

Introduction

Nowadays, electronic devices largely depend on complex hardware and software systems. Among them, medical instruments, traffic control units, and many more safety-critical systems are subject to particular quality standards. They all come along with the absolute need for reliability, as, in each case, the consequences of a breakdown may be incalculable.

1.1 Formal Methods

Many existing systems were unthinkable some years ago and their complexity is still rapidly growing so that it becomes more and more difficult to detect errors or to predict their incidence. Consequently, *formal methods* play an increasing role during the whole system-design process. The term “formal methods” hereby covers a wide range of mathematically derived and ideally mechanized approaches to system design and validation. More precisely, research on formal methods attempts to develop mathematical models and algorithms that may contribute to the tasks of *modeling*, *specifying*, and *verifying* software and hardware systems. Let us go into these subareas in more detail:

Modeling

To make a system (or the idea of a system) accessible to formal methods, we require it to be modeled mathematically. Unfortunately, we are faced with a dilemma: on the one hand, a model ideally preserves and reflects as many properties of the underlying system as possible. On the other hand, it should be compact enough to support algorithms for further system analysis. However, in general, a good balance between detailed modeling and abstraction will pay off. But not only does the modeling process lead to further interesting conclusions, it may also help, itself, to get a better understanding of the system at hand. Thus, the purposes of modeling a system are twofold. One is to understand and document its essential features. The other is to provide the

formal basis for a mathematical analysis. Both are closely related and usually accompany each other.

Preferably, the modeling takes place in an early stage of system design. The starting point, at a high level of abstraction, may be a rough, even if precisely defined, idea of the system to be, which is subsequently refined step-wise towards a full implementation. While, as mentioned, the latter might be too detailed to draw conclusions from, previous stages of the design phase can be consulted for that purpose. The models considered in this book are *communicating automata*, which, though they might abstract from many details, reflect the operational behavior of a distributed system in a suitable manner to make it accessible to formal methods.

Specification

Correctness of a system is always relative to a *specification*, a property or requirement that must be satisfied. Embedded into the formal-methods framework, a specification is often expressed within a logical calculus whose formulas can be interpreted over system models, provided they are based on a common semantic domain. Prominent examples are monadic second-order (MSO) logic [8, 44], the temporal logics LTL [83] and CTL [22], and the μ -calculus [54]. A specification might also be given in a high-level language that is closer to an implementation and often allows us to *synthesize* a system directly and automatically. In this regard, let us mention some process-algebra based languages such as CCS [71], ACP [9], and LOTOS [18] and other formal design notions like VHDL [81]. In this book, we focus on a monadic second-order logic, which might be used to formulate properties that a *given* system should satisfy, and *high-level message sequence charts*, which are employed at a rather early stage of system development.

Verification

Once a system is modeled and a specification is given, the next task might be to check if the specification is satisfied by the model. If the system or, rather, the model of a system passes successfully through a corresponding validation process, it may be called correct in a mathematical sense. Preferably, the verification process runs automatically. However, many frameworks are too complex to support fully mechanized algorithms. In this respect, we can distinguish two general approaches to verification: *model checking* [23], which is fully automatic, and *theorem proving* [85], which requires human assistance. If, otherwise, a system is synthesized directly from its specification, then it can be assumed to be correct a priori, provided the translation preserves the semantics of the specification.

Several phases of system design are depicted in Fig. 1.1, which, in addition, features the stage of *code generation* to gain from the system model an effective implementation thereof.

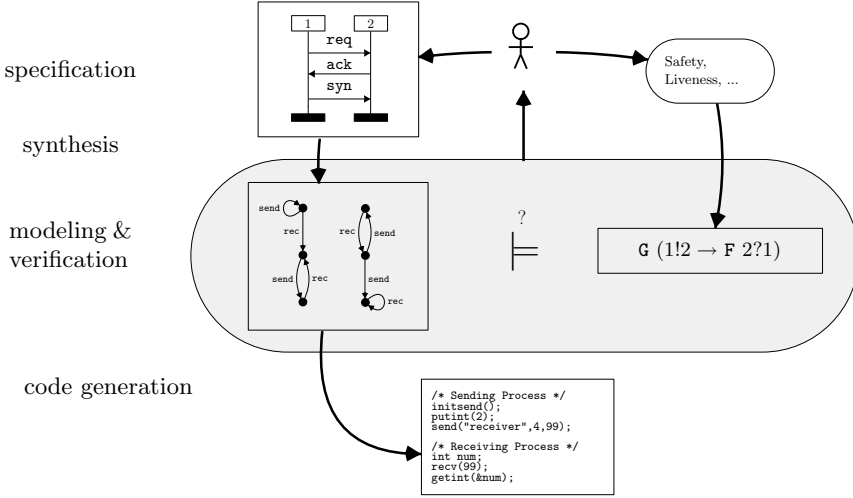


Fig. 1.1. Stages of system development

1.2 Partial Orders and Graphs

As mentioned above, it is desirable to apply formal methods even in the early stages of system design to avoid extensive reimplementation and redesign, which, in turn, might lead to explosive costs. A common design practice when developing communicating systems is to start with specifying scenarios to exemplify the intended interaction of the system to be. Usually, distributed systems operate concurrently, i.e., some actions do not depend on the occurrence of another. One possible single execution sequence of a distributed system is therefore often described by a partially ordered set (poset) (V, \leq) , such as depicted by Fig. 1.2a. The elements of V , which are also referred to as *events*, comprise actions that are executed during a system run. They are arranged according to the partial order $\leq \subseteq V \times V$ to reflect their interaction dependencies. Say, for example, we deal with events `send`, `rec` $\in V$ that form the send and receipt of a message. Naturally, sending a message precedes its receipt so that `send` \leq `rec`. Otherwise, there might be events that do not interfere with each other. For example, two read events `read1(x)` and `read2(x)` that independently read a shared variable x are not related, so neither `read1(x)` \leq `read2(x)` nor `read2(x)` \leq `read1(x)`, whereas the time of writing the variable does affect the value of what is read (cf. Fig. 1.2c).

A poset, in turn, might be represented by a graph (V, \rightarrow) whose edge relation $\rightarrow \subseteq V \times V$ gives rise to \leq when generating its reflexive transitive closure. Sometimes, \rightarrow allows a more concrete modeling of communication than \leq . Namely, writing `send` \rightarrow `rec` suggests that `send` and `rec` together

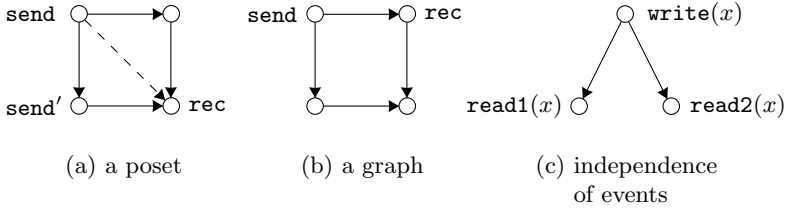


Fig. 1.2. Partially ordered sets and graphs

form the exchange of *one and the same* message (cf. Fig. 1.2b), whereas writing $\text{send} \leq \text{rec}$ is actually a weaker statement, just claiming that **rec** happens eventually after **send** but might be the receipt of another event **send'**, as illustrated in Fig. 1.2a.

Message sequence charts (MSCs) provide a prominent notion to further the partial-order and graph-based approaches. They are widely used in industry, are standardized [49, 50], and are similar to UML's sequence diagrams [7]. An MSC depicts a single partially ordered execution sequence of a system. In doing so, it defines a collection of processes, which, in their visual representation, are drawn as vertical lines and interpreted as time axes. Moreover, an arrow from one line to a second corresponds to the communication events of sending and receiving a message. An example MSC illustrating a part of *Bluetooth* [13], a specification for wireless communication, is depicted in Fig. 1.3. Using the *Host Control Interface* (HCI), which links a Bluetooth host (a portable PC, for example) with a Bluetooth controller (a PCMCIA card, for example), a host application attempts to establish a connection to another device. The connection-request phase, which is based on an asynchronous connectionless link (ACL), is heralded by Host-A sending a **HCI_Create_Connection** command to its controller to initiate a connection. Note that, usually, a command is equipped with parameters, which are omitted here. As HCI commands may take different amounts of time, their status is reported back to the host in the form of a **HCI_Command_Status** event. After that, HC-A defers the present request to HC-B, which, in turn, learns from Host-B that the request has been rejected, again accompanied by sending a status event. The controllers agree on rejection by exchanging messages **LMP_not_accepted** and **LMP_detach** and, afterwards, provide both Host-A and Host-B with **HCI_Connection_Complete** events. The execution sequence illustrated above depends on the visual arrangement of arrows. An endpoint of an arrow is a *send event* if it is the source of that arrow. Otherwise, it is a *receive event*. More specifically, we suppose events located on one and the same process line to be totally ordered and require a receive event to occur only if the corresponding send event has been executed. The above-mentioned partial order now arises from the reflexive transitive closure of those assumptions. Note that, in fact, some pairs of events cannot be ordered accordingly. Considering our example, re-

ceiving `HCI_Command_Status` by Host-A may occur before or after receiving `LMP_host_connection_req`, while the latter is supposed to happen after sending the former `HCI_Command_Status` event.

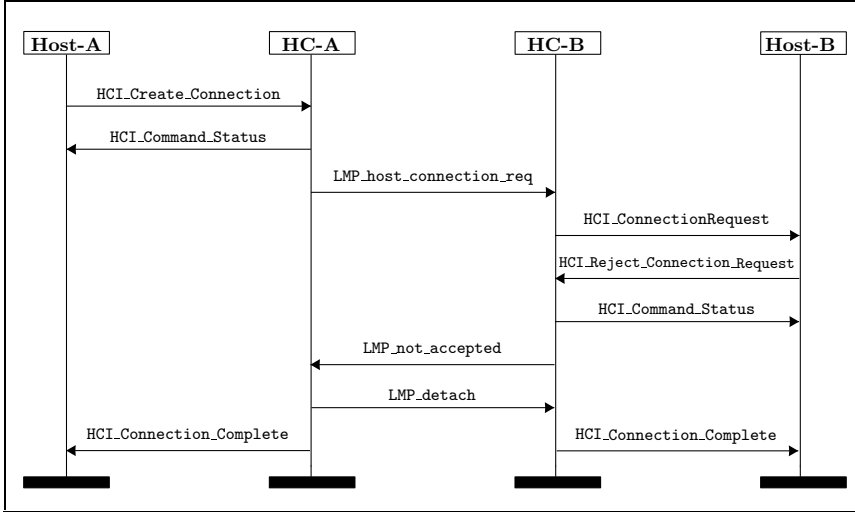


Fig. 1.3. An MSC modeling the ACL connection request phase

1.3 High-Level Specifications

Recall that a specification language might be used to formulate desirable properties of a given implementation or represent a first intuition of what the system has to do. A single graph or poset can, however, describe no more than one single execution sequence. Otherwise, a collection of graphs might capture all the scenarios that a designer wants the system under development to realize. Based on the notion of MSCs and the likewise partial-order based concept of *Mazurkiewicz traces* [27], several modeling and specification formalisms have been considered at a formal level, among them *high-level MSCs* [6, 45, 68, 76], which are capable of describing possibly infinitely many scenarios in a compact manner. From an algebraic point of view, high-level MSCs are rational expressions defining *rational languages* by means of choice, concatenation, and iteration. The study of algebraic language classes might then lead us to *recognizable languages* [73, 96], which can be characterized by certain *monoid automata*. Following the classical algebraic approach further, we will come across the class of *regular languages* whose linear extensions form a regular word language.

Moreover, there is a close connection between MSCs and Mazurkiewicz traces so that transferring the regularity notions for traces might be another axis to define regularity of sets of MSCs. Those aspects have been studied in [35, 56, 72, 73]. As we will see, the above language classes exhibit quite different properties in terms of *implementability*. Hereby, the notion of implementability is derived from a reference model, the poset- or graph-based counterpart of a finite automaton over words, which is explained in the next section in more detail.

MSO logic provides another specification formalism. But not only does MSO logic constitute an expressive specification language. Its relation to formalisms such as automata or high-level constructs over graphs and posets has also been a research area of great interest aiming at a deeper understanding of the latter’s logical and algorithmic properties (see [94] for an overview). Following the logical approach, one might likewise argue that we can call a set of graphs regular if it is definable in the corresponding MSO logic, because, in the domain of words, regularity and definability in MSO logic coincide [20, 32].

1.4 Towards an Implementation

The next step in system design might be to supply an implementation realizing or satisfying a specification. Recall that we are still interested in an abstract model rather than a concrete implementation in some low-level programming language. However, the view we are taking now is much closer to the latter. More precisely, we ask for *automata* models that are suited to accepting the system behavior described by, say, a high-level MSC, a logical formula, or a monoid automaton. Consequently, we are particularly interested in their expressiveness relative to the above-mentioned language classes.

To create formal methods tailored to a distributed system and to the associated mathematical model, it is generally helpful to study some of the model’s properties first and to learn more about its limitations along with algorithmic restrictions and its degree of abstraction. In this regard, typical questions to clarify are:

- Is my model of an implementation a suitable one, i.e., does it reflect all the aspects I want to verify?
- What is a suitable specification language; is any specification implementable?
- What kind of problem can I expect to be decidable?

Basically, that is what this book is all about. We will hereby concentrate on communicating systems, which occur whenever independent processes and objects interact, whether via message exchange through fifo (“first-in, first-out”) buffers or when attempting to write a shared variable. At the same time, we focus on issues related to the areas of system modeling and specification.

In particular, we will address the relation of several automata models with (fragments of) MSO logic to clarify its use as a specification language.

Concerning systems that are distributed in nature, the notion of a process is central. It seems therefore natural to consider each process as a single automaton and to define a notion of communication describing how these parallel systems work together. When, for example, we equip such local processes with message buffers, we obtain the model of a *message-passing automaton* or *communicating (finite-state) machine*. There is a precise logical characterization of communicating finite-state machines by a fragment of MSO logic, called existential MSO (EMSO) logic, so that any specification in terms of an EMSO expression has an implementation in terms of a communicating finite-state machine. Another model of communication is provided by *asynchronous automata*. Herein, local processes synchronize by executing certain actions (e.g., writing a variable) simultaneously, whereas others may be taken autonomously (e.g., reading the variable). Asynchronous automata were introduced originally by Zielonka in the framework of the partial-order model of Mazurkiewicz traces [97], and they were generalized by Droste et al. to run on even more general posets [29]. Asynchronous automata could also be shown to be expressively equivalent to EMSO logic relative to traces and CROW-posets, which are subject to an axiom that considers concurrent read and exclusive owner write. A quite general method of recognizing sets of partial orders and graphs is that of graph acceptors as introduced by Thomas [93]. They are known to be exactly as expressive as EMSO logic for arbitrary classes of graphs that have bounded degree. But they lack operational behavior and do not really reflect the dynamic causal nature of a system. We will, however, get to know *asynchronous cellular automata (with types)*, which combine the models of asynchronous automata, graph acceptors, communicating finite-state machines, and many other systems and allow us to treat them in a unifying framework. In particular, asynchronous cellular automata turn out to have the same expressive power as EMSO logic relative to any class of pomsets and dags.

1.5 An Overview of the Book

Chapter 2 recalls some basic notions and results concerning posets, monoids, and formal languages. It moreover presents the well-known halting problem of Turing machines, an undecidable problem that will be used to obtain related undecidability results with respect to communicating automata.

Chapter 3 introduces graphs in general and related notions, presents the corresponding MSO logic to express graph properties, and provides Thomas' fundamental result, which makes use of the famous theorem of Hanf and serves as the basis for upcoming logical characterizations: the expressive equivalence of graph acceptors and EMSO logic.

Chapter 4 recalls the well-known and thoroughly studied model of finite automata over words and their relation with MSO logic and the algebraic notions of recognizability and rationality. Though finite automata are considered to be a purely sequential model, they will, equipped with a communication medium, represent the building blocks of a distributed system.

Chapter 5 lays the foundation of subsequent chapters. It constitutes the basic parameter or architecture of a communicating system in terms of a distributed alphabet and introduces asynchronous cellular automata (with types) as a universal tool unifying finite automata, asynchronous automata, graph acceptors, communicating finite-state machines, and lossy channel systems. Asynchronous cellular automata turn out to be expressively equivalent to EMSO logic relative to dags over distributed alphabets and therefore cover the expressiveness results of all the above-mentioned models. Though, at first sight, asynchronous cellular automata appear as a rather complex and unintuitive model (e.g., compared with finite automata), they have been around since the end of the 80's and are a well-established tool to describe concurrent behavior. In this book, we deal with a particularly simple definition of asynchronous cellular automata to make them accessible to a broad readership. The reader is encouraged to study this model thoroughly; a good comprehension thereof will pay off and enable her to understand more specific concepts more easily (such as shared-memory and channel systems), to design her own automata models and to characterize them logically, and to sharpen the understanding of different phenomena of concurrency and their characteristics.

Chapter 6 presents asynchronous automata, a formal model of shared-memory systems. Naturally, asynchronous automata run over Mazurkiewicz traces, a class of graphs that describe the simultaneous access by several processes to common resources. The literature provides manifold approaches to modeling traces, and every approach has its strengths and weaknesses. In our setting, traces are best defined as graphs, as introduced in Chap. 3. We establish a logical characterization of asynchronous automata in terms of EMSO logic interpreted over traces. Note that the derivation of this equivalence is solely based on our considerations in Chap. 5 and, unlike other methods that have been applied so far, does not rely on further results whose proofs would have gone beyond the scope of this book. Finally, we recall the well-known theorem of Zielonka, which compares asynchronous automata with the notion of recognizability.

From Chap. 7 on, the book concentrates on systems that communicate through reliable or faulty (fifo) channels. In this regard, we first provide the notion of MSCs, followed by the definition of several classes of MSC languages, e.g., generated by high-level MSCs. Recall that a single MSC describes one possible run of the system at hand, whereas a set of MSCs (an MSC language) might be used to characterize the complete system behavior.

Traces are to asynchronous automata as MSCs are to communicating finite-state machines. The definition of communicating finite-state machines and lossy channel systems can be found in Chap. 8, which also deals with

their expressiveness relative to the previously proposed language classes. In particular, again exploiting Chap. 5, we can easily derive from the logical characterization of asynchronous cellular automata a logical characterization of communicating finite-state machines.

Finally, Chap. 9 studies the gap between MSO logic and its existential fragment, exemplified in the framework of communicating finite-state machines, which is also compared to the formalisms of high-level MSCs and recognizability. It will turn out that, as a specification language, the full MSO logic and (compositional) high-level MSCs are too powerful: we identify both MSO and high-level MSC specifications that cannot be implemented in terms of an automaton.

Formal Models of Communicating Systems
Languages, Automata, and Monadic Second-Order
Logic

Bollig, B.

2006, IX, 181 p., Hardcover

ISBN: 978-3-540-32922-0