0$. If $\alpha > 1$, then the input varies faster in time, whereas if $\alpha < 1$ then the input varies more slowly, but in both cases, the input pattern is the same. Then for this FSM, the outputs *heatOn*′ and *heatOff*′ are given by

$$\mathit{heatOn}'(t) = \mathit{heatOn}(\alpha \cdot t) \qquad \mathit{heatOff}'(t) = \mathit{heatOff}(\alpha \cdot t) \, .$$

Time-scale invariance means that scaling the time axis at the input results in scaling the time axis at the output, so the absolute time scale is irrelevant.

An alternative implementation for the thermostat would use a single temperature threshold, but instead would require that the heater remain on or off for at least a minimum amount of time, regardless of the temperature. The consequences of this design choice are explored in Exercise 2.

change to a new state. If no guard on any transition out of the current state evaluates to true, then the machine will remain in the same state.

It is possible for all inputs to be absent at a reaction. Even in this case, it may be possible for a guard to evaluate to true, in which case a transition is taken. If the input is absent and no guard on any transition out of the current state evaluates to true, then the machine will **stutter**. A **stuttering** reaction is one where the inputs and outputs are all absent and the machine does not change state. No progress is made and nothing changes.

Example 3.6: In Figure 3.4, if on any reaction both inputs are absent, then the machine will stutter. If we are in state 0 and the input *down* is *present*, then the guard on the only outgoing transition is false, and the machine remains in the same state. However, we do not call this a stuttering reaction because the inputs are not all *absent*.

Our informal description of the garage counter in Example 3.1 did not explicitly state what would happen if the count was at 0 and a car departed. A major advantage of FSM models is that they define all possible behaviors. The model in Figure 3.4 defines what happens in this circumstance. The count remains at 0. As a consequence, FSM models are amenable to formal checking, which determines whether the specified behaviors are in fact desirable behaviors. The informal specification cannot be subjected to such tests, or at least, not completely.

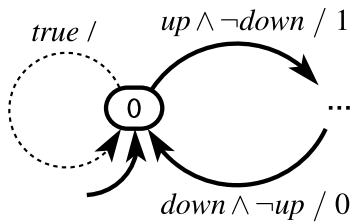


Figure 3.6: A default transition that need not be shown explicitly because it returns to the same state and produces no output.

Although it may seem that the model in Figure 3.4 does not define what happens if the state is 0 and *down* is *present*, it does so implicitly — the state remains unchanged and no output is generated. The reaction is not shown explicitly in the diagram. Sometimes it is useful to emphasize such reactions, in which case they can be shown explicitly. A convenient way to do this is using a **default transition**, shown in Figure 3.6. In that figure, the default transition is denoted with dashed lines and is labeled with "*true* / ". A default transition is enabled if no non-default transition is enabled and if its guard evaluates to true. In Figure 3.6, therefore, the default transition is enabled if $up \wedge \neg down$ evaluates to false, and when the default transition is taken the output is absent.

Default transitions provide a convenient notation, but they are not really necessary. Any default transition can be replaced by an ordinary transition with an appropriately chosen guard. For example, in Figure 3.6 we could use an ordinary transition with guard $\neg(up \wedge \neg down)$.

The use of both ordinary transitions and default transitions in a diagram can be thought of as a way of assigning priority to transitions. An ordinary transition has priority over a default transition. When both have guards that evaluate to true, the ordinary transition prevails. Some formalisms for state machines support more than two levels of priority. For example SyncCharts (André, 1996) associates with each transition an integer priority. This can make guard expressions simpler, at the expense of having to indicate priorities in the diagrams.

3.3.3 Update Functions

The graphical notation for FSMs defines a specific mathematical model of the dynamics of a state machine. A mathematical notation with the same meaning as the graphical notation sometimes proves convenient, particularly for large state machines where the graphical notation becomes cumbersome. In such a mathematical notation, a finite-state machine is a five-tuple

$$(States, Inputs, Outputs, update, initialState)$$

where

- *States* is a finite set of states;
- *Inputs* is a set of input valuations;
- *Outputs* is a set of output valuations;
- $update : States \times Inputs \rightarrow States \times Outputs$ is an **update function**, mapping a state and an input valuation to a *next* state and an output valuation;

- *initialState* is the initial state.

The FSM reacts in a sequence of reactions. At each reaction, the FSM has a *current state*, and the reaction may transition to a *next state*, which will be the current state of the next reaction. We can number these states starting with 0 for the initial state. Specifically, let $s: \mathbb{N} \to$ *States* be a function that gives the state of an FSM at reaction $n \in \mathbb{N}$. Initially, $s(0) =$ *initialState*.

Let $x: \mathbb{N} \to$ *Inputs* and $y: \mathbb{N} \to$ *Outputs* denote that input and output valuations at each reaction. Hence, $x(0) \in$ *Inputs* is the first input valuation and $y(0) \in$ *Outputs* is the first output valuation. The dynamics of the state machine are given by the following equation:

$$(s(n+1), y(n)) = \mathit{update}(s(n), x(n)) \tag{3.1}$$

This gives the next state and output in terms of the current state and input. The *update* function encodes all the transitions, guards, and output specifications in an FSM. The term **transition function** is often used in place of update function.

The input and output valuations also have a natural mathematical form. Suppose an FSM has input ports $P = \{p_1, \cdots, p_N\}$, where each $p \in P$ has a corresponding type V_p. Then

Software Tools Supporting FSMs

FSMs have been used in theoretical computer science and software engineering for quite some time (Hopcroft and Ullman, 1979). A number of software tools support design and analysis of FSMs. Statecharts (Harel, 1987), a notation for concurrent composition of hierarchical FSMs, has influenced many of these tools. One of the first tools supporting the Statecharts notation is STATEMATE (Harel et al., 1990), which subsequently evolved into Rational Rhapsody, sold by IBM. Many variants of Statecharts have arisen (von der Beeck, 1994), and some variant is now supported by nearly every software engineering tool that provides UML (unified modeling language) capabilities (Booch et al., 1998). SyncCharts (André, 1996) is a particularly nice variant in that it borrows the rigorous semantics of Esterel (Berry and Gonthier, 1992) for composition of concurrent FSMs. LabVIEW supports a variant of Statecharts that can operate within dataflow diagrams, and Simulink with its Stateflow extension supports a variant that can operate within continuous-time models.

Inputs is a set of functions of the form

$$i\colon P \to V_{p_1} \cup \cdots \cup V_{p_N} \cup \{\mathit{absent}\}\ ,$$

where for each $p \in P$, $i(p) \in V_p \cup \{\mathit{absent}\}$ gives the value of port p. Thus, a function $i \in \mathit{Inputs}$ is a valuation of the input ports.

Example 3.7: The FSM in Figure 3.4 can be mathematically represented as follows:

$$\begin{aligned} \mathit{States} &= \{0, 1, \cdots, M\} \\ \mathit{Inputs} &= (\{\mathit{up}, \mathit{down}\} \to \{\mathit{present}, \mathit{absent}\}) \\ \mathit{Outputs} &= (\{\mathit{count}\} \to \{0, 1, \cdots, M, \mathit{absent}\}) \\ \mathit{initialState} &= 0 \end{aligned}$$

The update function is given by

$$\mathit{update}(s, i) = \begin{cases} (s+1, s+1) & \text{if } s < M \\ & \quad \wedge\, i(\mathit{up}) = \mathit{present} \\ & \quad \wedge\, i(\mathit{down}) = \mathit{absent} \\ (s-1, s-1) & \text{if } s > 0 \\ & \quad \wedge\, i(\mathit{up}) = \mathit{absent} \\ & \quad \wedge\, i(\mathit{down}) = \mathit{present} \\ (s, \mathit{absent}) & \text{otherwise} \end{cases} \tag{3.2}$$

for all $s \in \mathit{States}$ and $i \in \mathit{Inputs}$. Note that an output valuation $o \in \mathit{Outputs}$ is a function of the form $o\colon \{\mathit{count}\} \to \{0, 1, \cdots, M, \mathit{absent}\}$. In (3.2), the first alternative gives the output valuation as $o = s + 1$, which we take to mean the constant function that for all $q \in Q = \{\mathit{count}\}$ yields $o(q) = s + 1$. When there is more than one output port we will need to be more explicit about which output value is assigned to which output port. In such cases, we can use the same notation that we use for actions in the diagrams.

Moore Machines and Mealy Machines

The state machines we describe in this chapter are known as **Mealy machine**s, named after George H. Mealy, a Bell Labs engineer who published a description of these machines in 1955 (Mealy, 1955). Mealy machines are characterized by producing outputs when a transition is taken. An alternative, known as a **Moore machine**, produces outputs when the machine is in a state, rather than when a transition is taken. That is, the output is defined by the current state rather than by the current transition. Moore machines are named after Edward F. Moore, another Bell Labs engineer who described them in a 1956 paper (Moore, 1956).

The distinction between these machines is subtle but important. Both are discrete systems, and hence their operation consists of a sequence of discrete reactions. For a Moore machine, at each reaction, the output produced is defined by the current state (at the *start* of the reaction, not at the end). Thus, the output at the time of a reaction does not depend on the input at that same time. The input determines which transition is taken, but not what output is produced by the reaction. Hence, a Moore machine is strictly causal.

A Moore machine version of the garage counter is shown in Figure 3.7. The outputs are shown in the state rather than on the transitions using a similar notation with a slash. Note, however, that this machine is *not* equivalent to the machine in Figure 3.4. To see that, suppose that on the first reaction, $up = present$ and $down = absent$. The output at that time will be 0 in Figure 3.7 and 1 in Figure 3.4. The output of the Moore machine represents the number of cars in the garage at the time of the arrival of a new car, not the number of cars after the arrival of the new car. Suppose instead that at the first reaction, $up = down = absent$. Then the output at that time is 0 in Figure 3.7 and *absent* in Figure 3.4. The Moore machine, when it reacts, always reports the output associated with the current state. The Mealy machine does not produce any output unless there is a transition explicitly denoting that output.

Any Moore machine may be converted to an equivalent Mealy machine. A Mealy machine may be converted to an almost equivalent Moore machine that differs only in that the output is produced on the *next* reaction rather than on the current one. We use Mealy machines because they tend to be more compact (requiring fewer states to represent the same functionality), and because it is convenient to be able to produce an output that instantaneously responds to the input.

inputs: *up*, *down*: pure
output: *count*: $\{0, \cdots, M\}$

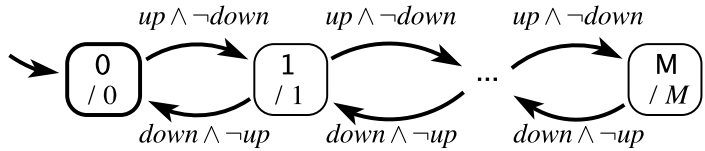


Figure 3.7: Moore machine for a system that keeps track of the number of cars in a parking garage. Note this machine is *not* equivalent to that in Figure 3.4.

3.3.4 Determinacy and Receptiveness

The state machines presented in this section have two important properties:

Determinacy: A state machine is said to be **deterministic** if, for each state, there is at most one transition enabled by each input value. The formal definition of an FSM given above ensures that it is deterministic, since *update* is a function, not a one-to-many mapping. The graphical notation with guards on the transitions, however, has no such constraint. Such a state machine will be deterministic only if the guards leaving each state are non-overlapping. Note that a deterministic state machine is **determinate**, meaning that given the same inputs it will always produce the same outputs. However, not every determinate state machine is deterministic.

Receptiveness: A state machine is said to be **receptive** if, for each state, there is at least one transition possible on each input symbol. In other words, receptiveness ensures that a state machine is always ready to react to any input, and does not "get stuck" in any state. The formal definition of an FSM given above ensures that it is receptive, since *update* is a function, not a partial function. It is defined for every possible state and input value. Moreover, in our graphical notation, since we have implicit default transitions, we have ensured that all state machines specified in our graphical notation are also receptive.

It follows that if a state machine is both deterministic and receptive, for every state, there is *exactly* one transition possible on each input value.

3.4 Extended State Machines

The notation for FSMs becomes awkward when the number of states gets large. The garage counter of Figure 3.4 illustrates this point clearly. If M is large, the bubble-and-arc notation becomes unwieldy, which is why we resort to a less formal use of "..." in the figure.

An **extended state machine** solves this problem by augmenting the FSM model with variables that may be read and written as part of taking a transition between states.

Example 3.8: The garage counter of Figure 3.4 can be represented more compactly by the extended state machine in Figure 3.8.

That figure shows a variable c, declared explicitly at the upper left to make it clear that c is a variable and not an input or an output. The transition indicating the initial state initializes the value of this variable to zero.

The upper self-loop transition is then taken when the input *up* is present, the input *down* is absent, and the variable c is less than M. When this transition is taken, the state machine produces an output *count* with value $c+1$, and then the value of c is incremented by one.

The lower self-loop transition is taken when the input *down* is present, the input *up* is absent, and the variable c is greater than zero. Upon taking the transition,

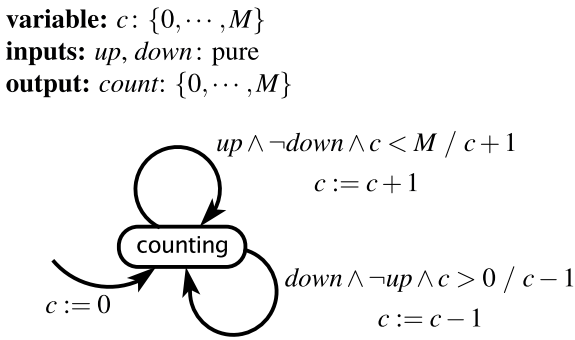


Figure 3.8: Extended state machine for the garage counter of Figure 3.4.

the state machine produces an output with value $c - 1$, and then decrements the value of c.

Note that M is a parameter, not a variable. Specifically, it is assumed to be constant throughout execution.

The general notation for extended state machines is shown in Figure 3.9. This differs from the basic FSM notation of Figure 3.3 in three ways. First, variable declarations are shown explicitly to make it easy to determine whether an identifier in a guard or action refers to a variable or to an input or an output. Second, upon initialization, variables that have been declared may be initialized. The initial value will be shown on the transition that indicates the initial state. Third, transition annotations now have the form

guard / output action
set action(s)

The guard and output action are the same as for standard FSMs, except they may now refer to variables. The **set action**s are new. They specify assignments to variables that are made when the transition is taken. These assignments are made *after* the guard has been evaluated and the outputs have been produced. Thus, if the guard or output actions reference a variable, the value of the variable is that *before* the assignment in the set action. If there is more than one set action, then the assignments are made in sequence.

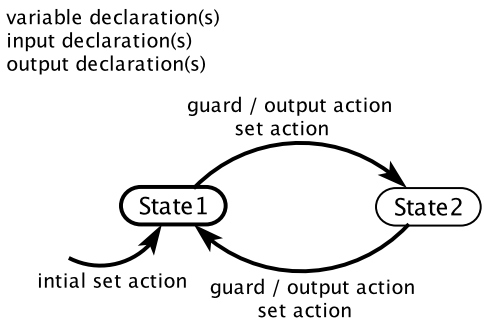


Figure 3.9: Notation for extended state machines.

Extended state machines can provide a convenient way to keep track of the passage of time.

Example 3.9: An extended state machine describing a traffic light at a pedestrian crosswalk is shown in Figure 3.10. This is a time triggered machine that assumes it will react once per second. It starts in the red state and counts 60 seconds with the help of the variable *count*. It then transitions to green, where it will remain until the pure input *pedestrian* is present. That input could be generated, for example, by a pedestrian pushing a button to request a walk light. When *pedestrian* is present, the machine transitions to yellow if it has been in state green for at least 60 seconds. Otherwise, it transitions to pending, where it stays for the remainder of the 60 second interval. This ensures that once the light goes green, it stays green for at least 60 seconds. At the end of 60 seconds, it will transition to yellow, where it will remain for 5 seconds before transitioning back to red.

The outputs produced by this machine are *sigG* to turn on the green light, *sigY* to change the light to yellow, and *sigR* to change the light to red.

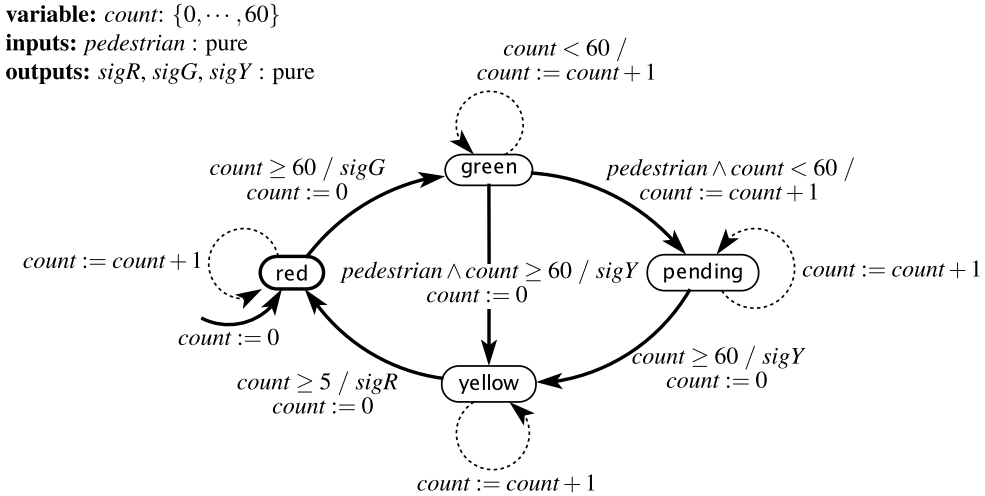


Figure 3.10: Extended state machine model of a traffic light controller that keeps track of the passage of time, assuming it reacts at regular intervals.

The state of an extended state machine includes not only the information about which discrete state (indicated by a bubble) the machine is in, but also what values any variables have. The number of possible states can therefore be quite large, or even infinite. If there are n discrete states (bubbles) and m variables each of which can have one of p possible values, then the size of the state space of the state machine is

$$|States| = np^m \ .$$

Example 3.10: The garage counter of Figure 3.8 has $n = 1$, $m = 1$, and $p = M + 1$, so the total number of states is $M + 1$.

Extended state machines may or may not be FSMs. In particular, it is not uncommon for p to be infinite. For example, a variable may have values in $\mathbb{N}$, the natural numbers, in which case, the number of states is infinite.

Example 3.11: If we modify the state machine of Figure 3.8 so that the guard on the upper transition is

$$up \wedge \neg down$$

instead of

$$up \wedge \neg down \wedge c < M$$

then the state machine is no longer an FSM.

Some state machines will have states that can never be reached, so the set of **reachable state**s — comprising all states that can be reached from the initial state on some input sequence — may be smaller than the set of states.

Example 3.12: Although there are only four bubbles in Figure 3.10, the number of states is actually much larger. The *count* variable has 61 possible values and there are 4 bubbles, so the total number of combinations is $61 \times 4 = 244$. The size of the state space is therefore 244. However, not all of these states are reachable. In particular, while in the yellow state, the *count* variable will have only one of 6 values in $\{0, \cdots, 5\}$. The number of reachable states, therefore, is $61 \times 3 + 6 = 189$.

3.5 Nondeterminism

Most interesting state machines react to inputs and produce outputs. These inputs must come from somewhere, and the outputs must go somewhere. We refer to this "somewhere" as the **environment** of the state machine.

Example 3.13: The traffic light controller of Figure 3.10 has one pure input signal, *pedestrian*. This input is *present* when a pedestrian arrives at the crosswalk. The traffic light will remain green unless a pedestrian arrives. Some other subsystem is responsible for generating the *pedestrian* event, presumably in response to a pedestrian pushing a button to request a cross light. That other subsystem is part of the environment of the FSM in Figure 3.10.

A question becomes how to model the environment. In the traffic light example, we could construct a model of pedestrian flow in a city to serve this purpose, but this would likely be a very complicated model, and it is likely much more detailed than necessary. We want to ignore inessential details, and focus on the design of the traffic light. We can do this using a nondeterministic state machine.

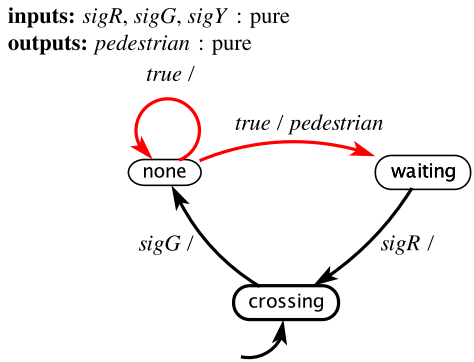


Figure 3.11: Nondeterministic model of pedestrians that arrive at a crosswalk.

Example 3.14: The FSM in Figure 3.11 models arrivals of pedestrians at a crosswalk with a traffic light controller like that in Figure 3.10. This FSM has three inputs, which are presumed to come from the outputs of Figure 3.10. Its single output, *pedestrian*, will provide the input for Figure 3.10.

The initial state is crossing. (Why? See Exercise 6.) When *sigG* is received, the FSM transitions to none. Both transitions from this state have guard *true*, indicating that they are always enabled. Since both are enabled, this machine is nondeterministic. The FSM may stay in the same state and produce no output, or it may transition to waiting and produce pure output *pedestrian*.

The interaction between this machine and that of Figure 3.10 is surprisingly subtle. Variations on the design are considered in Exercise 6, and the composition of the two machines is studied in detail in Chapter 5.

If for any state of a state machine, there are two distinct transitions with guards that can evaluate to *true* in the same reaction, then the state machine is **nondeterministic**. In a diagram for such a state machine, the transitions that make the state machine nondeterministic may be colored red. In the example of Figure 3.11, the transitions exiting state none are the ones that make the state machine nondeterministic.

It is also possible to define state machines where there is more than one initial state. Such a state machine is also nondeterministic. An example is considered in Exercise 6.

In both cases, a nondeterministic FSM specifies a family of possible reactions rather than a single reaction. Operationally, all reactions in the family are possible. The nondeterministic FSM makes no statement at all about how *likely* the various reactions are. It is perfectly correct, for example, to always take the self loop in state none in Figure 3.11. A model that specifies likelihoods (in the form of probabilities) is a **stochastic model**, quite distinct from a nondeterministic model.

3.5.1 Formal Model

Formally, a **nondeterministic FSM** is represented as a five-tuple, similar to a deterministic FSM,

$$(States, Inputs, Outputs, possibleUpdates, initialStates)$$

The first three elements are the same as for a deterministic FSM, but the last two are not the same:

- *States* is a finite set of states;
- *Inputs* is a set of input valuations;
- *Outputs* is a set of output valuations;
- $possibleUpdates : States \times Inputs \rightarrow 2^{States \times Outputs}$ is an **update relation**, mapping a state and an input valuation to a *set of possible* (next state, output valuation) pairs;
- *initialStates* is a set of initial states.

The form of the function *possibleUpdates* indicates there can be more than one next state and/or output valuation given a current state and input valuation. The codomain is the powerset of $States \times Outputs$. We refer to the *possibleUpdates* function as an update *relation*, to emphasize this difference. The term **transition relation** is also often used in place of update relation.

To support the fact that there can be more than one initial state for a nondeterministic FSM, *initialStates* is a set rather than a single element of *States*.

Example 3.15: The FSM in Figure 3.11 can be formally represented as follows:

$$\begin{aligned} States &= \{\text{none}, \text{waiting}, \text{crossing}\} \\ Inputs &= (\{sigG, sigY, sigR\} \to \{present, absent\}) \\ Outputs &= (\{pedestrian\} \to \{present, absent\}) \\ initialStates &= \{\text{crossing}\} \end{aligned}$$

The update relation is given below:

$$possibleUpdates(s,i) = \begin{cases} \{(\text{none}, absent)\} \\ \qquad \text{if } s = \text{crossing} \\ \qquad \wedge\, i(sigG) = present \\ \{(\text{none}, absent), (\text{waiting}, present)\} \\ \qquad \text{if } s = \text{none} \\ \{(\text{crossing}, absent)\} \\ \qquad \text{if } s = \text{waiting} \\ \qquad \wedge\, i(sigR) = present \\ \{(s, absent)\} \qquad \text{otherwise} \end{cases} \tag{3.3}$$

for all $s \in States$ and $i \in Inputs$. Note that an output valuation $o \in Outputs$ is a function of the form o: $\{pedestrian\} \to \{present, absent\}$. In (3.3), the second alternative gives two possible outcomes, reflecting the nondeterminism of the machine.

3.5.2 Uses of Nondeterminism

While nondeterminism is an interesting mathematical concept in itself, it has two major uses in modeling embedded systems:

Environment Modeling: It is often useful to hide irrelevant details about how an environment operates, resulting in a nondeterministic FSM model. We have already seen one example of such environment modeling in Figure 3.11.

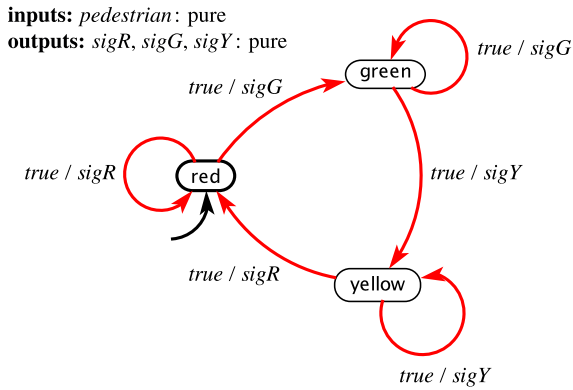


Figure 3.12: Nondeterministic FSM specifying order of signal lights, but not their timing. Notice that it ignores the *pedestrian* input.

Specifications: System specifications impose requirements on some system features, while leaving other features unconstrained. Nondeterminism is a useful modeling technique in such settings as well. For example, consider a specification that the traffic light cycles through red, green, yellow, in that order, without regard for the timing between the outputs. The nondeterministic FSM in Figure 3.12 models this specification. The guard *true* on each transition indicates that the transition can be taken at any step. Technically, it means that each transition is enabled for any input valuation in *Inputs*.

3.6 Behaviors and Traces

An FSM has discrete dynamics. As we did in Section 3.3.3, we can abstract away the passage of time and consider only the *sequence* of reactions, without concern for when in time each reaction occurs. We do not need to talk explicitly about the amount of time that passes between reactions, since this is actually irrelevant to the behavior of an FSM.

Consider a port p of a state machine with type V_p. This port will have a sequence of values from the set $V_p \cup \{absent\}$, one value at each reaction. We can represent this sequence as

a function of the form

$$s_p : \mathbb{N} \to V_p \cup \{absent\} \ .$$

This is the signal received on that port (if it is an input) or produced on that port (if it is an output).

A **behavior** of a state machine is an assignment of such a signal to each port such that the signal on any output port is the output sequence produced for the given input signals.

Example 3.16: The garage counter of Figure 3.4 has input port set $P = \{up, down\}$, with types $V_{up} = V_{down} = \{present\}$, and output port set $Q = \{count\}$ with type $V_{count} = \{0, \cdots, M\}$. An example of input sequences is

$$\begin{aligned} s_{up} &= (present, absent, present, absent, present, \cdots) \\ s_{down} &= (present, absent, absent, present, absent, \cdots) \end{aligned}$$

The corresponding output sequence is

$$s_{count} = (absent, absent, 1, 0, 1, \cdots) \ .$$

These three signals s_{up}, s_{down}, and s_{count} together are a behavior of the state machine. If we let

$$s'_{count} = (1, 2, 3, 4, 5, \cdots) \ ,$$

then s_{up}, s_{down}, and s'_{count} together *are not* a behavior of the state machine. The signal s'_{count} is not produced by reactions to those inputs.

Deterministic state machines have the property that there is exactly one behavior for each set of input sequences. That is, if you know the input sequences, then the output sequence is fully determined. That is, the machine is determinate. Such a machine can be viewed as a function that maps input sequences to output sequences. Nondeterministic state machines can have more than one behavior sharing the same input sequences, and hence cannot be viewed as a function mapping input sequences to output sequences.

The set of all behaviors of a state machine M is called its **language**, written $L(M)$. Since our state machines are receptive, their languages always include all possible input sequences.

A behavior may be more conveniently represented as a sequence of valuations called an **observable trace**. Let x_i represent the valuation of the input ports and y_i the valuation of the output ports at reaction i. Then an observable trace is a sequence

$$((x_0, y_0), (x_1, y_1), (x_2, y_2), \cdots) \ .$$

An observable trace is really just another representation of a behavior.

It is often useful to be able to reason about the states that are traversed in a behavior. An **execution trace** includes the state trajectory, and may be written as a sequence

$$((x_0, s_0, y_0), (x_1, s_1, y_1), (x_2, s_2, y_2), \cdots) \ ,$$

where $s_0 = \mathit{initialState}$. This can be represented a bit more graphically as follows,

$$s_0 \xrightarrow{x_0/y_0} s_1 \xrightarrow{x_1/y_1} s_2 \xrightarrow{x_2/y_2} \cdots$$

This is an execution trace if for all $i \in \mathbb{N}$, $(s_{i+1}, y_i) = \mathit{update}(s_i, x_i)$ (for a deterministic machine), or $(s_{i+1}, y_i) \in \mathit{possibleUpdates}(s_i, x_i)$ (for a nondeterministic machine).

Example 3.17: Consider again the garage counter of Figure 3.4 with the same input sequences s_{up} and s_{down} from Example 3.16. The corresponding execution trace may be written

$$0 \xrightarrow{up \wedge down \ /} 0 \xrightarrow{\ /\ } 0 \xrightarrow{up \ / \ 1} 1 \xrightarrow{down \ / \ 0} 0 \xrightarrow{up \ / \ 1} \cdots$$

Here, we have used the same shorthand for valuations that is used on transitions in Section 3.3.1. For example, the label "*up* / 1" means that *up* is present, *down* is absent, and *count* has value 1. Any notation that clearly and unambiguously represents the input and output valuations is acceptable.

For a nondeterministic machine, it may be useful to represent all the possible traces that correspond to a particular input sequence, or even all the possible traces that result from all possible input sequences. This may be done using a **computation tree**.

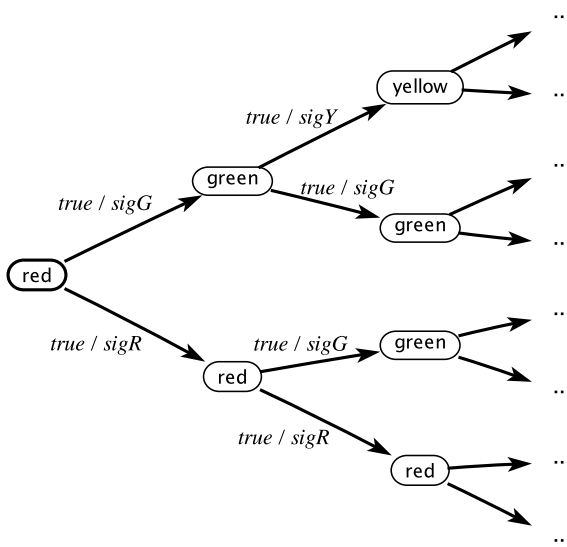


Figure 3.13: A computation tree for the FSM in Figure 3.12.

Example 3.18: Consider the nondeterministic FSM in Figure 3.12. Figure 3.13 shows the computation tree for the first three reactions with any input sequence. Nodes in the tree are states and edges are labeled by the input and output valuations, where the notation *true* means any input valuation.

Traces and computation trees can be valuable for developing insight into the behaviors of a state machine and for verifying that undesirable behaviors are avoided.

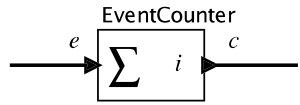
3.7 Summary

This chapter has given an introduction to the use of state machines to model systems with discrete dynamics. It gives a graphical notation that is suitable for finite state machines, and an extended state machine notation that can compactly represent large numbers of

states. It also gives a mathematical model that uses sets and functions rather than visual notations. The mathematical notation can be useful to ensure precise interpretations of a model and to prove properties of a model. This chapter has also discussed nondeterminism, which can provide convenient abstractions that compactly represent families of behaviors.

Exercises

1. Consider an event counter that is a simplified version of the counter in Section 3.1. It has an icon like this:



This actor starts with state i and upon arrival of an event at the input, increments the state and sends the new value to the output. Thus, e is a pure signal, and c has the form $c: \mathbb{R} \to \{absent\} \cup \mathbb{N}$, assuming $i \in \mathbb{N}$. Suppose you are to use such an event counter in a weather station to count the number of times that a temperature rises above some threshold. Your task in this exercise is to generate a reasonable input signal e for the event counter. You will create several versions. For all versions, you will design a state machine whose input is a signal $\tau: \mathbb{R} \to \{absent\} \cup \mathbb{Z}$ that gives the current temperature (in degrees centigrade) once per hour. The output $e: \mathbb{R} \to \{absent, present\}$ will be a pure signal that goes to an event counter.

   (a) For the first version, your state machine should simply produce a *present* output whenever the input is *present* and greater than 38 degrees. Otherwise, the output should be absent.

   (b) For the second version, your state machine should have hysteresis. Specifically, it should produce a *present* output the first time the input is greater than 38 degrees, and subsequently, it should produce a *present* output anytime the input is greater than 38 degrees but has dropped below 36 degrees since the last time a *present* output was produced.

   (c) For the third version, your state machine should implement the same hysteresis as in part (b), but also produce a *present* output at most once per day.

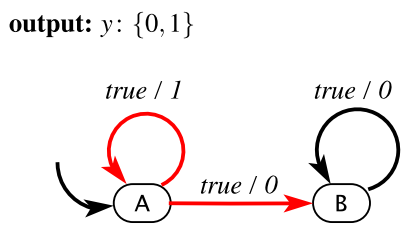
2. Consider a variant of the thermostat of example 3.5. In this variant, there is only one temperature threshold, and to avoid chattering the thermostat simply leaves the heat on or off for at least a fixed amount of time. In the initial state, if the temperature is less than or equal to 20 degrees Celsius, it turns the heater on, and leaves it on for at least 30 seconds. After that, if the temperature is greater than 20 degrees, it turns the heater off and leaves it off for at least 2 minutes. It turns it on again only if the temperature is less than or equal to 20 degrees.

(a) Design an FSM that behaves as described, assuming it reacts exactly once every 30 seconds.

(b) How many possible states does your thermostat have? Is this the smallest number of states possible?

(c) Does this model thermostat have the time-scale invariance property?

3. Consider the following state machine:

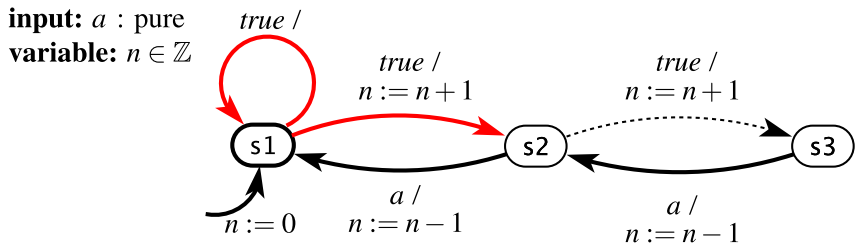


Determine whether the following statement is true or false, and give a supporting argument:

> The output will eventually be a constant 0, or it will eventually be a constant 1. That is, for some $n \in \mathbb{N}$, after the n-th reaction, either the output will be 0 in every subsequent reaction, or it will be 1 in every subsequent reaction.

Note that Chapter 13 gives mechanisms for making such statements precise and for reasoning about them.

4. How many reachable states does the following state machine have?



5. Consider the deterministic finite-state machine in Figure 3.14 that models a simple traffic light.

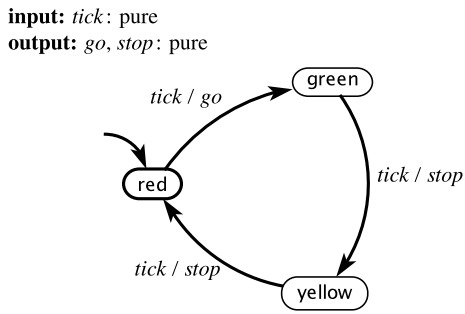


Figure 3.14: Deterministic finite-state machine for Exercise 5

(a) Formally write down the description of this FSM as a 5-tuple:

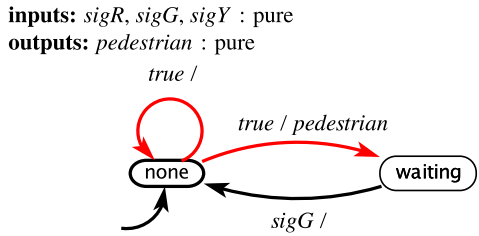
$$(States, Inputs, Outputs, update, initialState).$$

(b) Give an execution trace of this FSM of length 4 assuming the input *tick* is *present* on each reaction.

(c) Now consider merging the red and yellow states into a single stop state. Transitions that pointed into or out of those states are now directed into or out of the new stop state. Other transitions and the inputs and outputs stay the same. The new stop state is the new initial state. Is the resulting state machine deterministic? Why or why not? If it is deterministic, give a prefix of the trace of length 4. If it is nondeterministic, draw the computation tree up to depth 4.

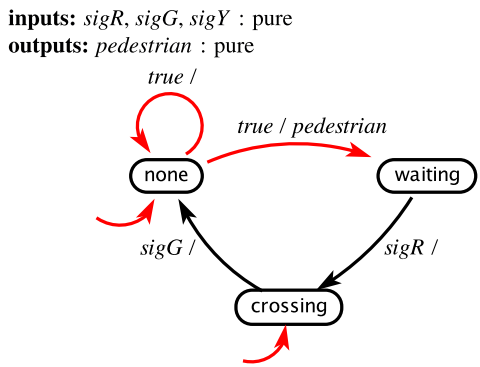
6. This problem considers variants of the FSM in Figure 3.11, which models arrivals of pedestrians at a crosswalk. We assume that the traffic light at the crosswalk is controlled by the FSM in Figure 3.10. In all cases, assume a time triggered model, where both the pedestrian model and the traffic light model react once per second. Assume further that in each reaction, each machine sees as inputs the output produced by the other machine *in the same reaction* (this form of composition, which is called synchronous composition, is studied further in Chapter 6).

(a) Suppose that instead of Figure 3.11, we use the following FSM to model the arrival of pedestrians:



Find a trace whereby a pedestrian arrives (the above machine transitions to waiting) but the pedestrian is never allowed to cross. That is, at no time after the pedestrian arrives is the traffic light in state red.

(b) Suppose that instead of Figure 3.11, we use the following FSM to model the arrival of pedestrians:



Here, the initial state is nondeterministically chosen to be one of none or crossing. Find a trace whereby a pedestrian arrives (the above machine transitions from none to waiting) but the pedestrian is never allowed to cross. That is, at no time after the pedestrian arrives is the traffic light in state red.

7. Consider the state machine in Figure 3.15. State whether each of the following is a behavior for this machine. In each of the following, the ellipsis "$\cdots$" means that the last symbol is repeated forever. Also, for readability, *absent* is denoted by the shorthand a and *present* by the shorthand p.

(a) $x = (p, p, p, p, p, \cdots)$, $\quad y = (0, 1, 1, 0, 0, \cdots)$

(b) $x = (p, p, p, p, p, \cdots)$, $\quad y = (0, 1, 1, 0, a, \cdots)$

(c) $x = (a, p, a, p, a, \cdots)$, $\quad y = (a, 1, a, 0, a, \cdots)$

(d) $x = (p, p, p, p, p, \cdots)$, $\quad y = (0, 0, a, a, a, \cdots)$

(e) $x = (p, p, p, p, p, \cdots), \quad y = (0, a, 0, a, a, \cdots)$

8. (NOTE: This exercise is rather advanced.) This exercise studies properties of discrete signals as formally defined in the sidebar on page 45. Specifically, we will show that discreteness is not a compositional property. That is, when combining two discrete behaviors in a single system, the resulting combination is not necessarily discrete.

   (a) Consider a pure signal $x : \mathbb{R} \to \{present, absent\}$ given by

   $$x(t) = \begin{cases} present & \text{if } t \text{ is a non-negative integer} \\ absent & \text{otherwise} \end{cases}$$

   for all $t \in \mathbb{R}$. Show that this signal is discrete.

   (b) Consider a pure signal $y : \mathbb{R} \to \{present, absent\}$ given by

   $$y(t) = \begin{cases} present & \text{if } t = 1 - 1/n \text{ for any positive integer } n \\ absent & \text{otherwise} \end{cases}$$

   for all $t \in \mathbb{R}$. Show that this signal is discrete.

   (c) Consider a signal w that is the merge of x and y in the previous two parts. That is, $w(t) = present$ if either $x(t) = present$ or $y(t) = present$, and is *absent* otherwise. Show that w is not discrete.

   (d) Consider the example shown in Figure 3.1. Assume that each of the two signals *arrival* and *departure* is discrete. Show that this does not imply that the output *count* is a discrete signal.

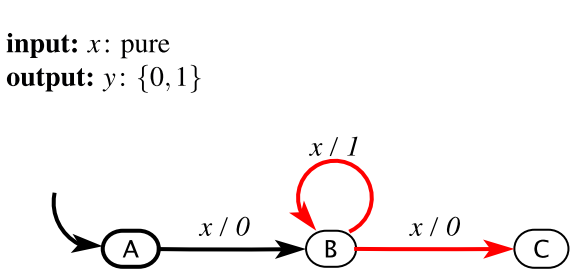


Figure 3.15: State machine for Exercise 7.

4 Hybrid Systems

Chapters 2 and 3 describe two very different modeling strategies, one focused on continuous dynamics and one on discrete dynamics. For continuous dynamics, we use differential equations and their corresponding actor models. For discrete dynamics, we use state machines.

Cyber-physical systems integrate physical dynamics and computational systems, so they commonly combine both discrete and continuous dynamics. In this chapter, we show that the modeling techniques of Chapters 2 and 3 can be combined, yielding what are known as **hybrid systems**. Hybrid system models are often much simpler and more understandable

than brute-force models that constrain themselves to only one of the two styles in Chapters 2 and 3. They are a powerful tool for understanding real-world systems.

4.1 Modal Models

In this section, we show that state machines can be generalized to admit continuous inputs and outputs and to combine discrete and continuous dynamics.

4.1.1 Actor Model for State Machines

In Section 3.3.1 we explain that state machines have inputs defined by the set *Inputs* that may be pure signals or may carry a value. In either case, the state machine has a number of input ports, which in the case of pure signals are either present or absent, and in the case of valued signals have a value at each reaction of the state machine.

We also explain in Section 3.3.1 that actions on transitions set the values of outputs. The outputs can also be represented by ports, and again the ports can carry pure signals or valued signals. In the case of pure signals, a transition that is taken specifies whether the output is present or absent, and in the case of valued signals, it assigns a value or asserts that the signal is absent. Outputs are presumed to be absent between transitions.

Given this input/output view of state machines, it is natural to think of a state machine as an actor, as illustrated in Figure 4.1. In that figure, we assume some number n of input ports named $i_1 \cdots i_n$. At each reaction, these ports have a value that is either *present* or *absent* (if the port carries a pure signal) or a member of some set of values (if the port carries a valued signal). The outputs are similar. The guards on the transitions define subsets of possible values on input ports, and the actions assign values to output ports. Given such an actor model, it is straightforward to generalize FSMs to admit continuous-time signals as inputs.

4.1.2 Continuous Inputs

We have so far assumed that state machines operate in a sequence of discrete reactions. We have assumed that inputs and outputs are absent between reactions. We will now generalize this to allow inputs and outputs to be continuous-time signals.

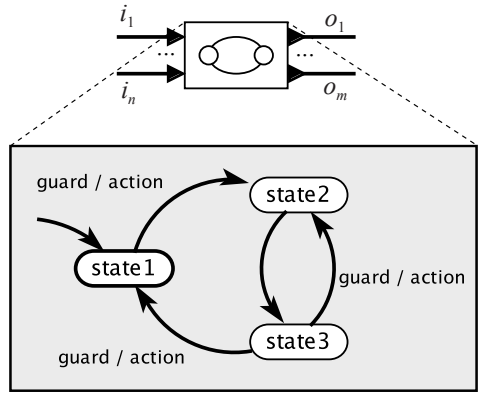


Figure 4.1: An FSM represented as an actor.

In order to get state machine models to coexist with time-based models, we need to interpret state transitions to occur on the same timeline used for the time-based portion of the system. The notion of discrete reactions described in Section 3.1 suffices for this purpose, but we will no longer require inputs and outputs to be absent between reactions. Instead, we will define a transition to occur when a guard on an outgoing transition from the current state becomes enabled. As before, during the time between reactions, a state machine is understood to not transition between modes. But the inputs and outputs are no longer required to be absent during that time.

Example 4.1: Consider a thermostat modeled as a state machine with states $\Sigma = \{\text{heating}, \text{cooling}\}$, shown in Figure 4.2. This is a variant of the model of Example 3.5 where instead of a discrete input that provides a temperature at each reaction, the input is a continuous-time signal $\tau : \mathbb{R} \to \mathbb{R}$ where $\tau(t)$ represents the temperature at time t. The initial state is cooling, and the transition out of this state is enabled at the earliest time t after the start time when $\tau(t) \leq 18$. In this example, we assume the outputs are pure signals *heatOn* and *heatOff*.

In the above example, the outputs are present only at the times the transitions are taken. We can also generalize FSMs to support continuous-time outputs, but to do this, we need the notion of state refinements.

4.1.3 State Refinements

A hybrid system associates with each state of an FSM a dynamic behavior. Our first (very simple) example uses this capability merely to produce continuous-time outputs.

> **Example 4.2:** Suppose that instead of discrete outputs as in Example 4.1 we wish to produce a control signal whose value is 1 when the heat is on and 0 when the heat is off. Such a control signal could directly drive a heater. The thermostat in Figure 4.3 does this. In that figure, each state has a refinement that gives the value of the output h while the state machine is in that state.

In a hybrid system, the current state of the state machine has a **state refinement** that gives the dynamic behavior of the output as a function of the input. In the above simple example, the output is constant in each state, which is rather trivial dynamics. Hybrid systems can get much more elaborate.

The general structure of a hybrid system model is shown in Figure 4.4. In that figure, there is a two-state finite-state machine. Each state is associated with a state refinement

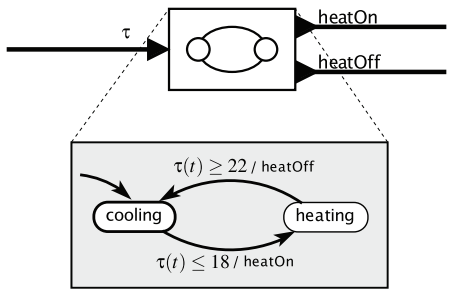


Figure 4.2: A thermostat modeled as an FSM with a continuous-time input signal.

labeled in the figure as a "time-based system." The state refinement defines dynamic behavior of the outputs and (possibly) additional continuous state variables. In addition, each transition can optionally specify set actions, which set the values of such additional state variables when a transition is taken. The example of Figure 4.3 is rather trivial, in that it has no continuous state variables, no output actions, and no set actions.

A hybrid system is sometimes called a **modal model** because it has a finite number of **mode**s, one for each state of the FSM, and when it is in a mode, it has dynamics specified by the state refinement. The states of the FSM may be referred to as modes rather than states, which as we will see, helps prevent confusion with states of the refinements.

The next simplest such dynamics, besides the rather trivial constant outputs of Example 4.2 is found in timed automata, which we discuss next.

4.2 Classes of Hybrid Systems

Hybrid systems can be quite elaborate. In this section, we first describe a relatively simple form known as timed automata. We then illustrate more elaborate forms that model nontrivial physical dynamics and nontrivial control systems.

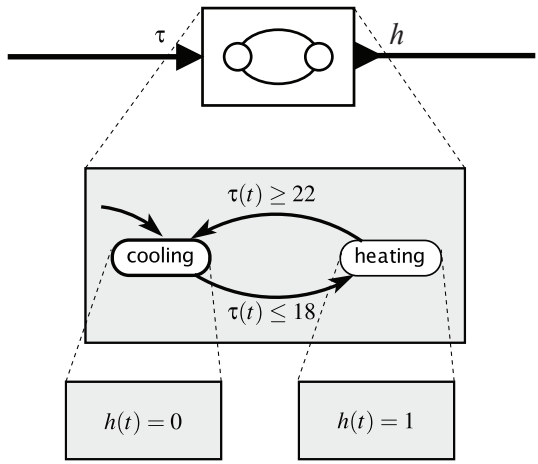


Figure 4.3: A thermostat with continuous-time output.

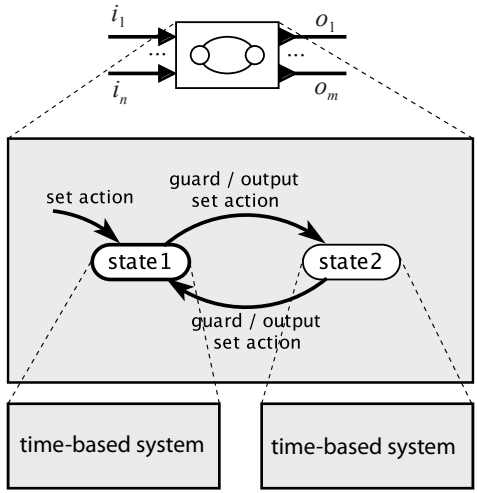


Figure 4.4: Notation for hybrid systems.

4.2.1 Timed Automata

Most cyber-physical systems require measuring the passage of time and performing actions at specific times. A device that measures the passage of time, a **clock**, has a particularly simple dynamics: its state progresses linearly in time. In this section, we describe **timed automata**, a formalism introduced by Alur and Dill (1994), which enable the construction of more complicated systems from such simple clocks.

Timed automata are the simplest non-trivial hybrid systems. They are modal models where the time-based refinements have very simple dynamics; all they do is measure the passage of time. A clock is modeled by a first-order differential equation,

$$\forall\, t \in T_m, \quad \dot{s}(t) = a,$$

where $s\colon \mathbb{R} \to \mathbb{R}$ is a continuous-time signal, $s(t)$ is the value of the clock at time t, and $T_m \subset \mathbb{R}$ is the subset of time during which the hybrid system is in mode m. The rate of the clock, a, is a constant while the system is in this mode.[1]

[1]The variant of timed automata we describe in this chapter differs from the original model of Alur and Dill (1994) in that the rates of clocks in different modes can be different. This variant is sometimes described in the literature as *multi-rate* timed automata.

Example 4.3: Recall the thermostat of Example 4.1, which uses hysteresis to prevent chattering. An alternative implementation that would also prevent chattering would use a single temperature threshold, but instead would require that the heater remain on or off for at least a minimum amount of time, regardless of the temperature. This design would not have the hysteresis property, but may be useful nonetheless. This can be modeled as a timed automaton as shown in Figure 4.5. In that figure, each state refinement has a clock, which is a continuous-time signal s with dynamics given by

$$\dot{s}(t) = 1 \ .$$

The value $s(t)$ increases linearly with t. Note that in that figure, the state refinement is shown directly with the name of the state in the state bubble. This shorthand is convenient when the refinement is relatively simple.

Notice that the initial state cooling has a set action on the dangling transition indicating the initial state, written as

$$s(t) := T_c \ .$$

As we did with extended state machines, we use the notation ":=" to emphasize that this is an assignment, not a predicate. This action ensures that when the thermostat starts, it can immediately transition to the heating mode if the temperature $\tau(t)$ is less than or equal to 20 degrees. The other two transitions each have set actions that reset the clock s to zero. The portion of the guard that specifies $s(t) \geq T_h$ ensures that the heater will always be on for at least time T_h. The portion of the guard that specifies $s(t) \geq T_c$ specifies that once the heater goes off, it will remain off for at least time T_c.

A possible execution of this timed automaton is shown in Figure 4.6. In that figure, we assume that the temperature is initially above the setpoint of 20 degrees, so the FSM remains in the cooling state until the temperature drops to 20 degrees. At that time t_1, it can take the transition immediately because $s(t_1) > T_c$. The transition resets s to zero and turns on the heater. The heater will remain on until time $t_1 + T_h$, assuming that the temperature only rises when the heater is on. At time $t_1 + T_h$, it will transition back to the cooling state and turn the heater off. (We assume here that a transition is taken as soon as it is enabled. Other transition semantics are possible.) It will cool until at least time T_c elapses and until

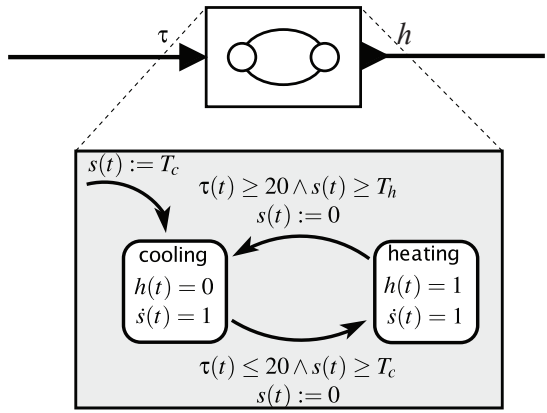


Figure 4.5: A timed automaton modeling a thermostat with a single temperature threshold, 20, and minimum times T_c and T_h in each mode.

the temperature drops again to 20 degrees, at which point it will turn the heater back on.

In the previous example the state of the system at any time t is not only the mode, heating or cooling, but also the current value $s(t)$ of the clock. We call s a **continuous state** variable, whereas heating and cooling are **discrete state**s. Thus, note that the term "state" for such a hybrid system can become confusing. The FSM has states, but so do the refinement systems (unless they are memoryless). When there is any possibility of confusion we explicitly refer to the states of the machine as modes.

Transitions between modes have actions associated with them. Sometimes, it is useful to have transitions from one mode back to itself, just so that the action can be realized. This is illustrated in the next example, which also shows a timed automaton that produces a pure output.

Example 4.4: The timed automaton in Figure 4.7 produces a pure output that will be present every T time units, starting at the time when the system begins

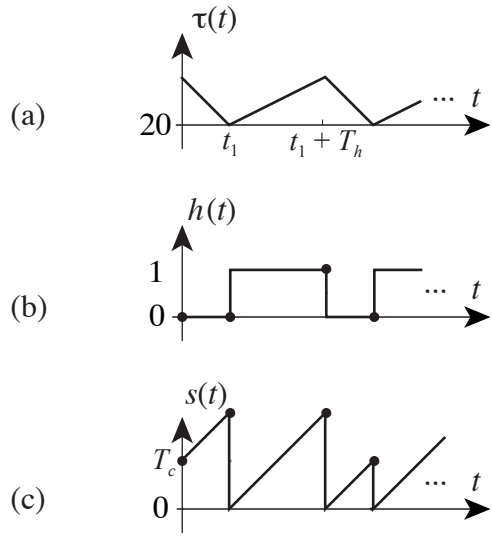


Figure 4.6: (a) A temperature input to the hybrid system of Figure 4.5, (b) the output h, and (c) the refinement state s.

executing. Notice that the guard on the transition, $s(t) \geq T$, is followed by an output action, *tick*, and a set action, $s(t) := 0$.

Figure 4.7 shows another notational shorthand that works well for simple diagrams. The automaton is shown directly inside the icon for its actor model.

Example 4.5: The traffic light controller of Figure 3.10 is a time triggered machine that assumes it reacts once each second. Figure 4.8 shows a timed automaton with the same behavior. It is more explicit about the passage of time in that its temporal dynamics do not depend on unstated assumptions about when the machine will react.

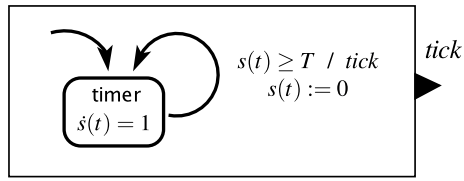


Figure 4.7: A timed automaton that generates a pure output event every T time units.

continuous variable: $x(t)$: $\mathbb{R}$
inputs: *pedestrian*: pure
outputs: *sigR*, *sigG*, *sigY*: pure

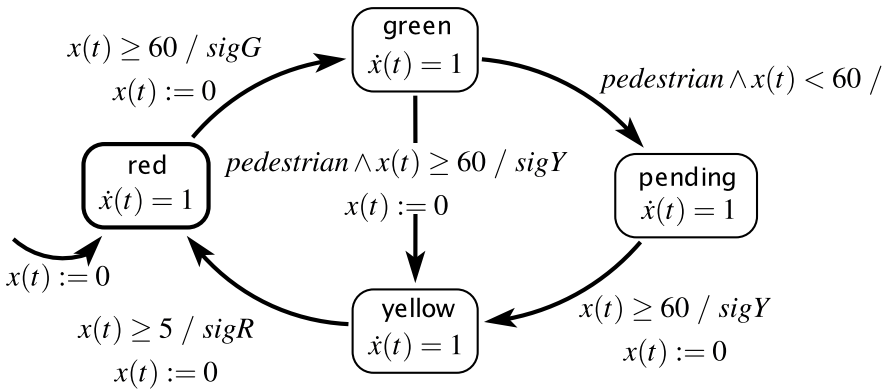


Figure 4.8: A timed automaton variant of the traffic light controller of Figure 3.10.

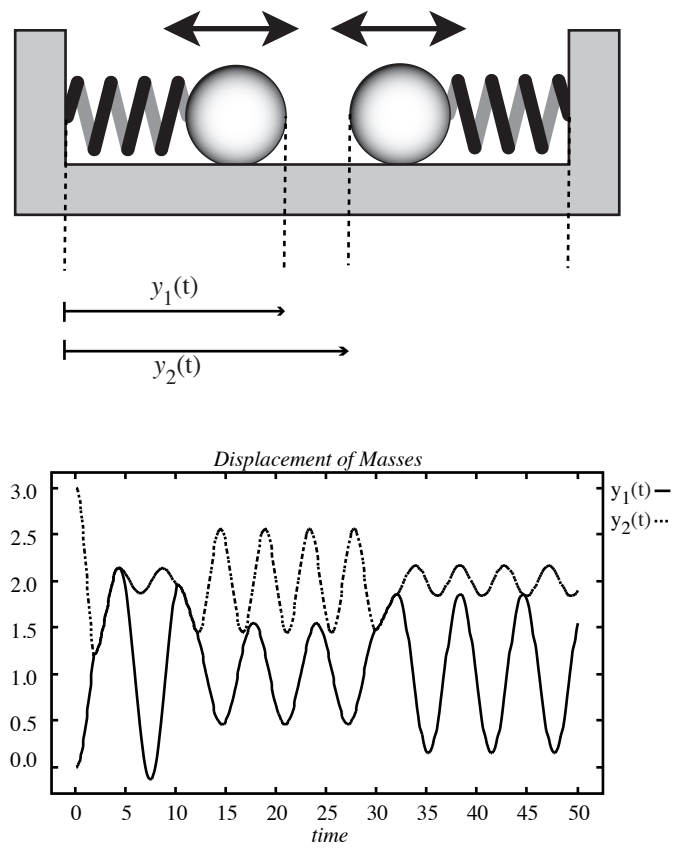


Figure 4.9: Sticky masses system considered in Example 4.6.

4.2.2 Higher-Order Dynamics

In timed automata, all that happens in the time-based refinement systems is that time passes. Hybrid systems, however, are much more interesting when the behavior of the refinements is more complex. Specifically,

Example 4.6: Consider the physical system depicted in Figure 4.9. Two sticky round masses are attached to springs. The springs are compressed or extended

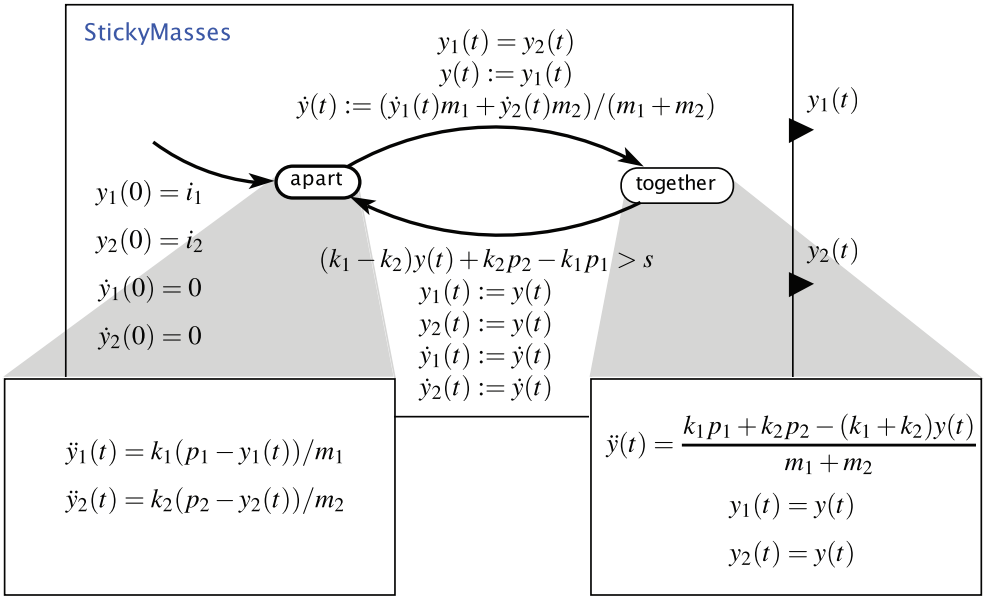


Figure 4.10: Hybrid system model for the sticky masses system considered in Example 4.6.

and then released. The masses oscillate on a frictionless table. If they collide, they stick together and oscillate together. After some time, the stickiness decays, and masses pull apart again.

A plot of the displacement of the two masses as a function of time is shown in Figure 4.9. Both springs begin compressed, so the masses begin moving towards one another. They almost immediately collide, and then oscillate together for a brief period until they pull apart. In this plot, they collide two more times, and almost collide a third time.

The physics of this problem is quite simple if we assume idealized springs. Let $y_1(t)$ denote the right edge of the left mass at time t, and $y_2(t)$ denote the left edge of the right mass at time t, as shown in Figure 4.9. Let p_1 and p_2 denote the neutral positions of the two masses, i.e., when the springs are neither extended nor compressed, so the force is zero. For an ideal spring, the force at time t on

the mass is proportional to $p_1 - y_1(t)$ (for the left mass) and $p_2 - y_2(t)$ (for the right mass). The force is positive to the right and negative to the left.

Let the spring constants be k_1 and k_2, respectively. Then the force on the left spring is $k_1(p_1 - y_1(t))$, and the force on the right spring is $k_2(p_2 - y_2(t))$. Let the masses be m_1 and m_2 respectively. Now we can use Newton's second law, which relates force, mass, and acceleration,

$$f = ma.$$

The acceleration is the second derivative of the position with respect to time, which we write $\ddot{y}_1(t)$ and $\ddot{y}_2(t)$. Thus, as long as the masses are separate, their dynamics are given by

$$\ddot{y}_1(t) = k_1(p_1 - y_1(t))/m_1 \tag{4.1}$$
$$\ddot{y}_2(t) = k_2(p_2 - y_2(t))/m_2. \tag{4.2}$$

When the masses collide, however, the situation changes. With the masses stuck together, they behave as a single object with mass $m_1 + m_2$. This single object is pulled in opposite directions by two springs. While the masses are stuck together, $y_1(t) = y_2(t)$. Let

$$y(t) = y_1(t) = y_2(t).$$

The dynamics are then given by

$$\ddot{y}(t) = \frac{k_1 p_1 + k_2 p_2 - (k_1 + k_2)y(t)}{m_1 + m_2}. \tag{4.3}$$

It is easy to see now how to construct a hybrid systems model for this physical system. The model is shown in Figure 4.10. It has two modes, apart and together. The refinement of the apart mode is given by (4.1) and (4.2), while the refinement of the together mode is given by (4.3).

We still have work to do, however, to label the transitions. The initial transition is shown in Figure 4.10 entering the apart mode. Thus, we are assuming the masses begin apart. Moreover, this transition is labeled with a set action that sets the initial positions of the two masses to i_1 and i_2 and the initial velocities to zero.

The transition from apart to together has the guard

$$y_1(t) = y_2(t)\ .$$

This transition has a set action which assigns values to two continuous state variables $y(t)$ and $\dot{y}(t)$, which will represent the motion of the two masses stuck together. The value it assigns to $\dot{y}(t)$ conserves momentum. The momentum of the left mass is $\dot{y}_1(t)m_1$, the momentum of the right mass is $\dot{y}_2(t)m_2$, and the momentum of the combined masses is $\dot{y}(t)(m_1 + m_2)$. To make these equal, it sets

$$\dot{y}(t) = \frac{\dot{y}_1(t)m_1 + \dot{y}_2(t)m_2}{m_1 + m_2}.$$

The refinement of the together mode gives the dynamics of y and simply sets $y_1(t) = y_2(t) = y(t)$, since the masses are moving together. The transition from apart to together sets $y(t)$ equal to $y_1(t)$ (it could equally well have chosen $y_2(t)$, since these are equal).

The transition from together to apart has the more complicated guard

$$(k_1 - k_2)y(t) + k_2p_2 - k_1p_1 > s,$$

where s represents the stickiness of the two masses. This guard is satisfied when the right-pulling force on the right mass exceeds the right-pulling force on the left mass by more than the stickiness. The right-pulling force on the right mass is simply

$$f_2(t) = k_2(p_2 - y(t))$$

and the right-pulling force on the left mass is

$$f_1(t) = k_1(p_1 - y(t)).$$

Thus,

$$f_2(t) - f_1(t) = (k_1 - k_2)y(t) + k_2p_2 - k_1p_1.$$

When this exceeds the stickiness s, then the masses pull apart.

An interesting elaboration on this example, considered in problem 11, modifies the together mode so that the stickiness is initialized to a starting value, but then decays according to the differential equation

$$\dot{s}(t) = -as(t)$$

where $s(t)$ is the stickiness at time t, and a is some positive constant. In fact, it is the dynamics of such an elaboration that is plotted in Figure 4.9.

As in Example 4.4, it is sometimes useful to have hybrid system models with only one state. The actions on one or more state transitions define the discrete event behavior that combines with the time-based behavior.

Example 4.7: Consider a bouncing ball. At time $t = 0$, the ball is dropped from a height $y(0) = h_0$, where h_0 is the initial height in meters. It falls freely. At some later time t_1 it hits the ground with a velocity $\dot{y}(t_1) < 0$ m/s (meters per second). A *bump* event is produced when the ball hits the ground. The collision is **inelastic** (meaning that kinetic energy is lost), and the ball bounces back up with velocity $-a\dot{y}(t_1)$, where a is constant with $0 < a < 1$. The ball will then rise to a certain height and fall back to the ground repeatedly.

The behavior of the bouncing ball can be described by the hybrid system of Figure 4.11. There is only one mode, called free. When it is not in contact with the ground, we know that the ball follows the second-order differential equation,

$$\ddot{y}(t) = -g, \tag{4.4}$$

where $g = 9.81$ m/sec^2 is the acceleration imposed by gravity. The continuous state variables of the free mode are

$$s(t) = \begin{bmatrix} y(t) \\ \dot{y}(t) \end{bmatrix}$$

with the initial conditions $y(0) = h_0$ and $\dot{y}(0) = 0$. It is then a simple matter to rewrite (4.4) as a first-order differential equation,

$$\dot{s}(t) = f(s(t)) \tag{4.5}$$

for a suitably chosen function f.

At the time $t = t_1$ when the ball first hits the ground, the guard

$$y(t) = 0$$

is satisfied, and the self-loop transition is taken. The output *bump* is produced, and the set action $\dot{y}(t) := -a\dot{y}(t)$ changes $\dot{y}(t_1)$ to have value $-a\dot{y}(t_1)$. Then (4.4) is followed again until the guard becomes true again.

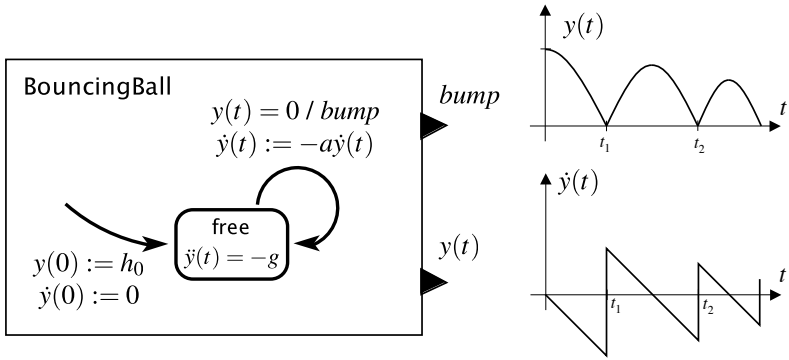


Figure 4.11: The motion of a bouncing ball may be described as a hybrid system with only one mode. The system outputs a *bump* each time the ball hits the ground, and also outputs the position of the ball. The position and velocity are plotted versus time at the right.

By integrating (4.4) we get, for all $t \in (0, t_1)$,

$$\begin{aligned} \dot{y}(t) &= -gt, \\ y(t) &= y(0) + \int_0^t \dot{y}(\tau)d\tau = h_0 - \frac{1}{2}gt^2. \end{aligned}$$

So $t_1 > 0$ is determined by $y(t_1) = 0$. It is the solution to the equation

$$h_0 - \frac{1}{2}gt^2 = 0.$$

Thus,

$$t_1 = \sqrt{2h_0/g}.$$

Figure 4.11 plots the continuous state versus time.

The bouncing ball example above has an interesting difficulty that is explored in Exercise 10. Specifically, the time between bounces gets smaller as time increases. In fact, it

gets smaller fast enough that an infinite number of bounces occur in a finite amount of time. A system with an infinite number of discrete events in a finite amount of time is called a **Zeno** system, after Zeno of Elea, a pre-Socratic Greek philosopher famous for his paradoxes. In the physical world, of course, the ball will eventually stop bouncing; the Zeno behavior is an artifact of the model. Another example of a Zeno hybrid system is considered in Exercise 13.

4.2.3 Supervisory Control

A control system involves four components: a system called the **plant**, the physical process that is to be controlled; the environment in which the plant operates; the sensors that measure some variables of the plant and the environment; and the controller that determines the mode transition structure and selects the time-based inputs to the plant. The controller has two levels: the **supervisory control** that determines the mode transition structure, and the **low-level control** that determines the time-based inputs to the plant. Intuitively, the supervisory controller determines which of several strategies should be followed, and the low-level controller implements the selected strategy. Hybrid systems are ideal for modeling such two-level controllers. We show how through a detailed example.

Example 4.8: Consider an **automated guided vehicle** (**AGV**) that moves along a closed track painted on a warehouse or factory floor. We will design a controller so that the vehicle closely follows the track.

The vehicle has two degrees of freedom. At any time t, it can move forward along its body axis with speed $u(t)$ with the restriction that $0 \leq u(t) \leq 10$ mph (miles per hour). It can also rotate about its center of gravity with an angular speed $\omega(t)$ restricted to $-\pi \leq \omega(t) \leq \pi$ radians/second. We ignore the inertia of the vehicle, so we assume that we can instantaneously change the velocity or angular speed.

Let $(x(t), y(t)) \in \mathbb{R}^2$ be the position relative to some fixed coordinate frame and $\theta(t) \in (-\pi, \pi]$ be the angle (in radians) of the vehicle at time t, as shown in Figure 4.12. In terms of this coordinate frame, the motion of the vehicle is given

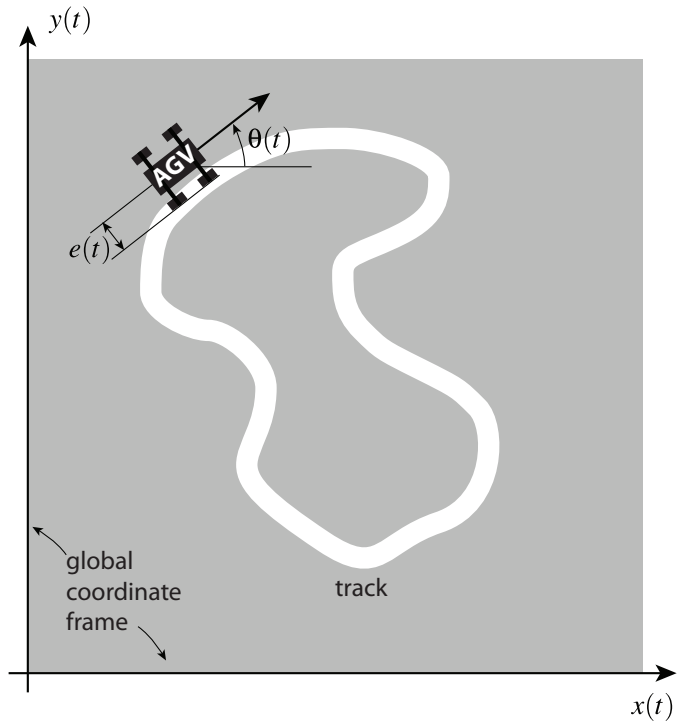


Figure 4.12: Illustration of the automated guided vehicle of Example 4.8. The vehicle is following a curved painted track, and has deviated from the track by a distance $e(t)$. The coordinates of the vehicle at time t with respect to the global coordinate frame are $(x(t), y(t), \theta(t))$.

by a system of three differential equations,

$$\begin{aligned} \dot{x}(t) &= u(t)\cos\theta(t), \\ \dot{y}(t) &= u(t)\sin\theta(t), \\ \dot{\theta}(t) &= \omega(t). \end{aligned} \tag{4.6}$$

Equations (4.6) describe the plant. The environment is the closed painted track. It could be described by an equation. We will describe it indirectly below by means of a sensor.

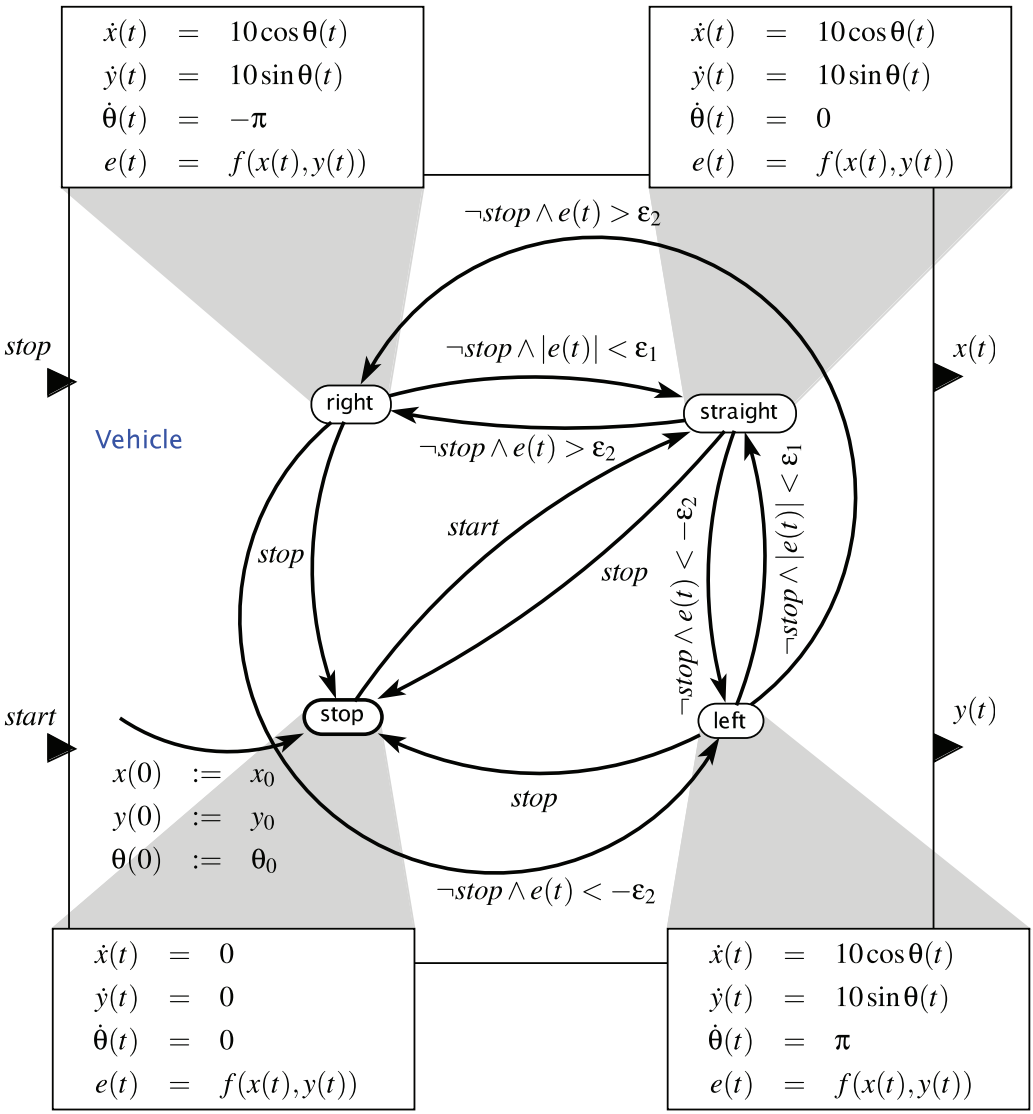


Figure 4.13: The automatic guided vehicle of Example 4.8 has four modes: stop, straight, left, right.

The two-level controller design is based on a simple idea. The vehicle always moves at its maximum speed of 10 mph. If the vehicle strays too far to the left of the track, the controller steers it towards the right; if it strays too far to the right of the track, the controller steers it towards the left. If the vehicle is close to the track, the controller maintains the vehicle in a straight direction. Thus the controller guides the vehicle in four modes, left, right, straight, and stop. In stop mode, the vehicle comes to a halt.

The following differential equations govern the AGV's motion in the refinements of the four modes. They describe the low-level controller, i.e., the selection of the time-based plant inputs in each mode.

straight

$$\begin{aligned} \dot{x}(t) &= 10\cos\theta(t) \\ \dot{y}(t) &= 10\sin\theta(t) \\ \dot{\theta}(t) &= 0 \end{aligned}$$

left

$$\begin{aligned} \dot{x}(t) &= 10\cos\theta(t) \\ \dot{y}(t) &= 10\sin\theta(t) \\ \dot{\theta}(t) &= \pi \end{aligned}$$

right

$$\begin{aligned} \dot{x}(t) &= 10\cos\theta(t) \\ \dot{y}(t) &= 10\sin\theta(t) \\ \dot{\theta}(t) &= -\pi \end{aligned}$$

stop

$$\begin{aligned} \dot{x}(t) &= 0 \\ \dot{y}(t) &= 0 \\ \dot{\theta}(t) &= 0 \end{aligned}$$

In the stop mode, the vehicle is stopped, so $x(t)$, $y(t)$, and $\theta(t)$ are constant. In the left mode, $\theta(t)$ increases at the rate of π radians/second, so from Figure 4.12

we see that the vehicle moves to the left. In the right mode, it moves to the right. In the straight mode, $\theta(t)$ is constant, and the vehicle moves straight ahead with a constant heading. The refinements of the four modes are shown in the boxes of Figure 4.13.

We design the supervisory control governing transitions between modes in such a way that the vehicle closely follows the track, using a sensor that determines how far the vehicle is to the left or right of the track. We can build such a sensor using photodiodes. Let's suppose the track is painted with a light-reflecting color, whereas the floor is relatively dark. Underneath the AGV we place an array of photodiodes as shown in Figure 4.14. The array is perpendicular to the AGV body axis. As the AGV passes over the track, the diode directly above the track generates more current than the other diodes. By comparing the magnitudes of the currents through the different diodes, the sensor estimates the displacement $e(t)$ of the center of the array (hence, the center of the AGV) from the track. We adopt the convention that $e(t) < 0$ means that the AGV is to the right of the track and $e(t) > 0$ means it is to the left. We model the sensor output as a function f of the AGV's position,

$$\forall t, \quad e(t) = f(x(t), y(t)).$$

The function f of course depends on the environment—the track. We now specify the supervisory controller precisely. We select two thresholds, $0 < \epsilon_1 < \epsilon_2$, as shown in Figure 4.14. If the magnitude of the displacement is small, $|e(t)| < \epsilon_1$, we consider that the AGV is close enough to the track, and the AGV can move straight ahead, in straight mode. If $e(t) > \epsilon_2$ ($e(t)$ is large and positive), the AGV has strayed too far to the left and must be steered to the right, by switching to right mode. If $e(t) < -\epsilon_2$ ($e(t)$ is large and negative), the AGV has strayed too far to the right and must be steered to the left, by switching to left mode. This control logic is captured in the mode transitions of Figure 4.13. The inputs are pure signals *stop* and *start*. These model an operator that can stop or start the AGV. There is no continuous-time input. The outputs represent the position of the vehicle, $x(t)$ and $y(t)$. The initial mode is stop, and the initial values of its refinement are (x_0, y_0, θ_0).

We analyze how the AGV will move. Figure 4.15 sketches one possible trajectory. Initially the vehicle is within distance ϵ_1 of the track, so it moves straight. At some

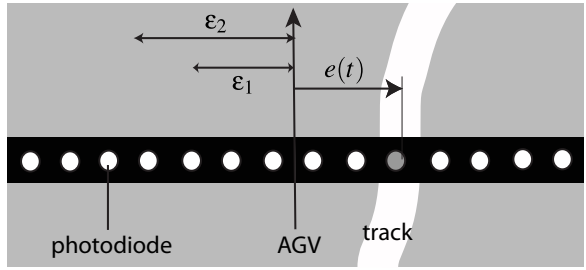


Figure 4.14: An array of photodiodes under the AGV is used to estimate the displacement e of the AGV relative to the track. The photodiode directly above the track generates more current.

later time, the vehicle goes too far to the left, so the guard

$$\neg stop \wedge e(t) > \epsilon_2$$

is satisfied, and there is a mode switch to right. After some time, the vehicle will again be close enough to the track, so the guard

$$\neg stop \wedge |e(t)| < \epsilon_1$$

is satisfied, and there is a mode switch to straight. Some time later, the vehicle is too far to the right, so the guard

$$\neg stop \wedge e(t) < -\epsilon_2$$

is satisfied, and there is a mode switch to left. And so on.

The example illustrates the four components of a control system. The plant is described by the differential equations (4.6) that govern the evolution of the continuous state at time t, $(x(t), y(t), \theta(t))$, in terms of the plant inputs u and ω. The second component is the environment—the closed track. The third component is the sensor, whose output at time t, $e(t) = f(x(t), y(t))$, gives the position of the AGV relative to the track. The fourth component is the two-level controller. The supervisory controller comprises the four modes and the guards that determine when to switch between modes. The low-level

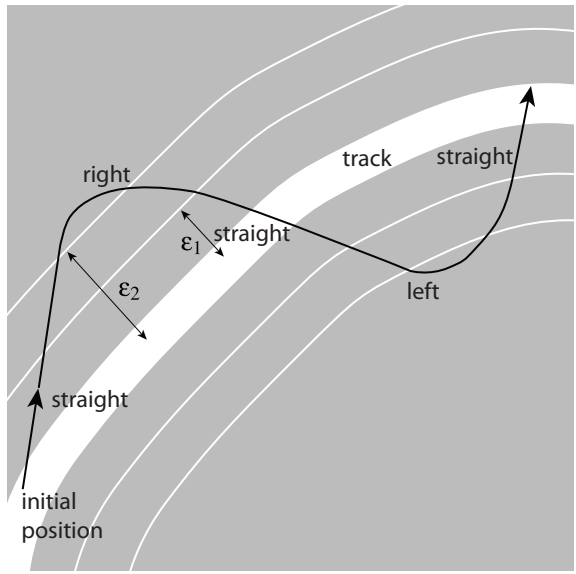


Figure 4.15: A trajectory of the AGV, annotated with modes.

controller specifies how the time-based inputs to the plant, u and ω, are selected in each mode.

4.3 Summary

Hybrid systems provide a bridge between time-based models and state-machine models. The combination of the two families of models provides a rich framework for describing real-world systems. There are two key ideas. First, discrete events (state changes in a state machine) get embedded in a time base. Second, a hierarchical description is particularly useful, where the system undergoes discrete transitions between different modes of operation. Associated with each mode of operation is a time-based system called the refinement of the mode. Mode transitions are taken when guards that specify the combination of inputs and continuous states are satisfied. The action associated with a transition, in turn, sets the continuous state in the destination mode.

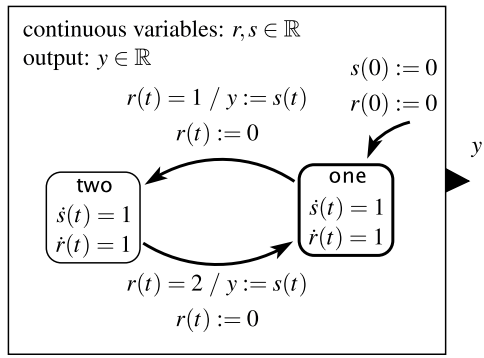
The behavior of a hybrid system is understood using the tools of state machine analysis for mode transitions and the tools of time-based analysis for the refinement systems. The design of hybrid systems similarly proceeds on two levels: state machines are designed to achieve the appropriate logic of mode transitions, and continuous refinement systems are designed to secure the desired time-based behavior in each mode.

Exercises

1. Construct (on paper is sufficient) a timed automaton similar to that of Figure 4.7 which produces *tick* at times $1, 2, 3, 5, 6, 7, 8, 10, 11, \cdots$. That is, ticks are produced with intervals between them of 1 second (three times) and 2 seconds (once).

2. The objective of this problem is to understand a timed automaton, and then to modify it as specified.

   (a) For the timed automaton shown below, describe the output y. Avoid imprecise or sloppy notation.

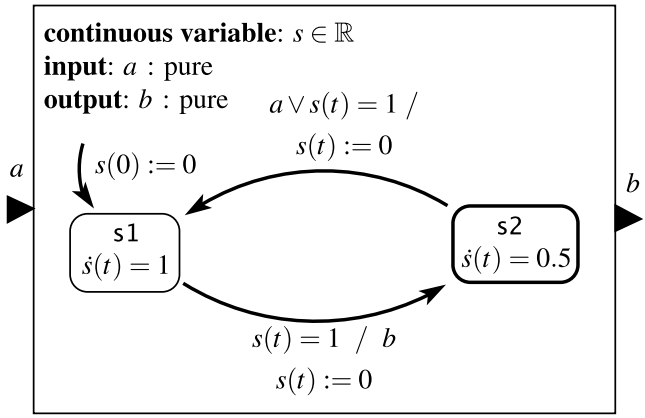


   (b) Assume there is a new pure input *reset*, and that when this input is present, the hybrid system starts over, behaving as if it were starting at time 0 again. Modify the hybrid system from part (a) to do this.

3. In Exercise 6 of Chapter 2, we considered a DC motor that is controlled by an input voltage. Controlling a motor by varying an input voltage, in reality, is often not practical. It requires analog circuits that are capable of handling considerable power. Instead, it is common to use a fixed voltage, but to turn it on and off periodically to vary the amount of power delivered to the motor. This technique is called pulse width modulation (PWM).

   Construct a timed automaton that provides the voltage input to the motor model from Exercise 6. Your hybrid system should assume that the PWM circuit delivers a 25 kHz square wave with a duty cycle between zero and 100%, inclusive. The input to your hybrid system should be the duty cycle, and the output should be the voltage.

4. Consider the following timed automaton:



Assume that the input signals a and b are discrete continuous-time signals, meaning that each can be given as a function of form $a: \mathbb{R} \to \{present, absent\}$, where at almost all times $t \in \mathbb{R}$, $a(t) = absent$. Assume that the state machine can take at most one transition at each distinct time t, and that machine begins executing at time $t = 0$.

(a) Sketch the output b if the input a is present only at times

$$t = 0.75, 1.5, 2.25, 3, 3.75, 4.5, \cdots$$

Include at least times from $t = 0$ to $t = 5$.

(b) Sketch the output b if the input a is present only at times $t = 0, 1, 2, 3, \cdots$.

(c) Assuming that the input a can be any discrete signal at all, find a lower bound on the amount of time between events b. What input signal a (if any) achieves this lower bound?

5. You have an analog source that produces a pure tone. You can switch the source on or off by the input event *on* or *off*. Construct a timed automaton that provides the *on* and *off* signals as outputs, to be connected to the inputs of the tone generator. Your system should behave as follows. Upon receiving an input event *ring*, it should produce an 80 ms-long sound consisting of three 20 ms-long bursts of the pure tone separated by two 10 ms intervals of silence. What does your system do if it receives two *ring* events that are 50 ms apart?

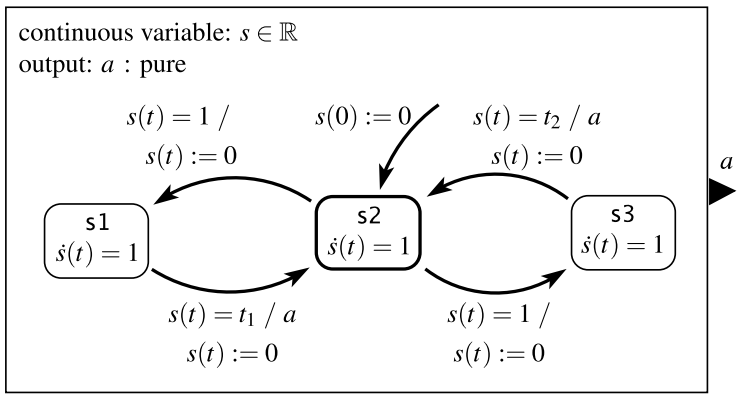
6. Automobiles today have the features listed below. Implement each feature as a timed automaton.

   (a) The dome light is turned on as soon as any door is opened. It stays on for 30 seconds after all doors are shut. What sensors are needed?

   (b) Once the engine is started, a beeper is sounded and a red light warning is indicated if there are passengers that have not buckled their seat belt. The beeper stops sounding after 30 seconds, or as soon the seat belts are buckled, whichever is sooner. The warning light is on all the time the seat belt is unbuckled. **Hint:** Assume the sensors provide a *warn* event when the ignition is turned on and there is a seat with passenger not buckled in, or if the ignition is already on and a passenger sits in a seat without buckling the seatbelt. Assume further that the sensors provide a *noWarn* event when a passenger departs from a seat, or when the buckle is buckled, or when the ignition is turned off.

7. A programmable thermostat allows you to select 4 times, $0 \leq T_1 \leq \cdots \leq T_4 < 24$ (for a 24-hour cycle) and the corresponding setpoint temperatures $a_1, \cdots, a_4$. Construct a timed automaton that sends the event a_i to the heating systems controller. The controller maintains the temperature close to the value a_i until it receives the next event. How many timers and modes do you need?

8. Consider the following timed automaton:



Assume t_1 and t_2 are positive real numbers. What is the minimum amount of time between events a? That is, what is the smallest possible time between two times when the signal a is present?

9. Figure 4.16 depicts the intersection of two one-way streets, called Main and Secondary. A light on each street controls its traffic. Each light goes through a cycle consisting of a red (R), green (G), and yellow (Y) phases. It is a safety requirement that when one light is in its green or yellow phase, the other is in its red phase. The yellow phase is always 5 seconds long.

   The traffic lights operate as follows. A sensor in the secondary road detects a vehicle. While no vehicle is detected, there is a 4 minute-long cycle with the main light having 3 minutes of green, 5 seconds of yellow, and 55 seconds of red. The secondary light is red for 3 minutes and 5 seconds (while the main light is green and yellow), green for 50 seconds, then yellow for 5 seconds.

   If a vehicle is detected on the secondary road, the traffic light quickly gives a right of way to the secondary road. When this happens, the main light aborts its green phase and immediately switches to its 5 second yellow phase. If the vehicle is detected while the main light is yellow or red, the system continues as if there were no vehicle.

   Design a hybrid system that controls the lights. Let this hybrid system have six pure outputs, one for each light, named *mG*, *mY*, and *mR*, to designate the main

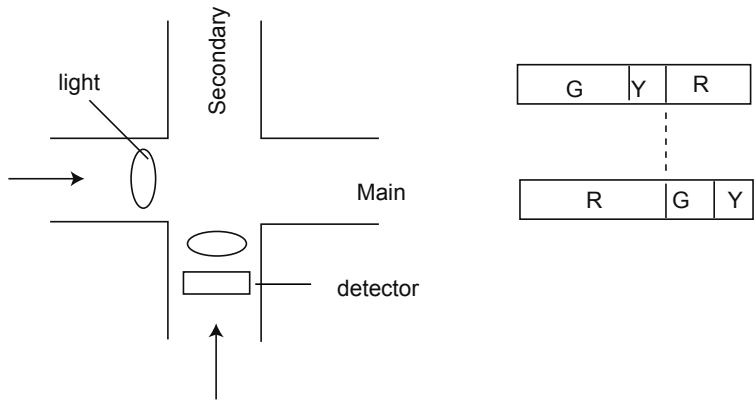


Figure 4.16: Traffic lights control the intersection of a main street and a secondary street. A detector senses when a vehicle crosses it. The red phase of one light must coincide with the green and yellow phases of the other light.

light being green, yellow, or red, respectively, and *sG*, *sY*, and *sR*, to designate the secondary light being green, yellow, or red, respectively. These signals should be generated to turn on a light. You can implicitly assume that when one light is turned on, whichever has been on is turned off.

10. For the bouncing ball of Example 4.7, let t_n be the time when the ball hits the ground for the n-th time, and let $v_n = \dot{y}(t_n)$ be the velocity at that time.

    (a) Find a relation between v_{n+1} and v_n for $n > 1$, and then calculate v_n in terms of v_1.

    (b) Obtain t_n in terms of v_1 and a. Use this to show that the bouncing ball is a Zeno system. **Hint:** The **geometric series identity** might be useful, where for $|b| < 1$,
    $$\sum_{m=0}^{\infty} b^m = \frac{1}{1-b}.$$

    (c) Calculate the maximum height reached by the ball after successive bumps.

11. Elaborate the hybrid system model of Figure 4.10 so that in the *together* mode, the stickiness decays according to the differential equation
    $$\dot{s}(t) = -as(t)$$
    where $s(t)$ is the stickiness at time t, and a is some positive constant. On the transition into this mode, the stickiness should be initialized to some starting stickiness b.

12. Show that the trajectory of the AGV of Figure 4.13 while it is in *left* or *right* mode is a circle. What is the radius of this circle, and how long does it take to complete a circle?

13. Consider Figure 4.17 depicting a system comprising two tanks containing water. Each tank is leaking at a constant rate. Water is added at a constant rate to the system through a hose, which at any point in time is filling either one tank or the other. It is assumed that the hose can switch between the tanks instantaneously. For $i \in \{1, 2\}$, let x_i denote the volume of water in Tank i and $v_i > 0$ denote the constant flow of water out of Tank i. Let w denote the constant flow of water into the system. The objective is to keep the water volumes above r_1 and r_2, respectively, assuming that the water volumes are above r_1 and r_2 initially. This is to be achieved

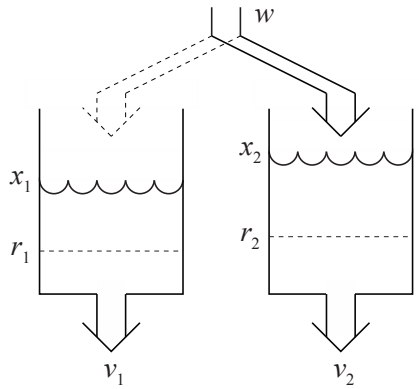


Figure 4.17: Water tank system.

by a controller that switches the inflow to Tank 1 whenever $x_1(t) \leq r_1(t)$ and to Tank 2 whenever $x_2(t) \leq r_2(t)$.

The hybrid automaton representing this two-tank system is given in Figure 4.18.

Answer the following questions:

(a) Construct a model of this hybrid automaton in Ptolemy II, LabVIEW, or Simulink. Use the following parameter values: $r_1 = r_2 = 0$, $v_1 = v_2 = 0.5$, and $w = 0.75$. Set the initial state to be $(q_1, (0, 1))$. (That is, initial value $x_1(0)$ is 0 and $x_2(0)$ is 1.)

Verify that this hybrid automaton is Zeno. What is the reason for this Zeno behavior? Simulate your model and plot how x_1 and x_2 vary as a function of time t, simulating long enough to illustrate the Zeno behavior.

(b) A Zeno system may be **regularized** by ensuring that the time between transitions is never less than some positive number ϵ. This can be emulated by inserting extra modes in which the hybrid automaton dwells for time ϵ. Use regularization to make your model from part (a) non-Zeno. Again, plot x_1 and x_2 for the same length of time as in the first part. State the value of ϵ that you used.

Include printouts of your plots with your answer.

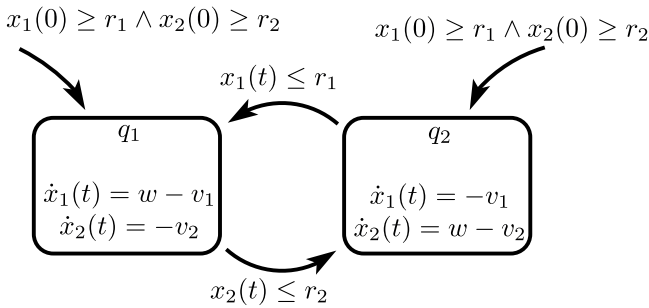


Figure 4.18: Hybrid automaton representing water tank system.