

Chapter 2

Specifications and Modeling

How can we describe the system which we would like to design and how can we represent intermediate design information? Models and description techniques allowing us to capture the initial specification as well as intermediate design information will be presented in this chapter.

2.1 Requirements

Consistent with the simplified design flow (see Fig. 1.8), we will first of all describe requirements and approaches for specifying embedded systems. Specifications for embedded systems provide **models** of the system under design (SUD). Models can be defined as follows:

Definition 2.1 (Jantsch [256]): “A model is a simplification of another entity, which can be a physical thing or another model. The model contains exactly those characteristics and properties of the modeled entity that are relevant for a given task. A model is minimal with respect to a task if it does not contain any other characteristics than those relevant for the task.”

Models are described in languages. Languages should be capable of representing the following features¹:

- **Hierarchy:** Human beings are generally not capable of comprehending systems containing many objects (states, components) having complex relations with each other. The description of all real-life systems needs more objects than human beings can understand. Hierarchy (in combination with **abstraction**) is a key mechanism

¹Information from the books of Burns et al. [81], Bergé et al. [542], and Gajski et al. [166] is used in this list.

helping to solve this dilemma. Hierarchies can be introduced such that humans need to handle only a small number of objects at any time.

There are two kinds of hierarchies:

- **Behavioral hierarchies:** Behavioral hierarchies are hierarchies containing objects necessary to describe the system behavior. States, events, and output signals are examples of such objects.
- **Structural hierarchies:** Structural hierarchies describe how systems are composed of physical components.
For example, embedded systems can be comprised of components such as processors, memories, actuators, and sensors. Processors, in turn, include registers, multiplexers, and adders. Multiplexers are composed of gates.
- **Component-based design** [464]: It must be “easy” to derive the behavior of a system from the behavior of its components. If two components are connected, the resulting new behavior should be predictable. For example, suppose that we add another component (say, some GPS unit) to a car. The impact of the additional component on the overall behavior of the system (including buses) should be predictable.
- **Concurrency:** Real-life systems are distributed, concurrent systems composed of components. It is therefore necessary being able to specify concurrency conveniently. Unfortunately, human beings are not very good at understanding concurrent systems and many problems with real systems are actually a result of an incomplete understanding of possible behaviors of concurrent systems.
- **Synchronization and communication:** Components must be able to communicate and to synchronize. Without communication, components could not cooperate and we would use each of them in isolation. It must also be possible to agree on the use of resources. For example, it is necessary to express mutual exclusion.
- **Timing behavior:** Many embedded systems are real-time systems. Therefore, explicit timing requirements are one of the characteristics of embedded systems. The need for explicit modeling of time is even more obvious from the term “cyber-physical system.” Time is one of the key dimensions of physics. Hence, timing requirements **must** be captured in the specification of embedded/cyber-physical systems.

However, standard theories in computer science model time only in a very abstract way. The O -notation is one of the examples. This notation just reflects growth rates of functions. It is frequently used to model run-times of algorithms, but it fails to describe real execution times. In physics, quantities have units, but the O -notation does not even have units. So, it would not distinguish between femtoseconds and centuries. A similar remark applies to termination properties of algorithms. Standard theories are concerned with proving that a certain algorithm *eventually* terminates. For real-time systems, we need to show that certain computations are completed in a given amount of time, but the algorithm as a whole should possibly run until power is turned off.

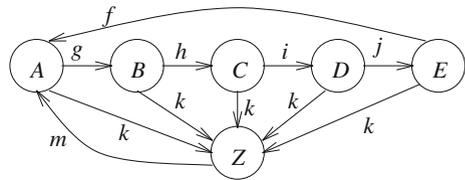
According to Burns and Wellings [81], modeling time must be possible in the following four contexts:

- Techniques for measuring **elapsed time**:
For many applications, it is necessary to check how much time has elapsed since some computation was performed. Access to a timer would provide a mechanism for this.
 - Means for **delaying of processes** for a specified time:
Typically, real-time languages provide some delay construct. Unfortunately, typical implementations of embedded systems in software do not guarantee precise delays. Let us assume that task τ should be delayed by some amount Δ . Usually, this delay is implemented by changing task τ 's state in the operating system from “ready” or “run” to “suspended.” At the end of this time interval, τ 's state is changed from “suspended” to “ready.” This does not mean that the task actually executes. If some higher priority task is executing or if preemption is not used, the delayed task will be delayed longer than Δ .
 - Possibility to specify **timeouts**:
There are many situations in which we must wait for a certain event to occur. However, this event may actually not occur within a given time interval and we would like to be notified about this. For example, we might be waiting for a response from some network connection. We would like to be notified if this response is not received within some amount of time, say Δ . This is the purpose of **timeouts**. Real-time languages usually also provide some timeout construct. Implementations of timeouts frequently come with the same problems which we mentioned for delays.
 - Methods for specifying **deadlines** and **schedules**:
For many applications, it is necessary to complete certain computations in a limited amount of time. For example, if the sensors of some car signal an accident, air bags must be ignited within about ten milliseconds. In this context, we must guarantee that the software will decide whether or not to ignite the air bags in that given amount of time. The air bags could harm passengers if they go off too late. Unfortunately, most languages do not allow to specify timing constraints. If they can be specified at all, they must be specified in separate control files, pop-up menus, etc. But the situation is still bad even if we are able to specify these constraints: Many modern hardware platforms do not have a very predictable timing behavior. Caches, stalled pipelines, speculative execution, task preemption, interrupts, etc. may have an impact on the execution time which is very difficult to predict. Accordingly, **timing analysis** (verifying the timing constraints) is a very hard design task.
- **State-oriented behavior**: It was already mentioned in Chap. 1 on p. 15 that automata provide a good mechanism for modeling reactive systems. Therefore, the state-oriented behavior provided by automata should be easy to describe. However, classical automata models are insufficient, since they cannot model timing and since hierarchy is not supported.

- **Event handling:** Due to the reactive nature of embedded systems, mechanisms for describing events must exist. Such events may be external events (caused by the environment) or internal events (caused by components of the system under design).
- **Exception-oriented behavior:** In many practical systems, exceptions do occur. In order to design dependable systems, it must be possible to describe actions to handle exceptions easily. It is not acceptable that exceptions must be indicated for each and every state (such as in the case of classical state diagrams).

Example 2.1: In Fig. 2.1, input k might correspond to an exception.

Fig. 2.1 State diagram with exception k



Specifying this exception at each state makes the diagram very complex. The situation would get worse for larger state diagrams with many transitions. On p. 47, we will show how all the transitions can be replaced by a single one (see Fig. 2.12). ▽

- **Presence of programming elements:** Popular programming languages have proven to be a convenient means of expressing computations that must be performed. Hence, programming language elements should be available in the specification technique used. Classical state diagrams do not meet this requirement.
- **Executability:** Specifications are not automatically consistent with the ideas in people's heads. Executing the specification is a means of plausibility checking. Specifications using programming languages have a clear advantage in this context.
- **Support for the design of large systems:** There is a trend toward large and complex embedded software programs. Software technology has found mechanisms for designing such large systems. For example, object orientation is one such mechanism. It should be available in the specification methodology.
- **Domain-specific support:** It would of course be nice if the same specification technique could be applied to all the different types of embedded systems, since this would minimize the effort for developing specification techniques and tool support. However, due to the wide range of application domains including those listed in Sect. 1.2, there is little hope that one language can be used to efficiently represent specifications in all such domains. For example, control-dominated, data-dominated, centralized and distributed application domains can all benefit from language features dedicated toward those domains.
- **Readability:** Of course, specifications must be readable by human beings. Otherwise, it would not be feasible to validate whether or not the specification meets

the real intent of the persons specifying the system under design. All design documents should also be machine-readable in order to process them in a computer. Therefore, specifications should be captured in languages which are readable by humans and by computers.

Initially, such specifications could use a natural language such as English or Japanese. Even this natural language description should be captured in a design document, so that the final implementation can be checked against the original document. However, natural languages are not sufficient for later design phases, since natural languages lack key requirements for specification techniques: It is necessary to check specifications for completeness and absence of contradictions. Furthermore, it should be possible to derive implementations from the specification in a systematic way. Natural languages do not meet these requirements.

- **Portability and flexibility:** Specifications should be independent of specific hardware platforms so that they can be easily used for a variety of target platforms. Ideally, changing the hardware platform should have no impact on the specification. In practice, small changes may have to be tolerated.
- **Termination:** It should be feasible to identify terminating processes from the specification. This means that we would like to use specifications for which the halting problem (the problem of figuring out whether or not a certain algorithm will terminate; see, for example, [469]) is decidable.
- **Support for non-standard I/O devices:** Many embedded systems use I/O devices other than those typically found in a PC. It should be possible to describe inputs and outputs for those devices conveniently.
- **Non-functional properties:** Actual systems under design must exhibit a number of non-functional properties, such as fault tolerance, size, extendability, expected lifetime, power consumption, weight, disposability, user friendliness, and electromagnetic compatibility (EMC). There is no hope that all these properties can be defined in a formal way.
- **Support for the design of dependable systems:** Specification techniques should provide support for designing dependable systems. For example, specification languages should have unambiguous semantics, facilitate formal verification, and be capable of describing security and safety requirements.
- **No obstacles to the generation of efficient implementations:** Since embedded systems must be efficient, no obstacles prohibiting the generation of efficient realizations should be present in the specification.
- **Appropriate model of computation (MoC):** The von-Neumann model of sequential execution combined with some communication techniques is a commonly used MoC. However, this model has a number of serious problems, in particular for embedded system applications. Problems include the following:
 - Facilities for describing timing are lacking.
 - Von-Neumann computing is implicitly based on accesses to globally shared memory (such as in Java). It has to guarantee mutually exclusive access to shared resources. Otherwise, multi-threaded applications allowing preemptions

at any time can lead to very unexpected program behaviors². Using primitives for ensuring mutually exclusive access can, however, very easily lead to deadlocks. Possible deadlocks may be difficult to detect and may remain undetected for many years.

Example 2.2: Lee [316] provided a very alarming example in this direction. Lee studied implementations of a simple observer pattern in Java. For this pattern, changes of values must be propagated from some producer to a set of subscribed observers. This is a very frequent pattern in embedded systems, but is difficult to implement correctly in a multi-threaded von-Neumann environment with preemptions. Lee’s code is a possible implementation of the observer pattern in Java for a multi-threaded environment:

```
public synchronized void addListener(listener) {...}
public synchronized void setValue(newValue)
{
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
        myListeners[i].valueChanged(newValue)    }
}
```

Method `addListener` subscribes new observers, and method `setValue` propagates new values to subscribed observers. In general, in a multi-threaded environment, threads can be preempted any time, resulting in an arbitrarily interleaved execution of these threads. Adding observers while `setValue` is already active could result in complications, i.e., we would not know if the new value had reached the new listener. Moreover, the set of observers constitutes a global data structure of this class. Therefore, these methods are synchronized in order to avoid changing the set of observers while values are already partially propagated. This way, only one of the two methods can be active at a given time. This mutual exclusion is necessary to prevent unwanted interleavings of the execution of methods in a multi-threaded environment. Why is this code problematic? It is problematic since `valueChanged` could attempt to get exclusive access to some resource (say, R). If that resource is allocated to some other method (say, A), then this access is delayed until A releases R. If A calls (possibly indirectly) `addListener` or `setValue` before releasing R, then these methods will be in a deadlock: `setValue` waits for R, releasing R requires A to proceed, and A cannot proceed before its call of `setValue` or `addListener` is serviced. Hence, we will have a deadlock.

This example demonstrates the existence of deadlocks resulting from using multiple threads which can be arbitrarily preempted and therefore require mutual exclusion for their access to critical resources. Lee showed [316] that many of the proposed “solutions” of the problem are problematic themselves. So, even this very simple pattern is difficult to implement correctly in a multi-threaded von-Neumann environment. This example shows that concurrency is really difficult to understand for humans and there may be the risk of oversights, even after very rigorous code inspections. ▽

²Examples are typically provided in courses on operating systems.

Lee came to the drastic conclusion that “*nontrivial software written with threads, semaphores, and mutexes is incomprehensible to humans*” and that “*threads as a concurrency model are a poor match for embedded systems. ... they work well only ... where best-effort scheduling policies are sufficient*” [318].

The underlying reasons for deadlocks have been studied in detail in the context of operating systems (see, for example, [483]). From this context, it is well-known that four conditions must hold at run-time to get into a deadlock: mutual exclusion, no preemption of resources, holding resources while waiting for more, and a cyclic dependency between threads. Obviously, all four conditions are met in the above example. The theory of operating systems provides no general way out of this problem. Rare deadlocks may be acceptable for a PC, but they are clearly unacceptable for a safety-critical system.

We would like to specify our SUD such that we do not have to care about possible deadlocks. Therefore, it makes sense to study non-von-Neumann MoCs avoiding this problem. We will study such MoCs from the next section onward. It will be shown that the observer pattern can be easily implemented in other MoCs.

From the list of requirements, it is already obvious that there will not be any single formal language meeting all these requirements. Therefore, in practice, we must live with compromises and possibly also with a mixture of languages (each of which would be appropriate for describing a certain type of problems). The choice of the language used for an actual design will depend on the application domain and the environment in which the design has to be performed. In the following, we will present a survey of languages that can be used for actual designs. These languages will demonstrate the essential features of the corresponding MoC.

2.2 Models of Computation

Models of computation (MoCs) describe the mechanism assumed for performing computations. In the general case, we must consider systems comprising components. It is now common practice to strictly distinguish between the computations performed in the components and communication. This distinction paves the way for **reusing components** in different contexts and enables *plug-and-play* for system components. Accordingly, we define models of computation as follows [255–257, 315]:

Definition 2.2: Models of computation (MoCs) define

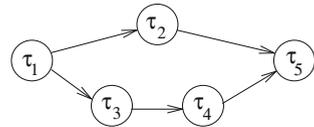
- **Components** and the organization of computations in such components: Procedures, processes, functions, finite state machines are possible components.
- **Communication protocols**: These protocols describe methods for communication between components. Asynchronous message passing and rendez-vous based communication are examples of communication protocols.

Relations between components can be captured in graphs. In such graphs, we will refer to the computations also as processes or tasks. Accordingly, relations

between these will be captured by **task graphs** and **process networks**. Nodes in the graph represent components performing computations. Computations map input data streams to output data streams. Computations are sometimes implemented in high-level programming languages. Typical computations contain (possibly non-terminating) iterations. In each cycle of the iteration, they consume data from their inputs, process the data received, and generate data on their output streams. Edges represent relations between components. We will now introduce these graphs at a more detailed level.

The most obvious relation between computations is their causal dependence: Many computations can only be executed after other computations have terminated. This dependence is typically captured in **dependence graphs**. Fig. 2.2 shows a dependence graph for a set of computations.

Fig. 2.2 Dependence graph



Definition 2.3: A dependence graph is a directed graph $G = (\tau, E)$, where τ is the set of **vertices** or **nodes** and E is the set of **edges**. $E \subseteq \tau \times \tau$ imposes a relation on τ . If $(\tau_1, \tau_2) \in E$ with $\tau_1, \tau_2 \in \tau$, then τ_1 is called an **immediate predecessor** of τ_2 and τ_2 is called an **immediate successor** of τ_1 . Suppose E^* is the transitive closure of E . If $(\tau_1, \tau_2) \in E^*$, then τ_1 is called a **predecessor** of τ_2 and τ_2 is called a **successor** of τ_1 .

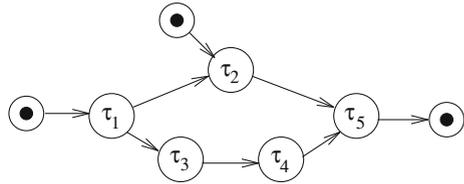
Such dependence graphs form a special case of task graphs. Task graphs may contain more information than modeled in Fig. 2.2. For example, task graphs may include the following extensions of dependence graphs:

1. **Timing information:** Tasks may have arrival times, deadlines, periods, and execution times. In order to show them graphically, it may be useful to include this information in the graphs. However, we will indicate such information separately from the graphs in this book.
2. **Distinction between different types of relations** between computations: Precedence relations just model constraints for possible execution sequences. At a more detailed level, it may be useful to distinguish between constraints for scheduling and communication between computations. Communication can also be described by edges, but additional information may be available for each of the edges, such as the time of the communication and the amount of information exchanged. Precedence edges may be kept as a separate type of edges, since there could be situations in which computations must execute sequentially even though they do not exchange information.

In Fig. 2.2, input and output (I/O) are not explicitly described. Implicitly, it is assumed that computations without any predecessor in the graph might be receiving input at some time. Also, they might generate output for the successor and

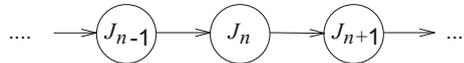
that this output could be available only after the computation has terminated. It is often useful to describe input and output more explicitly. In order to do this, another kind of relation is required. Using the same symbols as Thoen [515], we use partially filled circles for denoting input and output. In Fig. 2.3, partially filled circles identify I/O edges.

Fig. 2.3 Graph including I/O nodes and edges



3. **Exclusive access to resources:** Computations may be requesting exclusive access to some resource, for example, to some input/output device or some communication area in memory. Information about necessary exclusive access should be taken into account during scheduling. Exploiting this information might, for example, be used to avoid the priority inversion problem (see p. 206). Information concerning exclusive access to resources can be included in the graphs.
4. **Periodic schedules:** Many computations, especially in digital signal processing, are periodic. This means that we must distinguish more carefully between a task and its execution (the latter is frequently called a **job** [332]). Graphs for such schedules are infinite. Figure 2.4 shows a graph including jobs J_{n-1} to J_{n+1} of a periodic task.

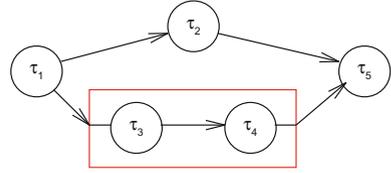
Fig. 2.4 Graph including jobs



5. **Hierarchical graph nodes:** The complexity of the computations denoted by graph nodes may be quite different. On the one hand, specified computations may be quite involved and contain thousands of lines of program code. On the other hand, programs can be split into small pieces of code so that in the extreme case, each of the nodes corresponds only to a single operation. The graph node complexity is also called their **granularity**. Which granularity should be used? There is no universal answer to this. For some purposes, the granularity should be as large as possible. For example, if we consider each of the nodes as one process to be scheduled by a real-time operating system (RTOS), it may be wise to work with large nodes in order to minimize context switches between different processes. For other purposes, it may be better to work with nodes modeling just a single operation. For example, nodes must be mapped to hardware or to software. If a certain operation (such as the frequently used discrete cosine transform, or DCT) can be mapped to special purpose hardware, then it should not be buried in a complex node that contains many other operations. It should rather be modeled as its own node. In order to avoid frequent changes of the granularity, hierarchical graph nodes are very useful. For example, at a high hierarchical level, the nodes

may denote complex tasks, at a lower level basic blocks³, and at an even lower level individual arithmetic operations. Figure 2.5 shows a hierarchical version of the dependence graph in Fig. 2.2, using a rectangle to denote a hierarchical node.

Fig. 2.5 Hierarchical task graph



As indicated above, MoCs can be classified according to the models of communication (reflected by edges in the task graphs) and the model of computations within the components (reflected by the nodes in the task graphs). In the following, we will explain prominent examples of such models:

- **Models of communication:**

We distinguish between two communication paradigms: **shared memory** and **message passing**. Other communication paradigms exist (e.g., entangled states in quantum mechanics [64]), but are not considered in this book.

- **Shared memory:**

For shared memory, communication is performed by accesses to the same memory from all components. Access to shared memory should be protected, unless access is totally restricted to reads. If writes are involved, exclusive access to the memory must be guaranteed while components are accessing shared memories. Segments of program code, for which exclusive access must be guaranteed, are called **critical sections**. Several mechanisms for guaranteeing exclusive access to resources have been proposed. These include semaphores, conditional critical regions, monitors, and spin locks (see books on operating systems like Stallings [483]). Shared memory-based communication can be very fast, but is difficult to implement in multiprocessor systems if no common memory is physically available.

- **Message passing:** In this case, messages are sent and received. Message passing can be implemented easily even if no common memory is available. However, message passing is generally slower than shared memory-based communication. We distinguish between three kinds of message passing:

- **Asynchronous message passing**, also called **non-blocking communication:** In asynchronous message passing, components communicate by sending messages through channels which can buffer the messages. The sender does not need to wait for the recipient to be ready to receive the message. In real life, this corresponds to sending a letter or an e-mail. A potential problem is the fact that messages must be stored and that message buffers can overflow. There

³Basic blocks are code blocks of maximum length not including any branch except possibly at their end and not being branched into.

are variations of this scheme, including communicating finite state machines (see p. 58) and data flow models (see p. 64).

- **Synchronous message passing or blocking communication, *rendez-vous* based communication:** In synchronous message passing, available components communicate in atomic, instantaneous actions called *rendez-vous*. The component reaching the point of communication first has to wait until the partner has also reached its point of communication. In real life, this corresponds to physical meetings or phone calls. There is no risk of overflows, but performance may suffer. Examples of languages following this model of computation include CSP (see p. 106) and Ada (see p. 107).
- **Extended *rendez-vous*, remote invocation:** In this case, the sender is allowed to continue only after an acknowledgment has been received from the recipient. The recipient does not have to send this acknowledgment immediately after receiving the message, but can do some preliminary checking before actually sending the acknowledgment.

- **Organization of computations within the components:**

- **Differential equations:** Differential equations are capable of modeling analog circuits and physical systems. Hence, they can find applications in cyber-physical system modeling.
- **Finite state machines (FSMs):** This model is based on the notion of a finite set of states, inputs, outputs, and transitions between states. Several of these machines may need to communicate, forming so-called **communicating finite state machines (CFSMs)**.
- **Data flow:** In the data flow model, the availability of data triggers the possible execution of operations.
- **Discrete event model:** In this model, there are events carrying a totally ordered time stamp, indicating the time at which the event occurs. Discrete event simulators typically contain a global event queue sorted by time. Entries from this queue are processed according to this order. The disadvantage is that this model relies on a global notion of event queues, making it difficult to map the semantic model onto parallel implementations. Examples include VHDL (see p. 94), SystemC (see p. 93), and Verilog (see p. 104).
- **Von-Neumann model:** This model is based on the sequential execution of sequences of primitive computations.

- **Combined models:** Actual languages are typically combining a certain model of communication with an organization of computations within components. For example, StateCharts (see p. 47) combines finite state machines with shared memories. SDL (see p. 58) combines finite state machines with asynchronous message passing. Ada (see p. 107) and CSP (see p. 106) combine von-Neumann execution with synchronous message passing. Table 2.1 gives an overview of combined models most of which we will consider in this chapter. This table also includes examples of languages for many of the MoCs.

Let us look at MoCs with a defined model for computations within components. For differential equations, Modelica [382], commercial languages such as

Table 2.1 Overview of MoCs and languages considered

Communication/organization of components	Shared memory	Message passing	
		Synchronous	Asynchronous
Undefined components	Plain text or graphics, use cases		
		(Message) sequence charts	
Differential equations	Modelica, Simulink [®] , VHDL-AMS		
Communicating finite state machines (CFSMs)	StateCharts		SDL
Data flow	Scoreboarding, Tomasulo algorithm		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, ...	
Discrete event (DE) model ^a	VHDL, Verilog, and SystemC	(Only experimental systems) Distributed DE in Ptolemy	
Von-Neumann model	C, C++, Java	C, C++, Java, ... with libraries	
		CSP, Ada	

^aThe classification of VHDL, Verilog, and SystemC is based on the implementation of these languages in simulators. Message passing can be modeled in these languages “on top” of the simulation kernel.

Simulink[®] [510], and the extension VHDL-AMS [236] of the hardware description language VHDL are examples of languages.

Scoreboarding and the Tomasulo algorithm are data flow-driven techniques for dynamically scheduling instructions in computer architectures. They are described in books in computer architecture (see, for example, Hennessy and Patterson [205]) and not presented in this book.

Some MoCs have advantages in certain application areas, while others have advantages in others. Choosing the “best” MoC for a certain application may be difficult. Being able to mix MoCs (such as in the Ptolemy framework [435]) can be a way out of this dilemma. Also, models may be translated from one MoC into another one. Non-von-Neumann models are frequently translated into von-Neumann models. The distinction between the different models is blurring if the translation between them is easy.

Designs starting from non-von-Neumann models are frequently called **model-based designs** [400]. The key idea of model-based design is to have some abstract model of the system under design (SUD). Properties of the SUD can then be studied at the level of this model, without having to care about software code. Software code is generated only after the behavior of the model has been studied in detail, and this software is generated automatically [453]. The term “model-based design” is usually associated with models of control systems, comprising traditional control system elements such as integrators and differentiators. However, this view may be too restricted, since we could also start with abstract models of consumer systems.

In the following, we will present different MoCs, using existing languages as examples for demonstrating their features. A related (but shorter) survey is provided by Edwards [141]. For a more comprehensive presentation, see [180].

2.3 Early Design Phases

The very first ideas about systems are frequently captured in a very informal way, possibly on paper. Frequently, only descriptions of the SUD in a natural language such as English or Japanese exist in the early phases of design projects. They are typically using a very informal style. These descriptions should be captured in some machine-readable document. They should be encoded in the format of some word processor and stored by a tool managing design documents. A good tool would allow link between the requirements, a dependence analysis as well as version management. DOORS® [221] exemplifies such a tool.

2.3.1 Use Cases

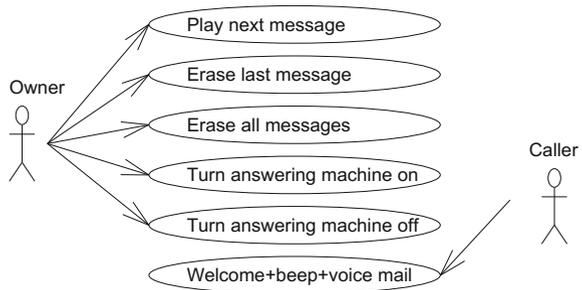
For many applications, it is beneficial to envision potential usages of the SUD. Such usages are captured in **use cases**. Use cases describe possible applications of the SUD. Different notations for use cases could be used.

Support for a systematic approach to early specification phases is the goal of the so-called UML™ standardization effort [161, 199, 413]. UML stands for “Unified Modeling Language.” UML was designed by leading software technology experts and is supported by commercial tools. UML primarily aims at the support of the software design process. UML provides a standardized form for use cases.

For use cases, there is neither a precisely specified model of the computations nor is there a precisely specified model of the communication. It is frequently argued that this is done intentionally in order to avoid caring about too many details during the early design phases.

Example 2.3: For example, Fig. 2.6 shows some use cases for an answering machine⁴. There are five use cases for the owner of the answering machine and one for potential callers. We have to make sure that all six use cases can be implemented correctly.

Fig. 2.6 Use case example



⁴We assume that UML is covered in-depth in a software engineering course included in the curriculum. Therefore, UML is only briefly discussed in this book.

Use cases identify different classes of users as well as the applications to be supported by the SUD. In this way, it is possible to capture expectations at a very high level.

2.3.2 (Message) Sequence Charts and Time/Distance Diagrams

At a more detailed level, we might want to explicitly indicate the sequences of messages which must be exchanged between components in order to implement some use of the SUD. **Sequence charts** (SCs)—earlier called **message sequence charts** (MSCs)—provide a mechanism for this. Sequence charts use one dimension (usually the vertical dimension) of a two-dimensional chart to denote sequences and the second dimension to reflect the different communication components. SCs describe partial orders between message transmissions, and they display a possible behavior of a SUD. SCs are also standardized in UML. UML 2.0 has extended SCs with elements allowing a more detailed description than UML 1.0.

Example 2.4: Figure 2.7 shows one of the use cases of the answering machine as an example. Dashed lines are so-called lifelines. Messages are assumed to be ordered

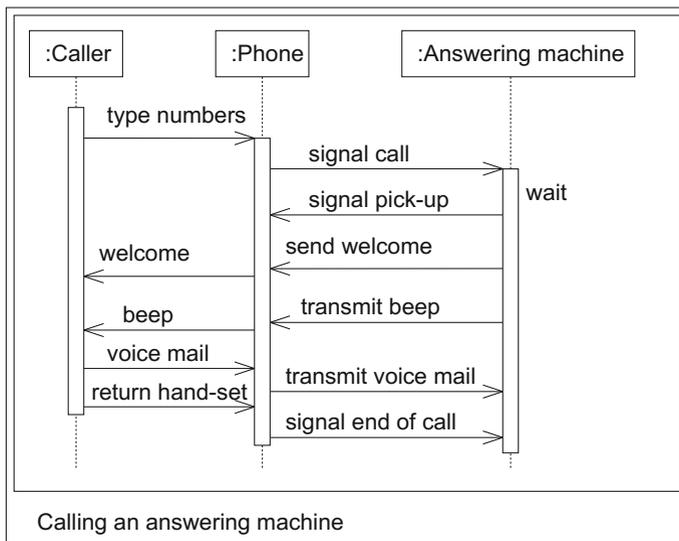


Fig. 2.7 Answering machine in UML™

according to their sequence along the lifeline. We assume that, in this example, all information is sent in the form of messages. Arrows used in this diagram denote asynchronous messages. This means several messages can be sent by a sender without waiting for the receipt to be confirmed. Boxes on top of lifelines represent active control at the corresponding component. In the example, the answering machine is waiting for the user to pick up the phone within a certain amount of time. If he or she fails to do so, the machine signals a pickup itself and sends a welcome message to the caller. The caller is then supposed to leave a voice mail message. Alternative sequences (e.g., an early termination of the call by the caller or the callee picking up the phone) are not shown. ▽

Complex control-dependent actions cannot be described by SCs. Other MoCs must be used for this. Frequently, certain preconditions must be met for a SC to apply. Such preconditions, a distinction between sequences which might happen and those which must happen, as well as other extensions are available in the so-called Live Sequence Charts [114].

Time/distance diagrams (TDDs) are a commonly used variant of SCs. In time/distance diagrams, the vertical dimension reflects real time, not just sequence. In some cases, the horizontal dimension also models the real distance between the components. TDDs provide the right means for visualizing schedules of trains or buses.

Example 2.5: Figure 2.8 is an example of a TDD. This example refers to trains between Amsterdam, Cologne, Brussels, and Paris. Trains can run from either

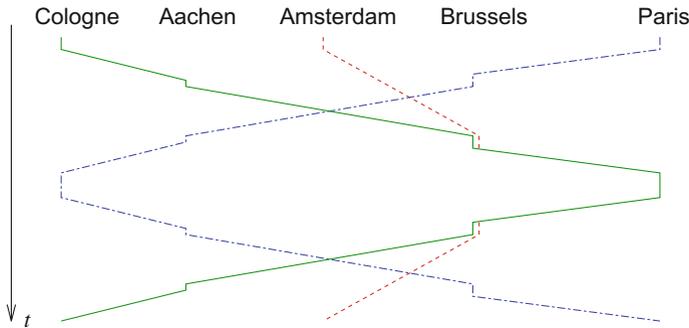


Fig. 2.8 Time/distance diagram

Amsterdam or Cologne to Paris via Brussels. Aachen is included as an intermediate stop between Cologne and Brussels. Vertical segments correspond to time spent at stations. For one of the trains, there is a timing overlap between the trains coming from Cologne and Amsterdam at Brussels. There is a second train which travels between Paris and Cologne which is not related to an Amsterdam train. This example and other examples can be simulated with the levi simulation software [473]. ▽

Example 2.6: A larger, more realistic example is shown in Fig. 2.9.

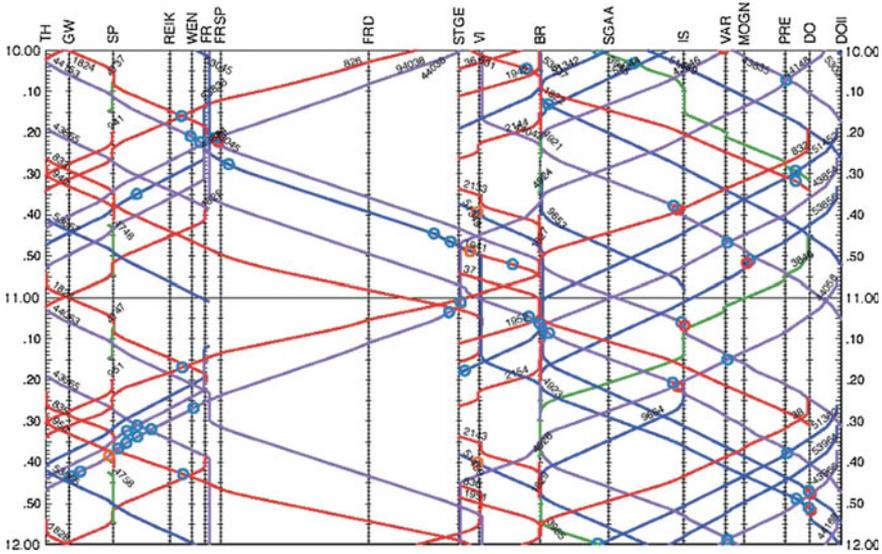


Fig. 2.9 Railway traffic displayed by a TDD (courtesy H. Brändli, IVT, ETH Zürich), © ETH Zürich

This example [215] describes simulated Swiss railway traffic in the Lötschberg area. Different station names are shown along the horizontal lines. The vertical dimension reflects real time. Slow and fast trains can be distinguished by their slope in the graph. Slow trains are characterized by steep slopes, possibly also containing significant waiting time at the stations (vertical slopes). For fast trains, slopes are almost flat. Trains are stopping only at a subset of the stations.

In the presented example, it is not known whether the timing overlap at stations happens coincidentally or whether some real synchronization for connecting trains is required. Furthermore, permissible deviations from the schedule (min/max timing behavior) are not visible. ▽

SCs and TDDs are very frequently used in practice. For example, they are valuable for applications of the IoT. One of the key distinctions between SCs and TDDs is that SCs do not include any reference to real time. TDDs are appropriate means for representing typical schedules. However, SCs and TDDs fail to provide information about necessary synchronization.

UML was initially not designed for real-time applications. UML 2.0 includes **timing diagrams** as a special class of diagrams. Such diagrams enable referring to physical time, similar to TDDs. Also, certain UML “profiles” (see p. 117) allow additional annotations to refer to time [352].

2.3.3 Differential Equations

Differential equations can be written in the language of mathematics. Inputs for design tools typically require certain variants of this language. We exemplify such a variant with Modelica [382], a language aiming at modeling cyber-physical systems.

Modelica has graphical as well as textual forms. Using the graphical form, systems can be described as sets of interconnected blocks. Each block can be described by equations. Connections between blocks denote common variables in the sense of mathematics. The information about each block together with information about connections can be transformed into a global set of equations. This process is called flattening of the hierarchy. Just like in mathematics, equations (and connections) have a bidirectional meaning (in contrast to programming languages).

Example 2.7: The following model⁵ represents the bouncing ball example introduced on p. 10:

```

model StickyBall
  type Height = Real(unit = "m");
  type Velocity = Real(unit = "m/s");
  parameter Real s = 0.8 "Restitution";
  parameter Height h0=1.0 "Initial hight";
  constant Velocity eps=1e-3 "small velocity";
  Boolean stuck;
  Height h;
  Velocity v;
initial equation
  v = 0;
  h = h0;
  stuck = false;
equation
  v = der(h);
  der(v) = if stuck then 0 else -9.81;
  when h <= 0.0 then
    stuck = abs(v) < eps;
    reinit(v, if stuck then 0 else -s*v);
  end when;
end StickyBall;

```

In the equations part, velocity v is defined as the derivative of the height h . The derivative of velocity (the acceleration) is set to standard gravity (-9.81), unless the ball is already sticking to the surface. Equations have a bidirectional meaning. For this set of equations, there are boundary conditions defined in the initial equation part. Mathematical equations can be integrated numerically. This procedure is exploited in the description of the bouncing: when-clauses can be used to define *events* which happen while solving the equations. In the particular example, an event is generated

⁵This model has been derived from the model published by M. Tiller [517].

when the height becomes less or equal to zero. Whenever this event is generated while the velocity is still sufficiently large, the velocity is inverted and reduced by a factor of s , called **restitution**. s is the square root of the so-called rebound coefficient r [157]. The reinit-clause effectively defines another boundary condition.

However, if the velocity is smaller than eps , the ball is assumed to become sticky and the velocity is set to zero, suppressing all future activities. The resulting model can be simulated, for example, with OpenModelica⁶.

The mathematical background is as follows: Let $v_0 = \sqrt{2gh_0}$ be the velocity just before the first bounce [157]. The time until the n -th bouncing is as follows [157]:

$$t_n = \frac{2v_0}{g} \sum_{k=0}^{n-1} s^k \quad (2.1)$$

As long as $s < 1$, this geometric series converges to

$$t_{final} = \lim_{n \rightarrow \infty} \frac{2v_0}{g} \sum_{k=0}^{n-1} s^k = \frac{2v_0}{g(1-s)} \quad (2.2)$$

This means that there is an upper bound on the time for the bounces, but not on the number of bounces. This corresponds to the fact that, mathematically speaking, infinite series may be converging to a finite value⁷.

Using sets of equations involving derivatives in Modelica brings us close to the language of mathematics and physics. However, events introduce sequential behavior. The implicit numerical integration procedure also introduces the hazard of numerical precision problems. In fact, already the test ≤ 0.0 reflects the fact that we might miss the case of h being exactly 0. Another hazard is present in the published model for the non-sticky ball [517]: Numerical precision problems result in an OpenModelica solution for which the ball penetrates the floor for large times t . This problem is caused by not generating events if the time distance between bounces is too small.

This example demonstrates very nicely the advantages and limitations of Modelica: On the one hand, it is feasible to describe even the physical part of cyber-physical systems. On the other hand, we are not exactly using the language of mathematics, and in this way, we are introducing modeling hazards. ∇

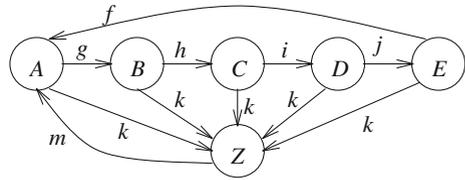
2.4 Communicating Finite State Machines (CFSMs)

In the following sections, we will consider the design of digital systems only. Compared to early design phases, we need more precise models of our SUD. We mentioned already on p. 15 and on p. 30 that we need to describe state-oriented behavior. State diagrams are a classical means of doing this. Figure 2.10 (the same as Fig. 2.1) shows an example of a classical state diagram, representing a **finite state machine (FSM)**.

⁶See <https://openmodelica.org/>.

⁷Note the link to the paradox of Achilles and the turtle [558].

Fig. 2.10 State diagram



Circles denote states. We will consider FSMs for which only one of their states is active. Such FSMs are called **deterministic** FSMs. Edges denote state transitions. Edge labels represent events. Let us assume that a certain state of the FSM is active, and that an event happens which corresponds to one of the outgoing edges for the active state. Then, the FSM will change its state from the currently active state to the one indicated by the edge. FSMs may be implicitly clocked. Such FSMs are called **synchronous** FSMs. For synchronous FSMs, state changes will happen only at clock transitions. FSMs may also generate output (not shown in Fig. 2.10). For more information about classical FSMs, refer to, for example, Kohavi et al. [290].

2.4.1 Timed Automata

Classical FSMs do not provide information about time. In order to model time, classical automata have been extended to also include timing information. Timed automata are essentially automata extended with real-valued variables. “The variables model the logical clocks in the system, that are initialized with zero when the system is started, and then increase synchronously with the same rate. Clock constraints, i.e., guards on edges, are used to restrict the behavior of the automaton. A transition represented by an edge can be taken when the clocks’ values satisfy the guard labeled on the edge. Clocks may be reset to zero when a transition is taken” [46].

Example 2.8: Figure 2.11 shows an example. The answering machine is usually in the initial state on the left. Whenever a *ring* signal is received, clock x is reset to 0 and a transition into a waiting state is made. If the called person lifts off the handset, talking can take place until the handset is returned. Otherwise, a transition to state *play text* can take place if time has reached a value of 4.

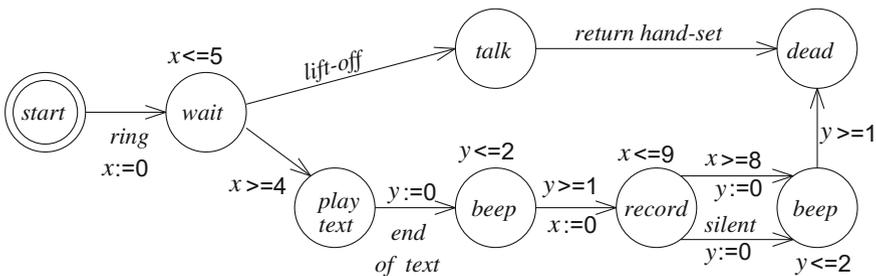


Fig. 2.11 Servicing an incoming line in an answering machine

Once the transition took place, a recorded message is played and this phase is terminated with a beep. Clock y ensures that this beep lasts at least one time unit. After the beep, clock x is reset to 0 again and the answering machine is ready for recording. If time has reached a value of 8 or if the caller remains silent, the next beep is played. This second beep again lasts at least one time unit. After the second beep, a transition is made into the final state. In this example, transitions are either caused by inputs (such as *lift-off*) or by so-called **clock constraints**. ∇

Clock constraints describe transitions which **can** take place, but they do not have to. In order to make sure that transitions actually take place, additional **location invariants** can be defined. Location invariants $x \leq 5$, $x \leq 9$, and $y \leq 2$ are used in the example such that transitions will take place no later than one time unit after the enabling condition became true. Using two clocks is for demonstration purposes only; a single clock would be sufficient.

Formally speaking, timed automata can be defined as follows [46]: Let C be a set of real-valued, non-negative variables representing clocks. Let Σ be a finite alphabet of possible inputs.

Definition 2.4: A **clock constraint** is a conjunctive formula of atomic constraints of the form $x \circ n$ or $(x - y) \circ n$ for $x, y \in C$, $\circ \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$.

Note that constants n used in the constraints must be integers, even though clocks are real-valued. An extension to rational constants would be easy, since they could be turned into integers with simple multiplications. Let $B(C)$ be the set of clock constraints.

Definition 2.5 (Bengtson [46]): A **timed automaton** is a tuple (S, s_0, E, I) where

- S is a finite set of states.
- s_0 is the initial state.
- $E \subseteq S \times B(C) \times \Sigma \times 2^C \times S$ is the set of edges. $B(C)$ models the conjunctive condition which must hold and Σ models the input which is required for a transition to be enabled. 2^C reflects the set of clock variables which are reset whenever the transition takes place.
- $I : S \rightarrow B(C)$ is the set of invariants for each of the states. $B(C)$ represents the invariant which must hold for a particular state S . This invariant is described as a conjunctive formula.

This first definition is usually extended to allow parallel compositions of timed automata. Timed automata having a large number of clocks tend to be difficult to understand. More details about timed automata can be found, for example, in papers by Dill et al. [128] and Bengtsson et al. [46].

Simulation and verification of timed automata is possible with the popular tool UPPAAL⁸. UPPAAL supports concurrency and data variables.

⁸See <http://www.uppaal.org> for the academic and <http://www.uppaal.com> for the commercial version.

Timed automata extend classical automata with timing information. However, many of our requirements for specification techniques are not met by timed automata. In particular, in their standard form, they do not provide hierarchy and concurrency.

2.4.2 StateCharts: Implicit Shared Memory Communication

The StateCharts language is presented here as a very prominent example of a language based on automata and supporting hierarchical models as well as concurrency. It does include a limited way of specifying timing.

The StateCharts language was introduced by David Harel [197] in 1987 and later described more precisely in [135]. According to Harel, the name was chosen since it was “*the only unused combination of flow or state with diagram or chart.*”

2.4.2.1 Modeling of Hierarchy

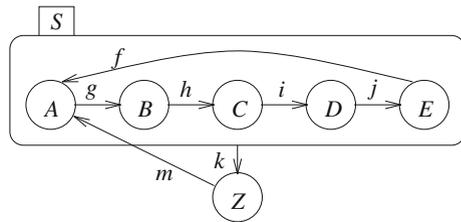
The StateCharts language describes extended FSMs. Due to this, they can be used for modeling state-oriented behavior. The key extension is **hierarchy**. Hierarchy is introduced by means of **super-states**.

Definition 2.6: States comprising other states are called **super-states**.

Definition 2.7: States included in super-states are called **sub-states** of the super-states.

Example 2.9: Figure 2.12 shows a StateCharts example. It is a hierarchical version of Fig. 2.10.

Fig. 2.12 Hierarchical state diagram



Super-state S includes states A , B , C , D , and E . Suppose the FSM is in state Z (we will also call Z to be an **active state**). Now, if input m is applied to the FSM, then A and S will be the new active states. If the FSM is in S and input k is applied, then Z will be the new active state, regardless of whether the FSM is in sub-states A , B , C , D , or E of S . In this example, all states contained in S are non-hierarchical states. ▽

In general, sub-states of S could again be super-states consisting of sub-states themselves. Also, **whenever a sub-state of some super-state is active, the super-state is active as well.**

Definition 2.8: Each state which is not composed of other states is called a **basic state**.

Definition 2.9: For each basic state s , the super-states containing s are called **ancestor states**.

The FSM of Fig. 2.12 can only be in one of the sub-states of super-state S at any time. Super-states of this type are called **OR super-states**⁹.

In Fig. 2.12, k might correspond to an exception for which state S has to be left. The example already shows that the hierarchy introduced in StateCharts enables a compact representation of exceptions.

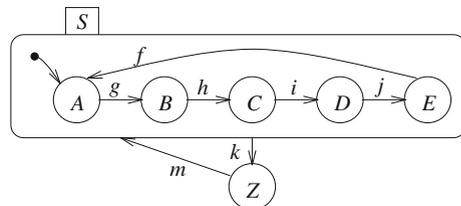
StateCharts allows hierarchical descriptions of systems in which a system description comprises descriptions of subsystems which, in turn, may contain descriptions of subsystems. The **hierarchy** of the entire system can be represented by a **tree**. The root of the tree corresponds to the system as a whole, and all inner nodes correspond to hierarchical descriptions (in the case of StateCharts called super-nodes). The leaves of the hierarchy are non-hierarchical descriptions (in the case of StateCharts called basic states).

So far, we have used explicit, direct edges to basic states to indicate the next state. The disadvantage of that approach is that the internal structure of super-states cannot be hidden from the environment. However, in a true hierarchical environment, we should be able to hide the internal structure so that it can be described later or changed later without affecting the environment. This is possible with other mechanisms for describing the next state.

The first additional mechanism is the **default state mechanism**. It can be used in super-states to indicate the particular sub-states that will become active if the super-states become active. In diagrams, default states are identified by edges starting at small filled circles.

Example 2.10: Figure 2.13 shows a state diagram using the default state mechanism. It is equivalent to the diagram in Fig. 2.12. Note that the filled circle does not constitute a state itself.

Fig. 2.13 State diagram using the default state mechanism



▽

⁹More precisely, they should be called XOR super-states, since the FSM is in **either** A , B , C , D , or E . However, this name is not commonly used in the literature.

Another mechanism for specifying next states is the **history mechanism**. With this mechanism, it is possible to return to the last sub-state that was active before a super-state was left. The history mechanism is symbolized by a circle containing the letter H. Please do not confuse circles comprising this letter with states! We will be using a different font for states and the history mechanism in order to reduce the risk of confusion. In order to define the next state for the very initial transition into a super-state, the history mechanism is frequently combined with the default mechanism.

Example 2.11: Figure 2.14 shows an example. The behavior of the FSM is now somewhat different. If we input m while the system is in Z , then the FSM will enter A if this is the very first time we enter S , and otherwise, it will enter the last state that we were in before leaving S . This mechanism has many applications. For example, if k denotes an exception, we could use input m to return to the state we were in before the exception. States $A, B, C, D,$ and E could also call Z like a procedure. After completing “procedure” Z , we would return to the calling state. In this way, we are adding elements of programming languages to StateCharts.

Fig. 2.14 State diagram using the history and the default state mechanism

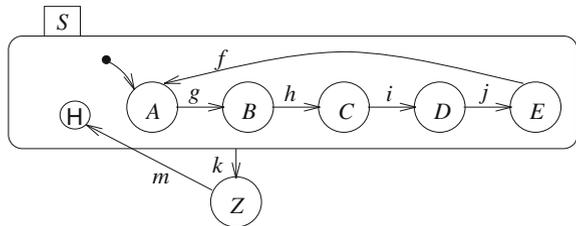


Fig. 2.15 Combining the symbols for the history and the default state mechanism

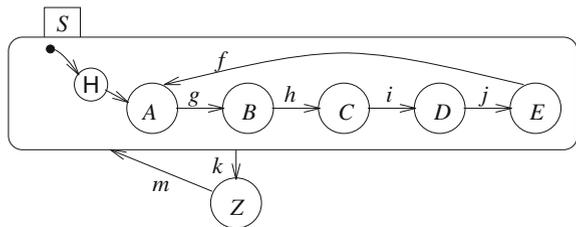


Figure 2.14 can also be redrawn as shown in Fig. 2.15. In this case, the symbols for the default and the history mechanism are combined. ∇

Specification techniques must also be able to describe concurrency conveniently. Toward this end, the StateCharts language provides a second class of super-states, so-called **AND** states.

Definition 2.10: Super-states S are called **AND super-states** if the system containing S will be in **all** of the sub-states of S whenever it is in S .

Example 2.12: An AND super-state is included in the answering machine example shown in Fig. 2.16. An answering machine normally performs two tasks concurrently: It is monitoring the line for incoming calls and the keys for user input. In Fig. 2.16, the corresponding states are called *Lwait* and *Kwait*. Incoming calls are processed in state *Lproc* while the response to pressed keys is generated in state *Kproc*. State *Lproc* is left whenever the caller hangs up the phone. Returning to state *Lwait* due to call termination by the owner is not modeled. Hence, this model provides no protection against stalking.

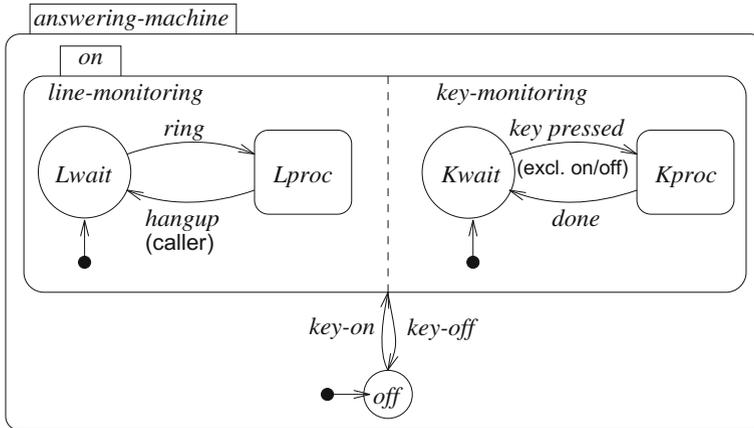


Fig. 2.16 Answering machine

For the time being, we assume that the on/off switch (generating events *key-off* and *key-on*) is decoded separately and pushing it does not result in entering *Kproc*. If the machine is switched off, the line monitoring state as well as the key monitoring state is left and reentered only if the machine is switched on. At that time, default states *Lwait* and *Kwait* are entered. While switched on, the machine will always be in the line monitoring state as well as in the key monitoring state. ∇

For AND super-states, the sub-states entered as a result of entering the super-state can be defined independently. There can be any combination of history, default, and explicit transitions. It is crucial to understand that **all** sub-states will always be entered, even if there is just one explicit transition to one of the sub-states. Accordingly, transitions out of an AND super-state will always result in leaving **all** the sub-states.

Example 2.13: For example, let us modify our answering machine such that the on/off switch, like all other switches, is decoded in state *Kproc* (see Fig. 2.17).

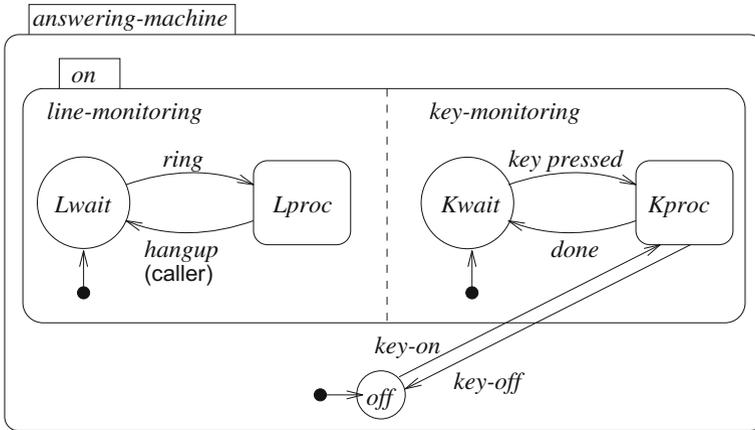


Fig. 2.17 Answering machine with modified on/off switch processing

If pushing that key is detected in *Kwait*, transitions are assumed first into state *Kproc* and then into the *off* state. The second transition results in leaving the line monitoring state as well. Switching the machine on again results in also entering the line monitoring state. ▽

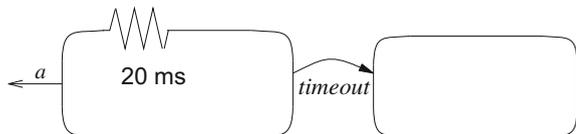
AND super-states provide the key mechanism for describing concurrency in break StateCharts. Each sub-state can be considered a state machine by itself. These machines are communicating with each other, forming **communicating finite state machines** (CFSMs). This term has been used as the title of this section.

Summarizing, we can state the following: **States in StateCharts diagrams are either AND states, OR states, or basic states.**

2.4.2.2 Timers

Due to the requirement to model time in embedded systems, StateCharts also provides timers. Timers are denoted by the symbol shown in Fig. 2.18 on the left.

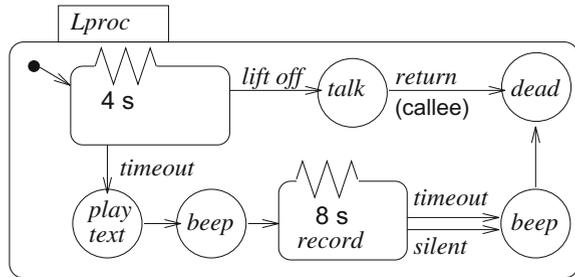
Fig. 2.18 Timer in StateCharts



After the system has been in the state containing the timer for the specified time, a time-out will occur and the system will leave the specified state. Timers can also be used hierarchically.

Timers can be employed, for example, at the next lower level of the hierarchy of the answering machine in order to describe the behavior of state *Lproc*. Figure 2.19 shows a possible behavior for that state. The timing specification is slightly different from the one in Fig. 2.11.

Fig. 2.19 Servicing the incoming line in *Lproc*



Due to the exception-like transition for hang-ups by the caller in Fig. 2.16, state *Lproc* is terminated whenever the caller hangs up. For hang-ups (returns) by the callee, the design of state *Lproc* results in an inconvenience: If the callee hangs up the phone first, the telephone will be dead (and quiet) until the caller has also hung up the phone.

The StateCharts language includes a number of other language elements. For a full description, refer to Harel [197]. A more detailed description of the semantics of StateCharts is described by Drusinsky and Harel [135].

2.4.2.3 Edge Labels and StateMate Semantics

Until now, we have not considered outputs generated by our extended FSMs. Generated outputs can be specified using edge labels. The general form of an edge label is “*event[condition]/reaction.*” All three label parts are optional. The *reaction* part describes the reaction of the FSM to a state transition. Possible reactions include the generation of events and assignments to variables. The *condition* part implies a test of the values of variables or a test of the current state of the system. The *event* part refers to a test of current events. Events can be generated either internally or externally. Internal events are generated as a result of some transition and are described in *reaction* parts. External events are usually described in the model environment.

Examples:

- *on-key / on := 1* (Event test and variable assignment),
- *[on = 1]* (Condition test for a variable value),
- *off-key [not in Lproc] / on := 0* (Event test, condition test for a state, variable assignment. The assignment is performed if the event has occurred and the condition is true).

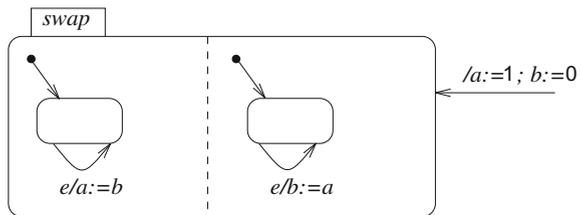
The semantics of edge labels can only be explained in the context of the semantics of StateMate [135], a commercial implementation of StateCharts. StateMate assumes a step-based execution of StateMate descriptions. Each step consists of three phases:

1. In the first phase, the impact of external changes on conditions and events is evaluated. This includes the evaluation of functions which depend on external events. This phase does not include any state changes. In our simple examples, this phase is not actually needed.
2. The next phase is to calculate the set of transitions that should be made in the current step. Variable assignments are evaluated, but the new values are only assigned to temporary variables.
3. In the third phase, state transitions become effective and variables obtain their new values.

The separation into phases 2 and 3 is important in order to guarantee a reproducible behavior of StateMate models.

Example 2.14: Consider the StateMate model of Fig. 2.20.

Fig. 2.20 Mutually dependent assignments



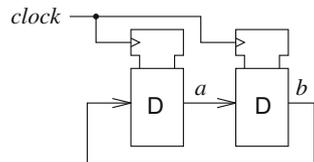
In the second phase, new values for a and b are stored in temporary variables, say a' and b' . In the final phase, temporary variables are copied into the user-defined variables:

```

phase 2:  $a' := b; b' := a;$ 
phase 3:  $a := a'; b := b'$ 
```

As a result, the values of the two variables will be swapped each time an event e happens. This behavior corresponds to that of two cross-coupled registers (one for each variable) connected to the same clock (see Fig. 2.21) and reflects the operation of a synchronous (clocked) finite state machine including those two registers¹⁰.

Fig. 2.21 Cross-coupled D-type registers



¹⁰We adopt IEEE standard schematic symbols [230] for gates and registers for all the schematics in this book. The symbols in Fig. 2.21 denote clocked D-type registers.

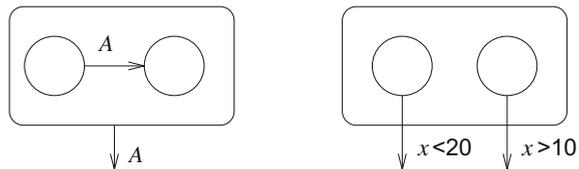
Without the separation into phases, the same value would be assigned to both variables. The result would depend on the sequence in which the assignments were performed. ∇

The separation into (at least) two phases is quite typical for languages that try to reflect the operation of synchronous hardware. We will find the same separation in VHDL (see p. 102). Due to the separation, the results do not depend on the order in which parts of the model are executed by the simulation. This property is extremely important. Otherwise, there could be simulation runs generating different results, all of which would be considered correct. This is not what we expect from the simulation of a real circuit with a fixed behavior and it could be very confusing in design procedures. There are different names for this property:

- Kahn [266] calls this property **determinate**.
- In other papers, this property is called **deterministic**. However, the term “deterministic” is employed with different meanings:
 - It is used in the context of deterministic finite state machines, FSMs, which can be only in one state at a time. In contrast, non-deterministic finite state machines can be in several states at the same time [212].
 - Languages may have non-deterministic operators. For these operators, different behaviors are legal implementations. Approximate, non-deterministic computations would be a relevant special case of non-deterministic operators.
 - Many authors consider systems to be non-deterministic if their behavior depends on some input not known before run-time.
 - The term “deterministic” has also been used in the sense of “determinate,” as introduced by Kahn.

In this book, we prefer to reduce possible confusion by following Kahn¹¹. Note that StateMate models can be determinate only if there are no other reasons for an undefined behavior. For example, conflicts between transitions may be allowed (see Fig. 2.22).

Fig. 2.22 **Left:** conflict between different nesting levels; **right:** conflict at the same nesting level



Consider Fig. 2.22 (**left**). If event A takes place while the system is in the left state, we must figure out which transition will take place. If these conflicts would be resolved arbitrarily, then we would have a non-determinate behavior. Typically,

¹¹In the first edition of the book, we used the term “deterministic” together with an additional explanation.

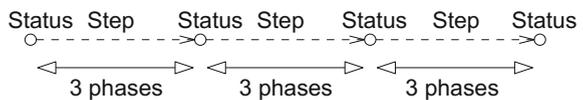
priorities are defined such that this type of a conflict is eliminated. Now, consider Fig. 2.22 (right). There will be a conflict for $x=15$. Such conflicts are difficult to detect. Achieving a determinate behavior requires the absence of conflicts that are resolved in an arbitrary manner.

Note that there may be cases in which we would like to describe non-determinate behavior (e.g., if we have a choice to read from two inputs). In such a case, we would typically like to explicitly indicate that this choice can be taken at run-time (see the **select** statement of Ada on p. 107).

Implementations of hierarchical state charts other than StateMate typically do not exhibit determinate behavior. These implementations correspond to a software-oriented view onto hierarchical state charts. In such implementations, choices are usually not explicitly described.

The three phases described on p. 52 have to be repeatedly executed. Each execution is called a **step** (see Fig. 2.23).

Fig. 2.23 Steps during the execution of a StateMate model



Steps are assumed to be executed each time events or variables have changed. The set of all values of variables, together with the set of events generated (and the current time), is defined as the **status**¹² of a StateMate model. After executing the third phase, a new status is obtained. The notion of steps allows us to define the semantics of **events** more precisely. Events are generated, as mentioned, either internally or externally. **The visibility of events is limited to the step following the one in which they are generated.** Thus, events behave like single bit values which are stored in permanently enabled registers at one clock transition and have an effect on the values stored at the next clock transition. They do not live forever.

Variables, in contrast, retain their values until they are reassigned. According to StateMate semantics, new values of variables are visible to all parts of the model from the step following the step in which the assignment was made onward. That means that StateMate semantics implies that new values of variables are propagated to all parts of a model between two steps. StateMate implicitly assumes a **broad-cast mechanism for updates on variables**. Hence, StateCharts or StateMate can be implemented easily for shared memory-based platforms but is less appropriate for message passing and distributed systems. These languages essentially assume shared memory-based communication, even though this is not explicitly stated. For distributed systems, it will be very difficult to update all variables between two steps. Due to this broadcast mechanism, StateMate is not an appropriate language for modeling distributed systems.

¹²We would normally use the term “state” instead of “status.” However, the term “state” has a different meaning in StateMate.

2.4.2.4 Evaluation and Extensions

StateCharts' main application domain is that of local, control-dominated systems. The capability of nesting hierarchies at arbitrary levels, with a free choice of AND and OR states, is a key advantage of StateCharts. Another advantage is that the semantics of StateMate is defined at a sufficient level of detail [135]. Furthermore, there are quite a number of commercial tools based on StateCharts. StateMate [220] and StateFlow [365] are examples of commercial tools based on StateCharts. Many of them are capable of translating StateCharts into equivalent descriptions in C or VHDL (see p. 94). From VHDL, hardware can be generated using synthesis tools. Therefore, StateCharts-based tools provide a complete path from StateCharts-based specifications down to hardware. Generated C programs can be compiled and executed. Hence, a path to software-based realizations exists as well.

Unfortunately, the efficiency of the automatic translation is sometimes a concern. For example, we could map sub-states of AND states to processes at the operating system level. This would hardly lead to efficient implementations on small processors. The productivity gain from object-oriented programming is not available in StateCharts, since it is not object-oriented. Furthermore, the broadcast mechanism makes it less appropriate for distributed systems. StateCharts do not comprise program constructs for describing complex computation and cannot describe hardware structures or non-functional behavior.

Commercial implementations of StateCharts typically provide some mechanisms for removing the limitations of the model. For example, C code can be used to represent program constructs and **module charts** of StateMate can represent hardware structures.

StateCharts allows timeouts. There is no straightforward way of specifying other timing requirements.

UML includes a variation of StateCharts and hence allows modeling state machines. In UML, these diagrams are called **state diagrams** in version 1 of UML and **state machine diagrams** from version 2.0 onward. Unfortunately, the semantics of state machine diagrams in UML is different from StateMate: The three simulation phases are not included.

2.4.3 Synchronous Languages

2.4.3.1 Motivation

Describing complex SUDs in terms of state machine diagrams is difficult. Such diagrams cannot express complex computations. Standard programming languages can express complex computations, but the sequence of executing several threads may be unpredictable. In a multi-threaded environment with preemptive scheduling, there can be many different interleavings of the different computations. Understanding all possible behaviors of such concurrent systems is difficult. A key reason for this is

that, in general, many different execution orders are feasible, i.e., the execution order is not specified. The order of execution may well affect the result. The resulting non-determinate behavior can have a number of negative consequences, such as problems with verifying a certain design. For distributed systems with independent clocks, determinate behavior is difficult to achieve. However, for non-distributed systems, we can try to avoid the problems of unnecessary non-determinate semantics.

For synchronous languages, finite state machines and programming languages are merged into one model. Synchronous languages can express complex computations, but the underlying execution model is that of finite automata. They describe concurrently operating automata. Determinate behavior is achieved by the following key feature: “... *when automata are composed in parallel, a transition of the product is made of the ‘simultaneous’ transitions of all of them*” [191]. This means that we do not have to consider all the different sequences of state changes of the automata that would be possible if each of them had its own clock. Instead, we can assume the presence of a single global clock. Each clock tick, all inputs are considered, new outputs and states are calculated, and then the transitions are made. This requires a fast broadcast mechanism for all parts of the model. This idealistic view of concurrency has the advantage of guaranteeing **determinate behavior**. This is a restriction if compared to the general communicating finite state machines (CFSM) model, in which each FSM can have its own clock. Synchronous languages reflect the principles of operation in synchronous hardware and also the semantics found in control languages such as IEC 60848 [223] and STEP 7 [463]. See Potop-Butucaru et al. [433] for a survey on synchronous languages.

2.4.3.2 Examples of Synchronous Languages: Esterel, Lustre, and SCADE

Guaranteeing a determinate behavior for all language features has been a design goal for the synchronous languages such as Esterel [63, 148], Lustre [193], and Quartz [455].

Esterel is a reactive language: When activated with an input event, Esterel models react by producing an output event. Esterel is a synchronous language: All reactions are assumed to be completed in zero time and it is sufficient to analyze the behavior at discrete moments in time. This idealized model avoids all discussions about overlapping time ranges and about events that arrive while the previous reaction has not been completed. Like other concurrent languages, Esterel has a parallelism operator, written `||`. Similar to StateCharts, communication is based on a broadcast mechanism. In contrast to StateCharts, however, communication is instantaneous. Instantaneous in this context means “within the same clock cycle.” This means that all signals generated in a particular clock cycle are also seen by the others parts of the model in the same clock cycle and these other parts, if sensitive to the generated signals, react in the same clock cycle. Several rounds of evaluations may be required until a stable state is reached. The computation of resulting worst-case reaction times is performed, for example, by Boldt et al. [59]. The propagation of values during

the same macroscopic instant of time corresponds to the generation of a next status for the same moment in time in StateMate, except that the broadcast is now instantaneous and not delayed until the next round of evaluations like in StateMate. For more and updated information about Esterel, refer to the Esterel home page [148].

Esterel and Lustre use different syntactic techniques to denote CFSMs. Esterel appears as a kind of imperative language, whereas Lustre looks more like a data flow language (see p. 64 for a description of data flow). SyncCharts is a graphical version of Esterel. In all three cases, semantics are explained by the closely related underlying CFSMs. The commercial graphical language SCADE [147] combines elements of all three languages. The so-called SCADE suite[®] is used for a number of safety-critical software components, for example, by Airbus.

Due to the three simulation phases in StateMate, this tool has the key attributes of synchronous languages and it is determinate if conflicts are resolved. According to Halbwachs, “*StateMate is almost a synchronous language and the only feature missing in StateMate is the instantaneous broadcast*” [192].

2.4.4 Message Passing: SDL as an Example

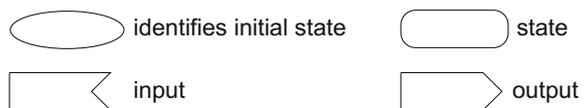
2.4.4.1 Features of the Language

StateCharts is not appropriate for modeling distributed communicating finite state machines. For distributed systems, message passing is the better communication paradigm. Therefore, we present a case of communicating finite state machines with asynchronous message passing.

We use SDL (specification and description language) as an example. SDL was designed for distributed applications. It dates back to the 1970s. Formal semantics have been available since the 1980s. The language was standardized by the ITU (International Telecommunication Union). The first standards document is the Z.100 Recommendation published in 1980, with updates in 1984, 1988, 1992 (SDL-92), 1996, and 1999. Relevant versions of the standard include SDL-88, SDL-92, SDL-2000, and SDL-2010 [444, 457].

Many users prefer graphical specification languages, while others prefer textual ones. SDL pleases both types of users since it provides textual as well as graphical formats. Processes are the basic elements of SDL. Processes represent components modeled as extended finite state machines. Extensions include operations on data. Figure 2.24 shows the graphical symbols used in the graphical representation of SDL.

Fig. 2.24 Symbols used in the graphical form of SDL



Example 2.15: As an example, we will consider how the state diagram in Fig. 2.25 can be represented in SDL. Figure 2.25 is the same as Fig. 2.13, except that output has been added, state Z has been deleted, and the effect of signal k has been changed.

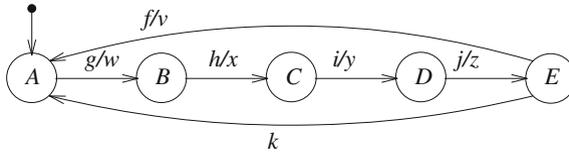


Fig. 2.25 FSM to be described in SDL

Figure 2.26 contains the corresponding graphical SDL representation.

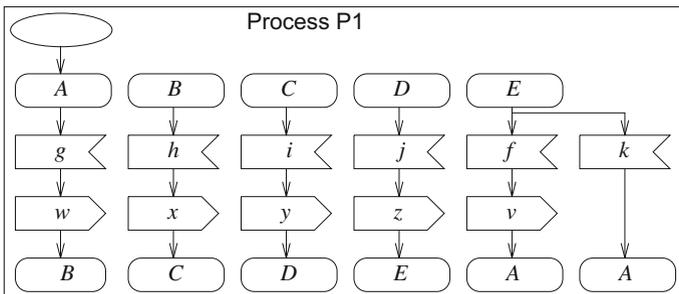
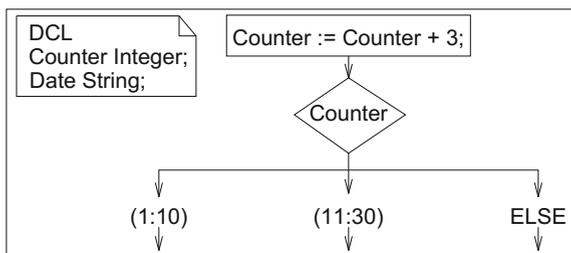


Fig. 2.26 SDL representation of Fig. 2.25

Obviously, the representation in Fig. 2.26 is equivalent to the state diagram of Fig. 2.25. ▽

As an extension to FSMs, SDL processes can perform operations on data. Variables can be declared locally for processes. Their type can either be predefined or defined in the SDL description itself. SDL supports abstract data types (ADTs). The syntax for declarations and operations is similar to that in other languages. Figure 2.27 shows how declarations, assignments, and decisions can be represented in SDL.

Fig. 2.27 Declarations, assignments, and decisions in SDL



SDL also contains programming language elements such as procedures. Procedure calls can also be represented graphically. Object-oriented features became available with version SDL-1992 of the language and were extended with SDL-2000.

Extended FSMs are just the basic elements of SDL descriptions. In general, SDL descriptions will consist of a set of interacting processes, or FSMs. Processes can send signals to other processes. Semantics of inter-process communication in SDL is based on asynchronous message passing and conceptually implemented through *first-in-first-out (FIFO)-queues* associated with processes. There is exactly one input queue per process. Signals sent to a particular process will be placed into the corresponding FIFO queue (see Fig. 2.28).

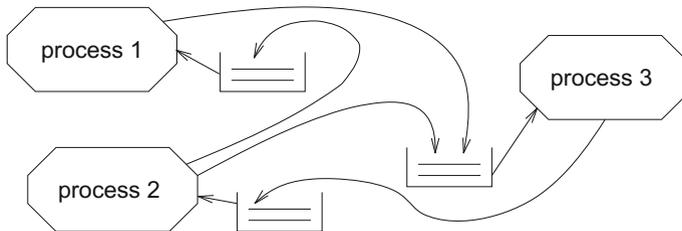


Fig. 2.28 SDL inter-process communication

Each process is assumed to fetch the next available entry from the FIFO queue and check whether it matches one of the inputs described for the current state. If it does, the corresponding state transition takes place and output is generated. The entry from the FIFO queue is ignored if it does not match any of the listed inputs (unless the so-called SAVE mechanism is used). FIFO queues are conceptually thought of as being of infinite length. This means that in the description of the semantics of SDL models, FIFO overflow is never taken into account. In actual systems, however, infinite FIFO queues cannot be implemented. They must be of finite length. This is one of the problems of SDL: In order to derive realizations from specifications, safe upper bounds on the length of the FIFO queues must be proven.

Process interaction diagrams can be used for visualizing which of the processes are communicating with each other. Process interaction diagrams include **channels** used for sending and receiving signals. In the case of SDL, the term “signal” denotes inputs and outputs of modeled automata.

Example 2.16: Figure 2.29 shows a process interaction diagram B1 with channels Sw1 and Sw2. Brackets include the names of signals propagated along a certain channel.

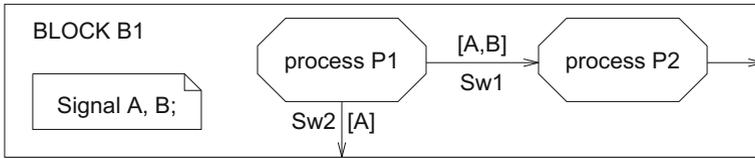


Fig. 2.29 Process interaction diagram



There are three ways of indicating the recipient of signals:

1. **Through process identifiers:** by using identifiers of recipient processes in the graphical output symbol (see Fig. 2.30 (left)).



Fig. 2.30 Left: process name identifies recipient; right: channel identifies recipient

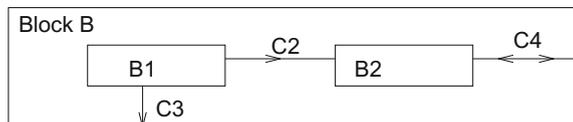
The number of processes does not need to be fixed at compile time, since processes can be generated dynamically at run-time. OFFSPRING represents identifiers of child processes generated dynamically by a process.

2. **Explicitly:** by indicating the channel name (see Fig. 2.30 (right)). Sw1 is the name of a channel.
3. **Implicitly:** If signal names imply the channel names, those channels are used. Example: For Fig. 2.29, signal B will implicitly always be communicated via channel Sw1.

No process can be defined within any other (processes cannot be nested). However, they can be grouped hierarchically into so-called **blocks**. Blocks at the highest hierarchy level are called **systems**. A system will not have any channels at its boundary if the environment is also modeled as a block. Process interaction diagrams are special cases of block diagrams. Process interaction diagrams are one level above the leaves of the hierarchical description.

Example 2.17: Block B1 of Example 2.16 can be used within intermediate-level blocks (such as within B in Fig. 2.31).

Fig. 2.31 SDL block



At the highest level in the hierarchy, we have the system (see Fig. 2.32).

Fig. 2.32 SDL system

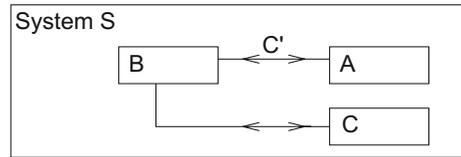
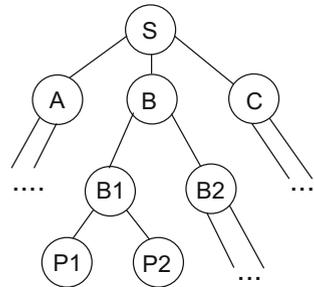


Figure 2.33 shows the hierarchy modeled by block diagrams Figs. 2.29, 2.31, and 2.32.

Fig. 2.33 SDL hierarchy



This example demonstrates that process interaction diagrams are next to the *leaves* of the hierarchical description, while system descriptions represent their *root*. ∇

Some of the restrictions of modeling hierarchy are removed in version SDL-2000 of the language. With SDL-2000, the descriptive power of blocks and processes is harmonized and replaced by a general *agent* concept.

In order to support the modeling of time, SDL includes **timers**. Timers can be declared locally for processes. They can be set using the SET primitive. This primitive has two parameters: an absolute time and a timer name. The absolute time defines a time at which the timer elapses. The built-in function *now* can be used to refer to the time at which the SET primitive is executed. Once a timer is elapsed, a signal is stored in the input queue. The name of this signal is obtained from the second parameter of the SET call. The signal will then typically cause a certain transition to take place in the FSM. However, this transition may be delayed by other entries in the input queue which have to be processed first. Hence, this timer concept is designed for soft timing constraints typically found in telecommunications and inappropriate for hard timing constraints. A second built-in function *expirytime* can be used to avoid some of the limitations of the *now* function.

Timers can be reset using the RESET primitive. This primitive will stop the counting process and—in case the signal has already been stored in the input queue—removes the signal from it. An implicit RESET is executed at the very beginning of executing a SET.

Example 2.18: Figure 2.34 shows the use of a timer T. The diagram corresponds to that of Fig. 2.26, with the exception that timer T is set to the current time now plus p during the transition from state D to E. For the transition from E to A, we now

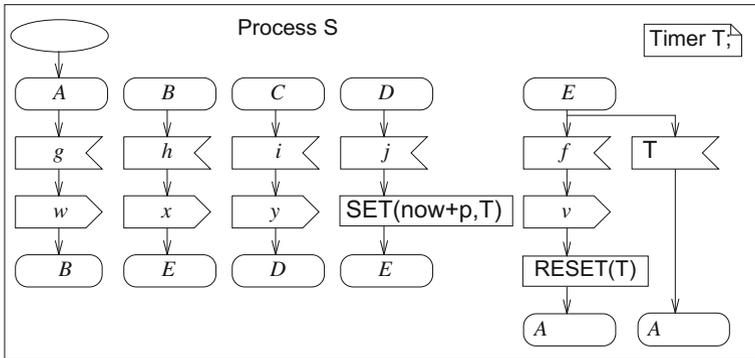


Fig. 2.34 Using timer T

have a timeout of p time units. If these time units have elapsed before signal f has arrived, a transition to state A is taken without generating output signal v. Strictly, periodic processing with a period of p is difficult to achieve this way, due to the possible delays by other entries in the input queue. ▽

Example 2.19: SDL can be used to describe protocol stacks found in computer networks, and SDL is very appropriate for this. Figure 2.35 shows three processors connected through a router. Communication between processors and the router is based on FIFOs. The processors as well as the router implement layered protocols

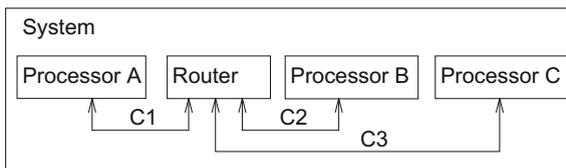


Fig. 2.35 Small computer network described in SDL

(see Fig. 2.36). Each layer describes communication at a more abstract level. The

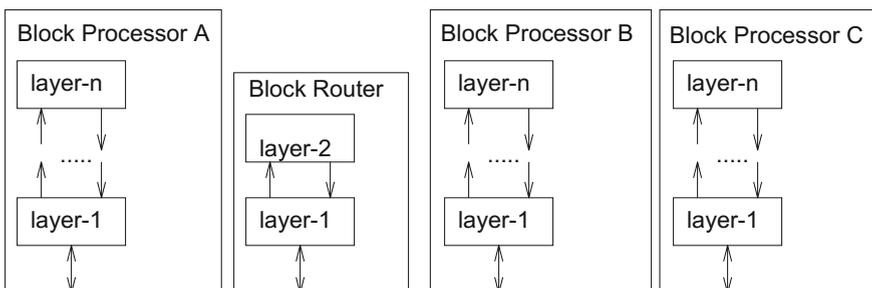


Fig. 2.36 Protocol stacks represented in SDL

behavior of each layer is typically modeled as a finite state machine. The detailed description of these FSMs depends on the network protocol and can be quite complex. Typically, this behavior includes checking and handling of error conditions, as well as sorting and forwarding of information packets. ▽

Available tools for SDL include interfaces to UML (see p. 117), and SCs (see page 40). A comprehensive list of tools is available from the SDL forum [458].

Estelle [76] is another language which was designed to describe communication protocols. Similar to SDL, Estelle assumes communication via channels and FIFO buffers. Attempts to unify Estelle and SDL failed.

2.4.4.2 Evaluation of SDL

SDL is excellent for distributed applications and has been used, for example, for specifying ISDN.

SDL is not necessarily determinate (the order, in which signals arriving at some FIFO at the same time are processed, is not specified).

Reliable implementations require the knowledge of an upper bound on the length of the FIFOs. This upper bound may be difficult to compute. The timer concept is sufficient for soft deadlines, but not for hard ones.

Hierarchies are not supported in the same way as in StateCharts.

There is no full programming support (but revisions of the standard changed this) and no description of non-functional properties.

SDL is very useful as a reference model for asynchronous message passing, but the interest in SDL is decreasing.

2.5 Data Flow

2.5.1 Scope

Data flow is a very “natural” way of describing real-life applications. Data flow models reflect the way in which data flows from component to component [140]. Each component transforms the data in one way or the other. The following is a possible definition of data flow :

Definition 2.11 ([554]): Data flow modeling “*is the process of identifying, modeling, and documenting how data moves around an information system. Data flow modeling examines processes (activities that transform data from one form to another), data stores (the holding areas for data), external entities (what sends data into a system or receives data from a system), data flows (routes by which data can flow)*”.

A **data flow program** is specified by a directed graph where the nodes (vertices), called **actors**, represent computations and the arcs represent communication channels. The computation performed by each actor is assumed to be functional, that is, based on the input values only. Each process in a data flow graph is decomposed into a sequence of firings, which are atomic actions. Each firing produces and consumes tokens.

Example 2.20: Figure 2.37 describes, as an example, the flow of data in a video-on-demand (VOD) system [287]. Viewers are entering the system via the network

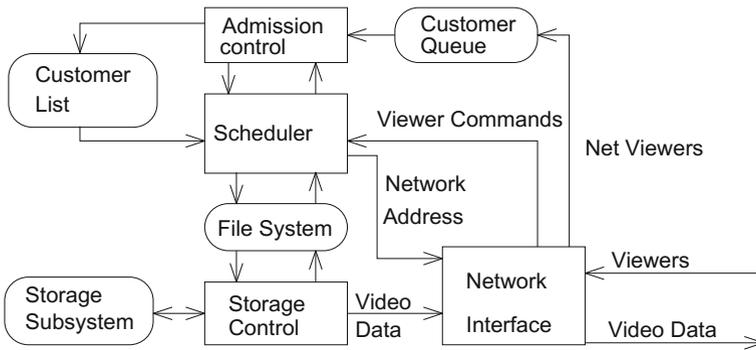


Fig. 2.37 Video-on-demand system

interface. Their admission request is added to the customer queue. Once they are admitted, their requests are scheduled for the file system. The file system, in cooperation with storage control, makes videos available to the customer. ▽

For unrestricted data flow, it is difficult to prove requested system properties. Therefore, restricted models are commonly used.

A special type of data flow is used for implementing out-of-order execution of instructions in computer architectures. This type of execution is also known as *dynamic scheduling* of instructions. Two algorithms for dynamic scheduling are well-known: scoreboarding and the Tomasulo algorithm [520]. Both algorithms are covered in detail in books on computer architecture (see, for example, Hennessy et al. [205]). Therefore, they are not included in this book. However, there are variants of these algorithms which are applied at task level (for example, see Wang et al. [537]).

2.5.2 Kahn Process Networks

Kahn process networks (KPN) [266] are a special case of data flow models. Like other data flow models, KPNs consist of nodes and edges. Nodes correspond to

computations performed by some program or task. KPN graphs, like all data flow graphs, show computations to be performed and their dependencies, but not the order in which the computations must be performed (in contrast to specifications in von-Neumann languages such as C). Edges imply communication via channels containing potentially infinite FIFOs. Computation times and communication times may vary, but communication is guaranteed to happen within a finite amount of time. Writes are non-blocking, since the FIFOs are assumed to be as large as needed. Reads must specify a single channel to be read from. A node cannot check whether data is available before attempting a read. A process cannot wait for data on more than one port at a time. Read operations block whenever an attempt is made to read from an empty FIFO queue. Only a single process is allowed to read from a certain queue, and only a single process is allowed to write into a queue. So, if output data has to be sent to more than a single process, duplication of data must be done inside processes. There is no other way for communication between processes except through FIFO queues.

In the following example, p1 and p2 are incrementing and decrementing the value received from the partner:

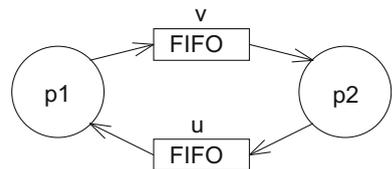
```

process p1(in int u, out int v){
  int i;
  i = 0;
  for (;;) {
    send(i,v);           /* send i via channel v */
    i = wait(u);        /* read i from channel u */
    i = i-1; }
}
process p2(in int v, out int u){
  int i;
  for (;;) {
    i = wait(v);
    i = i+1;
    send(i,u); }
}

```

Figure 2.38 shows a graphical representation of this KPN.

Fig. 2.38 Graphical representation of KPN



Obviously, we do not really need the FIFOs in this example, since messages cannot accumulate in the channels. This example and other examples can be simulated with the levi simulation software [471].

The restrictions are resulting in the **key beauty of KPNs**: The order in which a node is reading data from its channels is fixed by the sequence of read operations and does not depend on the order in which producers are transmitting data over the channels. This means that the sequence of operations is independent of the speed of the nodes producing data. **For a given set of input data, KPNs will always generate the same results, independently of the speed of the nodes.** This property is important, for example, for simulations: It does not matter how fast we are simulating the KPN, the result will always be the same. In particular, the result does not depend on using hardware accelerators for some of the nodes and a distributed execution will give the same result as a centralized one. This property has been called “determinate,” and we are following this use. SDL-like conflicts at FIFOs do not exist. Due to this nice property, KPNs are frequently used as an internal representation within a design flow.

Sometimes, KPNs are extended with a “merge” operator (corresponding to Ada’s **select** statement, see p. 107). This operator allows for queuing read commands containing a list of channels. The operator completes execution after the first of these channels has generated data. Such an operator introduces a non-determinate behavior: The order of processing inputs is not specified if both inputs arrive at the same time. This extension is useful in practice, but it destroys the key beauty of KPNs.

In general, Kahn processes require scheduling at run-time, since it is difficult to predict their precise behavior over time. These problems result from the fact that we do not make any assumptions regarding the speed of the channels and the nodes. Nevertheless, execution times are actually unknown during early design phases, and therefore, this model is very adequate.

KPNs are Turing complete, which means whatever can be computed by a Turing machine (the standard model for computability) can also be computed by a KPN. The proof is based on the fact that KPNs are a superset of so-called Boolean Dataflow (BDF), and according to Buck [75], BDF can simulate Turing machines. However, the number of processes has to be fixed at design time, which is an important limitation for many applications.

The question of whether or not finite-length FIFOs are sufficient for an actual KPN model is undecidable in the general case. However, useful scheduling algorithms [281] or proofs of the boundedness of the FIFOs [100] exist for some special cases. For example, these bounds can be derived for Polyhedral Process Networks (PPNs). For PPNs, the code for each of the nodes includes loops with bounds known at compile time. Derin [120] exploits knowledge about the code of the nodes for dynamic task migration.

2.5.3 *Synchronous Data Flow*

Scheduling becomes significantly easier, and questions regarding buffer sizes can decidably be answered if we impose restrictions on the timing of nodes and channels. Synchronous data flow (SDF) [319] is such a model. SDF can best be introduced

by referring to its graphical notation. SDF models include a directed graph, i.e., SDF models contain nodes and directed edges. Nodes are also called **actors**. Edges can store tokens, by default an unlimited number of them. In general, some of the edges will initially contain some tokens. Each edge has an incoming and an outgoing weight. The execution of an SDF model assumes a clock. For an actor to be enabled, it is necessary that for each of the edges leading to that actor the number of tokens on that edge is at least equal to the outgoing weight for that edge.

Example 2.21: Figure 2.39 (left) shows a synchronous data flow graph. Actor B is enabled since there is a sufficient number of tokens on the edges leading to B. Actor A is not enabled. Input edges like the one shown at the top for actor A are assumed to supply an infinite stream of tokens. Each clock tick, all enabled actors **fire**. As a

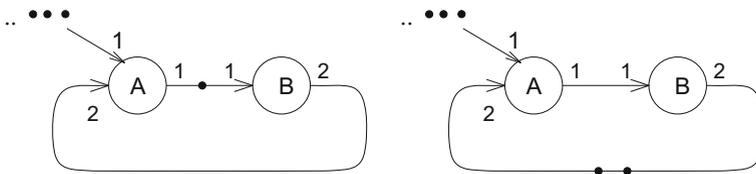


Fig. 2.39 Graphical representation of SDF: **left**: initial situation; **right**: after firing B

result, the number of tokens on the incoming edges get decreased by the incoming weight and the number of tokens on the outgoing edges are increased by the outgoing weight. Obviously, the number of tokens produced or consumed in a particular firing is static (does not vary during the execution of the model). For our example, the resulting number of tokens is shown in Fig. 2.39 on the right. ▽

In practice, tokens will represent data, actors will represent computations, and edges should correspond to FIFO buffers.

Buffers on the edges imply that SDF uses **asynchronous** message passing. Instead of using the default unlimited buffer capacities, we can express limited buffer capacities with backward edges. The initial number of tokens on these backward edges corresponds to the capacity of the FIFO buffer. This is shown in Fig. 2.40. The two models shown in Fig. 2.40 are equivalent.

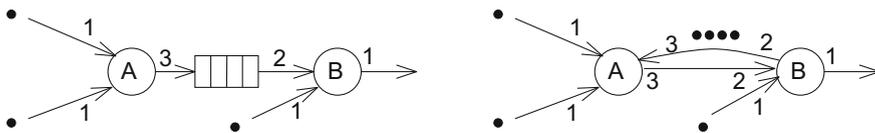


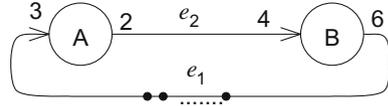
Fig. 2.40 Replacing explicit FIFO buffers by backward edges

For example, the first firing of A will consume three tokens from the backward edge, leaving only one token on the backward edge, corresponding to the one empty FIFO slot after the first firing of A on the left.

The property of producing and consuming a static number of tokens makes it possible to determine execution order and memory requirements at compile time. Hence, complex run-time scheduling of executions is avoided. SDF graphs can be translated into periodic schedules.

Example 2.22: Let us have a closer look at schedules of SDF models. Consider the example shown in Fig. 2.41. Suppose that initially there are six tokens for edge e_1 .

Fig. 2.41 SDF loop



Then, Table 2.2 (left) shows the resulting schedule for firings. Due to the limited number of initial tokens, only sequential firings are feasible.

Table 2.2 Schedules for loop in SDF: **left:** six initial tokens on e_1 , **right:** nine initial tokens on e_1

Clock	Tokens on edges		Next actor action	Clock	Tokens on edges		Next actor action
	e_1	e_2	A or B		e_1	e_2	A or B or (A and B)
0	6	0	A	0	9	0	A
1	3	2	A	1	6	2	A
2	0	4	B	2	3	4	A and B
3	6	0	A	3	6	2	A
4	3	2	A	4	3	4	A and B

Now, let us assume that there are nine initial tokens for edge e_1 . Then, the schedule of Table 2.2 (right) is produced. Under this assumption, A and B fire synchronously. ▽

During the generation of schedules, we could also consider constraints and objectives such as a limited number of available processors [60].

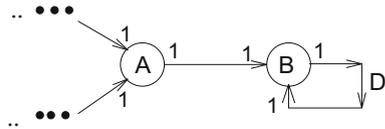
In this example, using edge labels 2, 3, 4, and 6 resulted in different execution rates of actors A and B. In general, edge labels facilitate the modeling of **multi-rate** signal processing applications, applications for which certain signals are generated at frequencies that are multiples of other frequencies. For example, in a TV set, some computations might be performed at a rate of 100Hz while others are performed at a rate of 50Hz. Ignoring some initial transient phase and considering longer periods, the number of tokens sent to an edge must be equal to the number of tokens consumed. Otherwise, tokens would accumulate in the FIFO buffers and no finite FIFO capacity would be sufficient. Let n_s be the number of tokens produced by some sender per firing, and let f_s be the corresponding rate. Let n_r be the corresponding number of tokens consumed per firing at the receiver, and let f_r be the corresponding rate. Then, we must have

$$n_s * f_s = n_r * f_r \tag{2.3}$$

This condition is met in the steady state for the example shown in Table 2.2.

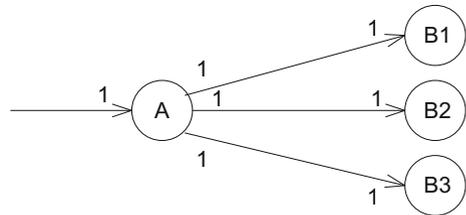
SDF graphs may include delays, denoted by the symbol D on an edge (see Fig. 2.42).

Fig. 2.42 SDF delay



The observer pattern, mentioned as a problem for modeling with von-Neumann languages on p. 31, can be easily implemented correctly in SDF (see Fig. 2.43). There is no risk of deadlocks. However, SDF does not allow adding new observers at run-time.

Fig. 2.43 Observer pattern in SDF



The letter S in SDF initially was meant to stand for the term **synchronous**, since enabled nodes fire synchronously. However, the two schedules in Table 2.2 demonstrate that cases of firing all actors synchronously may indeed be very rare. Therefore, the “S” in SDF has also been reinterpreted to denote the term “static” instead of “synchronous.”

SDF models are determinate, but they are not appropriate for modeling control flow, such as branches. Several extensions and variations of SDF models have been proposed (see, for example, Stuijk [491]):

- For example, we can have **modes** corresponding to states of an associated finite state machine. For each of the modes, a different SDF graph could be relevant. Certain events could then cause transitions between these modes.
- **Homogeneous synchronous data flow (HSDF)** graphs are a special case of SDF graphs. For HSDF graphs, the number of tokens consumed and produced per firing is always 1.
- For **cyclo-static data flow (CSDF)**, the number of tokens produced and consumed per firing can vary over time, but has to be periodic.

Complex SUDs including control flow must be modeled using more general computational graph structures.

2.5.4 Simulink

Computational graph structures are also frequently used in control engineering. For this domain, the Simulink[®] toolbox of MATLAB[®] [506, 510] is very popular. MATLAB is a modeling and simulation tool based on mathematical models including partial differential equations. Figure 2.44 shows an example of a Simulink model [349].

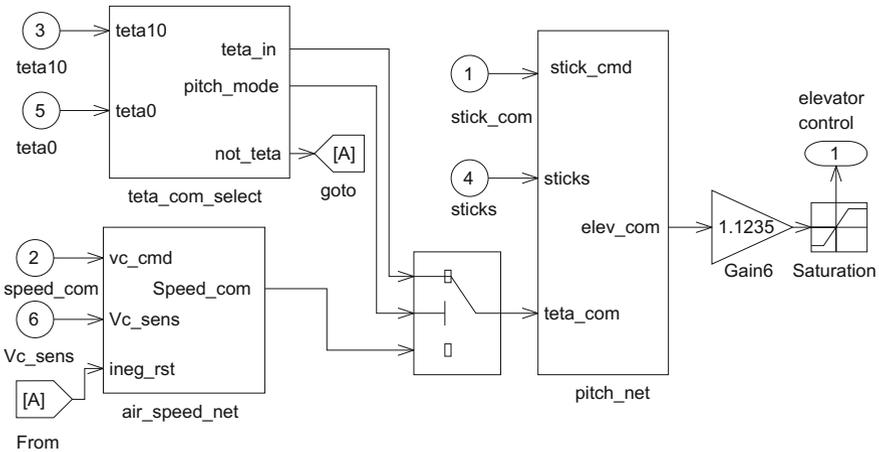


Fig. 2.44 Simulink model

The amplifier and the saturation component on the right demonstrate the inclusion of analog modeling. In the general case, the “schematic” could contain symbols denoting analog components such as integrators, differentiators. The switch in the center indicates that Simulink also allows some control flow modeling.

The graphical representation is intuitive and allows control engineers to focus on the control function, without caring about the code necessary to implement the function. The graphical symbols suggest that analog circuits are used as traditional components in control designs. A key goal is to synthesize software from such models. This approach is typically associated with the term **model-based design**.

Semantics of Simulink models reflect the simulation on a digital computer, and the behavior may be similar to that of analog circuits, but possibly not quite the same. What is actually the semantics of a Simulink model? Marian and Ma [349] describe the semantics as follows: “*Simulink uses an idealized timing model for block (node) execution and communication. Both happen infinitely fast at exact points in simulated time. Thereafter, simulated time is advanced by exact time steps. All values on edges are constant in between time steps.*” This means that we execute the model time step after time step. For each step, we compute the function of the nodes (in zero time) and propagate the new values to connected inputs. This explanation does

not specify the distance between time steps. Also, it does not immediately tell us how to implement the system in software, since even slowly varying outputs may be recomputed frequently.

This approach is appropriate for modeling physical systems such as cars or trains at a high level and then simulating the behavior of these systems. Also, digital signal processing systems can be conveniently modeled with MATLAB[®] and Simulink[®]. In order to generate implementations, MATLAB/Simulink models first must be translated into a language supported by software or hardware design systems, such as C or VHDL.

Components in Simulink models provide a special case of **actors**. We can assume that actors are waiting for input and perform their operation once all required inputs have arrived. SDF is another case of actor-based languages. In **actor-based languages**, there is no need to pass control to these actors, like in von-Neumann languages.

2.6 Petri Nets

2.6.1 Introduction

Very comprehensive descriptions of control flow are feasible with computational graphs known as Petri nets. Actually, Petri nets model **only** control and control dependencies. Modeling data as well requires extensions of Petri nets. Petri nets focus on the modeling of causal dependencies.

In 1962, Carl Adam Petri published his method for modeling causal dependencies, which became known as Petri nets [427]. Petri nets do not assume any global synchronization and are therefore especially suited for modeling distributed systems.

Conditions, events, and a flow relation are the key elements of Petri nets. Conditions are either satisfied or not satisfied. Events can happen. The flow relation describes the conditions that must be met before events can happen and it also describes the conditions that become true if events happen. Graphical notations for Petri nets typically use circles to denote conditions and boxes to denote events. Arrows represent flow relations.

Example 2.23: Figure 2.45 shows a first example. This example describes mutual exclusion for trains on a railroad track that must be used in both directions. A token is used to prevent collisions of trains going into opposite directions. In the Petri net, that token is symbolized by a condition in the center of the model. A partially filled circle (a circle containing a second, filled circle) denotes that a condition is met (this means that the track is available). When a train wants to travel to the right (also denoted by a partially filled circle in Fig. 2.45), the two conditions that are necessary for the event “train entering track from the left” are met. We call these two conditions **preconditions**. If the preconditions of an event are met, it can happen.

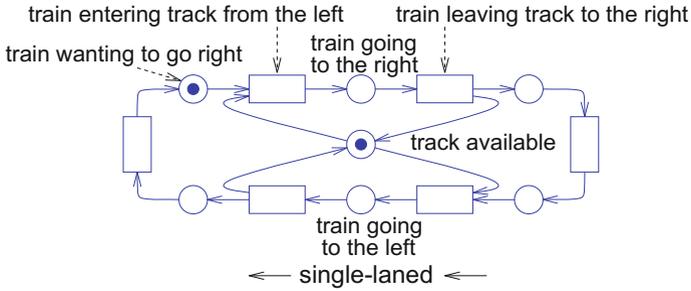


Fig. 2.45 Single-track railroad segment

As a result of that event happening, the token is no longer available and there is no train waiting to enter the track. Hence, the preconditions are no longer met and the partially filled circles disappear (see Fig. 2.46).

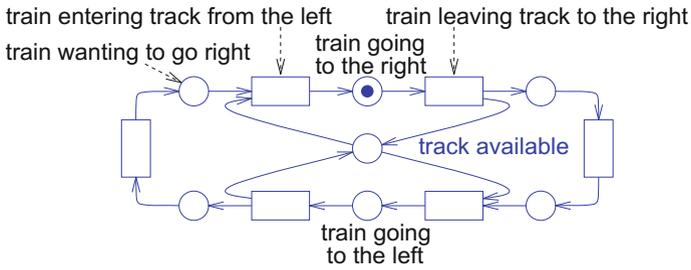


Fig. 2.46 Using resource “track”

However, there is now a train going on that track from the left to the right, and thus, the corresponding condition is met (see Fig. 2.46). A condition which is met after an event happened is called a **postcondition**. In general, an event can happen only if all its preconditions are true (or met). If it happens, the preconditions are no longer met and the postconditions become valid. Arrows identify those conditions which are preconditions of an event and those that are postconditions of an event. Continuing with our example, we see that a train leaving the track will return the token to the condition at the center of the model (see Fig. 2.47).

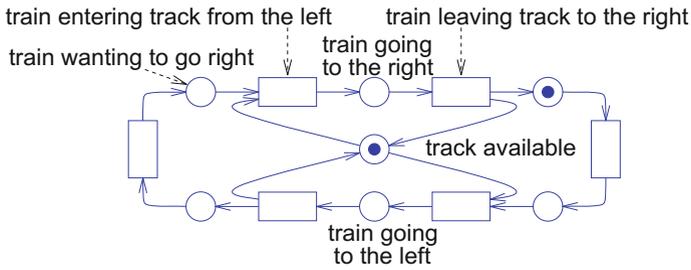


Fig. 2.47 Freeing resource “track”

If there are two trains competing for the single-track segment (see Fig. 2.48), only one of them can enter. In such situations, the next transition to be fired is non-deterministically chosen.

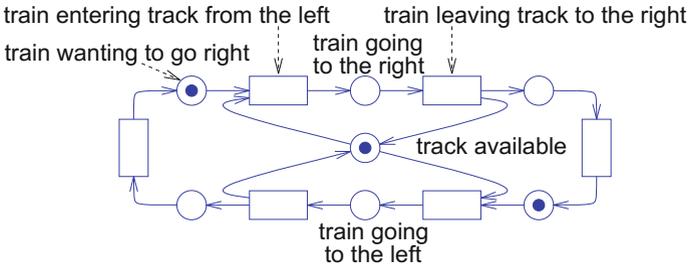


Fig. 2.48 Conflict for resource “track”

∇

Analyses of the net must consider all possible firing sequences. For Petri nets, we are intentionally modeling non-determinism.

A key advantage of Petri nets is that they can be the basis for formal proofs about system properties and that there are standardized ways of generating such proofs. In order to enable such proofs, we need a more formal definition of Petri nets. We will consider three classes of Petri nets: condition/event nets, place/transitions nets, and predicate transition nets.

2.6.2 Condition/Event Nets

Condition/event nets are the first class of Petri nets that we will define more formally.

Definition 2.12: $N = (C, E, F)$ is called a **net** iff the following holds:

1. C and E are disjoint sets.
2. $F \subseteq (E \times C) \cup (C \times E)$ is a binary relation, called flow relation.

The set C is called conditions and the set E is called events.

Definition 2.13: Let N be a net and let $x \in (C \cup E)$. $\bullet x := \{y | yFx, y \in (C \cup E)\}$ is called the **preset** of x . If x denotes an event, $\bullet x$ is also called the set of **preconditions** of x .

Definition 2.14: Let N be a net and let $x \in (C \cup E)$. $x^\bullet := \{y | xFy, y \in (C \cup E)\}$ is called the **postset** of x . If x denotes an event, x^\bullet is also called the set of **postconditions** of x .

The terms preconditions and postconditions are preferred if these sets actually denote conditions $\in C$, that is, if $x \in E$.

Definition 2.15: Let $(c, e) \in C \times E$. (c, e) is called a **loop** if $cFe \wedge eFc$.

Definition 2.16: Let $(c, e) \in C \times E$. N is called **pure** if F does not contain any loops (see Fig. 2.49, left).

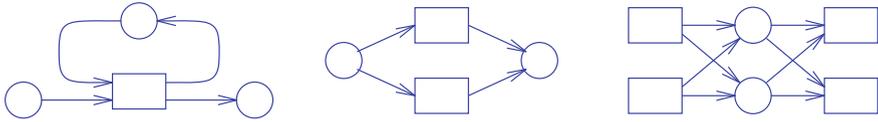


Fig. 2.49 Nets which are not pure (left) and not simple (center and right)

Definition 2.17: A net is called **simple** if no two transitions t_1 and t_2 have the same set of pre- and postconditions (see Fig. 2.49 (center and right)).

Simple nets with no isolated elements meeting some additional restrictions are called **condition/event nets**. Condition/event nets are a special case of bipartite graphs (graphs with two disjoint sets of nodes). We will not discuss those additional restrictions in detail since we will consider more general classes of nets in the following.

2.6.3 Place/Transition Nets

For condition/event nets, there is at most one token per condition. For many applications, it is useful to remove this restriction and to allow more tokens per condition. Nets allowing more than one token per condition are called place/transition nets. Places correspond to what we so far called conditions and transitions correspond to what we so far called events. The number of tokens per place is called a **marking**. Mathematically, a marking is a mapping from the set of places to the set of natural numbers extended by a special symbol ω denoting infinity.

Let \mathbb{N}_0 denote the natural numbers including 0. Then, formally speaking, place/transition nets can be defined as follows:

Definition 2.18: (P, T, F, K, W, M_0) is called a place/transition net \iff

1. $N = (P, T, F)$ is a net with places $p \in P$, transitions $t \in T$, and flow relation F .
2. Mapping $K : P \rightarrow (\mathbb{N}_0 \cup \{\omega\}) \setminus \{0\}$ denotes the capacity of places (ω symbolizes infinite capacity).
3. Mapping $W : F \rightarrow (\mathbb{N}_0 \setminus \{0\})$ denotes the weight of graph edges.
4. Mapping $M_0 : P \rightarrow \mathbb{N}_0 \cup \{\omega\}$ represents the initial marking of places.

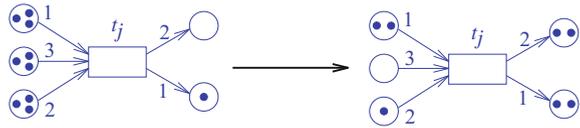
Edge weights affect the number of tokens that are required before transitions can happen and also identify the number of tokens that are generated if a certain transition takes place. Let $M(p)$ denote a current marking of place $p \in P$, and let $M'(p)$ denote a marking after some transition $t \in T$ took place. The weight of edges belonging to preconditions represents the number of tokens that are removed from places in the preset. Accordingly, the weight of edges belonging to the postconditions represents

the number of tokens that are added to the places in the postset. Formally, marking M' is computed as follows:

$$M'(p) = \begin{cases} M(p) - W(p, t), & \text{if } p \in \bullet t \setminus t^\bullet \\ M(p) + W(t, p), & \text{if } p \in t^\bullet \setminus \bullet t \\ M(p) - W(p, t) + W(t, p), & \text{if } p \in \bullet t \cap t^\bullet \\ M(p) & \text{otherwise} \end{cases}$$

Figure 2.50 demonstrates how transition t_j affects the current marking.

Fig. 2.50 Generation of a new marking



By default, unlabeled edges are considered to have a weight of 1 and unlabeled places are considered to have unlimited capacity ω .

We now need to explain the two conditions that must be met before a transition $t \in T$ can take place:

- for all places p in the preset, the number of tokens must at least be equal to the weight of the edge from p to t and
- for all places p in the postset, the capacity must be large enough to accommodate the new tokens which t will generate.

Transitions meeting these two conditions are called **M-activated**. Formally, this can be defined as follows:

Definition 2.19: Transition $t \in T$ is said to be M-activated \iff

$$(\forall p \in \bullet t : M(p) \geq W(p, t)) \wedge (\forall p' \in t^\bullet : M(p') + W(t, p') \leq K(p'))$$

Activated transitions can happen, but they do not need to. If several transitions are activated, the sequence in which they happen is not deterministically defined.

The impact of a firing transition t on the number of tokens can be represented conveniently by a vector \underline{t} associated with t . \underline{t} is defined as follows:

$$\underline{t}(p) = \begin{cases} -W(p, t), & \text{if } p \in \bullet t \setminus t^\bullet \\ +W(t, p), & \text{if } p \in t^\bullet \setminus \bullet t \\ -W(p, t) + W(t, p), & \text{if } p \in \bullet t \cap t^\bullet \\ 0 & \text{otherwise} \end{cases}$$

The new number M' of tokens, resulting from the firing of transition t , can be computed for all places p as follows:

$$M'(p) = M(p) + \underline{t}(p)$$

Using “+” to denote vector addition, we can rewrite this equation as follows:

$$M' = M + \underline{t}$$

The set of all vectors \underline{t} form an incidence matrix \underline{N} . \underline{N} contains vectors \underline{t} as columns.

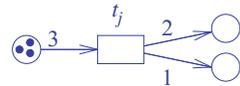
$$\underline{N} : P \times T \rightarrow \mathbb{Z}; \quad \forall t \in T : \underline{N}(p, t) = \underline{t}(p)$$

It is possible to formally prove system properties by using matrix \underline{N} . For example, we are able to compute sets of places, for which firing transitions will not change the overall number of tokens [445]. Such sets are called **place invariants**. Let us initially consider a single transition t_j in order to find such invariants. Let us search for sets $R \subseteq P$ of places such that the total number of tokens does not change if t_j fires. The following must hold for such sets:

$$\sum_{p \in R} \underline{t}_j(p) = 0 \tag{2.4}$$

Figure 2.51 shows a transition for which the total number of tokens does not change if it fires.

Fig. 2.51 Transition with a constant number of tokens



We are now introducing the characteristic vector \underline{c}_R of some set R of places:

$$\underline{c}_R(p) = \begin{cases} 1 & \text{iff } p \in R \\ 0 & \text{iff } p \notin R \end{cases}$$

With this definition, we can rewrite Eq. (2.4) as

$$\sum_{p \in R} \underline{t}_j(p) = \sum_{p \in P} \underline{t}_j(p) \cdot \underline{c}_R(p) = \underline{t}_j \cdot \underline{c}_R = 0. \tag{2.5}$$

which denotes the scalar product. Now, we search for sets of places such that firings of **any** transition will not change the total number of tokens. This means that Eq. (2.5) must hold for all transitions t_j :

$$\begin{aligned} \underline{t}_1 \cdot \underline{c}_R &= 0 \\ \underline{t}_2 \cdot \underline{c}_R &= 0 \\ &\dots \\ \underline{t}_n \cdot \underline{c}_R &= 0 \end{aligned} \tag{2.6}$$

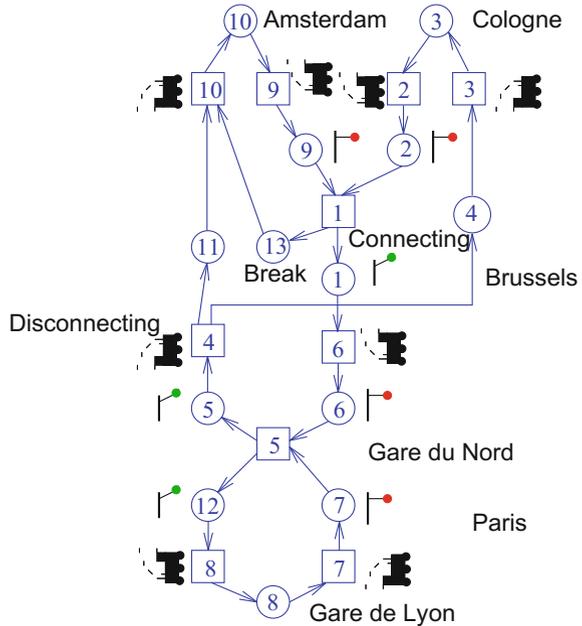
Equation (2.6) can be combined into the following equation by using the transposed incidence matrix \underline{N}^T :

$$\underline{N}^T \cdot \underline{c}_R = 0 \tag{2.7}$$

Equation (2.7) represents a system of linear, homogeneous equations. Matrix \underline{N} represents edge weights of our Petri nets. We are looking for solution vectors \underline{c}_R for this system of equations. Solutions must be characteristic vectors. Therefore, their components must be 1 or 0 (integer weights can be accepted if we use weighted sums of tokens). This is more complex than solving systems of linear equations with real-valued solution vectors. Nevertheless, it is possible to obtain information by solving equation (2.7). Using this proof technique, we can, for example, show that we are correctly implementing mutually exclusive access to shared resources.

Example 2.24: Let us now consider a larger example: We are again considering the synchronization of trains. In particular, we are trying to model high-speed Thalys trains traveling between Amsterdam, Cologne, Brussels, and Paris. Segments of the train run independently from Amsterdam and Cologne to Brussels. There, the segments get connected and then they run to Paris. On the way back from Paris, they get disconnected at Brussels again. We assume that Thalys trains must synchronize with some other train at Paris. The corresponding Petri net is shown in Fig. 2.52.

Fig. 2.52 Model of Thalys trains running between Amsterdam, Cologne, Brussels, and Paris



Places 3 and 10 model trains waiting at Cologne and Amsterdam, respectively. Transitions 2 and 9 model trains driving from these cities to Brussels. After their arrival at Brussels, places 2 and 9 contain tokens. Transition 1 denotes connecting the two trains. The cup symbolizes the driver of one of the trains, who will have a break at Brussels while the other driver is continuing on to Paris. Transition 5 models synchronization with other trains at the Gare du Nord station of Paris. These other trains connect Gare du Nord with some other stations (we have used Gare de Lyon as an example, even though the situation at Paris is somewhat more complex). Of course, Thalys trains do not use steam engines; they are just easier to visualize than modern high-speed trains. Table 2.3 shows matrix N^T for this example.

Table 2.3 N^T for the Thalys example

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}
t_1	1	-1							-1				1
t_2		1	-1										
t_3			1	-1									
t_4				1	-1						1		
t_5					1	-1	-1					1	
t_6	-1					1							
t_7							1	-1					
t_8								1				-1	
t_9									1	-1			
t_{10}										1	-1		-1

For example, row 2 indicates that firing t_2 will increase the number of tokens on p_2 by 1 and decrease the number of tokens on p_3 by 1. Using techniques from linear algebra, we are able to show that the following four vectors are solutions for this system of linear equations:

$$\begin{aligned}
 c_{R,1} &= (1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0) \\
 c_{R,2} &= (1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0) \\
 c_{R,3} &= (0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1) \\
 c_{R,4} &= (0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0)
 \end{aligned}$$

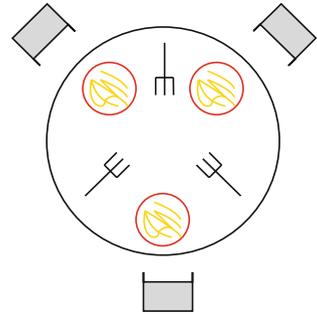
These vectors correspond to the places along the track for trains from Cologne, to the places along the track for trains from Amsterdam, to the places along the path for drivers of trains from Amsterdam, and to the places along the track within Paris, respectively. Therefore, we are able to show that the number of trains and drivers along these tracks is constant (something which we actually expect). This example demonstrates that place invariants provide us with a standardized technique for proving properties about systems. ∇

2.6.4 Predicate/Transition Nets

Condition/event nets as well as place/transition nets can quickly become very large for large examples. A reduction of the size of the nets is frequently possible with predicate/transition nets.

Example 2.25: We will demonstrate this, using the so-called dining philosophers problem as an example. The problem is based on the assumption that a set of philosophers is dining at a round table. In front of each philosopher, there is a plate containing spaghetti (see Fig. 2.53). Between each of the plates, there is just one

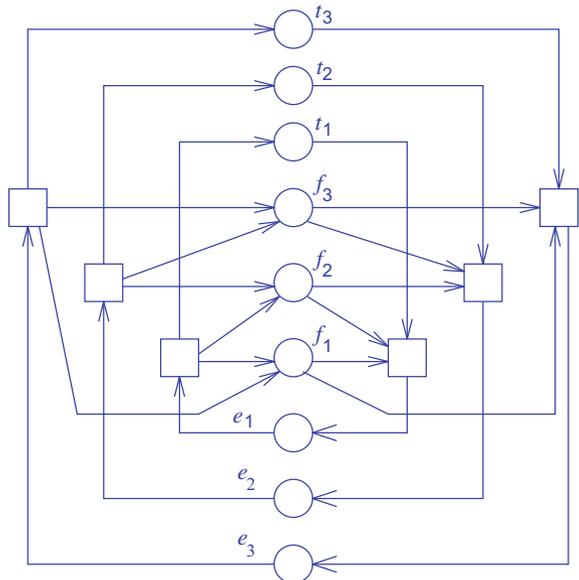
Fig. 2.53 The dining philosophers problem



fork. Each philosopher is either eating or thinking. Eating philosophers need their two adjacent forks for that, so they can only eat if their neighbors are not eating.

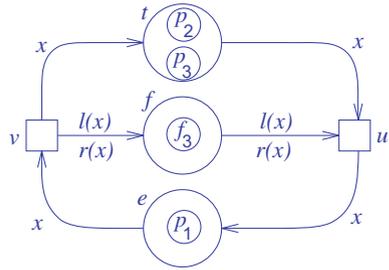
This situation can be modeled as a condition/event net, as shown in Fig. 2.54.

Fig. 2.54 Place/transition net model of the dining philosophers problem



Conditions t_j correspond to the thinking states, conditions e_j correspond to the eating states, and conditions f_j represent available forks. Considering the small size of the problem, this net is already very large. The size of this net can be reduced by using predicate/transition nets. Figure 2.55 is a model of the same problem as a predicate/transition net. With predicate/transition nets, tokens have an identity

Fig. 2.55 Predicate/transition net model of the dining philosophers problem



and can be distinguished from each other. Predicate/transition nets have also been called **Colored Petri Nets (CPN)**. See Jensen [260] for a survey of applications of CPNs for modeling of IT systems, including communication protocols. We use this in Fig. 2.55 in order to distinguish between the three different philosophers p_1 to p_3 and to identify fork f_3 . Furthermore, edges can be labeled with variables and functions. In the example, we use variables to represent the identity of philosophers and functions $l(x)$ and $r(x)$ to denote the left and right forks of philosopher x , respectively. These two forks are required as a precondition for transition u and returned as a postcondition by transition v . This model can be easily extended to $n > 3$ philosophers. We just need to add more tokens. In contrast to the net in Fig. 2.54, the structure of the net does not have to be changed. ∇

2.6.5 Evaluation

The key advantage of Petri nets is their power for modeling causal dependencies. Standard Petri nets have no notion of time, and all decisions can be taken locally by just analyzing transitions and their pre- and postconditions. Therefore, they can be used for modeling geographically distributed systems. Furthermore, there is a strong theoretical foundation for Petri nets, simplifying formal proofs of system properties. Petri nets are not necessarily determinate: Different firing sequences can lead to different results. The descriptive power of Petri nets encompasses that of other MoCs, including finite state machines.

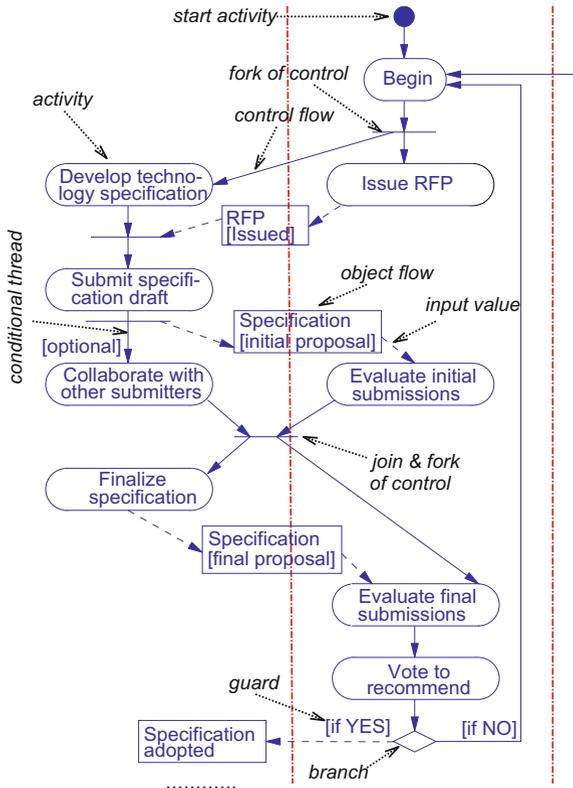
In certain contexts, their strength is also their weakness. If time is to be modeled, standard Petri nets cannot be used. Furthermore, standard Petri nets have no notion of hierarchy and no programming language elements, let alone object-oriented features. In general, it is difficult to represent data.

There are extended versions of Petri nets avoiding the mentioned weaknesses. However, there is no universal extended version of Petri nets meeting all requirements mentioned at the beginning of this chapter. Nevertheless, due to the increasing amount of distributed computing, Petri nets became more popular.

UML includes extended Petri nets called **activity diagrams**. Extensions include symbols denoting decisions (like in ordinary flow charts). The placement of symbols is similar to SDL.

Example 2.26: Figure 2.56 shows an activity chart of the procedure to be followed during a standardization process. Forks and joins of control correspond to transi-

Fig. 2.56 Activity diagram [288]



tions in Petri nets, and they use the symbols (horizontal bars) that were initially used for Petri nets as well. The diamond at the bottom shows the symbol used for decisions. Activities can be organized into “swim lanes” (areas between vertical dotted lines) such that the different responsibilities and the documents exchanged can be visualized. ▽

Interestingly, Petri nets were initially not a mainstream technique. Decades after their invention, they have become a popular technique due to their inclusion in UML.

2.7 Discrete Event-Based Languages

2.7.1 Basic Discrete Event Simulation Cycle

The discrete event-based model of computation is based on simulating the generation of events and processing them over time. We use a queue of future events, and these are sorted by the time at which they should be processed. Semantics is defined by fetching the event at the head of the queue, performing the corresponding actions, and possibly entering new events into the queue. Time is advanced whenever no action exists which should be performed at the current time. This is the basic algorithm:

```

loop
  fetch next entry from queue;
  perform function (e.g., assignment of variables as listed in the entry)
  (this may include the generation of new events);
until termination criterion is met;
    
```

Hardware description languages (HDLs) are typically based on the discrete event model. We will use HDLs as a prominent example of discrete event modeling.

Example 2.27: We demonstrate the application of this general scheme to simulate an RS latch (see Fig. 2.57). The latch consists of two cross-coupled NOR gates.

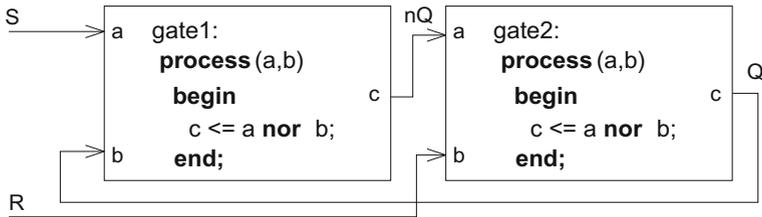


Fig. 2.57 Two cross-connected NOR gates forming an RS latch

The corresponding code in a hardware description language, in this case VHDL, is included in Fig. 2.57 as well. A representative sequence of values at the inputs and outputs is shown in Table 2.4.

Table 2.4 Sequence of values at inputs and outputs of RS latch

	$t < 0$	$t = 0$	$t > 0$		
R	0	1	1	1	1
S	0	0	0	0	0
Q	1	1	0	0	0
nQ	0	0	0	1	1

Let us assume that initially, the latch is set and this state is maintained, i.e., output Q is 1 and $R = S = 0$. The operation of both NOR gates is described by processes `gate1` and `gate2`. These processes are initially inactive, waiting for some event on their inputs a or b . This waiting is expressed by the lists (a, b) . `gate1` and `gate2` are said to be **sensitive** to the entries in that list.

Now, suppose that at time 0 , input R , the reset input, is changed to 1. We expect the latch to be reset. In terms of events, this works as follows: The change at input R is an event, which is stored in the queue of future events.

This event is immediately processed, since it is the only event in the queue. This event will wake up `gate2`, since this gate is sensitive to changes on its input b . `gate2` will compute the NOR function, with a result of 0 , and will then execute the assignment `c <= expression`. This notation indicates a **signal assignment**. This means that the new values will initially be stored only in the entries of future events. The actual assignment to the variable on the left becomes effective only when the time for processing this entry in the list of future events has been reached. In our example, an event requesting output c of `gate2` to be set to 0 will be created and stored in the event queue.

This event will be immediately fetched, since it is the only event. The event will set output c to 0 . This wakes up `gate1`, due to its sensitivity. `gate1` will compute the NOR function as well. This computation results in an event, requesting output c of `gate1` to be set to 1. This event will also be stored in the queue.

This event will also be immediately processed, setting the output as requested. This change will wake up `gate2` again. `gate 2` will again compute an output of 0 . Further details will depend somewhat on the mechanism which is used to detect stable situations not requiring further events to be generated.

We could have added delays in terms of real physical units to each of the signal assignments, which would have allowed us to keep track of elapsed time. Overall, this event-based simulation approximates the behavior of a real latch. ∇

2.7.2 Multi-valued Logic

Which values could we use for the signals in the above example? In this book, we are restricting ourselves to embedded systems implemented with binary logic. Nevertheless, it may be advisable or necessary to use more than two values for modeling such systems. For example, our systems might contain electrical signals of different strengths. It may be necessary to compute the strength and the logic level resulting from a connection of two or more sources of electrical signals. In the following, we will therefore distinguish between the **level** and the **strength** of a **signal**. While the former is an abstraction of the signal voltage, the latter is an abstraction of the impedance (resistance) of the voltage source. We will be using discrete sets of signal values representing the signal level and the strength. **Using discrete sets of strengths avoids the problems of having to solve Kirchhoff's network equations and enables us to avoid analog models used in electrical engineering.** We will also model unknown electrical signals by special signal values.

In practice, electronic design systems use a variety of value sets. Some systems allow only two, while others allow 9 or 46. The overall goal of developing discrete value sets is to avoid the problems of solving network equations and still model existing systems with sufficient precision.

In the following, we will present a systematic technique for building up value sets and relating these to each other. We will use the strength of electrical signals as the key parameter for distinguishing between various value sets. A systematic way of building up value sets, called CSA theory, was presented by Hayes [200]. CSA stands for “connector, switch, attenuator.” These three elements are key elements of this theory. We will later show how the standard value set used for most cases of VHDL-based modeling can be derived as a special case.

One Signal Strength (Two Logic Values)

In the simplest case, we will start with just two logic values, called '0' and '1'. These two values are considered to be of the same strength. This means that if two wires **connect** values '0' and '1', we will not know anything about the resulting signal level.

A single signal strength may be sufficient if no two wires carrying values '0' and '1' are connected and no signals of different strength meet at a particular node of electronic circuits.

Two Signal Strengths (Three and Four Logic Values)

In many circuits, there may be instances in which a certain electrical signal is not actively driven by any output. This may be the case, when a certain wire is not connected to ground, the supply voltage, or any circuit node.

For example, systems may contain open collector outputs (see Fig. 2.58, (left))¹³.

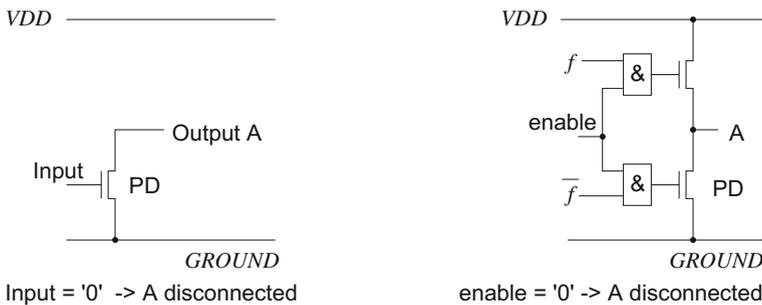


Fig. 2.58 Effectively disconnectable outputs: **left**: open collector output; **right**: tristate output

If the “pull-down” transistor PD is non-conducting, the output is effectively disconnected. For the tristate outputs (see Fig. 2.58, (right)), an enable signal of '0' will

¹³Schematics should **help** students to understand signal values, not make it more difficult. Students unfamiliar with schematics could just study logic values.

generate a '0' at the outputs of the AND gates (denoted by &), and will make both transistors non-conducting. As a result, output A will be disconnected¹⁴. Hence, using appropriate input signals, such outputs can be effectively disconnected from a wire.

The signal strength of disconnected outputs is the smallest strength that we can think of. We will denote the value at disconnected outputs as 'Z'. The signal strength of 'Z' is smaller than that of '0' and '1'. Furthermore, the signal level of 'Z' is unknown. If a signal of value 'Z' is connected to another signal, that other signal will always dominate. For example, if two tristate outputs are connected to the same bus and if one output contributes a value of 'Z', the resulting value on the bus will always be the value contributed by the second output (see Fig. 2.59).

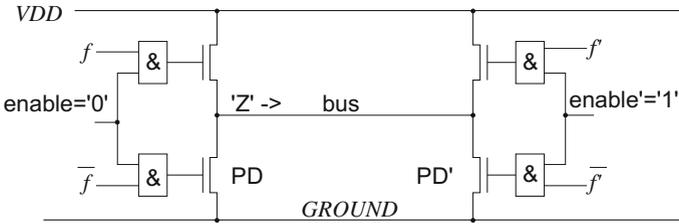
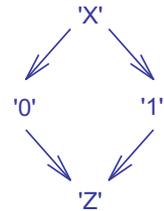


Fig. 2.59 Right output dominates bus

In most cases, three-valued logic sets {'0','1','Z'} are extended by a fourth value called 'X'. 'X' represents an unknown signal level of the same strength as '0' or '1'. More precisely, we are using 'X' to represent unknown values of signals that can be either '0' or '1' or some voltage representing neither '0' nor '1'¹⁵.

If multiple signals get connected, we have to compute the resulting value. This can be done easily if we make use of a partial order among the four signal values '0', '1', 'Z', and 'X'. The partial order is depicted in the **Hasse diagram** in Fig. 2.60.

Fig. 2.60 Partial order for value set {'0','1','Z','X'}



¹⁴Pull-up transistors may be depletion transistors, and the tristate outputs may be inverting.

¹⁵There are other interpretations of 'X' [67], but ours is the most useful in our context.

Edges in this figure reflect the domination of signal values. Edges define a relation $>$. If $a > b$, then a dominates b . '0' and '1' dominate 'Z'. 'X' dominates all other signal values. Based on the relation $>$, we define a relation \geq . $a \geq b$ holds iff $a > b$ or $a = b$.

We define an operation *sup* on two signals, which returns the **supremum** of the two signal values.

Definition 2.20: Let a and b be two signal values from a partially ordered set (S, \geq) . The **supremum** $c \in S$ of the two values a and b is the smallest value for which $c \geq a$ and $c \geq b$ hold.

For example, $sup('Z', '0') = '0'$ and $sup('Z', '1') = '1'$.

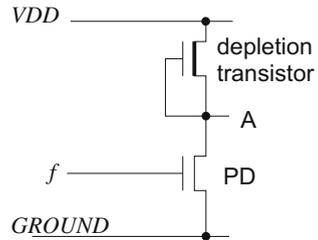
Lemma 2.1: Let a and b be two signals having values from a partially ordered set, where the partial order has been selected as shown above. Then, **the *sup* function computes resulting signal values if the two signals get connected.**

The supremum corresponds to the **connect** element of the CSA theory.

Three Signal Strengths (Seven Signal Values)

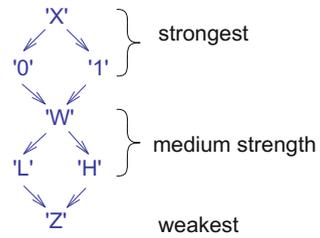
In many circuits, two signal strengths are not sufficient. A common case that requires more values is the use of depletion transistors (see Fig. 2.61).

Fig. 2.61 Output using depletion transistor



The effect of the depletion transistor is similar to that of a resistor providing a low conductance path to the supply voltage *VDD*. The depletion transistor as well as the “pull-down transistor” PD acts as drivers for node A of the circuit, and the signal value at node A can be computed using the supremum function. The pull-down transistor provides a driver value of '0' or 'Z', depending upon the input to PD. The depletion transistor provides a signal value, which is weaker than '0' and '1'. Its signal level corresponds to the signal level of '1'. We represent the value contributed by the depletion transistor by 'H', and we call it a “weak logic one.” Similarly, there can be weak logic zeros, represented by 'L'. The value resulting from the possible connection between 'H' and 'L' is called a “weak logic undefined,” denoted as 'W'. As a result, we have three signal strengths and seven logic values { '0'; '1'; 'L'; 'H'; 'W'; 'X'; 'Z' }. Computing the resulting signal value can again be based on a partial order among these seven values. The corresponding partial order is shown in Fig. 2.62.

Fig. 2.62 Partial order for value set $\{ '0', '1', 'L', 'H', 'W', 'X', 'Z' \}$



sup is also defined for this partially ordered set. For example, $sup('H','0') = '0'$, $sup('H','Z') = 'H'$ and $sup('H','L') = 'W'$.

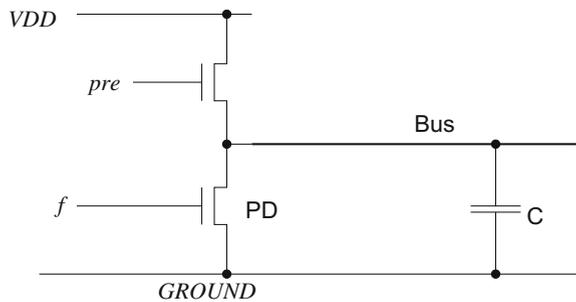
'0' and 'L' represent the same signal levels, but a different strength. The same holds for the pairs '1' and 'H'. Devices increasing signal strengths are called **amplifiers**, devices reducing signal strengths are called **attenuators**.

Ten Signal Values (Four Signal Strengths)

In some cases, three signal strengths are not sufficient. For example, there are circuits using charges stored on wires. Such wires are charged to levels corresponding to '0' or '1' during some phases of the operation of the electronic circuit. This stored charge can control the (high impedance) inputs of some transistors. However, if these wires get connected to even the weakest signal source (except 'Z'), they lose their charge and the signal value from that source dominates.

Example 2.28: In Fig. 2.63, we are driving a bus from a specialized output.

Fig. 2.63 Precharging a bus

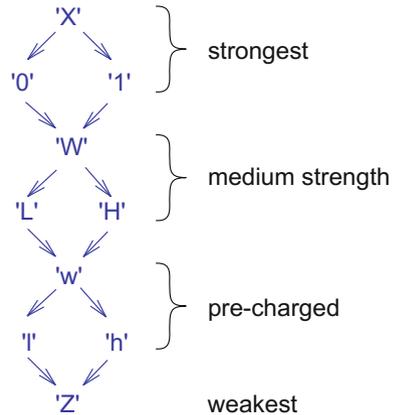


The bus has a high capacitive load C . While function f is still '0', we set pre to '1', charging capacitor C . Then, we set pre to '0'. If the real value of function f becomes known and it turns out to be '1', we discharge the bus. ∇

The key reason for using precharging is that charging a bus using an output such as the one shown in Fig. 2.61 is a slow process, since the resistance of depletion transistors is large. Discharging through regular pull-down transistors PD is a much faster process.

In order to model such cases, we need signal values which are weaker than 'H' and 'L', but stronger than 'Z'. We call such values “very weak signal values” and denote them by 'h' and 'l'. The corresponding very weak unknown value is denoted by 'w'. As a result, we obtain ten signal values {'0','1','Z','X','H','L','W','h','l','w'}. Using signal strengths, we can again define a partial order among these values (see Fig. 2.64).

Fig. 2.64 Partially ordered set {'0','1','Z','X','H','L','W','h','l','w'}



Note that precharging is not without risks. Once a precharged wire is discharged due to a transient signal, it cannot be recharged during the same clock period.

Five Signal Strengths

So far, we have ignored power supply signals. These are stronger than the strongest signals we have considered so far. Signal value sets taking power supply signals into account have resulted in the definition of initially popular 46-valued value sets [106]. However, such models are hardly used anymore.

2.7.3 Transaction-Level Modeling (TLM)

Discrete event simulation allows us to keep track of simulated time. However, it is not obvious how precisely we will be modeling time. A very precise model reflecting detailed timing of hardware signals will require long simulation times. In particular, very long simulation times are needed when we model electrical circuits. Faster simulation is feasible with cycle-accurate models reflecting the number of clock cycles in a clocked (synchronous) system implementation. More simulation speed can be gained from more coarse-grained timing models. In particular, transaction-level modeling (TLM) has received much attention. TLM has been defined as follows [184]:

Definition 2.21: “*Transaction-level modeling (TLM) is a high-level approach to modeling digital systems where details of communication among modules are separated from the details of the implementation of functional units or of the communication architecture. Communication mechanisms such as buses or FIFOs are modeled as channels, and are presented to modules using SystemC interface classes. Transaction requests take place by calling interface functions of these channel models, which encapsulate low-level details of the information exchange. At the transaction level, the emphasis is more on the functionality of the data transfers—what data is transferred to and from what locations—and less on their actual implementation, that is, on the actual protocol used for data transfer. This approach makes it easier for the system-level designer to experiment, for example, with different bus architectures (all supporting a common abstract interface) without having to recode models that interact with any of the buses, provided these models interact with the bus through the common interface.*”

A more detailed distinction between different timing models was described by Cai and Gajski [84]. They distinguish between timing models for communication and for computation¹⁶, and they consider different cases of timing models, depending upon how precisely communication and computation are modeled. Six cases are shown in Fig. 2.65.

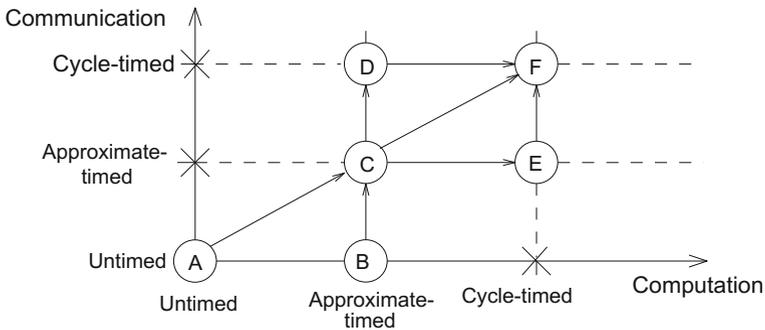


Fig. 2.65 Distinction between different timing models

For communication as well as for computations, we distinguish between untimed, approximately timed, and cycle-timed models. In diagram Fig. 2.65, crosses mark three unbalanced combinations of timing models, which have not been considered by Cai and Gajski. As a result, we consider six remaining cases [84]:

A Untimed models: In this case, we model only the functionality and do not consider timing at all. Such models are appropriate for early design phases. They can be called **specification model**.

¹⁶This is very much in line with the same distinction which we have made in Table 2.1 on p. 39.

- B In the specification model, we can replace pure functionality descriptions by descriptions of components using rough timing models. For example, we might know the WCET of some code running on a processor. We would still model communication by abstract communication primitives. As a result, we obtain node B in Fig. 2.65. Such a model can be called **component assembly model**.
- C In a model of type B, we could replace abstract communication primitives by communication models which are approximately timed. This means that we try to model access conflicts and their impact on the timing, but we do not model the impact of each and every signal, nor do we model any links to clock cycles. Such a model can be called **bus arbitration model**.
- D In a model of type C, we could replace rough communication timing models with cycle-timed models. This implies that we keep track of elapsed clock cycles in our simulation. We might even consider real, physical time. The resulting model, denoted as node D in Fig. 2.65, can be called a **bus functional model** [84].
- E In a model of type C, we could also replace rough computation timing models by cycle-accurate timing models of the computation. This allows us, for example, to capture memory references in detail. The resulting model can be called a **cycle-accurate computation model**.
- F The node labeled F is obtained when communication and computation are modeled in a cycle-accurate way. Such a model can be called an **implementation model**.

Design procedures need to traverse the diagram in Fig. 2.65 from node A to node F.

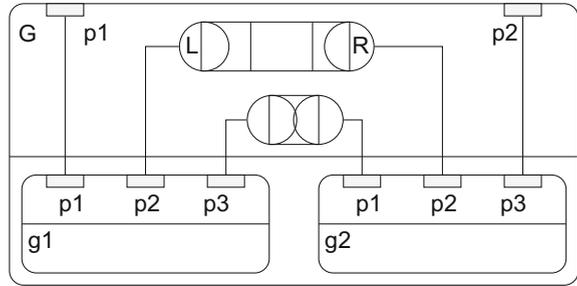
2.7.4 *SpecC*

The SpecC language [167] provides us with an nice example for demonstrating TLMs and a clear separation between communication and computation. SpecC models systems as hierarchical networks of behaviors communicating through channels. SpecC descriptions consist of behaviors, channels, and interfaces. Behaviors include ports, locally instantiated components, private variables and functions, and a public main function. Channels encapsulate communication. They include variables and functions, which are used for the definition of a communication protocol. Interfaces are linking behaviors and channels together. They declare the communication protocols which are defined in a channel.

SpecC can model hierarchies with nested behaviors.

Example 2.29: Figure 2.66 [167] shows a component G including sub-components g1 and g2 as leaves in the hierarchy.

Fig. 2.66 Structural hierarchy of SpecC example



This structural hierarchy is described in the following SpecC model:

```

01: interface L {void Write(int x); };
02: interface R {int Read(void); };
03: channel H implements L,R
04: {int Data; bool Valid;
05: void Write(int x) {Data=x; Valid=true;}
06: int Read (void)
07:   {while (!Valid) waitfor (10); return (Data);}
08: }
09: behavior G1(in int p1, L p2, in int p3)
10:   {void main (void) {/* ...*/ p2.Write(p1);} };
11: behavior G2(out int p1, R p2, out int p3)
12:   {void main(void) {/*...*/ p3=p2.Read(); } };
13: behavior G(in int p1, out int p2)
14: {int h1; H h2; G1 g1(p1, h2, h1); G2 g2(h1, h2, p2);
15: void main (void)
16:   {par {g1.main(); g2.main();}}
17: };

```

Concurrent execution of sub-components is denoted by the keyword **par** in line 16. As indicated in line 14, sub-components are communicating through integer h1 and through channel h2. Note that the interface protocol implemented in channel H (see line 03), consisting of methods for read and write operations (lines 05 and 06), can be changed without changing behaviors G1 and G2. For example, communication can be bit serial or parallel and the choice does not affect the models of G1 and G2. This is a necessary feature for reuse of hardware components or intellectual property (IP). The presented SpecC model does not include any timing information. Hence, it is a specification model (model of type A in Fig. 2.65). ▽

The design flow for SpecC was already shown in Fig. 1.9 on p. 20. The path in Fig. 2.65 is A, B, D, F [84]. At the specification level, SpecC can model any kind of communication and typically uses message passing. The communication model of SpecC has inspired the communication model in SystemC 2.0.

Note that SpecC is based on C and C++ syntax. The reason for this is the following:

There is the trend of implementing more and more functionality in software and using C for this purpose. For example, embedded systems implement standards such as MPEG 1/2/4 or decoders for mobile phone standards such as GSM, UMTS, or LTE. These standards are frequently available in the form of “reference implementations,” consisting of C programs not optimized for speed but providing the required functionality. The disadvantage of design methodologies based on special hardware description languages (like VHDL or Verilog, see below) is that these standards must be rewritten in order to generate systems. Furthermore, simulating hardware and software together requires interfacing software and hardware simulators. Typically, this involves a loss of simulation efficiency and inconsistent user interfaces. Also, designers would need to learn several languages.

Therefore, there has been a search for techniques for representing hardware structures in software languages. Some fundamental problems had to be solved before hardware could be modeled with software languages:

- **Concurrency**, as it is found in hardware, has to be modeled in software.
- There has to be a representation of simulated **time**.
- **Multiple-valued logic** as described earlier must be supported.
- The **determinate behavior** of almost all useful hardware circuits should be guaranteed.

For the SpecC language, as well as for other hardware description languages, these problems were solved.

2.7.5 *SystemC*TM

TLM modeling and the separation between communication and computation are also available in SystemCTM. SystemC (like SpecC) is based on C and C++. Similar to SpecC, SystemC provides channels, ports, and interfaces as abstract components for communication. The introduction of these mechanisms facilitates transaction-level modeling.

SystemCTM [416, 498] is a C++ class library. With SystemC, specifications can be written in C or C++, making appropriate references to the class library.

SystemC comprises a notion of processes executed concurrently. The execution of these processes is controlled by calls to **wait** primitives and **sensitivity lists** (lists of signals for which value changes start a re-execution of code). The sensitivity list concept includes dynamic sensitivity lists, i.e., the list of relevant signals can change during the execution.

SystemC includes a model of time. Earlier, SystemC 1.0 used floating point numbers to denote time. In the current standard, an integer model of time is preferred. SystemC also supports physical units such as picoseconds, nanoseconds, and microseconds.

SystemC data types include all common hardware types: Four-valued logic ('0','1','X' and 'Z') and bitvectors of different lengths are supported. Writing digital

signal processing applications is simplified due to the availability of fixed-point data types.

Determinate behavior (see p. 54) of SystemC is not guaranteed in general, unless a certain modeling style is used. Using a command line option, the simulator can be directed to run processes in different orders. This way, the user can check whether the simulation results depend on the sequence in which the processes are executed. However, for models of realistic complexity, only the presence of non-determinate behavior can be shown, not its absence.

Transaction-level modeling with SystemC has been described in a White Paper by Montoreano [384]. The White Paper distinguishes only between two types of TLM models:

- **Loosely timed models:** They are described as follows [384]: *“These models have a loose dependency between timing and data, and are able to provide timing information and the requested data at the point when a transaction is being initiated. These models do not depend on the advancement of time to be able to produce a response. Normally, resource contention and arbitration are not modeled using this style. Due to the limited dependencies and minimal context switches, these models can be made to run the fastest and are particularly useful for doing software development on a Virtual Platform.”*
- **Approximately timed models:** They are described as follows [384]: *“These models can depend on internal/external events firing and/or time advancing before they can provide a response. Resource contention and arbitration can be modeled easily with this style. Since these models must synchronize/order the transactions before processing them, they are forced to trigger multiple context switches in the simulation, resulting in performance penalties.”*

Hardware synthesis starting from SystemC has become available [207, 208]. A synthesizable subset of the language has been defined [7]. There are also commercial synthesis offerings. Commercial offerings are expected to support the synthesizable subset as a minimum. Methodology and applications for SystemC-based design are described in a book on that topic [390]. At the time of writing, the most recent version of SystemC is SystemC 2.3.1 [6].

2.7.6 VHDL

2.7.6.1 Introduction

VHDL is another language which is based on the discrete event paradigm. In contrast to SpecC and SystemC, it does not support a clear distinction between communication and computation, making reuse of components somewhat more difficult. However, VHDL is supported by many industrial and academic tools and is in widespread use. It is an example of a hardware description language (HDL). Having presented an initial example of event-based modeling already on p. 84, we would like to delve deeper into VHDL.

VHDL uses **processes** for modeling concurrency. Each process models one component of the potentially concurrent hardware. For simple hardware components, a single process may be sufficient. More complex components may need several processes for modeling their operations. Processes communicate through **signals**. Signals roughly correspond to physical connections (wires).

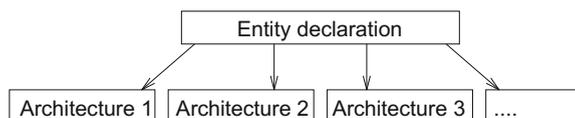
The origin of VHDL can be traced back to the 1980s. At that time, most design systems used graphical HDLs. The most common building block was the gate. However, in addition to using graphical HDLs, we can also use textual HDLs. The strength of textual languages is that they can easily represent complex computations including variables, loops, function parameters, and recursion. Accordingly, when digital systems became more complex in the 1980s, textual HDLs almost completely replaced graphical HDLs. Textual HDLs were initially a research topic at universities. See Mermet et al. [374] for a survey of languages designed in Europe at that time. MIMOLA was one of these languages, and the author of this book contributed to its design and applications [357, 362]. Textual languages became popular when VHDL and its competitor Verilog (see p. 104) were introduced.

VHDL was designed in the context of the VHSIC program of the Department of Defense (DoD) in the USA. VHSIC stands for *very high-speed integrated circuits*¹⁷. Initially, the design of VHDL (VHSIC hardware description language) was done by three companies: IBM, Intermetrics, and Texas Instruments. A first version of VHDL was published in 1984. Later, VHDL became an IEEE standard, called IEEE 1076. The first IEEE version was standardized in 1987; updates were published in 1993, 2000, 2002, and 2008 [229, 231–233, 235]¹⁸. VHDL-AMS [236] allows modeling analog and mixed-signal systems by including differential equations in the language. The design of VHDL used Ada (see p. 106) as the starting point, since both languages were designed for the DoD. Since Ada is based on PASCAL, VHDL has some of the syntactical flavor of PASCAL. However, the syntax of VHDL is much more complex and it is necessary not to get distracted by the syntax. In the current book, we will just focus on some concepts of VHDL which are useful also in other languages. A full description of VHDL is beyond the scope of this book. The standard is available from IEEE (see, for example, [235]).

2.7.6.2 Entities and Architectures

VHDL, like all other HDLs, includes support for modeling concurrent operation of hardware components. Components are modeled by so-called **design entities** or **VHDL entities**. Entities contain **processes** used to model concurrency. According to the VHDL grammar, design entities are composed of two types of ingredients: an **entity declaration** and one (or several) **architectures** (see Fig. 2.67).

Fig. 2.67 Entity consisting of an entity declaration and architectures



¹⁷The design of the Internet was also part of the VHSIC program.

¹⁸The next update can be expected for 2017.

For each entity, the most recently analyzed architecture will be used by default. The use of other architectures can be specified. Architectures may contain several processes.

Example 2.30: We will discuss a full adder as an example. Full adders have three input ports and two output ports (see Fig. 2.68).

Fig. 2.68 Full adder and its interface signals



An entity declaration corresponding to Fig. 2.68 is the following:

```

entity full_adder is                                -- entity declaration
  port (a, b, carry_in: in Bit;                      -- input ports
        sum, carry_out: out Bit);                    -- output ports
end full_adder;
    
```

Two hyphens (--) are starting comments. They extend until the end of the line. ▽

Architectures consist of architecture headers and architectural bodies. We can distinguish between different styles of bodies, in particular between structural and behavioral bodies. We will show how the two are different using the full adder as an example. Behavioral bodies include just enough information to compute output signals from input signals and the local state (if any), including the timing behavior of the outputs.

Example 2.31: The following is an example of this:

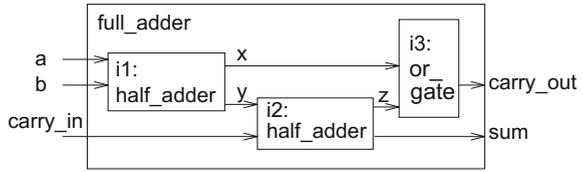
```

architecture behavior of full_adder is              -- architecture
begin
  sum      <= (a xor b) xor carry_in after 10 ns;
  carry_out <= (a and b) or (a and carry_in) or
              (b and carry_in) after 10 ns;
end behavior;
    
```

VHDL-based simulators are capable of displaying output signal waveforms resulting from stimuli applied to the inputs of the full adder described above.

In contrast, structural bodies describe the way entities are composed of simpler entities. For example, the full adder can be modeled as an entity consisting of three components (see Fig. 2.69). These components are called i1 to i3 and are of type half_adder or or_gate.

Fig. 2.69 Schematic describing structural body of the full adder



In the 1987 version of VHDL, these components must be declared in a so-called component declaration. This declaration is very similar to (and it serves the same purpose) as forward declarations in other languages. This declaration provides the necessary information about the component even if the full description of that component is not yet stored in the VHDL database (this may happen in the case of so-called top-down designs). From the 1992 version of VHDL onward, such declarations are not required if the relevant components are already stored in the component database.

Connections between local component and entity ports are described in **port maps**. The following VHDL code represents the structural body shown in Fig. 2.69:

```

architecture structure of full_adder is                                -- architecture head
  component half_adder
    port (in1, in2: in Bit; carry: out Bit; sum: out Bit);
  end component;
  component or_gate
    port (in1, in2: in Bit; o: out Bit);
  end component;
  signal x, y, z: Bit;                                               --local signals
begin                                                                -- port map section
  i1: half_adder                                                    -- introduction of half_adder i1
    port map (a, b, x, y);                                          --connections between ports
  i2: half_adder port map (y, carry_in, z, sum);
  i3: or_gate    port map (x, z, carry_out);
end structure;
    
```



2.7.6.3 Assignments

Example 2.31 contains several assignments. Let us look at assignments more closely! Assignments are special cases of statements. In VHDL, there are two kinds of assignments:

- **Variable assignments:** The syntax of variable assignments is

```

variable := expression
    
```

Whenever control reaches such an assignment, the expression is computed and assigned to the variable. Such assignments behave like assignments in common programming languages.

- **Signal assignments:** Signal assignments (as mentioned already on pp. 83 and 95) are evaluated concurrently. Signals and signal assignments are introduced in an attempt to model electrical signals in real hardware systems. Signals associate values with instances in time. In VHDL, such a mapping from time to values is represented by **waveforms**. Waveforms are computed from signal assignments. The syntax of signal assignments is

```

signal <= expression;
signal <= transport expression after delay;
signal <= expression after delay;
signal <= reject time inertial expression after delay;

```

Whenever control reaches such an assignment, the expression is computed and used to extend predicted future values of the waveform. In VHDL, each signal is associated with a so-called signal **driver**. Computing the value resulting from the contributions of multiple drivers to the same signal is called **resolution**, and resulting values are computed by functions called **resolution functions**. In this way, the *sup* function mentioned in the context of CSA theory is implemented if signals are connected.

In order to compute future values, **simulators are assumed to include a queue of events to happen later than the current simulated time**. This queue is sorted by the time at which future events (e.g., updates of signals) should happen. Executing a signal assignment results in the creation of entries in this queue. Each entry contains a time for executing the event, the affected signal, and the value to be assigned. For signal assignments not containing any **after** clause (first syntactical form), the entry will contain the current simulation time as the time at which this assignment has to be performed. In this case, the change will take place after an infinitesimally small amount of time, called δ -delay (see below). This allows us to update signals without changing macroscopic time.

For signal assignments containing a **transport** prefix (second syntactical form), the update of the signal will be delayed by the specified amount. This form of the assignment is following the so-called **transport delay model**. This model is based on the behavior of simple wires: Wires are (as a first order of approximation) delaying signals. Even short pulses propagate along wires. The transport delay model can be used for logic circuits, even though its main application is to model wires.

Example 2.32: Suppose that we model a simple OR gate using a transport delay signal assignment:

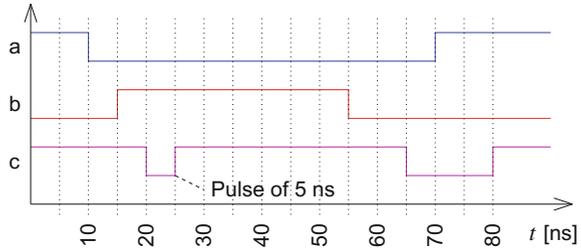
```

c <= transport a or b after 10 ns;

```

Such a model would propagate even short pulses (see Fig. 2.70).

Fig. 2.70 Gate modeled with transport delay



Output signal c includes a short pulse of 5 ns, which would be suppressed for a transport delay model. ▽

Transport delay signal assignments will delete all entries in the queue corresponding to the time of the computed update or later times (if we first execute an assignment with a rather large delay and then execute an assignment with a smaller delay, then the entry resulting from the first assignment will be deleted).

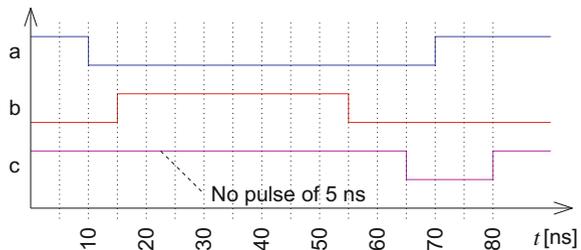
For signal assignments containing an after clause, but no transport clause, **inertial delay** is assumed. The inertial delay model reflects the fact that real circuits come with some “inertia.” This means that short spikes will be suppressed. For the third syntactical form of the signal assignment, all signals changes which are shorter than the specified delay are suppressed. For the fourth form, all signal changes which are shorter than the indicated amount are removed from the predicted waveform. The subtle rules for removals are not repeated here.

Example 2.33: Suppose that we model a simple OR gate using inertial delay:

```
c <= a or b after 10 ns;
```

For such a model, short spikes would be suppressed (see Fig. 2.71).

Fig. 2.71 Gate modeled with inertial delay



For output signal c, there is no short pulse of 5 ns, but the 15 ns pulse arrives at the output. ▽

2.7.6.4 VHDL Processes

Assignments are just a shorthand for VHDL processes. More control over signal evaluations is available with processes. The general syntax for processes is as follows:

```

label:                                -- optional
process
  declarations                          --optional
begin
  statements                             --optional
end process;

```

In addition to assignments, processes may contain **wait** statements. Such statements can be used to explicitly suspend a process. There are the following kinds of **wait** statements:

```

wait on signal list; --suspend until one of the signals in the list changes;
wait until condition;      --suspend until condition is met, e.g., a='1';
wait for duration;        --suspend for a specified period of time;
wait;                      --suspend indefinitely.

```

As an alternative to explicit **wait** statements, a list of signals can be added to the process header. In that case, the process is activated whenever one of the signals in that list changes its value.

Example 2.34: The following model of an AND gate will execute its body once and will restart from the beginning every time one of the inputs changes its value:

```

process(x, y) begin
  prod <= x and y ;
end process;

```

This model is equivalent to

```

process begin
  prod <= x and y ;
  wait on x,y;
end process;

```

where there is an explicit **wait** statement at the end. ▽

2.7.6.5 The VHDL Simulation Cycle

According to the original standards document [229], the execution of a VHDL model is described as follows: “*The execution of a model consists of an **initialization phase** followed by the **repetitive execution of process statements** in the description of that model. Each such repetition is said to be a **simulation cycle**. In each cycle, the values*

of all signals in the description are computed. If as a result of this computation an event occurs on a given signal, process statements that are sensitive to that signal will resume and will be executed as part of the simulation cycle.”

The initialization phase takes signal initializations into account and executes each process once. It is described in the standards as follows¹⁹:

“At the beginning of initialization, the current time, T_c , is assumed to be 0 ns. The initialization phase consists of the following steps²⁰:

- The driving value and the effective value of each explicitly declared signal are computed, and the current value of the signal is set to the effective value. This value is assumed to have been the value of the signal for an infinite length of time prior to the start of the simulation. ...
- Each ... process in the model is executed until it suspends. ...
- The time of the next simulation cycle (which in this case is the first simulation cycle), T_n , is calculated according to ... step e of the simulation cycle, below.”

Each simulation cycle starts with setting the current time to the next time at which changes must be considered. This time T_n was either computed during the initialization or during the last execution of the simulation cycle. Simulation terminates when the current time reaches its maximum, *TIME'HIGH*. The standard describes the simulation cycle as follows: “A simulation cycle consists of the following steps:

- (a) The current time, T_c , is set equal to T_n . Simulation is complete when $T_n = \text{TIME'HIGH}$ and there are no active drivers or process resumptions at T_n .
- (b) Each active explicit signal in the model is updated. (Events may occur as a result.)” ...

In the cycle preceding the current cycle, future values for some signals have been computed. If T_c corresponds to the time at which these values become valid, they are now assigned. Values of newly computed signals are not assigned before the next simulation cycle, at the earliest. Signals that change their value generate events which, in-turn, may release processes that are sensitive to that signal.

- (c) “For each process P , if P is currently sensitive to a signal S and if an event has occurred on S in this simulation cycle, then P resumes.
- (d) Each ... process that has resumed in the current simulation cycle is executed until it suspends.
- (e) T_n (the time of the next simulation cycle) is set to the earliest of
 1. *TIME'HIGH* (this is the end of simulation time).
 2. The next time at which a driver becomes active (this is the next instance in time, at which a driver specifies a new value), or
 3. The next time at which a process resumes (as computed from **wait for** statements).

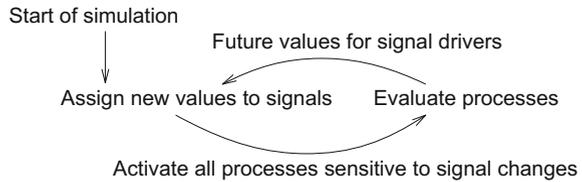
If $T_n = T_c$, then the next simulation cycle (if any) will be a delta cycle.”

The iterative nature of simulation cycles is shown in Fig. 2.72.

¹⁹We leave out the discussion of implicitly declared signals and so-called postponed processes.

²⁰Some sections of the standard are omitted in the citation (indicated by “...”).

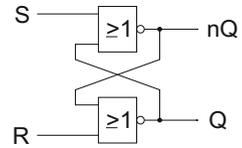
Fig. 2.72 VHDL simulation cycles



Delta (δ) simulation cycles have been the source of discussions. They introduce an infinitesimally small delay if the user did not specify any.

Example 2.35: Let us come back to our latch example and look more closely at timing. Figure 2.73 shows the latch again, this time using standard schematic symbols.

Fig. 2.73 RS Flip-flop



The flip-flop is modeled in VHDL as follows:

```

entity RS_Flipflop is
  port (R: in BIT; -- reset
         S: in BIT; -- set
         Q: inout BIT; -- output
         nQ: inout BIT; ); -- Q-bar
end RS_Flipflop;
architecture one of RS_Flipflop is
  begin
    process: (R,S,Q,nQ)
    begin
      Q <= R nor nQ; nQ <= S nor Q;
    end process;
  end one;
  
```

Ports Q and nQ must be of mode **inout** since they are also read internally, which would not be possible if they were of mode **out**. Table 2.5 shows the simulation times at which signals are updated for this model. During each cycle, updates are

Table 2.5 δ cycles for RS flip-flop

	<0ns	0ns	0ns+ δ	0ns+2* δ	0ns+3* δ
R	0	1	1	1	1
S	0	0	0	0	0
Q	1	1	0	0	0
nQ	0	0	0	1	1

propagated through one of the gates. Simulation terminates after three δ cycles. The last cycle does not change anything, since Q is already '0'. ▽

δ cycles correspond to an infinitesimally small unit of time, which will always exist in reality. δ cycles ensure that simulation respects causality.

The results do not depend on the order in which parts of the model are executed by the simulation. This feature is enabled by the separation between the computation of new values for signals and their actual assignment. In a model containing the lines

```
a <= b;
b <= a;
```

signals *a* and *b* will always be swapped. If the assignments were performed immediately, the result would depend on the order in which we execute the assignments (see also p. 53). **VHDL models are therefore determinate.** This is what we expect from the simulation of a real circuit with a fixed behavior.

There can be arbitrarily many δ cycles before the current time T_c is advanced. This possibility of infinite loops can be confusing. One of the options of avoiding this possibility would be to disallow zero delays, which we used in our model of the flip-flop.

The propagation of values using signals also allows an easy implementation of the observer pattern (see p. 31). In contrast to SDF, the number of observers can vary, depending on the number of processes waiting for changes on a signal.

What is the communication model behind VHDL? The description of the semantics of VHDL relies heavily on a **single, centralized** queue of future events, storing values of all signals in the future. The purpose of this queue is **not** to implement asynchronous message passing. Rather, this queue is supposed to be accessed by the simulation kernel, one entry at a time, in a non-distributed fashion. Attempts to perform distributed VHDL simulations are typically suffering from a poor performance. All modeled components can access values of signals and variables which are in their scope without any message-based communication. Therefore, we tend toward associating VHDL with a shared memory-based implementation of the communication. However, FIFO-based message passing could be implemented in VHDL on top of the VHDL simulator as well.

2.7.6.6 IEEE 1164

In VHDL, there is no predefined number of signal values, except for some basic support for two-valued logic. Instead, the used value sets can be defined in VHDL itself and different VHDL models can use different value sets.

However, portability of models would suffer in a very severe manner if this capability of VHDL was applied in this way. In order to simplify exchanging VHDL models, a standard value set was defined and standardized by the IEEE. This standard is called IEEE 1164 and is employed in many system models. IEEE 1164 has nine values: {'0','1','L','H','X','W','Z', 'U','-'}. The first seven values correspond to the seven signal values described from pp. 84 to 89. 'U' denotes an uninitialized value. It is used by simulators for signals that have not been explicitly initialized.

'-' denotes the **input don't care**. This value needs some explanation. Frequently, hardware description languages are used for describing Boolean functions. The VHDL **select** statement is a very convenient means for doing that. The **select** statement corresponds to **switch** and **case** statements found in other languages, and its meaning is different from the **select** statement in Ada (see p. 107).

Example 2.36: Suppose that we would like to represent the Boolean function

$$f(a, b, c) = a\bar{b} + bc$$

Furthermore, suppose that f should be undefined for the case of $a = b = c = '0'$. A very convenient way of specifying this function would be the following:

```
f <= select a & b & c           -- & denotes concatenation
    '1' when "10-"             -- corresponds to first term
    '1' when "-11"             -- corresponds to second term
    'X' when "000"
```

This way, functions given above could be easily translated into VHDL. Unfortunately, the **select** statement denotes something completely different. Since IEEE 1164 is just one of a large number of possible value sets, it does not include any knowledge about the “meaning” of '-'. Whenever VHDL tools evaluate select statements such as the one above, they check if the selecting expression ($a \& b \& c$ in the case above) is equal to the values in the when-clauses. In particular, they check if, e.g., $a \& b \& c$ is equal to "10-". In this context, '-' behaves like any other value: VHDL systems check if c has a value of '-'. Since '-' is never assigned to any of the variables, these tests will never be true. ∇

Therefore, '-' is of limited benefit. The non-availability of convenient input don't care values is the price that one has to pay for the flexibility of defining value sets in VHDL itself²¹.

The nice property of the general discussion on pp. 84 to 89 is the following: It allows us to immediately draw conclusions about the modeling power of IEEE 1164. The IEEE standard is based on the seven-valued value set described on p. 85, and therefore, is capable of modeling circuits containing depletion transistors. It is, however, not capable of modeling charge storage²².

2.7.7 Verilog and SystemVerilog

Verilog is another hardware description language. Initially, it was a proprietary language, but it was later standardized as IEEE standard 1364, with versions called IEEE

²¹This problem was corrected in VHDL 2006 [326].

²²As an exception, if the capability of modeling depletion transistors or pull-up resistors is not needed, one could interpret weak values as stored charges. This is, however, not very practical since pull-up resistors are found in most actual systems.

standard 1364–1995 (Verilog version 1.0) and IEEE standard 1364–2001 (Verilog 2.0). Some features of Verilog are quite similar to VHDL. Just like in VHDL, designs are described as a set of connected design entities, and design entities can be described behaviorally. Also, processes are used to model concurrency of hardware components. Just like in VHDL, bitvectors and time units are supported. There are, however, some areas in which Verilog is less flexible and focuses more on comfortable built-in features. For example, standard Verilog does not include the flexible mechanisms for defining enumerated types such as the ones defined in the IEEE 1164 standard. However, support for four-valued logic is built into the Verilog language, and the standard IEEE 1364 also provides multiple-valued logic with eight different signal strengths. Multiple-valued logic is more tightly integrated into Verilog than into VHDL. The Verilog logic system also provides more features for transistor-level descriptions. However, VHDL is more flexible. For example, VHDL allows hardware entities to be instantiated in loops. This can be used to generate a structural description for, e.g., n -bit adders without having to specify n adders and their interconnections manually.

Verilog has a similar number of users as VHDL. While VHDL is more popular in Europe, Verilog is more popular in the USA.

Verilog versions 3.0 and 3.1 are also known as SystemVerilog. They include numerous extensions to Verilog 2.0. These extensions include [237, 494]:

- additional language elements for modeling behavior,
- C data types such as `int` and type definition facilities such as `typedef` and `struct`,
- definition of interfaces of hardware components as separate entities,
- standardized mechanism for calling C/C++ functions and, to some extent, to call built-in Verilog functions from C,
- significantly enhanced features for describing an environment (called test bench) for the hardware circuit under design (called CUD), and for using the test bench to validate the CUD by simulation,
- classes known from object-oriented programming for use within test benches,
- dynamic process creation,
- standardized inter-process communication and synchronization, including semaphores,
- automatic memory allocation and deallocation,
- language features that provide a standardized interface to formal verification (see p. 231).

Due to the capability of interfacing with C and C++, interfacing to SystemC models is also possible. Improved facilities for simulation- as well as for formal verification-based design validation and the possible interfacing to SystemC will potentially create a very good acceptance. Verilog and SystemVerilog have been merged into one standard, IEEE 1800–2009 [234].

2.8 Von-Neumann Languages

The sequential execution and explicit control flow of von-Neumann languages are their common characteristics. Also, such languages allow an almost unrestricted access to global variables and we may need explicit communication and synchronization. Model-based design using CFSMs and computational graphs is very appropriate for embedded system design. Nevertheless, the use of standard von-Neumann languages is still widespread. Therefore, we cannot ignore these languages.

Also, the distinction between models such as KPNs and properly restricted von-Neumann languages is blurring. For KPNs, we do also have sequential execution of the code for each of the nodes. We are still keeping the distinction between KPN and von-Neumann languages since the KPN style of modeling has its advantages like determinate execution.

For the first two languages covered next, communication is built into the languages. For the remaining languages, focus is on the computations and communication can be replaced by selecting different libraries.

2.8.1 CSP

CSP (*communicating sequential processes*) [209] is one of the first languages comprising mechanisms for inter-process communication. Communication is based on channels.

Example 2.37: Consider input/output for channel *c* in this example:

<pre> process A var a .. a := 3; c!a; -- output to channel c end;</pre>	<pre> process B var b c?b; -- input from channel c end;</pre>
--	--

Both processes will wait for the other process to arrive at the input or output statement. This is a case of **rendez-vous**-based, **blocking**, or **synchronous message passing**. ▽

CSP is determinate, since it relies on the commitment to wait for input from a particular channel, like in Kahn process networks.

CSP has laid the foundation for the OCCAM language that was proposed as a programming language of the **transputer** [378]. The focus on communication channels has been picked up again in the design of the XS1 processor [575].

2.8.2 Ada

During the 1980s, the Department of Defense (DoD) in the USA realized that the dependability and maintainability of the software in its military equipment could soon become a major source of problems, unless some strict policy was enforced. It was decided that all software should be written in the same real-time language. Requirements for such a language were formulated.

No existing language met the requirements, and, consequently, the design of a new one was started. The language which was finally accepted was based on PASCAL. It was called Ada (after Ada Lovelace, regarded as being the first (female) programmer). Ada'95 [81, 274] is an object-oriented extension of the original standard.

One of the interesting features of Ada is the ability to have nested declarations of processes (called tasks in Ada). Tasks are started whenever control passes into the scope in which they are declared.

Example 2.38: The following code has been adopted from Burns et al. [81]:

```

procedure example1 is
  task a;
  task b;
  task body a is
    -- local declarations for a
    begin
      -- statements for a
    end a;
  task body b is
    -- local declarations for b
    begin
      -- statements for b
    end b;
begin
  -- body of procedure example1
end;

```

Tasks a and b will start before the first statement of the code of example1. ▽

The communication concept of Ada is another key concept. It is based on the synchronous *rendez-vous* paradigm. Whenever two tasks want to exchange information, the task reaching the “meeting point” first has to wait until its partner has also reached a corresponding point of control. Syntactically, procedures are used for describing communication. Procedures which can be called from other tasks must be identified by the keyword **entry**.

Example 2.39: This code has also been adopted from Burns et al. [80]:

```
task screen_out is
  entry call (val : character; x, y : integer);
end screen_out;
```

Task `screen_out` includes a procedure named `call` which can be called from other processes. Some other task can call this procedure by prefixing it with the name of the task:

```
screen_out.call('Z',10,20);
```

The calling task has to wait until the called task has reached a point of control, at which it accepts calls from other tasks. This point of control is indicated by the keyword **accept**:

```
task body screen_out is
  ...
  begin
    ...
    accept call (val : character; x, y : integer) do
      ...
    end call;
    ...
  end screen_out;
```

Obviously, task `screen_out` may be waiting for several calls at the same time. The Ada **select** statement provides this capability:

```
task screen_output is
  entry call_ch(val:character; x, y: integer);
  entry call_int(z, x, y: integer);
end screen_output;
task body screen_output is
  ...
  select
    accept call_ch ... do...
    end call_ch;
  or
    accept call_int ... do ..
    end call_int;
  end select;
  ...
```

In this case, task `screen_output` will be waiting until either `call_ch` or `call_int` is called. ▽

Due to the presence of the **select** statement, Ada is not determinate. Ada has been the preferred language for military equipment produced in the Western hemisphere for some time. Information about Ada is available from a number of Web sites (see, for example, [275]).

2.8.3 *Java*

For Java, communication can be selected by choosing between different libraries. Computation is strictly sequential.

Java was designed as a platform-independent language. It can be executed on any machine for which an interpreter of the internal byte-code representation of Java programs is available. This byte-code representation is a very compact representation, which requires less memory space than a standard binary machine code representation. Obviously, this is a potential advantage in system-on-a-chip applications, where memory space is limited.

Also, Java was designed to be a safe language. Many potentially dangerous features of C or C++ (like pointer arithmetic) are not available in Java. Java supports exception handling, simplifying recovery in case of run-time errors. There is no danger of memory leakages due to missing memory deallocation, since Java provides automatic garbage collection. This feature avoids potential problems in applications that must run for months or even years without ever being restarted. Java also meets the requirement to support concurrency since it includes threads (lightweight processes).

In addition, Java applications can be implemented quite fast, since Java supports object orientation and since Java development systems come with powerful libraries.

However, standard Java is not really designed for real-time and embedded systems. A number of characteristics which would make it a real-time and embedded programming language are missing:

- The size of Java run-time libraries has to be added to the size of the application itself. These run-time libraries can be quite large. Consequently, only really large applications benefit from the compact representation of the application itself.
- For many embedded applications, direct control over I/O devices is necessary (see p. 30). For safety reasons, no direct control over I/O devices is available in standard Java.
- Automatic garbage collection requires some computing time. In standard Java, the instance in time at which automatic garbage collection is started cannot be predicted. Hence, the worst-case execution time is very difficult to predict. Only extremely conservative estimates can be made.
- Java does not specify the order in which threads are executed if several threads are ready to run. As a result, worst-case execution time estimates must be even more conservative.
- Java programs are typically less efficient than C programs. Hence, Java is less recommended for resource constrained systems.

Proposals for solving the problems were made by Nilsen [402]. Proposals include hardware-supported garbage collection, replacement of the run-time scheduler, and tagging of some of the memory segments.

Currently (2017), relevant Java programming environments include the Java Enterprise Edition (J2EE), the Java Standard Edition (J2SE), the Java Micro

Edition (J2ME), and CardJava [492]. CardJava is a stripped-down version of Java with emphasis on security for SmartCard applications. J2ME is the relevant Java environment for all other types of embedded systems. Two library profiles have been defined for J2ME: CDC and CLDC. CLDC is used for mobile phones, using the so-called MIDP 1.0/2.0 as its standard for the application programming interface (API). CDC is used, for example, for TV sets and powerful mobile phones. Currently, relevant sources for Java real-time programming include book by Wellings [549], Dibble [126] and Bruno [73] as well as Web sites [16, 258] and the annual JTRES on “Java Technologies for Real-time and Embedded Systems” (see <http://jtres2016.compute.dtu.dk/> for the latest edition).

2.8.4 *Communication Libraries*

Standard von-Neumann languages do not come with built-in communication primitives. However, communication can be provided by libraries. There is a trend toward supporting communication within some local system as well as communication over longer distances. The use of Internet Protocols is becoming more popular.

2.8.4.1 MPI

Multi-core programming with imperative programs is possible with the message passing interface MPI. MPI is a very frequently used library, initially designed for high-performance computing. It allows a choice between synchronous and asynchronous message passing. For example, synchronous message passing is possible with the `MPI_Send` library function [376]:

`MPI_Send(buffer, count, type, dest, tag, comm)` where

- `buffer` is the address of data to be sent,
- `count` is the number of data elements to be sent,
- `type` is the data type of data to be sent (e.g., `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`),
- `dest` is the process id of the target process,
- `tag` is a message id (for sorting incoming messages),
- `comm` is the communication context (set of processes for which destination field is valid) and
- function result indicates success.

The following is an asynchronous library function:

`MPI_Isend(buffer, count, type, dest, tag, comm, request)` where

- `buffer`, `count`, `type`, `dest`, `tag`, `comm` are same as above, and
- the system issues a unique “request number.” The programmer uses this system assigned “handle” later (in a `WAIT` type routine) to determine completion of the non-blocking operation.

For MPI, the partitioning of computations among various processors must be done explicitly and the same is true for the communication and the distribution of data. Synchronization is implied by communication, but explicit synchronization is also possible. As a result, much of the management code is explicit and causes a major amount of work for the programmer. Also, it does not scale well when the number of processors is significantly changed [530].

In order to apply the MPI style of communication to real-time systems, a real-time version of MPI, called MPI/RT, has been defined [476]. MPI/RT does not cover issues such as thread creation and termination. MPI/RT is conceived as a potential layer between the operating system and standard (non-real-time) MPI.

MPI is available on a variety of platforms and also considered for multiple processors on a chip. However, it is based on the assumption that memory accesses are faster than communication operations. Also, MPI is mainly targeting at homogeneous multiprocessors. These assumptions are not true for multiple processors on a chip.

MPI has recently been extended to cover shared memory-based communication as well.

2.8.4.2 OpenMP

OpenMP is a compiler-based solution for shared memory-based communication. For OpenMP, parallelism is mostly explicit, whereas computation partitioning, communication, synchronization, etc. are implicit. Parallelism is expressed with pragmas: For example, loops can be preceded by pragmas indicating that they should be parallelized.

Example 2.40: The following program demonstrates a small parallel loop [417]:

```
void a1(int n, float *a, float *b)
{int i;
 #pragma omp parallel for
   for (i=1; i<n; i++) /* i is private by default */
     b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Note that a simple pragma is sufficient to indicate parallel programming. ∇

This means that OpenMP requires a relatively small amount of effort for parallelization for the user. However, this also means that the user cannot control partitioning [530]. There are some applications for MPSoCs (see, for example, Marian et al. [350]).

More techniques for multi-core programming will be described in the section on system software (see p. 225).

2.8.5 *Additional Languages*

Pearl [122] was designed for industrial control applications. It does include a large repertoire of language elements for controlling processes and referring to time. It requires an underlying real-time operating system. Pearl has been very popular in Europe, and a large number of industrial control projects have been implemented in Pearl. Pearl supports semaphores which can be used to protect communication based on shared buffers.

Chill [564] was designed for telephone exchange stations. It was standardized by the CCITT and used in telecommunication equipment. Chill is a kind of extended PASCAL.

IEC 60848 [223] and STEP 7 [49] are specialized languages that are used in control applications. Both provide graphical elements for describing the system functionality.

2.9 Levels of Hardware Modeling

In practice, designers start design cycles at various levels of abstraction. In some cases, these are high levels describing the overall behavior of the system to be designed. In other cases, the design process starts with the specification of electrical circuits at lower levels of abstraction. For each of the levels, a variety of languages exists, and some languages cover various levels. In the following, we will describe a set of possible levels. Some lower end levels are presented here for context reasons. Specifications should not start at those levels. The following is a list of frequently used names and attributes of levels:

- **System-level models:** The term system level is not clearly defined. It is used here to denote the entire embedded system and the system into which information processing is embedded (“the product”), and possibly also the environment (the physical input to the system, reflecting, e.g., the roads and weather conditions). Obviously, such models include mechanical as well as information processing aspects and it may be difficult to find appropriate simulators. Possible solutions include VHDL-AMS (the analog extension to VHDL), Verilog-AMS, SystemC, Modelica, COMSOL (see <https://www.comsol.com/>), or MATLAB/Simulink. MATLAB/Simulink and VHDL-AMS support modeling partial differential equations, which is a key requirement for modeling mechanical systems. It is a challenge to model information processing parts of the system in such a way that the simulation model can also be used for the synthesis of the embedded system. If this is not possible, error-prone manual translations between different models may be needed.
- **Algorithmic level:** At this level, we are simulating the algorithms that we intend to use within the embedded system. For example, we might be simulating MPEG video encoding algorithms in order to evaluate the resulting video quality. For such simulations, no reference is made to processors or instruction sets.

Data types may still allow a higher precision than the final implementation. For example, MPEG standards use double precision floating point numbers. The final embedded system will hardly include such data types. If data types have been selected such that every bit corresponds to exactly one bit in the final implementation, the model is said to be **bit-true**. Translating non-bit-true into bit-true models should be done with tool support (see p. 344).

Models at this level may consist of single processes or of sets of cooperating processes.

- **Instruction set level:** In this case, algorithms have already been compiled for the instruction set of the processor(s) to be used. Simulations at this level allow counting the executed number of instructions. There are several variations of the instruction set level:
 - In a coarse-grained model, only the effect of the instructions is simulated and their timing is not considered. The information available in assembly reference manuals (instruction set architecture (ISA)) is sufficient for defining such models.
 - **Transaction-level modeling:** In transaction-level modeling (see also p. 89), transactions, such as bus reads and writes, and communication between different components are modeled. Transaction-level modeling includes fewer details than cycle-true modeling (see below), enabling significantly superior simulation speeds [105].
 - In a more fine-grained model, we might have **cycle-true instruction set simulation**. In this case, the exact number of clock cycles required to run an application can be computed. Defining cycle-true models requires a detailed knowledge about processor hardware in order to correctly model, for example, pipeline stalls, resource hazards, and memory wait cycles.
- **Register-transfer level (RTL):** At this level, we model all the components at the register-transfer level, including arithmetic/logic units (ALUs), registers, memories, multiplexers, and decoders. Models at this level are always cycle-true. Automatic synthesis from such models is not a major challenge.
- **Gate-level models:** In this case, models contain gates as the basic components. Gate-level models provide accurate information about signal transition probabilities and can therefore also be used for power estimations. Also delay calculations can be more precise than for the RTL. However, typically no information about the length of wires and hence no information about capacitances is available. Hence, delay and power consumption calculations are still estimates. The term “gate-level model” is sometimes also employed in situations in which gates are only used to denote Boolean functions. Gates in such a model do not necessarily represent physical gates; we are only considering the behavior of the gates, not the fact that they also represent physical components. More precisely, such models should be called “Boolean function models²³,” but this term is not frequently used.

²³These models could be represented with binary decision diagrams (BDDs) [546].

- **Switch-level models:** Switch-level models use switches (transistors) as their basic components. Switch-level models use digital values models (refer to p. 84 for a description of possible value sets). In contrast to gate-level models, switch-level models are capable of reflecting bidirectional transfer of information. Switch-level models can be simulated with ternary simulation [74].
- **Circuit-level models:** Circuit theory and its components (current and voltage sources, resistors, capacitances, inductances, and frequently possible macro-models of semiconductors) form the basis of simulations at this level. Simulations involve partial differential equations. These equations are linear if and only if the behavior of semiconductors is linearized (approximated). The most frequently used simulator at this level is SPICE [533] and its variants.
- **Layout models:** Layout models reflect the actual circuit layout. Such models include **geometric** information. Layout models cannot be simulated directly, since the geometric information does not directly provide information about the behavior. Behavior can be deduced by correlating the layout model with a behavioral description at a higher level or by extracting circuits from the layout, using knowledge about the representation of circuit components at the layout level.
In a typical design flow, the length of wires and the corresponding capacitances are extracted from the layout and **back-annotated** to descriptions at higher levels. This way, more precision can be gained for delay and power estimations. Also, layout information may be essential for thermal modeling.
- **Process and device models:** At even lower levels, we can model fabrication processes. Using information from such models, we can compute parameters (gains, capacitances, etc.) for devices (transistors). Due to a growing complexity of the fabrication process, these models are also becoming more complex.

2.10 Comparison of Models of Computation

2.10.1 Criteria

Models of computation can be compared according to several criteria. For example, Stuijk [491] compares MoCs according to the following criteria:

- **Expressiveness** and **succinctness** indicate which systems can be modeled and how compact they are.
- **Analyzability** relates to the availability of schedulability tests and scheduling algorithms. Also, analyzability is affected by the need for run-time support.
- The **implementation efficiency** is influenced by the required scheduling policy and the code size.

Figure 2.74 classifies data flow models according to these criteria.

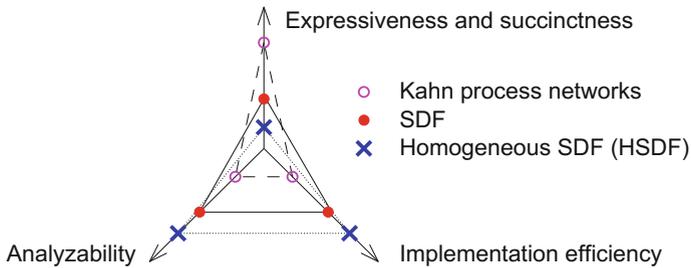


Fig. 2.74 Comparison between data flow models

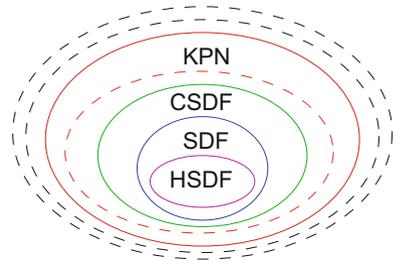
This figure reflects the fact that Kahn process networks are expressive: They are Turing complete, meaning that any problem which can be computed on a Turing machine can also be computed in a KPN. Turing machines are used as the standard model of universal computers [206]. However, termination properties and upper bounds on buffer sizes of KPNs are difficult to analyze. While Kahn process networks are Turing complete, cyclo-static data flow (CSDF, see p. 70) is not Turing complete. Also, SDF graphs are not Turing complete. The underlying reason is that they cannot model control flow. However, deadlock properties and upper bounds on buffer sizes of SDF graphs are easier to analyze. Homogeneous SDF (HSDF) graphs (graphs for which all rates are equal to one) are even less expressive, but also easier to analyze.

We could compare MoCs also with respect to the type of processes supported:

- The **number of processes** can be either **static** or **dynamic**. A static number of processes simplifies the implementation and is sufficient if each process models a piece of hardware and if we do not consider “hot-plugging” (dynamically changing the hardware architecture). Otherwise, dynamic process creation (and termination) should be supported.
- Processes can either be statically **nested** or all declared at the same level. For example, StateCharts allows nested process declarations while SDL (see p. 58) does not. Nesting provides encapsulation of concerns.
- Different techniques for **process creation** exist. Process creation can result from an elaboration of the process declaration in the source code, through the fork and join mechanism (supported for example in Unix), and also through explicit process creation calls.

The expressiveness of different data flow-oriented models of computation is also shown in Fig. 2.75 [43]. MoCs not discussed in this book are indicated by dashed lines.

Fig. 2.75 Expressiveness of data flow models



None of the MoCs and languages presented so far meets all the requirements for specification languages for embedded systems. Table 2.6 presents an overview over some of the key properties of some of the languages.

Table 2.6 Language comparison

Language	Behavioral hierarchy	Structural hierarchy	Programming language elements	Exceptions supported	Dynamic process creation
StateCharts	+	-	-	+	-
VHDL	+	+	+	-	-
SDL	+--	+--	+--	-	+
Petri nets	-	-	-	-	+
Java	+	-	+	+	+
SpecC	+	+	+	+	+
SystemC	+	+	+	+	+
Ada	+	-	+	+	+

Interestingly, SpecC and SystemC meet all listed requirements. However, some other requirements (like a precise specification of deadlines) are not included. It is not very likely that a single MoC or language will ever meet all requirements, since some of the requirements are essentially conflicting. A language supporting hard real-time requirements may well be inconvenient to use for less strict real-time requirements. A language appropriate for distributed control-dominated applications may be poor for local data flow dominated applications. Hence, we can expect that we will have to live with compromises and possibly with mixed models.

Which compromises are actually used in practice? In practice, assembly language programming was very common in the early years of embedded systems programming. Programs were small enough to handle the complexity of problems in assembly languages. The next step was the use of C or derivatives of C. Due to the increasing complexity of embedded system software, higher level languages are to follow the introduction of C. Object-oriented languages and SDL are languages which provide

the next level of abstraction. Also, languages like UML are required to capture specifications at an early design stage. The trend is to move toward model-based designs [453]. In practice, languages can be used like shown in Fig. 2.76.

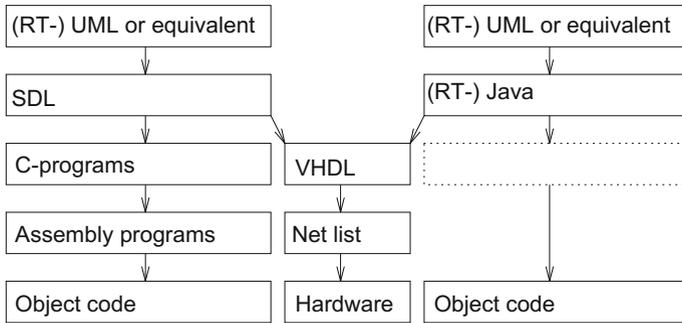


Fig. 2.76 Using various languages in combination

According to Fig. 2.76, languages like SDL or StateCharts can be translated into C. These C descriptions are then compiled. Starting with SDL or StateCharts also opens the way to implementing the functionality in hardware, if translators from these languages to VHDL are provided. Both C and VHDL will certainly survive as intermediate languages for many years. Java does not need intermediate steps but does also benefit from good translation concepts to assembly languages. In a similar way, translations between various graphs are feasible. For example, SDF graphs can be translated into a subclass of Petri nets [491]. Also, they correspond to a subclass of the **computation graph model** proposed by Karp and Miller [270]. Linking the various models of computation is facilitated by formal techniques [96].

Several languages for embedded system design are covered in a book edited by M. Radetzki [439]. Popovici et al. [432] use a combination of Simulink and SystemC.

We have skipped the discussion of algebraic languages such as LOTOS [246] and Z [480]. These languages enable precise specifications and formal proofs, but they are not executable.

2.10.2 UML™

UML™ is a language including diagrams reflecting several MoCs. Table 2.7 classifies the UML diagrams mentioned so far with respect to our table of MoCs.

Table 2.7 Models of computation available in UML™

Communication/ organization of components	Shared memory	Message passing	
		Synchronous	Asynchronous
Undefined components		Use cases	
		Sequence charts, timing diagrams	
Differential equations		–	
Finite state machines	State diagrams	–	–
Data flow	–	Data flow diagrams	
Petri nets	(Not useful)	Activity charts	
Distributed event model	–	–	
Von-Neumann model	–	–	

This figure shows how UML covers several models of computation, with a focus on early design phases. Semantics of communication is typically imprecisely defined. Therefore, our classification cannot be precise in this respect. In addition to the diagrams already mentioned, the following diagrams can be modeled:

- **Deployment diagrams:** These diagrams are important for embedded systems, and they describe the “execution architecture” of systems (hardware or software nodes).
- **Package diagrams:** Package diagrams represent the partitioning of software into software packages. They are similar to module charts in StateMate.
- **Class diagrams:** These diagrams describe inheritance relations of object classes.
- **Communication diagrams** (called **Collaboration diagrams** in UML™1.x): These graphs represent classes, relations between classes, and messages that are exchanged between them.
- **Component diagrams:** They represent the components used in applications or systems.
- **Object diagrams, interaction overview diagrams, composite structure diagrams:** This list consists of three types of diagrams which are less frequently used. Some of them may actually be special cases of other types of diagrams.

Available tools provide some consistency checking between the different diagram types. Complete checking, however, seems to be impossible. One reason for this is that the semantics of UML initially was left undefined. It has been argued that this was done intentionally, since one does not like to bother about the precise semantics during the early phases of the design. As a consequence, precise, executable specifications can only be obtained if UML is combined with some other, executable languages. Available design tools have combined UML with SDL [219] and C++. There are, however, also some first attempts to define the semantics of UML.

Version 1.4 of UML was not designed for embedded systems. Therefore, it lacks a number of features required for modeling embedded systems (see p. 27). In particular, the following features are missing [368]:

- the partitioning of software into tasks and processes cannot be modeled,
- timing behavior cannot be described at all,
- the presence of essential hardware components cannot be described.

Due to the increasing amount of software in embedded systems, UML is gaining importance for embedded systems as well. Hence, several proposals for UML extensions to support real-time applications have been made [132, 368]. These extensions have been considered during the design of UML 2.0. UML 2.0 includes 13 diagram types (up from nine in UML 1.4) [12]. Special profiles are taking the requirements of real-time systems into account [352]. Profiles include class diagrams with constraints, icons, diagram symbols, and some (partial) semantics. There are UML profiles for the following [352]:

- Schedulability, Performance, and Time Specification (SPT) [409],
- Testing [412],
- Quality of Service (QoS) and Fault Tolerance [412],
- a Systems Modeling Language called SysML [410],
- Modeling and Analysis of Real-Time Embedded Systems (MARTE), [411]
- UML and SystemC interoperability [446],
- the SPRINT profile for reuse of intellectual property (IP) [481].

Using such profiles, we can—for example—attach timing information to sequence charts. However, profiles may be incompatible. Also, UML has been designed for modeling and frequently leaves too many semantical issues open to allow automatic synthesis of implementations [352].

2.10.3 *Ptolemy II*

The Ptolemy project [435] focuses on modeling, simulation, and design of heterogeneous systems. Emphasis is on embedded systems that mix different technologies and, accordingly, also MoCs. For example, analog and digital electronics, hardware and software, and electrical and mechanical devices can be described. Ptolemy supports different types of applications, including signal processing, control applications, sequential decision making, and user interfaces. Special attention is paid to the generation of embedded software. The idea is to generate this software from the MoC which is most appropriate for a certain application. Version 2 of Ptolemy (Ptolemy II) supports the following MoCs and corresponding domains (see also p. 37):

1. Communicating sequential processes (CSP)
2. Continuous time (CT): This model is appropriate for mechanical systems and analog circuits. Hence, this model supports differential equations. Tools include extensible differential equation solvers.

3. Discrete event model (DE): This is the model used by many simulators, e.g., VHDL simulators.
4. Distributed discrete events (DDE). Discrete event systems are difficult to simulate in parallel, due to the inherent centralized queue of future events. Attempts to distribute this data structure have not been very successful so far. Therefore, this special (experimental) domain is introduced. Semantics can be defined such that distributed simulation becomes more efficient than in the DE model.
5. Finite state machines (FSM)
6. Process networks (PN), using Kahn process networks (see p. 65).
7. Synchronous dataflow (SDF)
8. Synchronous/reactive (SR) MoC: This model uses discrete time, but signals do not need to have a value at every clock tick. Esterel (see p. 57) is a language following this style of modeling.

This list clearly shows the focus on different models of computation in the Ptolemy project.

2.11 Problems

We suggest solving the following problems either at home or during a flipped classroom session:

2.1: What is a (design) model?

2.2: Prepare a list of up to six requirements for specification/modeling languages for embedded systems!

2.3: Why could our specification lead to deadlocks?

2.4: What is a “model of computation (MoC)”?

2.5: What is a “job” and how is it different from “tasks”?

2.6: Which are the two key techniques for communication in computers?

2.7: Which description techniques can be used for capturing initial ideas about the system to be designed?

2.8: Simulate trains between Paris, Brussels, Amsterdam, and Cologne, using the levi simulation software [473]! Modify the examples included with the software such that two independent tracks exist between any two stations and demonstrate an (arbitrary) schedule involving 10 trains!

2.9: Download the OpenModelica simulation software. Develop a simulation model for Newton’s cradle (see, for example, https://en.wikipedia.org/wiki/Newton%27s_cradle).

2.10: Modify the answering machine of Example 2.8 such that the owner can intervene at any time during the playing of precoded text or the recording of the message.

2.11: Model your daily schedule with a timed automaton. Hours are reflected by a variable h , days by a variable d . $d = 1$ means Monday, $d = 7$ means Sunday. On a weekend ($d = 6$ or $d = 7$), you leave the sleeping state between $h = 10$ and $h = 11$, spend 1–2h getting yourself ready for the day, stay with your friend until some time in the range $h = 20$ to $h = 21$, walk back home and enter the sleeping state between $h = 22$ and $h = 23$. During the week ($d = 1$ or ... or $d = 5$), you leave the sleeping state between $h = 7$ and $h = 8$, spend 1–2h getting yourself ready for the day, study until some time in the range $h = 20$ to $h = 21$, walk back home and enter the sleeping state between $h = 22$ and $h = 23$. Model your schedule! Do not forget to increase the day d at the end of each day.

2.12: Suppose the StateCharts model in Fig. 2.77 (left) model is given.

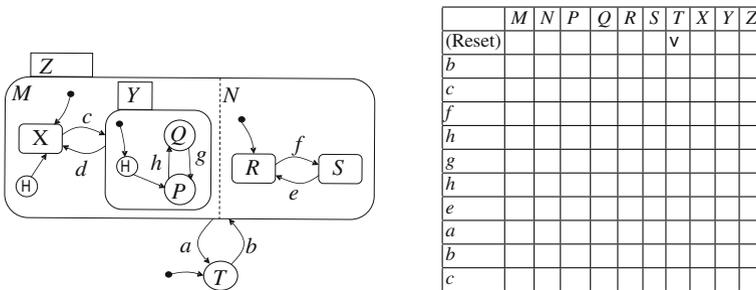


Fig. 2.77 StateCharts example: **left:** graphical model; **right:** table of states

Also, suppose that we have the following sequence of input events: $b c f h g h e a b c$. In the diagram in Fig. 2.77 (right), mark all the states the StateCharts model will be in after a particular input has been applied! Note that H denotes the history mechanism.

2.13: Are StateCharts determinate models if we follow the StateMate semantics? Please explain your answer!

2.14: Is SDL a determinate language? Please explain your answer!

2.15: Let us assume that you have been asked to help modeling the flow of visitors in the hypothetical Museum of Fine Future Information Nuggets (MUFFIN). We consider a steady state with no visitors entering or exiting the museum. The museum will have three exhibition halls. In front of each hall, there is space for a waiting line. The exit of this space is connected to the entry of the hall. Each of the hall exits is connected to each entry of the waiting spaces. Visitors leaving one of the halls are free to choose any of the other halls as their next one. We assume that each hall can be described as a process in a meaningful way, with some randomness of the time that a visitor stays in a hall. Assume that you would like to model this situation in SDL. Show a diagram with explicit processes and FIFO queues!

2.16: Download the levi simulation software for KPNs [471] and develop a KPN model computing Fibonacci numbers in a distributed fashion (i.e., just using a single KPN node is illegal).

2.17: Which three types of Petri nets did we discuss in this book?

2.18: One of the types of Petri nets allows several non-distinguishable tokens per place. Which components are used in a mathematical model of such nets? Hint: $N = (P, \dots, \dots)$

2.19: Draw the following condition/event system: $N = (C, E, F)$, given

- *Conditions:* $C = \{c_1, c_2, c_3, c_4\}$,
- *Events:* $E = \{e_1, e_2, e_3\}$,
- *Relation:*

$$F = \{(c_1, e_1), (c_1, e_2), (e_1, c_2), (e_1, c_3), (e_2, c_2), (e_2, c_3), (e_2, c_4), (c_2, e_3), (c_3, e_3), (c_4, e_3), (e_3, c_1), (e_3, c_4)\}$$

Specify the precondition of e_3 as well as the postcondition of e_1 . Is N *simple* or/and *pure*? Given it is not, which edge(s) need(s) to be removed in order to turn N into a pure net? Substantiate or prove your answers **concisely**.

2.20: What does a compact model of the dining philosophers problem look like?

2.21: CSA theory leads to 2, 3, and 4 logic strengths, corresponding to 4, 7, and 10 logic values. How many strengths and values are we using in IEEE 1164? Please show the partial order among the values of IEEE 1164 in a diagram! Which of the values of IEEE 1164 are not included in the partial order and what is the meaning of these values?

2.22: Suppose that a bus as shown in Fig. 2.78 is given. Rectangles containing an &-sign denote AND gates.



<http://www.springer.com/978-3-319-56043-4>

Embedded System Design
Embedded Systems Foundations of Cyber-Physical
Systems, and the Internet of Things

Marwedel, P.

2018, XXIV, 423 p. 284 illus., 121 illus. in color.,

Hardcover

ISBN: 978-3-319-56043-4