

Chapter 2

Models of Computation and Languages

This chapter looks at several important models of computation and languages for embedded systems design. We do not attempt to draw sharp distinctions between models and languages. Thus, the topics in this section are not divided strictly to either models of computation or languages. A model of computation is commonly used for defining the semantics of a language. However, when a model of computation is expressed with a simple syntax, it can also be called a language. For example, synchronous dataflow (SDF) is usually thought of as a model of computation. But as soon as it is expressed with a simple graphical syntax consisting of a network of blocks connected with arrows, it is not incorrect to call it a language.

2.1 Finite State Machine

An FSM [7] can be defined as a tuple of six elements:

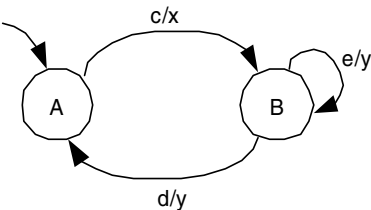
- Q is a finite set of states
- Σ is a set of input symbols
- Δ is a set of output symbols
- δ is a transition function mapping $Q \times \Sigma$ to $Q \times \Delta$
- q_0 is the initial state

An FSM reacts to inputs by entering the next state and producing outputs as defined by the transition function δ . The state transition diagram is a common way of representing an FSM. A simple state transition diagram is shown in Fig. 2.1.

In this case $Q = \{A, B\}$; $\Sigma = \{c, d, e\}$; $\Delta = \{x, y\}$; $\delta(A, c) = (x, B)$, $\delta(B, d) = (A, y)$ and $\delta(B, e) = (B, y)$, $q_0 = A$.

States are denoted by circles. Transitions are denoted by arcs. Every transition is associated with a guard and an action. This is labelled as *guard/action* on the corresponding arc. *Guard* denotes the enabling input event $i \in \Sigma$ that causes a transition

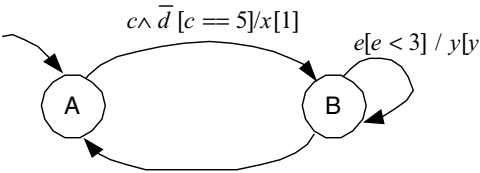
Fig. 2.1 Simple state transition diagram



Current state	A	A	B	B
Input symbol	d	c	e	d
Next state	A	B	B	A
Output symbol	-	x	y	y

Fig. 2.2 Possible trace for FSM in Fig 3.1

Fig. 2.3 FSM with valued events



from one state to another. *Action* denotes the output event $o \in \Delta$ that is produced as result of the transition.

If a non-enabling input event occurs in a certain state, the implicit self-transition is assumed. A sequence of reactions, sometimes called *trace*, shows a sequence of states and a sequence of output symbols caused by a sequence of input symbols. A possible trace for the FSM in Fig. 2.1 is shown in Fig. 2.2

The FSM model is closely related to the finite automata (FA) model. An FA is designed to recognize whether a sequence of input symbols belongs or does not belong to a certain language. Thus it accepts or rejects a sequence of input symbols. An FSM also produces a sequence of output symbols in addition to changing states.

Input and output events can be *pure* or they can carry a *value*. If an event is pure, it can only be *present* or *absent*. A valued event, besides being present or absent, also has a value when it is present.

Valued events increase the expressiveness of an FSM. A Boolean expression representing the guard of a transition can contain values of events. Values of output events can be expressed in terms of arithmetic operations. An example of a state transition diagram with valued events is given in Fig. 2.3.

In the above example, “=” is used for input events to test equality, whereas “=” is used for output events as an assignment operator. A Boolean expression in a

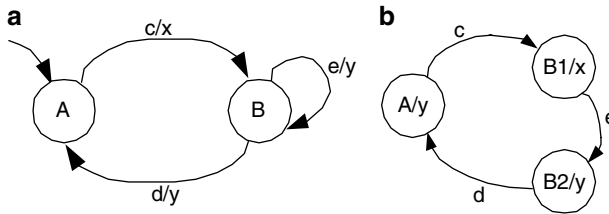


Fig. 2.4 Mealy and Moore machines

transition guard may require that some events be absent for the transition to take place. In Fig. 2.3, $c \wedge \bar{d}[c = 5]$ means that the transition occurs if c is present and has the value of 5 and d is absent.

Generally, two types of FSMs are commonly used. A *Mealy machine* is an FSM where outputs are associated with the transition. A *Moore machine* is an FSM where outputs are associated with the present state of the FSM. The FSMs presented so far are Mealy machines since their outputs are associated with transitions.

A Moore machine may have more states than the equivalent Mealy machine. Figure 2.4 shows the Mealy machine (part (a)) from Fig. 2.1 with the equivalent Moore machine (part (b)).

FSMs can be specified formally in a clear way. Their strong formal properties make them attractive for safety critical applications. It is easier to avoid undesirable states with FSMs than with if-else, goto and other statements found in programming languages.

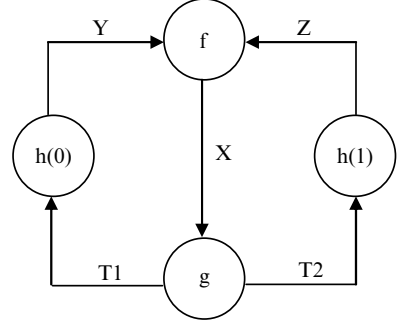
A single FSM could hardly be used to cover the entire control behaviour of a larger system because it would have an impractically large number of states. The usefulness of FSMs was largely increased when Harel introduced Statecharts (described in Sect. 2.7). In Statecharts, a single FSM state can be refined to another FSM. Thus, a hierarchical description of system behaviour is possible. The other major innovation in Statecharts is the possibility of having two or more states that are active at the same time. This helps in describing concurrent behaviours. A simple flat FSM has no means to describe hierarchy and concurrency, the two features that are commonly found in embedded systems.

2.2 Kahn Process Networks

Synchronous dataflow (SDF), which is used in DFCharts, belongs to the group of dataflow models in which processes communicate through first-in-first-out (FIFO) channels using blocking reads. In order to understand the properties of SDF, we need to explore the most general model in the dataflow group, called Kahn Process Networks (KPN) [27].

KPN processes communicate through FIFO channels using blocking reads and non-blocking writes. At any point during the execution of a Kahn process network, a process

Fig. 2.5 A Kahn process network example



can either be waiting for an input or doing computations. Since blocking reads are used, a process cannot test a channel for the presence of data. If a process attempts to read from an empty channel it will become blocked until data arrives on the channel.

Figure 2.5 shows a KPN example that was used in [27]. The two instances of process h copy data from input to output, but initially they output 0 and 1. Without these initial tokens the network would be deadlocked from the start. Process g copies data from input channel X to output channels $T1$ and $T2$, alternately. Process f copies data alternately from input channels Y and Z to output channel X .

In the denotational semantics of Kahn process networks, processes are mathematically defined as functions that map potentially infinite input streams to output streams. A stream is a sequence of data elements $X = [x_1, x_2, x_3, x_4, \dots]$, where indices are used to specify temporal features of the sequence. The empty sequence is marked by the symbol \perp . A relation on streams called *prefix ordering* [10] is useful for analyzing the mathematical properties of Kahn process networks. For example the sequence $X = [0]$ is a prefix of the sequence $Y = [0, 1]$ which is in turn a prefix of $Z = [0, 1, 2]$. The relation “ X is a prefix of Y or equal to Y ” is written as $X \subseteq Y$.

A *chain* is an ordered set in which any two elements are comparable. Alternate names for a chain are *linearly ordered set* and *totally ordered set* [29]. In the context of Kahn process networks, elements of a chain are sequences and they are compared with the prefix ordering relation \subseteq . Any increasing chain $\vec{X} = (X_1, X_2, \dots)$ with $X_1 \subseteq X_2 \subseteq \dots$ has a least upper bound $\Pi \vec{X}$ (symbol Π is used to denote a least upper bound). A least upper bound is a sequence whose length tends towards infinity:

$$\lim_{i \rightarrow \infty} X_i = \Pi \vec{X}$$

The set of all sequences is a complete partial order (c.p.o.) with the relation \subseteq , since any increasing chain of sequences in this set has a least upper bound.

In Kahn process networks, a process f maps input streams to output streams. A process f is continuous if and only if for any increasing chain $\vec{X} = (X_1, X_2, \dots)$:

$$f(\Pi \vec{X}) = \Pi f(\vec{X})$$

If a process is continuous, it is also monotonic (the opposite is not necessarily true). Monotonicity means that:

$$X \subseteq Y \Rightarrow f(X) \subseteq f(Y)$$

A process network can be described with a set of equations, with one equation for each process. For example, the process network in Fig. 2.5 can be represented by the following set of equations:

$$T_1 = g_1(X), T_2 = g_2(X), X = f(Y, Z), Y = h(T_1, 0), Z = h(T_2, 1)$$

The system of equations above can be reduced to a single equation. For instance the equation for X is:

$$X = f(h(g_1(X), 0), h(g_2(X), 1))$$

If all processes are continuous the set of equations has a unique least fixpoint solution. The solution represents the histories of tokens that appeared on the communication channels. For example, the solution for X is an infinite sequence of alternating 0's and 1's – $X=f(Y,Z)=[0,1,0,1 \dots]$. The proof by induction can be found in [27].

The blocking read semantics of Kahn process networks ensures that processes are continuous. Therefore a set of equations describing a Kahn process network will have a unique least fixed point solution. This leads to a very useful property of KPN. Any execution order of processes will yield the same solution i.e. the same histories of tokens on the communication channels.

While the execution order cannot influence the histories of tokens, it can greatly impact memory requirements (buffer sizes). Since writes are non-blocking, there are no restrictions on buffer sizes. There are two major methods for scheduling Kahn process networks, data-driven scheduling and demand driven scheduling [30]. In data driven scheduling, the semantics of the Kahn process networks is satisfied in a simple way – a process is unblocked as soon as data is available. Data driven scheduling can lead to unbounded accumulation of tokens on communication channels.

An alternative strategy is to use demand driven scheduling of processes, where a process is activated only when the tokens it produces are needed by another process. Kahn and MacQueen describe a demand driven scheduling method in [31]. A process that needs tokens is marked as *hungry* and that causes the producer of those tokens to be activated. That, in turn, can cause another activation. All the scheduling is done by a single process.

Regardless of the type of scheduling employed, decisions in KPN have to be made at run time. Thus, context switching becomes inevitable if multiple processes run on a single processor. Valuable time has to be spent on saving the state of the current thread before the control can be transferred to another thread.

2.3 Synchronous Dataflow

Synchronous dataflow (SDF) [23, 32] imposes limitations on KPN in order to make static scheduling possible. An SDF network is composed of *actors* that are connected by FIFO channels. When an actor fires, it consumes tokens from input channels and produces tokens on output channels. Firings of an SDF actor create a process. The firing rule of an actor specifies how many tokens are consumed on each input. In SDF, the constant number of tokens is consumed on each input in every firing i.e. the firing rule remains the same. It should also be emphasised that an SDF actor has to output a constant number of tokens on each output in every firing. Due to constant consumption and production rates of tokens it is possible to make very efficient static schedules.

Figure 2.6 shows an SDF network that consists of three actors. Consumption and production rates are labelled on each channel. For example, from the direction of the ch1, it can be seen that its production rate is RA1 and its consumption rate is RC1.

There are two steps in constructing a static schedule for an SDF graph. The first step is to determine how many times each actor should fire during an iteration. An iteration is a series of actor firings that return the channels to their original state. The number of tokens in a channel is the same before and after an iteration. The first step is accomplished by solving the set of balance equations [32]. Balance equations state that production and consumption of tokens must be equal on all channels. The balance equations for the SDF graph from Fig. 2.6 are shown below.

$$FA \times RA1 = FC \times RC1$$

$$FA \times RA2 = FB \times RB1$$

$$FB \times RB2 = FC \times RC2$$

FA, FB, FC are integers showing how many times actors A, B and C fire in a single iteration. They form a *firing* or *repetition vector*. The least positive integer solution is taken. For example if RA1=2, RA2=2, RB1=3, RB2=3, RC1=6 and RC2=6 then FA=3, FB=2 and FC=1.

If the only solution to the set of equations is zeros, the SDF graph is said to be inconsistent [33]. This means that production and consumption of tokens cannot be balanced on all channels. As a result, the execution of an inconsistent SDF graph

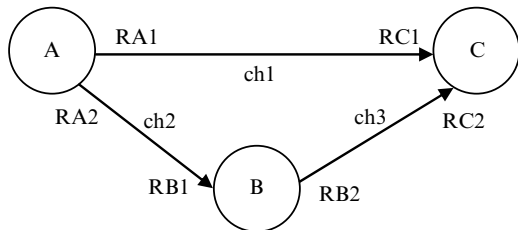


Fig. 2.6 An SDF graph

results in unbounded accumulation of tokens on channels. The graph from Fig. 2.6 would become inconsistent if RC1 were equal to 5, for example.

The second step is to analyse data dependencies between SDF actors in order to determine the order of firings. Multiple valid execution orders can emerge from the analysis due to different interleaving of actor firings. For example, the execution orders AAABBC and AABABC can both be used for the SDF graph from Fig. 2.6.

An SDF graph can contain a cycle that is formed by two or more channels. Initial tokens must be placed on a channel in the cycle so that deadlock does not occur. This was relevant for the KPN example from Fig. 2.5 where the two *h* processes produced initial values.

SDF is suitable for a wide range of digital signal processing (DSP) systems with constant data rates. It has efficient static scheduling and always executes in bounded memory. These properties are very useful in embedded systems design. For this reason, SDF graphs have been adopted as a part of DFCharts. For systems with variable rates, KPN or dynamic dataflow models can be used. There are many dataflow models whose expressiveness falls between SDF and KPN, such as boolean dataflow (BDF) [34], cyclostatic dataflow (CSDF) [26], parameterized synchronous dataflow (PSDF) [35], multidimensional synchronous dataflow [36], synchronous piggyback dataflow [37] among others.

A large amount of research has been done on synchronous dataflow resulting in numerous techniques and algorithms for memory optimization [38–44], simulation [45, 46], software synthesis [47, 48], hardware synthesis [49–51], and HW/SW codesign [52–54].

2.4 Synchronous/Reactive Model

The synchronous reactive (SR) model of computation [11] is the underlying model for the group of synchronous languages which includes Esterel [15], Argos [24], Lustre [16] and Signal [17]. A brief description of all four languages can be found in [55]. In the SR model of computation, time is divided into discrete instants. In each instant (tick), inputs are read and outputs are computed instantaneously. This is the central assumption in the *synchrony hypothesis* of the SR model. Instantaneous computation and communication makes outputs synchronous to inputs. The status of each signal has to be defined in each tick. It can be either present (true) or absent (false).

This model is similar to synchronous digital circuits that are driven by clocks. As a result, SR models can be efficiently synthesised into hardware. Software synthesis is also possible. In that case, the time between two successive instants is usually not constant.

The assumption of instantaneous computation facilitates hierarchical specification of systems. When a process is broken down into several other processes, they will all have instantaneous computation.

Zero delay communication represents a challenge for compilers of synchronous languages. An SR compiler has to be able to deal with causality loops that arise as

a result of zero delays. When resolving the status of each signal in a tick three general outcomes are possible:

- There is a single solution. The signal is either present or absent.
- There is no solution. The model does not make sense.
- Both the presence and absence of the signal satisfy the model. Thus, the system is non-deterministic.

The first outcome is the desired one. The last two outcomes should be rejected by the compiler and an error should be reported to the user. The three possible cases are illustrated in Sects. 2.3 and 2.4 with Esterel and Argos programs.

There are two distinct styles of synchronous modelling [11], which emerged during the development of the synchronous languages. The first one is known as *State Based Formalisms* (SBF), the second one is known as *Multiple Clocked Recurrent Systems* (MCRS's). The oldest and most developed synchronous language Esterel, uses the first style. Argos is also an SBF-style synchronous language. On the other hand, declarative dataflow languages Lustre and Signal use the second style.

State based formalisms are convenient for specifying control-dominated systems but they are not efficient in dataflow modelling. It is the opposite with MCRS's. Their main use is in specifying signal processing systems but it is more difficult to specify systems that step through different states. There have been attempts to unify two styles in a single environment as in [56, 57].

Synchronous programs can always be compiled into finite state machines. This property is very important since it greatly facilitates formal verification and ensures that memory requirements are known at compile time.

In Kahn process networks and related dataflow models, events are partially ordered. Events on a single channel are totally ordered, but they in general have no relation with events on other channels. In synchronous models, events are totally ordered. This has an important impact in modelling reactive systems, which have to promptly respond to every event that comes from the external environment. Reactive systems often have to wait for several events simultaneously. A KPN process cannot wait on multiple channels at the same time since it must implement blocking reads in order to achieve determinism. On the other hand, a synchronous process can test a channel before reading it and still preserve determinism. The downside of the total ordering of events is that it may unnecessarily reduce the implementation space by overspecifying the system, especially in the case of data-dominated systems.

It is interesting to note that due to the differences in event ordering, synchronous dataflow (SDF) is not an appropriate name when KPN based dataflow models are compared against synchronous models. To avoid confusion, a better name would be statically scheduled dataflow (SSDF) as suggested in [6].

2.5 Discrete Event Model

Discrete event (DE) [9] is the only MoC that incorporates the notion of physical time. Every event in DE carries a value and a time stamp. A DE block is activated when it receives an event to process. Events are processed chronologically. It is the

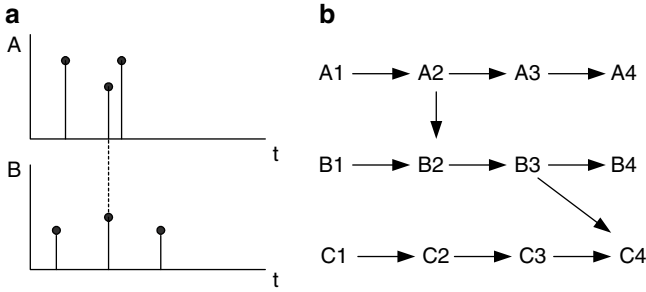
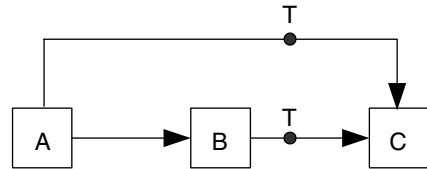


Fig. 2.7 Event ordering in DE and process networks. (a) Discrete event (b) Kahn process network

Fig. 2.8 Simultaneous events in DE



task of a DE scheduler to ensure that events with the smallest time stamp are processed first.

Events in DE are globally ordered. Figure 2.7 illustrates the difference in terms of ordering of events between the DE model and Kahn process networks.

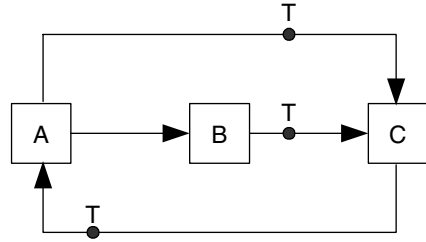
In the DE model any two events are comparable even if they belong to different signals. Two events can either be simultaneous or one occurs before the other. This is shown in Fig. 2.7a. When a model employs a partial ordering scheme, as in Kahn process networks for instance, events that belong to the same signal are totally ordered but events across different signals may not be comparable at all. This is illustrated in Fig. 2.7b. There are three signals A, B and C. For example, events A1 and C1 are not related.

Total ordering of events can overspecify systems, which makes implementation more difficult. It is easier to build parallel systems when events across signals are not related. In theory the DE model should be suitable for modelling distributed systems, but creating a DE simulator for distributed systems can be a difficult task since there may be a very large number of events that need to be sorted.

It was mentioned in the previous section that events in SR are also totally ordered. However, the total ordering of events is easier to implement in SR than in DE, since every event in SR is related to the global clock. The SR compilers rely on the global clock to sort events.

Simultaneous events and feedback loops with zero delay are the two major problems in DE model. Figure 2.8 illustrates the problem with simultaneous events. Block B produces zero delay. The time stamp of an event that passes through block B remains unchanged. Therefore block C receives two events with the identical time stamp T, one from block A, and another from block B. The DE scheduler has to

Fig. 2.9 Instantaneous feedback in DE



determine which event block C should process first or whether it should take two events at the same time, in which case both events are covered in one firing.

There are three different methods that the current DE simulators use in dealing with this problem. The ambiguity in Fig. 2.8 may be left unresolved. The processing of the two events is scheduled randomly. The result is a nondeterministic system which is generally undesirable.

The second approach is to break the relevant time instant into microsteps with infinitesimal time delays between microsteps. By doing that, a two-dimensional time model is effectively introduced. The time between two microsteps can be marked as Δt for instance. For example when an event passes through block B its time stamp is increased by Δt . The event appears at the output of block B with the time stamp $T + \Delta t$. When block C is invoked it processes first the event from block A which has the time stamp t . It then fires again and accepts the event from block B with the time stamp $T + \Delta t$. The infinitesimal time delays are not visible to the user. They are only used internally by the DE simulator.

The third approach is based on the analysis of data precedences statically within a single time instant, which is done in the Ptolemy [20] DE simulator. Arcs in a DE graph are assigned different priorities. The order of actions is always known when two simultaneous events appear at the same input.

Adding a feedback loop from block C to block A with C having zero processing time would create a situation that could not be resolved. This is shown in Fig. 2.9. An event circulates around the instantaneous loop without any increment in time stamp.

Digital hardware systems can be well described in DE. Both VHDL and Verilog simulators use DE as the underlying model of computation.

DE is mainly used for simulation. Global ordering of events with time stamps is expensive in real implementations.

2.6 Communicating Sequential Processes

A system described in communicating sequential processes (CSP) [13] consists of processes that execute independently in a sequential manner, but have to synchronize when they are communicating. When a process reaches a point of synchronisation with another process it has to stop and wait for the other process regardless of whether

it has to read or write. Both reads and writes are blocking. Processes have to participate in common operations at the same time. This type of communication where processes have to synchronize in order to communicate is called *rendezvous*.

Every process is defined with its alphabet, which contains the names of events that are relevant to the description of the process. The actual occurrence of an event is regarded as instantaneous or atomic. When an activity with some duration needs to be represented, it should have a starting event and an ending event.

In [13] processes that describe various vending machines are used as examples. For example, a simple vending machine has only two relevant events – coin and choc (chocolate). When the machine gets a coin it gives out a chocolate. Generally upper case letters are used for processes and lower case letters are used for events. Let the process be called VM. Then its alphabet is written as

$$\alpha VM = \{\text{coin}, \text{choc}\} \alpha \text{ is used to denote an alphabet}$$

In order to explain the behaviour of CSP, several key CSP operators are introduced, together with examples, in the following paragraphs. Some operators that are less frequently used are omitted. A detailed description can be found in [13].

The sequential behaviour of a process is described with the prefix operator “ \rightarrow ”. Let x be an event and let P be a process. Then $x \rightarrow P$ (pronounced “ x then P ”) describes an object that which first engages in the event x and behaves as described by P . The prefix operator cannot be used between two processes. It would be incorrect to write $P \rightarrow Q$.

For example a simple vending machine that breaks after receiving a coin is described as:

$$VM1 = \text{coin} \rightarrow \text{STOP}$$

STOP is a process that indicates unsuccessful termination. STOP does not react to any events. A successful termination also exists in CSP and will be introduced later. If a process gets into that state, no event can occur. A simple vending machine that serves two customers before it breaks is described as follows:

$$VM2 = \text{coin} \rightarrow \text{choc} \rightarrow \text{coin} \rightarrow \text{choc} \rightarrow \text{STOP}$$

The description indicates that the event choc can happen only after the event coin. The machine will not give a chocolate unless a coin is inserted.

Many processes will never stop. After executing a certain action they will go back to their initial state. Processes of that kind can best be described recursively. CSP supports recursive definitions, i.e. definitions in which a process name appears on both sides of the equation. For example the process CLOCK has only one event called tick. Thus the alphabet of the clock is $\alpha \text{CLOCK} = \{\text{tick}\}$. The process is recursively defined as

$$\text{CLOCK} = \text{tick} \rightarrow \text{CLOCK}$$

The above definition is equivalent to $\text{CLOCK} = \text{tick} \rightarrow \text{tick} \rightarrow \text{CLOCK}$,

$\text{CLOCK} = \text{tick} \rightarrow \text{tick} \rightarrow \text{tick} \rightarrow \text{CLOCK}$ etc. This sequence can be unfolded as many times as necessary. Obviously recursive definitions are very useful in process descriptions.

Hoare seems to prefer another form of recursive definition that is more formal. For example a good vending machine that does not break is defined recursively in the way shown above as follows:

$$\text{VM3} = \text{coin} \rightarrow \text{choc} \rightarrow \text{VM3}$$

The alternative and more formal definition according to Hoare is in the form of $\mu X:A.F(X)$ where the letter μ is used to denote a recursive expression, X is a local variable used in the recursive expression, and A is the alphabet of the expression i.e. the set of the names of events that appear in the expression. The alphabet is often omitted. Instead of X any other letter can be used, for example Y etc. VM3 is alternatively defined by μ with the definition below:

$$\text{VM3} = \mu X : \{\text{coin}, \text{choc}\}.(\text{coin} \rightarrow \text{choc} \rightarrow X)$$

Another important operator in CSP is the choice operator written as the bar $|$. The choice operator allows the environment in which the process operates to choose a sequence of actions that the process should perform. For example a vending machine may offer a choice of slots for inserting a 2p coin or a 1p coin. A customer decides which slot to use. The choice operator is used in conjunction with the prefix operator. In the expression below, two distinct events x and y initiate two distinct streams of behaviour:

$$(x \rightarrow P \mid y \rightarrow Q)$$

If x occurs before y , the subsequent behaviour of the object is defined by P . Similarly if y occurs before x , the subsequent behaviour of the object is defined by Q . The environment decides which event occurs first and thus which path is taken. The choice operator has to be used with the prefix operator. It would be incorrect to write $P \mid Q$.

As an example for the choice operator, a vending machine that offers a choice to the customer is defined below. The customer inserts a 5p coin and then chooses which combination of change to take.

$$\begin{aligned} \text{VM4} = & \text{in5p} \rightarrow (\text{out1p} \rightarrow \text{out1p} \rightarrow \text{out1p} \rightarrow \text{out2p} \rightarrow \text{VM4} \\ & \mid \text{out2p} \rightarrow \text{out1p} \rightarrow \text{out2p} \rightarrow \text{VM4}) \end{aligned}$$

In all of the examples above only single processes were considered. In CSP processes can be made to run in parallel by using the concurrency operator \parallel . $P \parallel Q$ means that the system is composed of two processes that are running concurrently. The two processes have to synchronize on any event that is common to their alphabets. Events that are not common are executed independently by one of the two processes.

An interesting example that illustrates the use of the concurrency operator is given in [13]. Two processes are defined and then composed into a system using \parallel . The two processes are called NOISYVM (noisy vending machine) and CUST (customer). The alphabet of NOISYVM is defined below.

$$\alpha\text{NOISYVM} = \{\text{coin}, \text{choc}, \text{clink}, \text{clunk}, \text{toffee}\}$$

The machine offers chocolate or toffee. The event clink is the sound that a coin makes when it is inserted. The event clunk is another sound that the machine makes on completion of a transaction. This time the machine has run out of toffee and it only outputs a chocolate after receiving a coin.

$$\text{NOISYVM} = (\text{coin} \rightarrow \text{clink} \rightarrow \text{choc} \rightarrow \text{clunk} \rightarrow \text{NOISYVM})$$

The customer prefers to get toffee. When he doesn't get toffee he says a curse, which is included in the alphabet of CUST given below.

$$\alpha\text{CUST} = \{\text{coin}, \text{choc}, \text{curse}, \text{toffee}\}$$

$$\text{CUST} = (\text{coin} \rightarrow (\text{toffee} \rightarrow \text{CUST} \mid \text{curse} \rightarrow \text{choc} \rightarrow \text{CUST}))$$

The system that results from the concurrent composition of the two processes is defined below.

$$(\text{NOISYVM} \parallel \text{CUST}) = \mu X. (\text{coin} \rightarrow (\text{clink} \rightarrow \text{curse} \rightarrow \text{choc} \rightarrow \text{clunk} \rightarrow X \\ \mid \text{curse} \rightarrow \text{clink} \rightarrow \text{choc} \rightarrow \text{clunk} \rightarrow X))$$

It should be noted that the two processes are synchronized on the event choc: the machine outputs it and the customer takes it. Choc appears in both alphabets. On the other hand, the events curse and clink occur asynchronously. They can occur one before the other or simultaneously. If they occur simultaneously it does not matter which one is recorded first.

This example clearly illustrates the essence of CSP: processes have to synchronize only on common events. Otherwise, they are asynchronous. Therefore CSP is not a completely synchronous MoC like synchronous/reactive model. In the SR model all actions are executed in lock-step. In CSP, processes synchronize only at rendezvous points.

Several other operators are briefly mentioned in the rest of the section.

When two processes P and Q are composed into a system, but do not have to synchronize on any events, their composition is written as $P \parallel\parallel Q$ which is read "P interleave Q". The events of the two processes are arbitrarily interleaved.

The unsuccessful termination is represented with the process STOP. The successful termination also exists and is represented by the process SKIP. SKIP does not react to any events in the same way as STOP. If a process finishes with SKIP it may be followed by another process. This is written as $P;Q$. Q is started after P successfully terminates.

In the examples shown so far, events could not be classified as inputs or outputs. In CSP the distinction between inputs and outputs is made using *channels*. Channels carry events. For example a simple process that inputs a message and then outputs the same message is defined as follows

$$\text{COPYBIT} = \mu X.(\text{in} ? x \rightarrow (\text{out} ! x \rightarrow X))$$

where $\alpha \text{in}(\text{COPYBIT}) = \alpha \text{out}(\text{COPYBIT}) = \{0, 1\}$

The alphabet of the input and output channels shows that the events 0 and 1 can occur on both channels.

Around the same time CSP was being developed, another similar model was emerging. The model is called Calculus of Communicating Systems (CCS). The model was created by Milner who later wrote a book about it [14]. Generally the two models are fairly similar, partly because the developers influenced each other while working on them. The basis of CCS are also processes that independently operate but have to synchronize on common events. Both models had a large impact on the research in concurrent systems. One of the main reasons for that is a sound formal treatment behind both models. An important property of concurrent systems such as deadlock can be formally analysed in CSP and CCS. Another example of formal analysis available in CSP and CCS is determining whether an implementation of a process satisfies its specification.

2.7 Petri Nets

Petri nets [12] is a graphical model that emerged in 1960s. Since then, many applications have been modeled with Petri nets and many research papers related to them have been published. While Petri nets is a tool for graphical modelling, it can also be mathematically analysed. In this section, though, only the basic features of Petri nets are introduced through examples.

Petri nets can be used to describe a wide range of applications. Systems that are suitable to be described by Petri nets are asynchronous, concurrent, distributed, and nondeterministic.

A Petri net is a bipartite directed graph. It consists of two kinds of nodes: places and transitions. Places are usually represented as circles while transitions are usually represented as bars or boxes. Places and transitions are joined by arcs. An arc can be drawn between a place and a transition but it is not allowed to join two places or two transitions. An arc can go from a transition to a place in which case it is marked as (t_i, p_i) or it can go from a place to a transition in which case it is marked as (p_i, t_i) . It cannot go from a transition to a transition (t_i, t_j) or from a place to a place (p_i, p_j) . With respect to a particular transition, a place can either be *input place* if the direction of the arc is (p, t) or *output place* if the direction of the arc is (t, p) .

Places hold one or more tokens. Tokens are usually marked as dots inside places. When a Petri net *fires*, i.e. a transition is made, the numbers of tokens in various

A Petri net is a 5-tuple, $PN = (P, T, F, W, M_0)$ where:

$P = \{p_1, p_2, \dots, p_m\}$ is a finite set of places,

$T = \{t_1, t_2, \dots, t_n\}$ is a finite set of transitions

$F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation)

$W: F \rightarrow \{1, 2, 3, \dots\}$ is a weight function

$M_0: P \rightarrow \{0, 1, 2, 3, \dots\}$ is the initial marking

$P \cap T = \emptyset$ and $P \cup T \neq \emptyset$

A Petri net structure $PN = (P, T, F, W)$ without any specific initial marking is denoted by N .

A Petri net with the given initial marking is denoted by (N, M_0) .

Fig. 2.10 Formal definition of Petri net

places change. The numbers of tokens in places represent a state of a Petri net which is in Petri nets terminology called *marking*. Each place is marked with the number of tokens it currently holds. The marking of a place is labelled as $M(p_i)$. The initial marking of a Petri net is denoted as M_0 .

Arcs are weighted. The weighting of an arc shows how many tokens can flow through the arc during a transition. The weighting of an arc is labelled as $w(p,t)$ or $w(t,p)$ depending on the direction of an arc. Everything that has been said so far about Petri net can be summarized in the formal definition in Fig. 2.10.

The behaviour of a Petri net can be analyzed by observing the set of states (markings) that a Petri net steps through. A state change occurs upon a transition when markings of places (numbers of tokens they hold) are changed. The rule regarding transitions is described in the three points below:

- A transition is *enabled* in each input place is marked with at least $w(p,t)$ tokens where $w(p,t)$ is the weighting of the corresponding arc.
- An enabled transition may or may not occur.
- When a transition occurs $w(p,t)$ tokens are removed from each input place and $w(t,p)$ tokens are added to each output place.

A transition that has no input place is called *source* transition. A source transition is unconditionally enabled. A transition that has no output place is called *sink* transition. A sink transition consumes tokens but it does not produce any.

A transition p and a place t are called *self-loop* if p is both the input and output place of t . A Petri net that does not contain any self-loops is called *pure*. A Petri net is called *ordinary* if the weighting of each of its arcs is one.

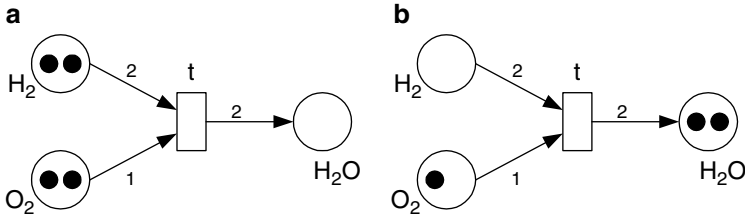


Fig. 2.11 Petri Net modeling chemical reaction. (a) Before transition (b) after transition

A simple example in Fig. 2.11 illustrates the transition rule. The Petri net in Fig. 2.11 models the well known chemical reaction $2\text{H}_2 + \text{O}_2 \rightarrow 2\text{H}_2\text{O}$. Part (a) shows that transition t is enabled since each of the input places is marked with enough input tokens. The initial marking of the net is $M_0 = (M(\text{H}_2) \ M(\text{O}_2) \ M(\text{H}_2\text{O})) = (2 \ 2 \ 0)$. Part (b) shows the state of the net after the transition has occurred. Two tokens have been removed from the input place H_2 as the weighting of the arc (H_2, t) is two. One token has been removed from the input place O_2 as the weighting of the arc (O_2, t) is one. Two tokens have been added to the output place H_2O as the weighting of the arc $(t, \text{H}_2\text{O})$ is two. After the transition the new marking is $M_1 = (0 \ 1 \ 2)$. In part (b) the transition t is disabled since the place H_2 has no tokens and it has to have at least two.

The Petri net in Fig. 2.11 has an *infinite capacity*, because there is no bound on the number of tokens in each place. In practice, though, it may be more realistic to put a limit on the number of tokens in each place. Petri nets whose places are bounded are called *finite capacity* Petri nets. The capacity of place p is labelled as $K(p)$.

When a Petri net is a finite capacity net, a transition has to account for the finite capacity of places. After a transition, the number of tokens in the output places must not exceed their capacities. This kind of transition rule that takes into account the capacity constraints is called *strong* transition rule. In the above example, *weak* transition rule was applied since the Petri net was an infinite capacity net, so the capacity constraints were not relevant.

In fact there are two options that can be used on a finite capacity net denoted as (N, M_0) . Either the strong transition rule can be used or the weak transition rule can be used on the transformed net (N', M'_0) . The transformation from (N, M_0) to (N', M'_0) consists of the two steps shown below:

- For each place p in (N, M_0) add a complementary place in (N', M'_0) . The initial marking of the complementary place should be $M'_0(p') = K(p) - M_0(p)$.
- New arcs must be added to connect complementary places to transitions. For every arc (t, p) add a new arc (p', t) with weighting $w(p', t) = w(t, p)$. For every arc (p, t) add a new arc (t, p') with weighting $w(t, p') = w(p, t)$.

The effect of this transformation is that the sum of tokens in a place p and its complementary place p' is equal to the capacity of the place p ($K(p)$) before and after transition when the weak transition rule is applied. Figure 2.12 gives an example of the transformation.

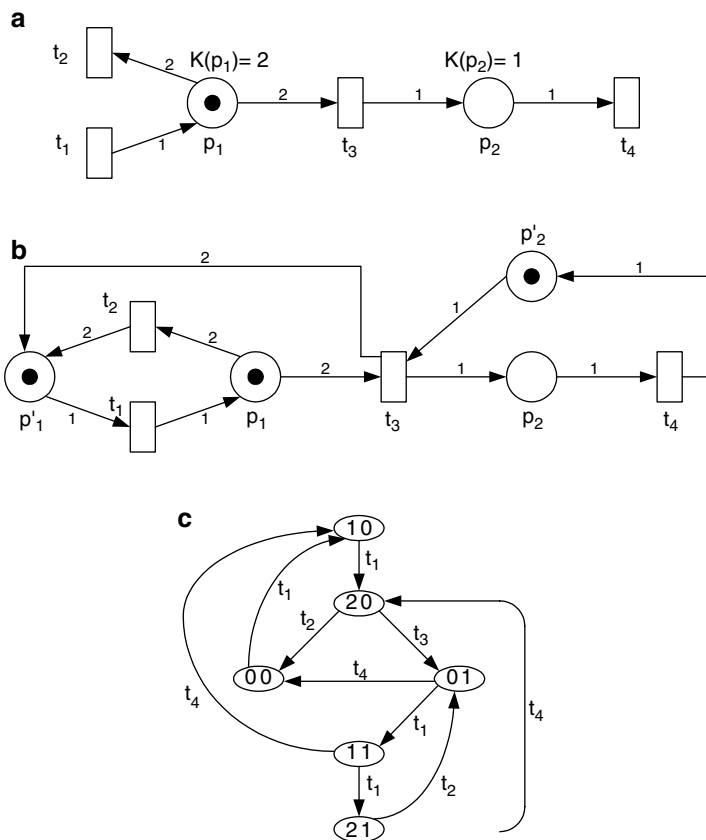


Fig. 2.12 Finite capacity net, its transformation and reachability graph (a) Finite capacity Petri net (b) finite capacity net from (a) transformed (c) reachability graph for Petri net in (a)

Part (a) shows a finite capacity net. The initial marking is $M_0 = (1 \ 0)$ and the only enabled transition is t_1 . After t_1 fires the new marking is $M_1 = (2 \ 0)$. The transitions t_2 and t_3 are now enabled. If t_2 fires the new marking is $M_2 = (0 \ 0)$. Otherwise if t_3 fires the new marking is $M_3 = (0 \ 1)$. By repeating this process a reachability graph can be drawn which is shown in Fig. 2.12c.

The finite capacity net (N, M_0) in part (a) can be transformed into (N', M'_0) shown in part (b). The first step is to add complementary places p'_1 and p'_2 with their initial markings $M'_0(p'_1) = K(p_1) - M_0(p_1) = 2 - 1 = 1$ and $M'_0(p'_2) = K(p_2) - M_0(p_2) = 1 - 0 = 1$. The second step is to add new arcs to connect the complementary places to the transitions. For example for $p'_2, (t_4, p'_2)$ is added with $w(t_4, p'_2) = 1$ since $w(p_2, t_4) = 1$ and (p'_2, t_3) with $w(p'_2, t_3) = 1$ since $w(t_3, p_2) = 1$. Similarly for p'_1 three arc are drawn: (t_2, p'_1) , (p'_1, t_1) and (t_3, p'_1) with $w(t_2, p'_1) = w(p_1, t_2) = 2$, $w(p'_1, t_1) = w(t_1, p_1) = 1$ and $w(t_3, p'_1) = w(p_1, t_3) = 2$.

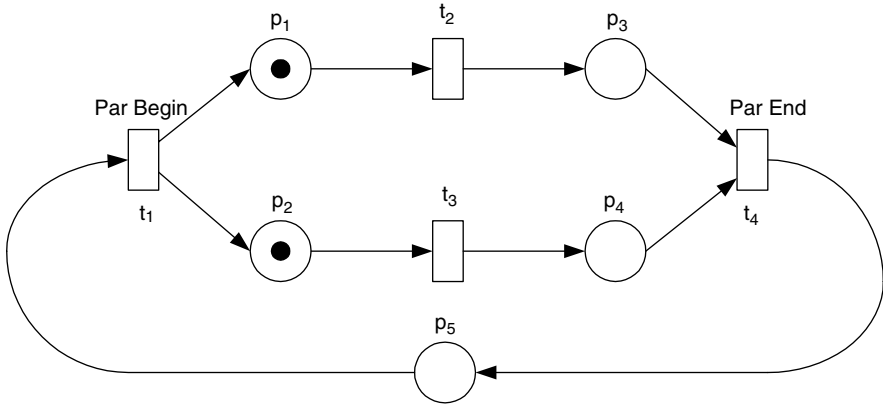


Fig. 2.13 A Petri net with concurrent activities

The reachability graph for (N', M_0') in Fig. 2.12b can be constructed in the same way as the reachability graph for (N, M_0) in Fig. 2.12a which is shown Fig. 2.12c. It can be shown that the two reachability graphs are isomorphic. This means that the two nets (N, M_0) and (N', M_0') are equivalent in the sense that both have the same set of all possible firing sequences.

There is a variety of applications that can be modelled with Petri nets. Considering all of them would take an enormous amount of space. One example is selected where it is shown how concurrency can be represented using Petri nets. A Petri net that contains two concurrent activities is shown in Fig. 2.13.

In the Petri net in Fig. 2.13 two concurrent activities begin when t_1 fires. Transitions t_2 and t_3 can fire independently of each other. This is enabled by the fact that all places in the two parallel branches have only one incoming and one outgoing arc (p_5 is also like that). The Petri net in Fig. 2.13 belongs to the subclass of Petri nets called *marked graph*. In a marked graph all places have exactly one incoming and one outgoing arc.

In embedded systems applications a clear disadvantage of Petri nets is the lack of hierarchy. Also, Petri nets are highly nondeterministic because of nondeterministic firings of transitions. Nondeterminism can make the analysis of a large embedded system difficult. It has been shown, however, that many smaller embedded applications can be successfully represented with Petri nets.

2.8 Statecharts/Statemate

It was pointed out in Sect. 2.1 that flat sequential FSMs are inadequate for representing complex control systems. The number of states and transitions becomes very large and the whole system becomes unmanageable. Harel introduced in [22]

Fig. 2.14 A basic FSM

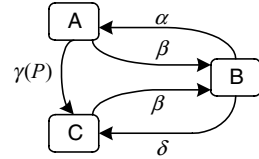
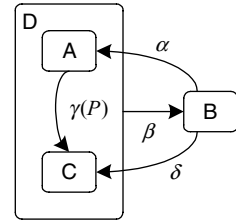


Fig. 2.15 Hierarchical equivalent of Fig. 2.14



some innovations that largely improved the usefulness of FSMs. The resulting visual language was called Statecharts. A tool called StateMate based around Statecharts was also developed. Both are discussed in this section.

In Harel's paper an FSM transition is labelled as $a[b]/c$. This type of notation was already introduced in Sect. 2.1. a is the event that causes the transition, b represents the condition which has to be fulfilled for the transition to take place when a occurs, and c is the output event generated by the transition which causes other transitions as will be illustrated later in the section. In Statecharts, the output events are called *actions*.

The basic FSM is neither hierarchical nor concurrent. Hierarchy and concurrency are common features in embedded systems. In Statecharts the two key innovations are OR states used to describe hierarchy and AND states used to describe concurrency. Figures 2.14 and 2.15 illustrate the use of OR states. Figure 2.14 shows a basic flat FSM with three states A, B, C. Figure 2.15 shows an equivalent hierarchical FSM.

In the FSM in Fig. 2.14 the same event β leads from states A and C to state B. It is convenient to cluster these two states into a *superstate* called D in Fig. 2.15. State D contains two OR states A and C. When D is entered, either A or C becomes active but not both. So OR states actually behave as exclusive OR states. When the event β occurs D is left which means that either A or C is left, and B is entered. It should be noticed that in Fig. 2.14 two arcs are used to represent this change while in Fig. 2.15 only one arc is used. In this simple example the FSM in Fig. 2.14 can be easily understood, and reducing the number of transitions may not seem significant, but in complex systems reducing the number of transitions brings large benefits. By introducing hierarchy with OR states the number of transitions is reduced and it is generally easier to understand the system.

The second key innovation in Statecharts is the use of AND states to represent concurrency. Figure 2.16 shows how AND states are represented. State Y is an

Fig. 2.16 Statechart with AND states

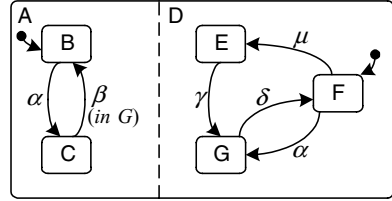
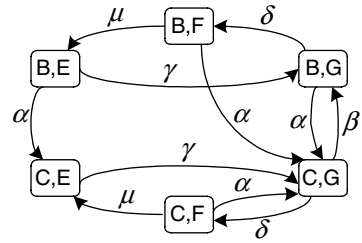


Fig. 2.17 Basic FSM equivalent to statechart in Fig. 2.16



orthogonal product of states A and D. *Orthogonality* is the term that is often used in [22] to denote concurrency. Being in state Y means that both states A and D are simultaneously active. Thus A and D are AND states. Graphically AND states are separated by a dashed line.

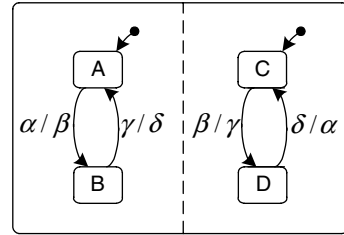
State A contains OR states B and C, while state D contains OR states F, E and G. Arrows pointed to B and F mean that these two states are the default states for A and D, respectively. This means that, when Y is entered from outside, states B and F become simultaneously active. Simultaneous transitions are possible in AND states. For example if B and F are active and the event α occurs states C and G are entered at the same time. Some events can cause a change in only one AND state. For example if F is active and μ occurs, E is entered but there is no change in A. So while AND states are concurrent they can operate independently.

The equivalent basic FSM of the statechart in Fig. 2.16 is shown in Fig. 2.17. The FSM in Fig. 2.17 contains six states since in Fig. 2.16 A contains two states and D contains three states. If A and D contained thousand states each the FSM in Fig. 2.17 would contain million states. This is so called state explosion problem, which makes it very difficult to represent large concurrent systems with basic FSMs.

There are some other features of Statecharts such as history, condition and selection entrances, timeouts and delays, which are not discussed here. Those features are less important than OR and AND states described above.

In the examples so far, FSM transitions did not produce any actions. Those transitions were in the form of $a[b]$ rather than $a[b]/c$. Actions can trigger other transitions so they are not really different from other events. In Statecharts, concurrent states communicate by the broadcast mechanism. Actions are instantaneous and they are immediately visible to all states. A state does not have to be prepared to receive an action as in rendezvous.

Fig. 2.18 Infinite loop due to instantaneous broadcast



The broadcast mechanism can create some interesting semantic problems. Harel outlined in [22] some of those problems and did not provide an immediate solution. That triggered lots of research on Statecharts resulting in at least 20 different variants. Von der Beeck provides a good summary of Statecharts variants in [68]. Figure 2.18 illustrates one of the problems related to communication in Statecharts.

It is not difficult to see that an infinite loop of transitions is produced when any of the four events occurs. A tool that analyses Statecharts must be able to detect directed cycles like this and report an error.

Statecharts can represent the reactive part of an embedded system but they cannot be used for transformative data functions. Stateate [82] is a tool based around Statecharts that allows specification and analysis of a complete embedded system including data parts.

Stateate incorporates *module-charts*, *activity-charts* and *statecharts*. In the examples above it was shown how an action produced in a transition can cause other transitions. In Stateate actions can also trigger *activities*. Unlike actions, activities take time. Activities are represented by activity-charts. The purpose of activity-charts is to represent data parts of the system under development (SUD) in Stateate. SUD is the term used in [82], a paper that describes Stateate.

In Stateate, statecharts are used to represent the reactive part of a system, but they are also used to control activity-charts, which, represent the data part of a system. Module-charts represent statecharts and activity-charts at a level that is closer to physical implementation. For example module-charts indicate how an activity-chart maps to a particular processor.

Statecharts control activity-charts by actions produced upon state transitions. Examples of some actions are start(*a*), suspend(*a*), resume(*a*), stop(*a*) where *a* is an activity. Statecharts and activity charts can be mixed at any level of hierarchy.

Atomic activities (those that cannot be further decomposed) are described by a programming language, but it is not indicated in [82] which programming language has to be used. Perhaps, several different programming languages may be used.

After a system has been described in Stateate using module-charts, activity-charts and statecharts, it can be simulated. Stateate can also synthesize the system description into C-code.

Although Stateate is equipped to support a complete design of embedded systems, statecharts that are used to model reactive behaviour are definitely the part that attracted the most attention. It is interesting to note that in Statecharts it is possible for an arc to connect two states at different hierarchical levels as seen in some

of the examples above. This jumping across hierarchy has been criticized by some researchers who believe it compromises the modularity of a design. In fact some variants of Statecharts like Argos allow declaration of local signals inside a state, which cannot be seen by higher level states.

2.9 Argos

A single, flat FSM is suitable for specifying sequential controllers, but it can hardly handle more complicated embedded systems where support for concurrency and hierarchy becomes necessary. This weakness was addressed in Statecharts [22], which introduced hierarchical and concurrent compositions of FSMs and several other useful features. It was described in [22] that concurrent FSMs communicate using synchronous broadcast with the assumption of instantaneous communication. However, many questions regarding instantaneous loops were left unanswered. This semantic gap resulted in more than 20 different versions of Statecharts [68] approaching causality problems in different ways. Argos is a version of Statecharts that uses the principles of Esterel to deal with semantic challenges caused by instantaneous communication. Unlike other Statecharts variants, Argos compiler rejects all programs that contain non-determinism that the programmer did not intend to have, i.e. implicit non-determinism. Another distinguishing feature of Argos is that it forbids inter-level transitions, which connect two states on different hierarchical levels. This emphasizes modularity.

The three basic Argos operators that are applied on FSMs are: synchronous parallel, refinement and hiding. They were introduced in [69]. Later, a few additional operators were described in [24]. Figure 2.19 shows an Argos specification with the three main operators.

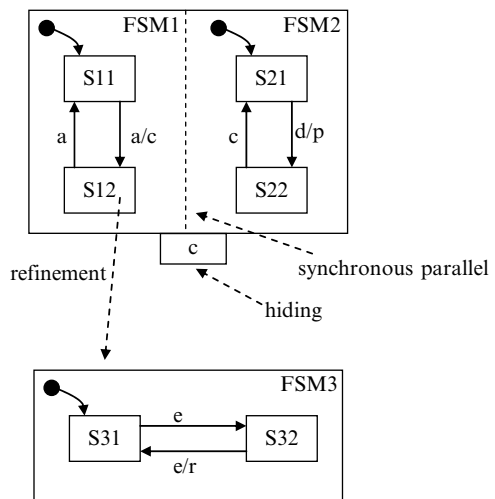


Fig. 2.19 Argos example

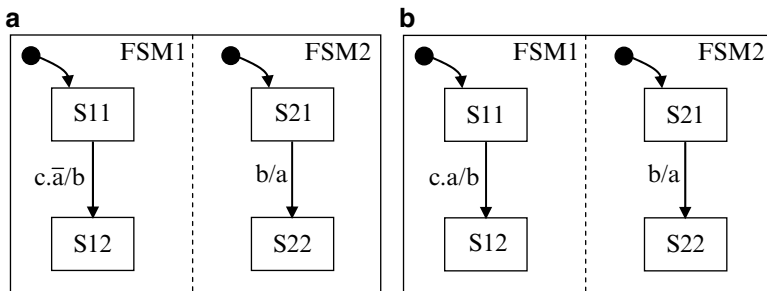


Fig. 2.20 Causality errors (a) no solution (b) multiple solutions

The top level FSMs are connected by the synchronous parallel operator and communicate using the local signal c . When FSM1 is in S11 and a is present, it makes the transition to S12 and emits c . If FSM2 is in S22, the presence of c instantaneously triggers the transition to S21. When FSM1 makes the transition from S12 to S11 it preempts FSM3. Only weak abort is available in Argos. Thus FSM3 is allowed to react in the tick in which it is pre-empted. For example if it is in S32 and e is present in the instant of pre-emption, r will be emitted.

As in Esterel, causality problems due to instantaneous communication can also appear in Argos. Two examples are shown in Fig. 2.20. A dot between input signals means logical AND (conjunction). A line above the name of a signal denotes its absence.

The specification in Fig. 2.20a has no meaningful behaviour. If c is present and a is absent, FSM1 makes the transition from S11 to S12 and emits b . Since b is present FSM2 makes the transition from S21 to S22 and emits a . This means that a is present and absent simultaneously, which is not possible. Figure 2.20b illustrates non-determinism. If c is present, both FSMs can take transitions and emit signals a and b that are necessary for their activations or transitions are not taken and a and b are not emitted.

2.10 Esterel

Esterel is an imperative language based on the synchronous reactive model of computation that was described in Sect. 2.3. Since the synchrony hypothesis is used in Esterel, it is assumed that all communications and computations take zero time. As a result outputs are synchronous to inputs. Communication is achieved by synchronous broadcast. Events produced by one process are immediately visible to other processes. Esterel is a deterministic language. For a given sequence of inputs only one sequence of outputs is possible.

Esterel programs communicate with external environment through signals and sensors. Signals are used as inputs and outputs. Sensors can only be used as inputs.

Table 2.1 Esterel kernel statements

Statement	Interpretation
nothing	Dummy statement
halt	Halting statement
$X := \text{exp}$	Assignment statement
call P (<i>variable-list</i>) (<i>expression-list</i>)	External procedure call
emit S(<i>exp</i>)	Signal emission
$\text{stat}_1; \text{stat}_2$	Sequence
loop <i>stat</i> end	Infinite loop
if <i>exp</i> then stat_1 else stat_2 end	Conditional
abort <i>stat</i> when S	Watchdog
$\text{stat}_1 \parallel \text{stat}_2$	Parallel statement
trap T in <i>stat</i> end	Trap definition
exit T	Exit from trap
var X : <i>type</i> in <i>stat</i> end	Local variable declaration
signal S (combine <i>type</i> with <i>comb</i>) in <i>stat</i> end	Local signal declaration

Signals can be *pure* or *valued*. Pure signals only have status at each instant. The status of a signal indicates whether the signal is present or absent. Valued signals also carry a value besides status. Valued signal S is denoted as S(*v*) where *v* represents the value. In expressions, the value of the signal S is denoted as ?S. If different values of the same signal are produced by multiple processes at the same time instant, it has to be specified how those values are combined. For example different values can be added to produce the resulting value. It can also be specified that it is forbidden to have multiple values for a signal at any time instant.

There are two types of Esterel statements: *primitive* statements and *derived* statements. Derived statements are actually derived from primitive statements. From the point of view of the programmer, derived statements are user-friendly and they also make programs shorter. An example of how a statement is derived will be shown later in this section. Primitive statements can be divided in two groups: basic imperative statements and temporal statements. Basic imperative statements take no time, they are executed instantaneously. Temporal statements handle signals and they take time. Table 2.1 lists the primitive statements, which form the basic Esterel kernel. The table was taken from [15] slightly modified to reflect recent changes in the Esterel syntax.

Most of the statements in Table 2.1 are found in other imperative languages. The statements *emit* and *abort when* are specific to Esterel. *Nothing* does nothing and takes no time. *Halt* does nothing but it never terminates so it takes time. The assignment statement and external procedure call are both instantaneous. *Emit* evaluates the value of the signal, emits the signal with its value and terminates. The emission is instantaneous. The sequence of statements is executed sequentially but since the computation is assumed to be infinitely fast the whole sequence takes no time. For example, a sequence

$$X := 1; X := X + 1$$

results in X being equal to 2. The sequence above takes zero time, but it is executed in the correct order. The number of statements in a sequence has to be finite. For example statements such as

$$X := 0; \text{ loop } X := X + 1 \text{ end}$$

are not allowed. The loop construct never terminates. When the body of a loop terminates the loop is instantly restarted. The expression in *if then else* conditional can be composed of both signals and variables in the latest version of Esterel. Thus, the *present* statement which is used exclusively for testing signals became obsolete.

The execution of *abort when* can end in two ways. The body inside the construct can finish before signal S occurs. If S occurs before the body finishes, the body is terminated instantly without being allowed to execute in the instant of termination. This type of abort is *strong*. It is also possible to create a *weak* abort simply by writing *weak abort* instead of *abort*. Weak abort can also be made using *trap*. When weak abort is used, the body is allowed to execute in the instant of termination.

The branches of a parallel statement start simultaneously. A parallel statement terminates when all of its branches have terminated.

The *trap exit* construct is a powerful control mechanism also found in some other languages. *Trap* defines a block of statements that instantly terminate when the corresponding *exit* statement is executed.

There are several useful derived statements in Esterel such as *await*, *every*, *each*, *sustain*. It is shown below what *await* is equivalent to.

await S	<u>instead of</u>	abort
		halt
		when S

The basic unit of an Esterel program is called *module*. A module consists of the *declaration* part and *statement part*. In the declaration part, types, constants, functions and procedures used in the module are declared. Esterel is used in combination with a host language, which can be C for instance. Earlier Esterel versions had only basic types and operators built in, such as integer, Boolean, basic arithmetic and logic operators. More complex operators and types had to be imported from the host language. The latest Esterel version [58] includes more complicated data types that are mainly geared towards hardware modelling. The declaration part can also specify relations between signals. For example a relation can state that two signals never occur at the same time. This helps the Esterel compiler in optimisation. An example of an Esterel module is given in Fig. 2.21.

The first top level parallel branch emits O with the value of 1 after either A or B occurs. This operation can be pre-empted by R , which is emitted by the second parallel branch when C occurs.

It was mentioned in Sect. 2.3 that the powerful zero delay communication mechanism of the SR model can create causality problems. In particular, two kinds of problems were mentioned: lack of solution and multiple solutions.

Fig. 2.21 Esterel example

```

module est_example:

  input A, B, R;
  output O : integer;

  loop
    abort
      [ await A || await B ];
    emit O(1)
  when R
  end
  ||
  loop
    await C;
    emit R
  end

end module

```

Those two problems will now be illustrated on Esterel programs. The program below demonstrates the first problem – no solution.

```

signal S in
  present S else emit S end
end

```

If it is assumed that signal S is not present then it is emitted so it is present. This contradiction cannot be resolved since the whole statement is executed in a single instant. The second program demonstrates the second causality problem – multiple solutions.

```

signal S1, S2 in
  present S1 else emit S2 end
  ||
  present S2 else emit S1 end
end

```

The program has two solutions. In one solution S1 is present and S2 is absent. In the other solution S2 is present and S1 is absent. Hence the program is nondeterministic. An Esterel compiler has to reject both programs.

The first Esterel compiler [15] converts an Esterel program into a flat FSM. The resulting code is very fast but it is impractical for large systems due to exponential increase in code size. The second Esterel compiler [59] translates an Esterel program into a synchronous digital circuit. The resulting code is compact, but its execution is very slow since every gate has to be evaluated in every reaction. Recent Esterel compilers [60–62] produce fast and compact code by simulating

the reactive features of Esterel, but they cannot compile a number of correct cyclic Esterel programs.

Esterel's reactive statements have inspired several modifications of C/C++ and Java to make them more suitable for embedded systems. Some examples are Modeling reactive systems in Java [63], ECL [64], Jester [65], Reactive-C[66], JavaTime [67].

2.11 Lustre and Signal

Lustre [36] and Signal [37] also belong to the group of synchronous languages. Unlike Esterel, which is an imperative language, Lustre and Signal are declarative dataflow languages. The main motivation behind the creation of Lustre and Signal was the fact that most embedded system designers have background in signal processing and control systems, not computer science. Signal processing/control systems are often modelled with equations in which variables are expressed as functions of time. Therefore, according to the creators of Lustre and Signal, dataflow languages would be a natural choice for control/signal processing engineers rather than imperative languages computer scientists are used to.

In Lustre and Signal, every variable refers to a *flow* which is a sequence of values associated with a clock. Lustre and Signal have more powerful clocking schemes than Esterel. For example, in Lustre, it is very easy to derive slower clocks using flows of Boolean values. This is illustrated in Table 2.2.

A clock derived from a Boolean flow represents a sequence of instants where the flow is true. The basic clock is the fastest clock. In the table above, the Boolean flow B1 is associated with the fastest clock. B2 is a Boolean flow that is associated with the clock derived from B1. A clock can also be derived from B2.

Synchronous dataflow languages can successfully describe some systems where clock rates are multiples of each other. It is more difficult to describe systems where data arrives irregularly. In that case, events that indicate absence of samples can be used, but in this way the system specification tends to become inefficient.

In [36] it is stated that Lustre can effectively describe both signal processing and reactive systems. Some examples that show how Lustre can describe reactive systems are given. It is true that Lustre can specify reactive systems, but it is not straightforward to do that. Synchronous dataflow is convenient for specifying some signal processing systems, but not reactive systems since states are not easily specified.

Table 2.2 Boolean flows and clocks in Lustre

Basic clock	1	2	3	4	5	6	7	8
Values of B1	True	True	False	True	False	True	False	False
Clock derived from B1	1	2		3		4		
Values of B2	False	True		True		False		
Clock derived from B2		1		2				

Synchronous dataflow languages are often regarded as being close to Kahn process networks. However there is one important difference – synchronous dataflow languages are synchronous, while Kahn process networks are asynchronous. Kahn process networks use buffers for communication between nodes. This is not the case in Lustre and Signal.

2.12 SystemC

SystemC [70–72] is an attempt to expand the widely used programming language C++ with features that would make it suitable for specifying embedded systems. SystemC is a class library of C++ and can be compiled on any C++ compiler. This is a clear advantage since there are well-developed, mature tools for C++. SystemC includes constructs to support specification of embedded systems characteristics such as timing, concurrency and reactive behaviour. Such construct cannot be found in the standard C/C++ programming environment.

SystemC enables designers to represent both hardware and software parts of an embedded system in the single environment. A specification in SystemC is executable because it can be simulated. In fact, SystemC allows specifying and simulating a system at different levels of abstraction, ranging from highly abstract system models down to cycle based models. The ability to simulate a high level system specification brings large benefits to the design process. System functionality can be verified before implementation begins. Mistakes can be uncovered early in the design process, when it is much cheaper to remove them than in well advanced design stages.

The main idea behind SystemC version 1.0 is to enable the designers to describe both software and hardware using the same language. For that purpose some features of hardware description languages were incorporated in the SystemC class library. Later SystemC 1.0 evolved into SystemC 2.0. The current version is 2.2 but we will focus only on major changes that occurred between the releases 1.0 and 2.0. With respect to SystemC 1.0, SystemC 2.0 is better equipped for system-level modelling. This is mainly due to new communication mechanisms that appeared in SystemC 2.0. The second version still supports everything from the first version. In the following paragraphs SystemC 1.0 is firstly described. Later in the section, the features that defined SystemC 2.0 are outlined.

A specification in SystemC consists of modules. A module can instantiate lower level modules, thus supporting hierarchy. Modules contain processes that describe system functionality. Three kinds of processes exist as will be described below. Modules are connected through ports which can be bi-directional or uni-directional. Ports are connected with signals. There is a wide range of signal types available in SystemC in order to support different levels of abstraction. Clocks are also available as a special signal type.

Breaking a system into modules enables division of tasks among designers. A module can be modified such that its external interface and functionality remain the same and the only thing that is changed is the way in which the function is described. In this way other modules in the design are unaffected. The external interface of a

module is represented by its ports. There are three types of ports: input, output and input – output ports. Every port also has a data type associated with it. Modules are declared with the SystemC keyword `SC_MODULE`. For example the declaration of a module that describes a fifo (first-in-first-out) buffer is given below.

```
SC_MODULE (fifo) {
    sc_in<bool>load;
    sc_in<bool>read;
    sc_inout<int>data;
    sc_out<bool>full;
    sc_out<bool>empty;
    // module description not shown
}
```

Only the ports of the module are shown, not the functionality. The module has two input ports, two output ports and one bi-directional input-output port. The input and output ports are used for control and are all of type boolean. The bi-directional data port is of type integer.

Ports of different modules are connected with signals. A signal declaration only indicates which data type is carried by the signal, while the direction (in, out, inout) is not specified as it is in the case of ports. The direction of data flow in a signal is determined by ports that are connected together by the signal.

Local variables can be used to store data received through ports. Local variables are only visible inside the module in which they are declared, unless they are explicitly made to be visible in other modules. Local variables can be of any C++ data type, SystemC data type or user-defined data type.

The functionality in a module is defined by processes. Processes are registered with the SystemC kernel. Processes are sensitive to signal changes. A process starts executing when there is a change in at least one of the signals the process is sensitive to. Some processes are similar to C++ functions. They execute sequentially and return control once they have stepped through all statements. Other processes are different in that they may be suspended by halting statements and then resumed at some later instant.

There are three types of processes in SystemC: methods, threads and clocked threads. Methods behave like C++ functions. A method starts executing statements sequentially when called. It returns control to the calling mechanism when it reaches the end. The execution of a method has to be completed. For that reason, it is recommended in SystemC User's Guide that designers should be careful to avoid making infinite loops inside a method.

Threads may be suspended and resumed. A thread is suspended when the `wait()` function is encountered. It is resumed when one of the signals in its sensitivity list changes.

Clocked threads are a special case of threads. Clocked threads are suspended and resumed in the same way as threads. However the sensitivity list of a clocked thread contains only one signal, and that signal is a clock. Furthermore, a clocked thread is sensitive to only one edge of the clock. Clocked threads resemble the way in which synchronous digital circuits are specified. For example, a synchronous process in VHDL contains only a clock signal in its sensitivity list.

```

#include "systemc.h"

SC_MODULE (count) {
    sc_in <bool>    load;
    sc_in <int>     din;
    sc_in <bool>    clock;
    sc_out <int>    dout;

    int count_val;    // internal data storage

    void count_up( );

    SC_CTOR (count) {
        SC_METHOD (count_up);    // Method process
        sensitive_pos << clock;
    }
};

void count : : count_up ( ) {
    if (load) {
        count_val = din;
    } else {
        count_val = count_val + 1;
    }
    dout = count_val;
}

```

Fig. 2.22 SystemC description of a counter

The code in Fig. 2.22 illustrates how a counter is specified in SystemC inside a module.

The behaviour of the module is described by the process `count_up`, which is of the type `method`. The process is triggered by the positive edge of the clock. This is specified in the line “`sensitive_pos << clock`”. When the process is triggered, the value on input port `load` is checked. If it is true, variable `count_val` is assigned the value of input port `din`. Otherwise, `count_val` is incremented by one. `count_val` is a local variable, visible only in the module. It should be noted that every module is initialized by a constructor. The keyword `SC_CTOR` is used for constructors.

In SystemC 1.0 modules communicate through ports which are connected by hardware signals. The behaviour of those signals is essentially the same as in VHDL and Verilog. From the point of view of the system level designer, communication using only hardware signals is insufficient to capture highly abstract features that are apparent at the system level. For that reason new communication mechanisms were added leading to the second release. For example, in SystemC 2.0, a semaphore or a mutex are available to protect shared data used by communicating modules. Modules can also communicate using FIFOs, another type of communication not directly supported in SystemC 1.0. Moreover, it is possible for users to define

their own communication mechanisms. In order to do that, the user has to define a new *channel* and its *interfaces*. Channels and interfaces are two new kinds of objects in SystemC 2.0. A channel defines a communication mechanism. An interface is used to connect a port of a module to a channel. FIFO, mutex and semaphore are examples of built-in SystemC 2.0 channels.

With abstract communication mechanisms, it becomes possible to work at levels of abstraction that are higher than RTL. *Transaction level modelling* (TLM) [71] is an example of such a level. In TLM, processing elements typically communicate by procedure calls, whereby a whole packet of data can be transferred. Implementation details seen at the register transfer level are omitted. As a result, TLM simulations are much faster than RTL simulations. Extensive research in modelling embedded systems at the TLM level has been carried out recently [73–78].

Designs in SystemC can be simulated with testbenches. A testbench typically contains a module that generates inputs for the design under test (DUT), the DUT itself, and a module that checks the outputs of the DUT. It is often the case that a testbench does not contain a module for checking outputs; instead the designer manually checks outputs. Designs at various levels of abstractions can be simulated.

It is worth mentioning that the development of interfaces and channels in SystemC 2.0 was significantly influenced by another system-level language based on C, called SpecC [79]. While SystemC libraries are defined using the standard C/C++ syntax, SpecC extends the standard ANSI-C by adding new constructs to support hardware and system-level features. HardwareC [80] and Handel-C [81] are also examples of languages that extend C in order to support hardware design.

2.13 Ptolemy

Ptolemy is an environment for modelling heterogeneous systems. Embedded systems are often heterogeneous in that they encompass different models of computation. Their heterogeneity also stems from the fact that they are composed of software and hardware components.

The basic object in Ptolemy is called *block*. In the first version of Ptolemy [83] blocks are described in C++. In the second, expanded version [84] blocks are described in Java. Blocks communicate with the external environment through port-holes. Blocks can use different methods for communication. Other objects in Ptolemy are derived from blocks. It should be noted that Ptolemy intensively uses object oriented programming.

Ptolemy has several domains. Domains represent models of computation. Examples of the domains in Ptolemy are process networks (PN), dynamic dataflow (DDF), Boolean data flow (BDF), synchronous dataflow (SDF), discrete event (DE), synchronous/reactive (SR). SDF is the most developed domain in Ptolemy. In fact, Ptolemy's predecessors only supported the dataflow model of computation. A *star* is an object derived from a block. A star always belongs to a certain domain. An object called *galaxy* can be formed from stars. Galaxies are hierarchical – they can contain other galaxies.

Another object derived from a block is called *target*. There are two types of targets in Ptolemy: simulation and code generation. An object called *scheduler* is needed to determine the order of execution of stars. Like stars, schedulers are also related to domains. Different domains have different schedulers. Finally, putting together stars, galaxies, schedulers and a target results in a complete application is called *universe*. What is done with a universe depends on its target – it can be either simulated or code can be generated from it (C and VHDL code generators are available).

The key feature of Ptolemy is that stars from different domains can be mixed. This allows multiple models of computation to be present in a single system. It is important to emphasize that one domain can be embedded inside another, hierarchically, but different domains cannot be at the same hierarchical level. A domain can be embedded inside another by the mechanism called *wormhole*. Externally, a wormhole behaves according to the semantics of the domain it is in, just like any other star or galaxy that belong to that domain. However, the internal behaviour of a wormhole is entirely different because it contains another domain with different semantics. The interface between the domain that resides in a wormhole and the external domain in which the wormhole is placed is called *Event Horizon*. Two conversions take place on the event horizon. One of them is the conversion between particles that cross the interface from one domain to another. For example, if DE (discrete event) is embedded inside SDF (synchronous data flow), a particle going from SDF to DE has to be attached a time stamp, i.e. an SDF particle has to be transformed into a DE particle. The converse is true in the opposite direction. The other conversion is to do with schedulers. The schedulers in two interacting domains have to synchronize.

The above description gives an idea of how different models of computation are treated in Ptolemy. Different models of computation are kept pure. The main focus is placed on the interface between different models. The opposite approach in handling multiple models of computation is to compose them more tightly, mix their properties so that basically a new model results, which contains everything that its ingredients contain. The developers of Ptolemy criticize this brute-force approach because it results in what they call in [20] *emergent* behaviour. The designer expects that the resulting model of computation will have the properties of the models from which it was created. Instead, it often happens that the resulting model exhibits unexpected and undesired behaviour. It is also difficult to analyse the model.

On the other hand, it is questionable whether the approach in Ptolemy entirely preserves the properties of combined models. Some models are so different that it is really difficult to make them interact in a meaningful way.

In order to illustrate a system specification in Ptolemy this section focuses on combinations between FSM and other models of computation. The reason for choosing this combination is that it probably has the greatest chance of successfully representing mixed control/dataflow systems.

The combination of FSM and other models of computation is called **charts* [85] in Ptolemy (pronounced star charts) where *** denotes a wildcard indicating that it is possible to combine various models of computation with FSM. In **charts* the concurrency semantics is decoupled from the FSM model. The concurrency semantics depends on the model that is combined with FSM. It is possible to have multiple concurrency semantics in a single system. In [85], the authors discuss combining

FSM with DF, DE and SR. In this section the FSM / DF combination is mostly discussed, in particular combining FSM with SDF (synchronous data flow).

Figure 2.23 shows a system that is composed of FSM and dataflow domains. There are five levels of hierarchy marked as (a), (b), (c), (d), (e). A hierarchical decomposition as in Fig. 2.23 is typical in Ptolemy. The hierarchy can be arbitrarily deep. Each hierarchical level belongs to a particular domain or model of computation. The model of computation determines how the blocks at that level communicate. For example the communication at level (a) is governed by SDF. Two or more models of computation cannot coexist in parallel at the same level of computation. In terms of what happens inside blocks, the black-box approach is taken. Blocks can be defined with different models of computation. It is only the communication between blocks that is governed by a single model of computation.

In *charts, an FSM can be used to describe a block. A state of an FSM can also be refined to another model. One of the problems with refining a state of an FSM in Ptolemy is that the refining system has to complete an iteration when the state is left. This is possible in SDF, for example, where a complete cycle can be defined as an iteration. A complete cycle returns buffers to their initial state. It would be more difficult to refine an FSM state with a Kahn process network, since a Kahn process network operates on infinite streams and cannot be divided into iterations.

A *homogenous* SDF actor consumes / produces only one token on each input/output. A homogenous SDF actor fits naturally into the FSM model. It is more difficult to handle a *nonhomogenous* actor. For example, in Fig. 2.23 actor A consumes two tokens from input a and one token from input b , so it is a nonhomogenous actor (numbers in brackets indicate how many tokens are consumed or produced). It is refined to an FSM. At the FSM level, the two tokens from the input a are treated as events. It was decided [85] to order multiple events with the notation from the language Signal. a denotes the most recent event while $a\$1$ denotes the second most recent event. Suppose that the FSM is in the state α and the SDF actor fires. The FSM takes the transition from α to β if both a and $a\$1$ are present. The presence or absence of an event is explicitly encoded in the corresponding SDF token. The transition produces the output event x which appears on the output $x(1)$ of the SDF actor A. The SDF actor D at level (d) is also refined to an FSM. The output of SDF actor B produces two tokens $y(2)$, but the FSM below produces only one event y . The interpretation is that y is present while $y\$1$ is always absent since it is not mentioned in the transition. The events y and $y\$1$ at level (e) are connected to $y(2)$ at level (d).

At levels (d) and (e) an FSM is activated whenever the SDF block refined by the FSM fires. It is often useful not to activate an FSM when the higher level SDF block fires. This is especially the case when one or more states of the FSM are refined by SDF blocks. Therefore there are two types of firings associated with SDF/FSM combinations [85]. *Type A* firing is when an SDF block fires but the FSM that refines it is not activated, instead the system that refines the FSM is activated. *Type B* firing is when an SDF block fires and the FSM that refines it is activated, as well as the system that refines the FSM. For example suppose that the schedule of the SDF network at level (a) is {D,C,C,C,E} and the FSM at level B is in state β . The first two firings of C at level (a) will be of type A. The FSM at level (b) will simply ignore all events. It will just pass tokens down to level (c). The third firing of C will

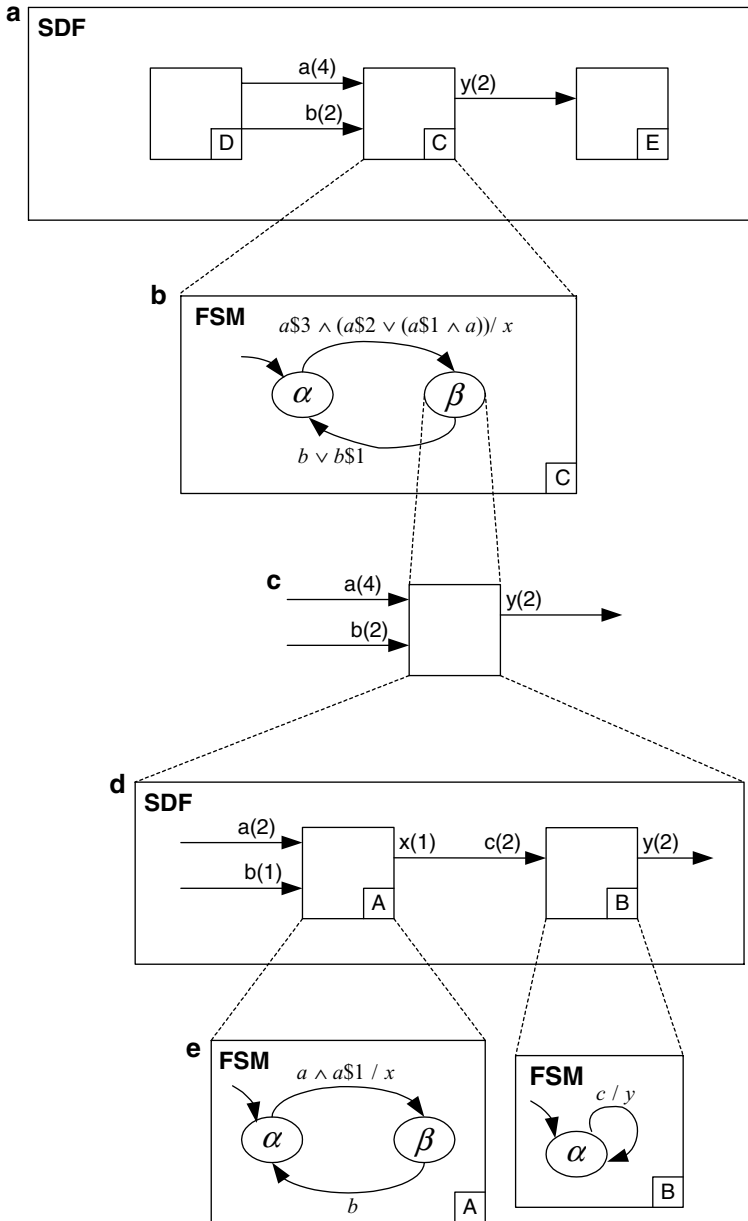


Fig. 2.23 Mixing FSM and SDF in Ptolemy

be of type B. This time the FSM will pass tokens to level (c), but it will also respond to them by producing the transition from β to α if the condition $b \vee b\$1$ is true.

Difficulties in combining FSM and SDF arise when states of an FSM are refined into SDF subsystems that do not consume the same number of tokens. If the FSM

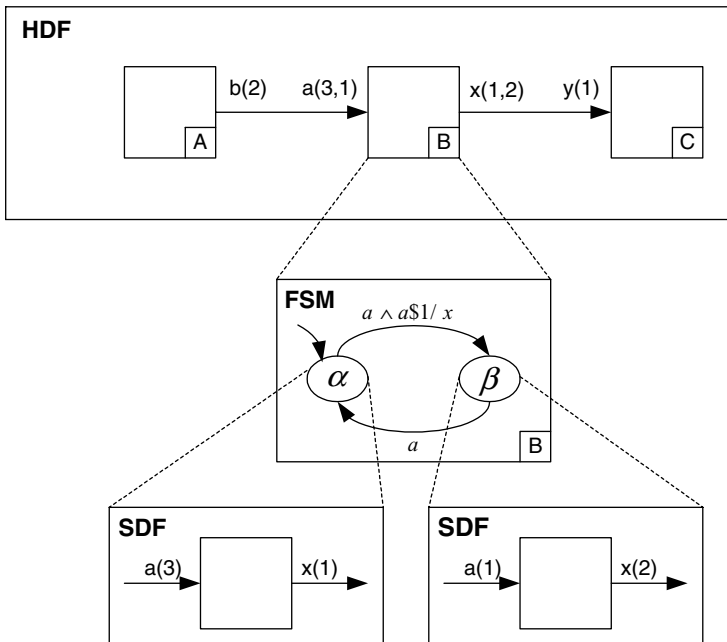


Fig. 2.24 HDF with FSM and SDF

refines an SDF block, the number of tokens consumed by the SDF block cannot be constant. Therefore, it seems that it is necessary to use dynamic dataflow. However, in that case, all advantages of SDF such as static scheduling would be lost. In [85], a new model called *heterochronous dataflow* (HDF) is proposed. In HDF, the numbers of tokens consumed by inputs and the numbers of tokens produced by outputs are not constant, but the list of possibilities is finite so static scheduling is still possible. An example of a system that uses HDF is shown in Fig. 2.24.

In Fig. 2.24, state α of the FSM is refined into an SDF subsystem that consumes three tokens and produces one token, while state β is refined into an SDF subsystem that consumes one token and produces two tokens. As a result of that, block A at the HDF level can either consume three tokens and produce one token or consume one token and produce two tokens. It is possible to do scheduling of an HDF system statically. Details can be found in [85].

In Ptolemy, the basic idea is to keep different models of computations separate and concentrate on their communication. Having each level of hierarchy defined by one model of computation should make the analysis of complex systems easier. Furthermore, hierarchical composition in Ptolemy encourages clean, modular design. Unfortunately, interaction between different models of computation is rarely straightforward. Models have to adjust to each other in order to be able to communicate, which often leads to restrictions in their expressive power.

Embedded Systems Design Based on Formal Models of
Computation

Radojevic, I.; Salcic, Z.

2011, XV, 183 p., Hardcover

ISBN: 978-94-007-1593-6