Chapter 5

Functional Design

In this book, we chose to given particular emphasis to functionality because it exposes some of the most interesting aspects of a rigorous design methodology and because a deep analysis of the representation of the functionality of the design can expose design errors early. We will show that the mathematical foundations used in this chapter can be of use also in the representation of platforms and of their instances.

The use of documents written in natural languages, such as English, has been traditionally adopted to hand over system specification from customers to developers. This approach has been the source of a number of problems for the supply chain because of the ambiguous nature of natural language specifications since they are not executable and a precise conformal test is impossible.

We argued that for true system level design, we must be able to capture the functionality of the design at the highest possible level of abstraction without using implicit assumptions about an implementation choice. Since a most appealing feature of capturing the functionality of the design is to be able to execute it on a computer for verification and analysis, it is natural that designers and researchers cast the problem in terms of design languages.In this chapter, we first give a historical perspective on some of the approaches used in the hardware and software embedded system community to specify in executable forms complex designs. While this approach is certainly a step forward, we believe that specifying and handling concurrency and heterogeneous behaviors needs a more abstract point of view. The rest of the chapter is dedicated to abstract system behavior formalisms called *Models of computation.* A precise definition of these concepts is still open for debate.

We simply say that a model of computation provides two essential features: a *syntax* to write structurally correct models and an associated *semantics* to given a meaning to them. A model of computation defines the way in which computation inside processes, communication among processes and coordination of concurrent actions are executed. These components define the set of possible behaviors of a model.

5.1 Historical Perspective

5.1.1 Languages for Hardware Design

Because of the popularity and the efficiency of C, several approaches for raising the levels of abstraction for hardware design are based on C and its variants[1]. C has been used successfully to represent the high-level functional behavior of hardware systems and to simulate its behavior. A simulation amounts to running the compiled C code and hence is very fast, limited by the speed of the simulation host and by the quality of the compiler. *The main problem with this approach is the lack of concurrency and of the concept of time in C.* In fact, hardware is inherently concurrent and time is essential to represent its behavior accurately. In addition, C has been designed with standard programming application in mind and in its constructs, it relies upon standard communication mechanisms through memory that are inadequate, to say the least, for hardware representation. For these reasons, a number of derivative languages have been introduced, some with more success than others. The pioneering work in this field was done by De Micheli and students [52, 104] who discussed the main problems of using C as a hardware description language from which a Register Transfer Level (RTL) description could be synthesized. Commercial offerings such as Mentor CatapultC, Celoxica Handel-C [31], C2Verilog, Bach [54] defined a subset of ANSI C to do either synthesis or verification.

More recently, there has been a strong interest in languages that are derived from C or C++ and that *explicitly capture the particular aspects of hardware.* In particular, SystemC [133, 68] and SpecC [46] stand out.

SystemC is a class library of the C++ language while SpecC is a super set of ANSI C. Both have special constructs to represent hardware concepts, to support concurrency and a rich set of communication primitives. The resemblance to C of these languages is then mainly syntactical while their semantics are quite different.

The usefulness to a designer of a language is not only the capability of representing his/her intent but also the support given in terms of verification, formal analysis and synthesis. Both SystemC and SpecC are not directly synthesizable nor formally verifiable. To verify formally or synthesize a design expressed in System C or SpecC, we need either to subset the language (SystemC) or to go through a set of manual or automatic transformations to yield a synthesizable representation (SpecC).

SystemC is used mainly for simulation. Several SystemC simulation engines are available (one is open source). Of course the performance of the simulation and the leverage in design representation comes from the level of abstraction of the models described in these languages. There are a few synthesis tools that generate RTL from SystemC-like languages. Companies like Mentor, Synopsys and Forte Design offer tools in this domain. The jury is still out as the degree of acceptance of hardware designers for this kind of tools as the quality of the results is mixed. Sometimes the quality is comparable and even better than the one of human designs, sometimes it is definitely worse.

An alternative approach to raising the level of abstraction is to extend existing RTLs to cover constructs that help in describing and verifying higher levels of abstraction. In this context, SystemVerilog [132, 58] has been accepted with interest by the hardware design community as it builds upon the widely used Hardware

[1] We recommend to the interested reader the excellent survey paper by Edwards for a critical review of C-derived languages [50].

Description Language (HDL) Verilog. While SystemVerilog can be used almost for everything SystemC can do, the opposers of this approach list as drawbacks the difficulty of using SystemVerilog for system designers with software background and the difficulty in expressing some of the concepts important to system design.

An interesting approach to hardware synthesis and verification is offered by BlueSpec [19]. BlueSpec takes as input a SystemVerilog or a SystemC subset and manipulates it with technology derived from Term Rewriting Systems (TRS) [10] initially developed at MIT by Arvind. The idea of term rewriting was developed in computer science and is the basis of several compiler techniques. It offers a nice environment to capture successive refinements to an initial high-level design that are guaranteed correct by the system. The appeal of the approach is that designers can maintain their intent throughout their design process and control the synthesis steps. This is a significant deviation from the work on high-level synthesis pioneered by the CMU school [45] where from a high-level design representation, both architecture and micro-architecture implementations were automatically generated. The general consensus today is that the chasm between an algorithmic description and an implementation is just too wide to be able to obtain a good implementation.

5.1.2 Languages for Embedded Software Design

Traditionally, abstract analysis and design have been divorced from implementation concerns, helped by the Turing abstraction that has simplified the task of programming by decoupling functionality and the physical world. Because of this dichotomy, today, embedded software designers use low-level facilities of a real-time operating system (RTOS), tweaking parameters such as priorities until the system seems to work. The result is, of course, quite brittle.

The main difficulty faced by embedded software designers are programmer productivity and design correctness. This is particularly true when the software is to be distributed over a network, as in the case of avionics and automotive applications. In this case, side effects due to the interaction of different tasks implemented on different elements of the design make the verification of the correct behavior of the system very difficult since the traditional paradigm of adjusting priorities does not apply.

Most of the design methodologies in use in the embedded system industry have been borrowed by standard software design practices and emphasis is placed on development processes rather than on the content of the design. However, in the industrial sectors where safety is a primary concern, there has been an interesting shift towards the use of languages that have intrinsic correctness guarantees and for which powerful analysis and synthesis methods are possible. Ironically, while in the case of hardware system design the state-of-the-art is the attempt at adapting languages like C and C++ typically used for software, the most advanced paradigm in embedded software design is borrowed from hardware design! In particular, the most elegant approach is the extension of the synchronous design paradigm to software design.

Synchronous Languages . The goal of synchronous languages is to offer strong formal semantics that make the verification and the code generation problem easier *by construction*. The work of the French school on Synchronous Languages [15], and in particular, Esterel [16], Lustre [75], and Signal [72] with their industrial derivatives (e.g., Esterel Studio and Lustre-SCADE from Esterel Technology and Signal-RT Builder from TN-Software Division of Valiosys), has been at the forefront of a novel way of thinking about embedded software. The synchronous design languages approach has made some significant in-roads in the safety critical domain especially in avionics.

The synchronous languages adopt the synchronous hardware design paradigm in the sense that they assume that computation occurs in two separate phases, computation and communication, that do not overlap. Often this concept is described as communication and computation taking zero time, while a better way would be to say that the actual time taken for communication and computation does not matter from a functional point of view as they do not overlap. The notion of time is then one of *logical time* that relates to phases of behavior and sequencing. In this model, behavior is predictable and independent of the actual implementation platform. This is similar to synchronous hardware where as long as the critical path of the combinational hardware between latches takes less time than the clock cycle, the behavior of the hardware is independent of the actual delays of the combinational logic.

The adoption of these languages in the embedded software community has been limited to safety-critical domains such as aviation and automotive, but I believe that they could and should have a larger application.

5.2 Examples and Uses

As we will see later in this chapter, there are many formalisms to capture a design specification. Each formalism bears intrinsic properties that make it suitable to model only a particular class of systems. In Section 5.3.2 we clarify this statements with two examples of specifications for two different classes of systems: data-dominated and control-dominated. The choice of the *language*, or formalism, is very a very important decision since each language also comes with a set of tools such as simulation, verification and synthesis, that only depend on the language semantics and can, therefore, be applied to any specification. To show the advantages of using a precise formalism, in this section we show two examples of system specifications: a vending machine and a cell-phone keypad controller. Consider, for instance, a simple automatic machine that sells coffee. The machine accepts one-dollar bills up to two dollars, returns change in quarters and sells only two products: a small coffee for one dollar and a large coffee for $ 1.25.

How do we formally capture the behavior of the vending machine? Natural languages, like simple documentation written in English, have been traditionally used as the starting point of the design process. The specification is written together by the customer (i.e. the person or company that commissions the implementation of the system) and the developer (i.e. the company that will implement and deliver the final product). The specification is then given to the design team that proceeds with the implementation. The behavior of the vending machine can be described as follows:

> *If a customer inserts one dollar and asks for a small coffee, then serve the coffee and wait for the next customer.*
>
> *If the customer inserts two dollars and asks for a large coffee, then serve the coffee, return three quarters change and wait for the next customer.*
>
> *If the customer inserts two dollars and asks for a small coffee, then serve the coffee, return four quarters change and wait for the next customer.*

The first obvious drawback of this specification is that it is not executable. There is no way for a designer who receives this document to execute (i.e. simulate) the specification and have a better understanding of the way in which the system is supposed to work. To build an executable model, the English document has to be interpreted by the designer who has to use another language, such as C, to write a program that can be then executed on a host machine. Having the executable model allows to test the behavior of the system under different input scenarios and use the simulation traces as a reference to validate the final implementation.

However, this is only one of the problems with the English specification. For instance, we may want to make sure that the system satisfies two important properties:

1. no coffee is served for free,
2. no money is taken without serving coffee.

This two properties are about the correctness of the specification of the system and should be satisfied by the specification and by all its refinements, meaning by its final implementation. The only way to check if the description given in natural language satisfies these two properties is by manual inspection. Checking the first property is not impossible, even if some misunderstanding may arise. In fact, the specification does not mention what happens if a coffee is requested without inserting any money. The designer, driven by her common sense, will make sure during the implementation that no coffee is served for free. This is an example of ambiguity that, in this simple example, would cause not harm, but that could be fatal in complex systems for safety critical applications. The second property is more difficult to verify. Consider the case where a customer inserts one dollar in the vending machine but does not ask for any coffee. This scenario is not taken into account by the specification that only captures the cases where a customer inserts money and asks for coffee. The customer, who may not be familiar with the vending machine, may think that the machine is broken and leave, and maybe file a complain to get his dollar back. A second customer could insert a one dollar bill and ask for a large coffee making the system violate two properties: money were taken without serving coffee, and a large coffee was served to a customer who did not paid the full amount. Of course, avoiding that a customer makes a “gift” to another customer cannot be avoided. However, the machine could be design in such a way that, if a customer does not ask for a coffee within a certain amount time from the insertion of a bill, then the money are returned (and maybe some indications are given to correctly operate the machine). This example shows that the possible ambiguities in the specification may be eliminated by describing the behavior of a system under all possible circumstances, which makes the specification quickly difficult to read and implement correctly. For complex systems, it is not unusual to find specifications with thousands of scenarios. In real cases, it is often the case that a designer judges the specification to be incomplete and asks for more information to be included in the description of the system. However, the complexity of the specification makes the process of interpreting the functionality difficult and error-prone.

5.2.1 Finite State Machine Description of the Vending Machine

To provide a way to capture the vending machine behavior in a formal way we need to define a language that is based on mathematical rules which do not leave space for subject interpretations. A language is defined by two important concepts: its *syntax* and its *semantics*. The syntax defines the symbols used to compose words in the language, and words are used to compose sentences (or programs in the computer science jargon.) Moreover, the syntax also defines the valid words and the valid sentences. The semantics gives a meaning to valid sentences. There are two different ways of defining the semantics of a language, namely *denotational* and *operational* semantics. A denotational semantics defines the meaning of a program as a function from its inputs to its outputs. This function can be defined by a relation or more simply by a table where each column defines the outputs to associate to a given set of inputs. A pioneering work in this area is the one from Scott and Strachey [130, 125]. An operational semantics provides an algorithm that executes a program belonging to a certain language. Operational semantics were first defined by Plotkin [115]. In this section we use Finite State Machines to describe the behaviors of the vending machine example. This type of description is and operation one as an algorithm can be given to execute the machine. We will also show how this description can be used to automatically verify that the two properties defined in the previous section are satisfied by the specification.

The formal definition of a finite state machine (FSM) is given in Section 5.4.1. For the sake of this discussion, we define a finite state machine as a labeled graph where each node represents a state and it is labeled by a name, and each edge represents a transition between two states and it is labeled by a letter drawn from a finite set Σ that represents the events that the FSM can generate, or is sensitive to. The alphabet of the vending machine has six elements:

- d is the event that signals the insertion of a one dollar bill;
- sc is the event that signals the request of a small coffee;
- hc is the command that serves half cup of coffee;
- lc is the event that signals the request of a large coffee;
- fc is the command that serves a full cup of coffee;
- q is the event that signals the issuing of a quarter change;

Figure 5.1 shows the finite state machine capturing all possible behaviors of the vending machine VM. The semantics of this model is very intuitive. The state machine has an initial state *idle* marked by an arrow with no predecessor node. From any state, the FSM can make a transition to a successor state only if the event that labels the transition occurs. For instance, if the FSM is in state *idle* and a one dollar bill is inserted (i.e. event d occurs), then the FSM makes a transition to state s_1. Notice that when the machine is in state s_2, the state transition diagram does not prescribe any specific transition with a d label. In this informal introduction to finite state machines we simply assume that such input sequence is not accepted by the model. Without this assumption, if a third dollar bill were to be inserted in the machine, nothing is said about how the vending machine would behave. In Section 5.4.1, we will introduce the notion of *incompletely specified* finite state machines where this assumption is not made.

Besides being able to simulate the model by providing events and observing state transitions, finite state machines are amenable to automatic verification. The verification problem is reduced to the one of exploring the set of reachable states of a model. Given a finite state machine, we can in principle compute the set of all its execution traces, where a trace is a sequence of state transitions. Consider,

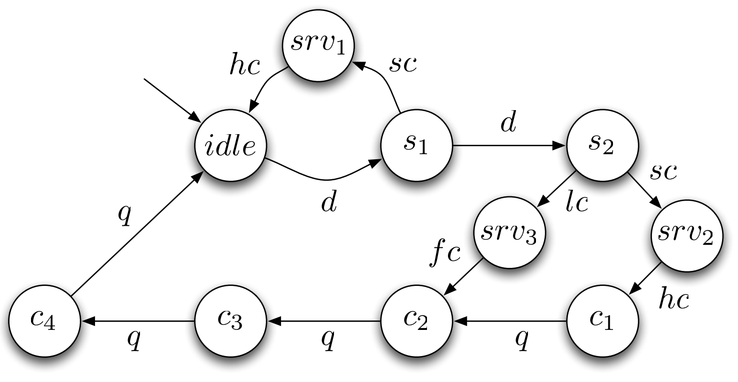


Figure 5.1: A finite state machine encoding the valid sequences of events of the vending machine.

for instance, the properties of our vending machine. Any correct execution trace that reaches state srv_1 must contain at least one d event, asserting that the user has inserted one dollar in the vending machine. To check this property, we build a companion finite state machine that executes synchronously with the vending machine and that tacks customer credit (i.e. total amount of money that have been inserted but not used).

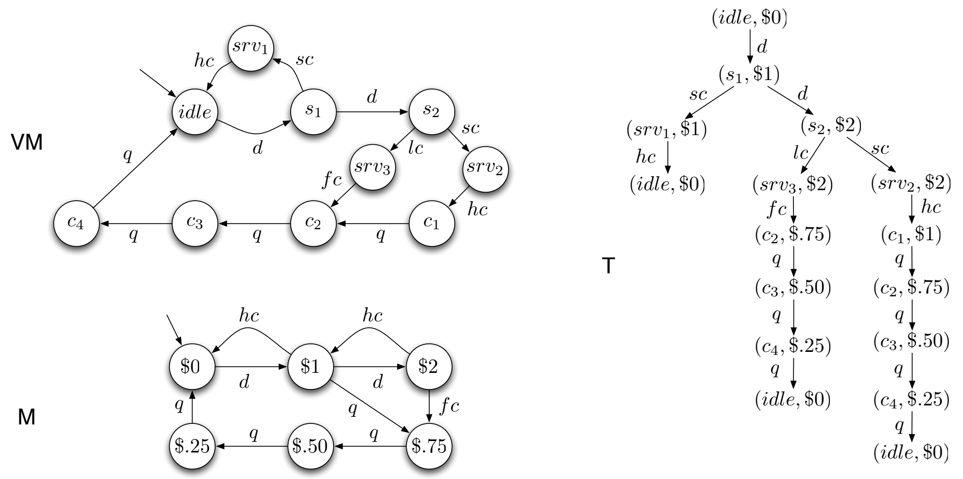


Figure 5.2: The vending machine VM and the the companion FSM M that is used to count the total amount of money inserted in VM but not used.

Figure 5.2 shows the combined model where VM is the vending machine, and M is the credit counter. The states of M are labeled with the total user credit. The set of labels on the arcs of M is a subset of the set of labels on the arcs of VM. The two finite state machines move synchronously meaning that, upon occurrence of an event, they change state together. For instance, if M is in state \$0, VM is in state *idle* and event d occurs, the two finite state machines move together to states

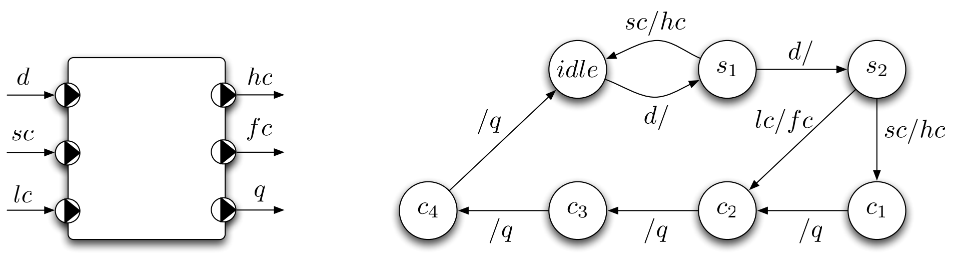


Figure 5.3: Mealy machine description of the vending machine.

\$1 and s_1, respectively.

We can now state the property that no coffee is served without the appropriate amount of money being paid as illegal combinations of states of M and VM as follows:

> ***P1:*** *For all possible traces of events, the following conditions are always false:*
>
> - *VM is in state srv_1 and M is in state \$0, or*
> - *VM is in state srv_2 and M is in state \$0, or*
> - *VM is in state srv_3 and M is in state \$0 or \$1*

To verify that this property is satisfied, the set of pairs of states that can be reached by the two state machines must be explored. The reachable states are encoded in a tree, T, shown in Figure 5.2. Each node in T is labeled with a pair of states of VM and M, respectively. The root node corresponds to the pair of initial states of VM and M. If a dollar is inserted in the vending machine, then VM makes a transition to s_1 and M makes a transition to \$1. The new state, with label $(s_1, \$1)$, is reachable from the initial state and is added to T together with an arrow denoting the transition. Checking that the vending machine satisfies property P1 is equivalent to checking that the pairs $(srv_1, \$0)$, $(srv_2, \$0)$, $(srv_3, \$0)$ and $(srv_3, \$1)$ are not present in T (i.e. these states are not reachable.)

So far, we have treated the alphabet of an FSM as a set of actions. The alphabet can be partitioned in two sets I and O of the input and output events, respectively. Figure 5.3 shows the input-output representation of the vending machine. A denotational way of describing the behavior of the state machine with inputs and outputs is the following. The set of inputs is $I = \{d, sc, lc\}$ and the set of outputs is $O = \{hc, fc, q\}$. The state space is the one of Figure 5.1, namely $S = \{idle, s_1, s_2, c_4, c_3, c_2, c_1\}$. Now, we define two functions to capture the behavior of the vending machine:

- a state transition function $\delta : I \times S \rightarrow S$ that associates a new state to each pair of current state and input;
- an output function $\lambda : I \times S \rightarrow O$ that associates and output to each pair of current state and input.

For instance, $\delta(d, idle) = s_1$ means that if the current state is $idle$ and a one dollar bill is inserted, the new state becomes s_1. Similarly, $\lambda(sc, s_1) = hc$ and $\delta(sc, s_1) = idle$ means that if a small coffee is requested while in state s_1 (i.e. after inserting a one-dollar bill), the machine serves half a cup of coffee and goes back to the initial state. Since the behavior of the state machine is captured by two functions that, to each combination of input and state associate one and only one value for the new state and output, this description is called *deterministic*. In non-deterministic descriptions, one pair of an input and a state can have multiple next states, in which case δ is no longer a function but it is rather a relation (we will explain this in details in Section 5.4.1).

With the introduction of some additional notation, we can use state transition diagrams also in the case where the distinction between inputs and outputs is made explicit. The change is notation is the following. Labels on the edges of the state transition diagram are of the form i/o where i is an input event and o is an output event. Two other labels are allowed: $/o$ which means that the transition is taken without any event occurrence, and $i/$ which means that no output event is emitted by the state machine on that transition. Figure 5.3 shows the state transition diagram of the modified vending machine.

5.2.2 Finite State Machine Description of the User Interface for a Mobile Phone

In this section we show an example of specification of the user interface of a mobile phone. A popular interface is the one adopted by Nokia phones. When the phone is idle, the keypad is locked preventing any unchecked use of the cellphone. To unlock the phone, the user has to select the "Menu" option and then press the "*" (star) key. When in unlock mode, the user can again select the "Menu" option and get access to a number of other functions like browsing contacts and making phone calls.

Instead of using plain English to describe the functionality of the the user interface, we may attempt to use the C programming language which can then be compiled into code on a host machine and simulated. Without entering into specific problems regarding the automatic verification of the properties of a C model, the main drawback of a C program is its sequential nature. The following program is a C description of the user interface.

```
int main(){
    enum{LK,UNLK,LK_MENU,MENU,CONTACTS,CALL...} state ;
    state = LK ;
    string command ;
    while( 1 ) {
        read(&command);
        switch(state) {
        case LK:
            if (command == "Menu") {
                state = LK_MENU ;
            }
            break ;
        case LK_MENU:
            if (command == "*") {
                state = MENU ;
            }
            break;
        ...
        case MENU:
            if (command = "call") {
                string number = getnumber() ;
                call(number) ;
                state = LK ;
            }
            break;
         ...
        }
    }
}
```

The `state` variable enumerates the set of possible states. The program is an infinite loop that reads an input `command` from the user and depending on the actual state performs an action and computes the next state. For instance, if the current state is `LK` and the `Menu` command is issued, the next state becomes `LK_MENU`. At this point, if the star command is issued, the phone is unlocked and the user can select one of the possible options provided by the menu. It is possible, for instance, to make a phone call. If this options is selected, the user interface program gets a number to call (also provided by the user either directly or through a selection of a contact stored on the phone) and runs a service `call`. When the service returns, the user interface goes back to the `LK` state. This model has the undesired behavior that during a phone call the user interface is inactive because the thread of control is inside the `call` function. What if the user is asked to check for the phone number of a person during a phone call? This feature requires the user interface to run concurrently with the `call` service.

A possible solution to this problem is to use different threads for the two functions. The following modification of our model should allow to use the menu options while in a phone conversation:

```
case MENU:
    if (command = "call") {
      string number = getnumber() ;
      create_thread(&tid,call,number) ;
      state = LK ;
    }
break;
```

A new thread is created to run the `call` function while the user interface can update its state and continue running. Unfortunately, the use of threads has its own drawbacks. Threads, differently from processes, share the same memory space. Therefore, they can potentially use the same memory locations to store data. Since the execution of each thread depends on the memory content (that holds their states), subtle side effects can happen that can lead to non-deterministic behavior.

Set	**Elements**
I	*menu*, *star*, 1, 2, ...
O	*startcall*, *sendsms*, ...
S	*lk*, *nowstar*, *unlk*, *contacts* ,...

Table 5.1: Definition of the inputs, outputs and states of the state machine describing the keypad controller.

The hard task of a multi-threading programmer is to make sure that side effects are avoided and that the execution of the program is deterministic.

One question that the reader may ask is whether it is possible to build an automatic tool that, given a sequential C program is able to extract the parts that can run concurrently and generate threads automatically, also checking and avoiding possibly unsafe memory usage. The reality is that this is not possible in the general case because the problem of verifying that two threads do not interfere (i.e. do not use the same memory location for storing their data) is undecidable.[2] Unfortunately, many questions, such as if a program will eventually finish, are undecidable in the context of the C programming language, which makes verification difficult at best. This is a strong argument in favor of using formal models that can be automatically "processed" to carry on verification and code generation.

The keypad controller can be modeled as a state machine, although this will not result in the desirable feature of being able to brown the list of contacts while in a phone call. In the finite state machine model of computation there is no notion of "concurrency", meaning the concurrent execution of two state machines. Luckily, extensions ot this model exist that allow to define concurrent actions to happen (see for instance the Statechart model of computation in Section 5.4.1).

The inputs to our controller are all the commands coming from the keypad, while the outputs are the commands sent to other functions to start specific activities (such as making a call). For instance, the key 1 represents an input while the *startcall* command that starts a phone call is an output. The set of inputs, outputs and states are described in Table 5.1.

The state diagram of the keypad controller is shown in Figure 5.4. The initial state is *lk*. If the user presses the *menu* button, the state changes to *nowstar* and, upon pressing the *star* key, the keypad controller switches to the *unlk* state. In this state the menu can be entered by pressing *menu* again (state m). A number of options are available while in this state. For each selection made by the user, the state machine moves into a new state that represents a position in the tree of options. This model is monolithic in the sense that the entire controller is described by one state machine.

Figure 5.5 shows a model where the lock/unlock finite state machine, and the machines that describe the different menu options are kept separate. Therefore, we need to define the way in which they interact. For each transition, we use a label e/a where e is the event that enables the transition and a is the event that is activated, or emitted, when the transition is taken. Different state machines communicate by synchronizing on the events that are globally visible. For instance, when the main controller is in state *nowstar* and the key *star* is pressed, an event u is emitted. This event triggers the transition of the contact service to move from the *idle* state to the *unlk* state. If the *menu* button is pressed, the main controller switches to the

[2]An undecidable problem is one for which an algorithm that solves the problem is not guaranteed to finish its computation.

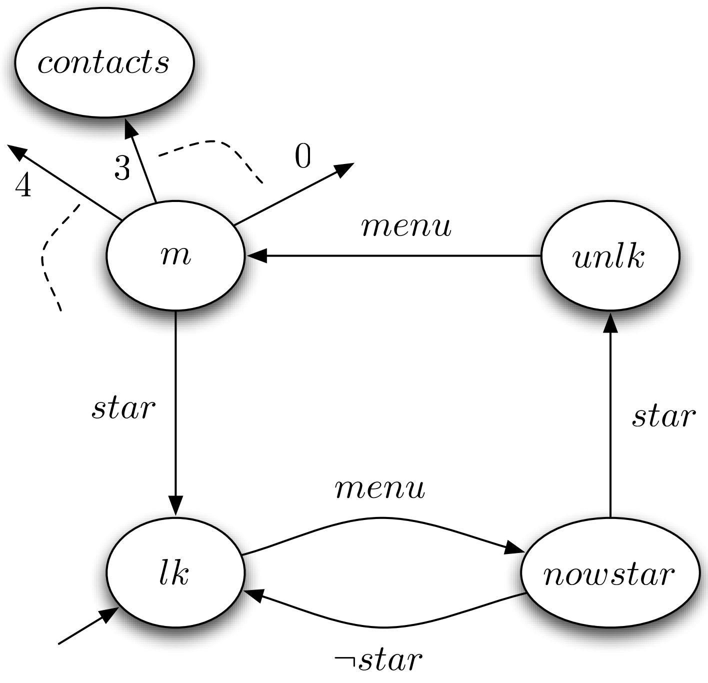


Figure 5.4: State diagram representation of the keypad controller.

m state and the contact service switches to the *mainmenu* state. Now, if option 3 is selected, the contact service switches to the *contacts* state, meaning that the contact menu is entered, and event *start* is emitted. This event triggers a transition of the main controller to a state *srv* indicating that some service has been initiated by the user. The main controller exits this state when the user finishes to use the service and event *done* is emitted.

5.3 Models of Computation

A design process starts with the idea of the services, or functionality, that a system should provide to the end users. The initial idea, that is in general abstract and that lacks many of the details of the final implementation, has to be captured in the form of a design specification. A team of designers will use the specification to drive the design process toward a satisfying and cost-effective implementation.

Typically, in the early stages of the design process, the specification is not detailed. It is refined in the course of the design by adding details that the engineers request to their customers or to the marketing department. Often times, clarifications are needed simply because the specification documents may contain ambiguity that must be resolved before implementation. Ambiguities arise form the non-objective interpretation of statements written in natural language or to the interpretation of block diagrams that describe the system or part of it. Block diagrams are widely used to exchange information about the partitioning of a system into sub-systems, each performing a specific piece of the entire functionality. A

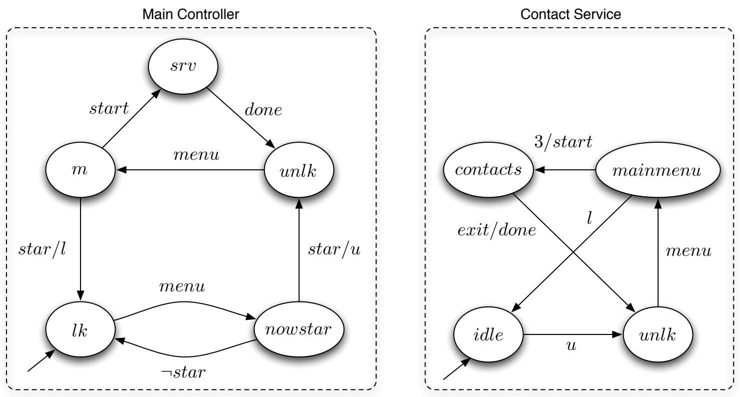


Figure 5.5: State diagram of the keypad controller divided into a main controller and a set of services. In this figure, an example of part of the contact service is represented.

block diagram is a model of a system described using a graphical language. Usually, the syntax of such language contains labeled circles or squares, and arrows connecting them. The interpretation of its meaning, or better the semantics of the language, is left to the intuition of the reader: circles or squares represent certain input/output functions and arrows represent data dependencies among functions. Because the meaning, even if intuitive, is never precisely defined, questions may arise: How are data exchanged? How are the functions computed and scheduled?

In this chapter we introduce a set of languages, that we refer to as Models of Computation (MoC), to capture design specifications. The semantics of these languages will be formally defined, leaving no ambiguity in their interpretation. We start with a general definition of a MoC.

Definition 5.3.1 (Model of Computation) *A model of computation is a mathematical description that has a syntax and rules for the computation of the behavior described by the syntax, also referred to as* semantics. *A model of computation is used to specify the semantics of computation and concurrency.*

The definition is explicit in mentioning two important ingredients of a model of computation: syntax and semantics. The syntax defines the symbols of the language and the valid composition of symbols (i.e. the valid programs). The semantics defines the rules that give a meaning to a program. The rules, for instance, define the way in which data are exchanged or accessed, and the order in which processes can be executed. The reader may be familiar with dataflow graphs that are commonly used in signal processing. The following example shows the use of the dataflow language to describe a simple Finite Impulse Response (FIR) filter.

Example 5.3.2 (A FIR filter) An Finite impulse response filter is a digital filter that processes a sequence of input samples of a signal and generates a sequence of output samples. The general law to compute the output sequence from the input sequence is the following:

$$y_n = \sum_{i=0}^{k-1} x_{n-i} h_i$$

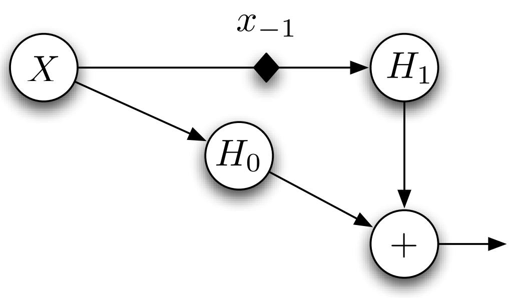


Figure 5.6: Block diagram of the FIR filter.

where $x = \{x_i\}$ is the input sequence (with x_j denoting the j-th element of the sequence) and h is the discrete-time impulse response. Notice that h is a finite sequence of length k which is also called the number of taps of the FIR filter. Consider the case of a simple FIR filter with two taps:

$$y_n = h_0 x_n + h_1 x_{n-1}$$

Figure 5.6 shows a block diagram of a system implementing the FIR equation. Circles represent processes while arrows represent communication channels. Process X generates the input stream. Processes H_0 and H_1 multiply the input samples by a constant (h_0 and h_1, respectively). Process $+$ adds the input streams and generates the output.

The block diagram does not define the behavior of the system. Here, we informally list a set of rules (i.e. the semantics) associated with the block diagram that define the set of its valid execution. Each arrow is a first-in-first-out (FIFO) queue, and black diamonds are used to denote the presence of tokens in the queue. For example, an initial token is placed in the queue between process X and process H_1. This token defines the value of x_{-1} that is necessary to compute y_0. Each process executes according to the following rule: the process is enabled when there is at least one token in each of its input queues; if a process is enabled, then it can fire by reading one token from each input queue and generating one token to all its output queues. A special attention should be paid to process X. In fact, this process does not have any input. In this example we neglect this technical detail and we just assume that process X is always enabled.

Figure 5.7 shows the first few snapshots of a possible execution of the block diagram that complies with these rules. Figure 5.7a shows the state of the system after firing process X. Two tokens with value x_0 are generated on both its outputs. Figure 5.7b shows the state of the system after firing both H_0 and H_1. Notice that these two processes can be fired together because there are no dependencies among them. Finally, process $+$ is enabled and fires (Figure 5.7c) generating the first output y_0.

In defining the semantics of the block diagram in Example 5.3.2, we used some concepts, such as process, enabling and firing, without being too much precise about them. In Section 5.3, we will present several models of computation to describe the behavior of systems and we will be very precise in defining their semantics. However, the definition of a model of computation comprises always three elements:

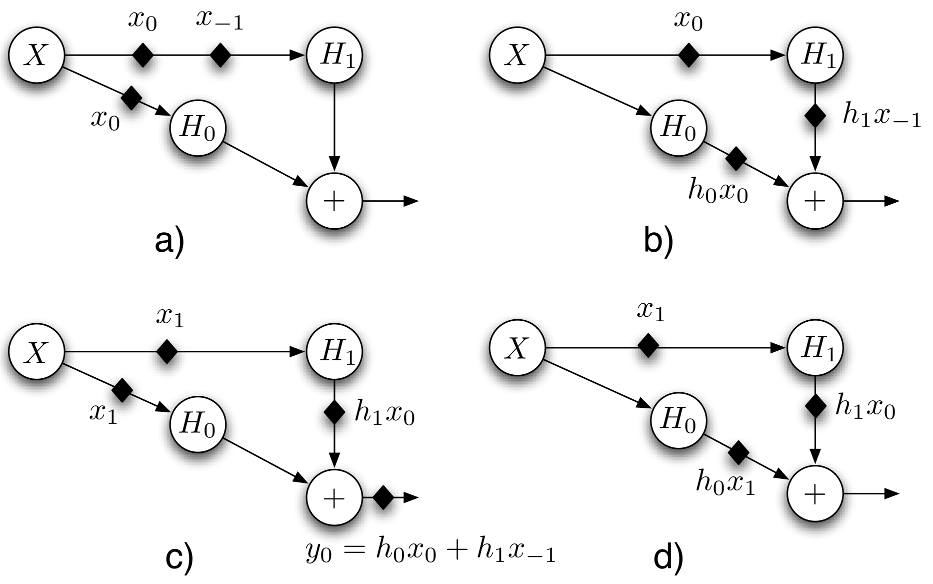


Figure 5.7: Snapshots of the execution of the FIR filter

the actions (or computation), the coordination of the actions and the exchange of data (or communication).

5.3.1 Elements of a Model of Computation

A model of computation is defined by three fundamental constituents: *computation*, *coordination* and *communication*. Example 5.3.2 already presented all these elements: The actions, or computations, correspond to firing processes and executing a function that computes output tokens starting from input tokens; the communication semantics is defined in terms of FIFO queues on the arcs of the block diagram, and the coordination among processes, i.e. the order in which processes are fired, depends on the distribution of tokens on the FIFO queues.

Computation is specified by a set of objects that transform data. Such objects can be functions from the inputs to the outputs of a process, or transitions of a finite state machine. *Coordination* rules define "when" actions can be executed. The coordination rules among actions (e.g execution of a process, or transition from one state to another in a state machine) are very important to define concurrency in a model of computation. The *Communication* semantics specifies how data are exchanged among processes (e.g. shared variables, unbounded FIFO queues, blocking or non-blocking read-write).

5.3.2 How to choose an MoC

In Section **??**, we will study a wide variety of models of computation that differ for their syntax and semantics. The main reason for having such a variety of models is that each of them offers features that may be well suited to capture specifications in one application domain, but that limit the expressiveness of the language and consequently its suitability to other application domains.

Each MoC allows to expose relevant properties of a class of systems (i.e. the ones that can be captured using the language provided by the MoC). These properties can be leveraged to develop analysis and design tools that depend only on the the MoC and not on the specific system model. The availability of tools is also another criterion to decide which MoC to adopt in a design process. Before listing all the criteria that should be considered when comparing MoCs and deciding the most suitable in a particular application, we give a couple of examples that illustrate how different application domains demand for different modeling paradigm.

Example 5.3.3 (Control-dominated applications) Generally speaking, a controller is sensitive to some *input events* and implements a *control law* that reacts to the inputs and, depending on the value of some *internal state variables*, generates some *output events*. For instance, if we want to control the temperature T of a room to track a desired value T_0, we use a temperature sensor and a controller. If the temperature drops below T_0, the controller turns the heater on. When the temperature rises above T_0, the controller turns the heater off. Clearly, the input events are $T < T_0$ and $T > T_0$. It is also clear that the controller needs to know if the heater is on or off in order to generate the right output event. In fact, the sole input event $T < T_0$ does not contain enough information to make the decision of whether the heater should be turned on or off. The *state* of the controller is an internal variable used to keep the memory of what has happened previously to the system. Using the state of the system, we can decide if the heater should be tuned on upon receiving the $T < T_0$ event.

Protocols are also examples of these class of systems. Figure 5.8 shows as simple system composed of two nodes equipped with a wireless radio. The sequence of symbols that are received through the wireless channel are demodulated by the radio and sent to the upper layer of the protocol stack. The upper layer is composed of two interfaces. Interface TX is used to send data while RX is used to receive data. The radio completely hides the presence of the wireless channel. In fact, if the channel were wired, the interfaces would not behave any differently from the wireless case.

The *UML sequence diagram* in Figure 5.8 shows the way in which the interfaces interact. When the TX interface receives a new data to be sent over the wireless channel, it issues a request to the RX interface of the other node and waits for an acknowledgement. If the acknowledgement arrives, the TX interface sends the data and waits for another acknowledgement. The sequence diagram also shows the interaction of the TX and RX interfaces with the external world (the upper layer of the protocol stack). This diagram describes only one possible communication scenario but may others are possible. For instance, the RX interface may send back a negative acknowledgement signal to alert the transmitter that the received data was corrupted and should be sent again.

All these scenarios can be captured by the state machines shown in Figure 5.9. The FSM description of the two interfaces relies on a graphical language. The syntax of the language is composed of labelled circles and labelled directed arcs. An arc must connect two circles. The labels in the circles are strings while the labels on the arcs have the following syntax: *input condition/ output list*. The input condition is any proposition on the inputs while the output list is an assignment of values to the output.

The semantics these language defines the meaning of the labels, the way in which different machines interact (i.e. the communication semantics), and the meaning

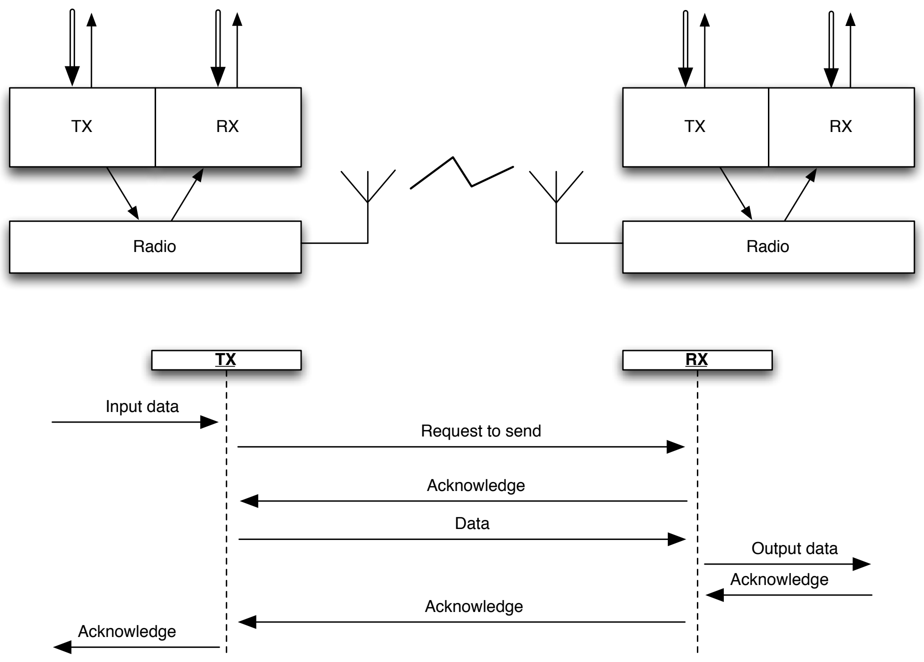


Figure 5.8: A simple system with a transmitter and a receiver and the sequences of messages exchanged between them.

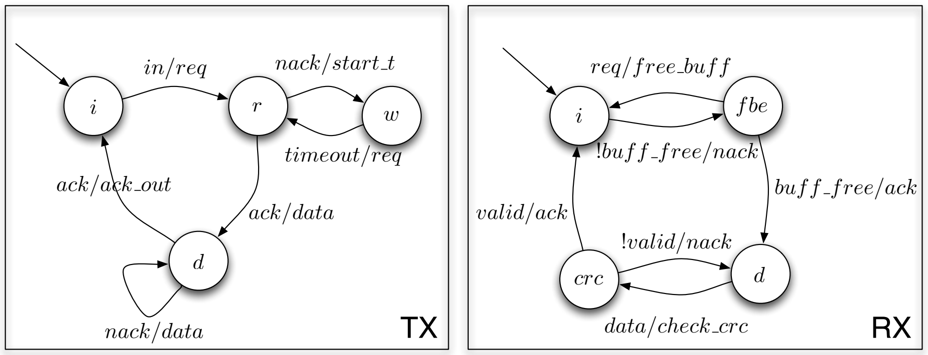


Figure 5.9: The finite state machines capturing the behavior of the TX and RX interfaces.

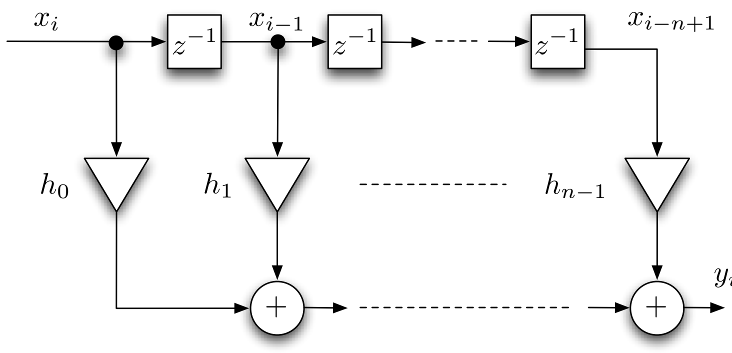


Figure 5.10: The classical graphical representation of a Finite Impulse Response filter.

of the state transitions represented by the arcs, i.e. the conditions that allow a transition to be executed. We postpone to Sections **??** the formal description of the semantics of finite state machines. Informally, the FSMs communicate by means of instantaneous events that are exchanged on memoryless connections. While in a state, the FSM observes its inputs and continuously checks whether some input condition on the outgoing arcs is satisfied. When the input condition on an arc is satisfied, the transition can be taken and the outputs in the output list are emitted (i.e. instantaneous events are generated on those outputs). It is possible to verify that the sequence of actions in the sequence diagram of Figure 5.8 complies with the FSM specification of the two interfaces.

The protocol example shows an important characteristic of control-dominated applications. For instance, suppose that a data has been sent to the receiver. The transmitter is waiting for the final acknowledgement. Therefore, the transmitter is in state *d* and the receiver is either in state *crc* or *d*. Also, suppose that the data has been correctly transmitted and received so that the CRC test is successful. If the upper layer sends signal *in* to the transmitter before the final *ack* from the receiver has been sent, such event is lost and the transmitter will simply receive and *ack* and sit in the *i* state. If the *in* signal arrives after the final ack, the transmitter will first go to the *i* state and then starts a new request going to the *r* state.

The behavior of the system is quite different depending on the relative order of the input events.

In control-dominated systems, the state plays an important role in deciding the system's behavior and should be captured explicitly. The behavior is sensitive to the relative order of the events in the system.

Example 5.3.4 (Data-dominated systems) Finite Impulse Response filters, like the one of Example 5.3.2, fall in the brad class of data-dominated applications. These systems are described by functions operating on streams of data. Traditionnaly, filters are represented by diagrams like the one shown in Figure 5.10.

This type of diagrams are widely used by designers of digital signal processing systems to describe their algorithms. The meaning associated to this diagram is very intuitive. The block with label z^{-1} is a delay. A delay is implemented as a buffer with one space only that holds the previous value of a sequence. The other blocks carry out simple operations. The triangle is a constant scaling that multiplies each element of the input sequence by a constant factor. The circles are adders that sum the two input sequences. If in_1 and in_2 are the two inputs and *out* is the output, then for all indexes i, $out_i = in_{1,i} + in_{2,i}$.

Notice that there are some hidden assumptions that define the meaning to this model. It is implicitly assumed that when a new element of a sequence arrives at the input of a block, the tokens held by the delay are shifted from the input to the output. Moreover, the constant scaling and the adders can perform their computation before the next element of the sequence appears at the input. Finally, the black dots are duplication points such that the same value is copied to the inputs all all the blocks connected to the same point. In Section **??** we will see a model of computation to capture this kind of systems called *data-flow*.

These systems have states. In fact, the past n values of the stream x contribute to the computation of the current value of y. Because the value of the i-th sample x_i is usually discrete (e.g. between 0 and 255), the number of states that of this system is finite. In principle, a state machine may be defined that describe the filter. Each state corresponds to one combination of values of the n samples held in the buffers. However, such representation would be impractical. The state in this systems is represented directly by the memory elements holding the previous values of the stream.

Consider the adder element. Independently from the relative arrival time of $in_{1,i}$ and $in_{2.i}$, the output is always going to be out_i. The result of the computation does not really depend on the relative timing of the two inputs as long as the i-th element of both sequences are taken for the computation of the output. In fact, there is no real time information associated with the streams, but there is only a ordering relation between the inputs and the outputs.

Example 5.3.3 and example 5.3.4 suggest that depending on the type of systems that we are trying to capture, one modelling language is more appropriate that another. The adoption of one MoC rather than another should be guided by the potential benefits that a modeling language offers. Given the application domain, an appropriate MoC should have the following properties

Expressiveness It should be possible to model all the systems that pertain to the application domain of interest.

Simplicity The use of the MoC should not be hardened by unnecessary complication that would negatively counterbalance its benefits.

Verifiability It should possible to *verify that the model is correct with respect to some properties.* In order to perform verification it is sufficient to check that a malicious behavior does not belong to the model. Notice that, even if defined in this way verification is always possible, the verification of a properties could require and exaustive search of the set of all possible behaviors of a model.

Sinthesizability The rules governing the model of computation are known and can therefore be used at lower level of abstractions to generate models that include implementation details and that mimic the same behavior of the original model. This allows a model to be synthesized on the selected target platform. Moreover, it is possible to verify that the implementation behavior, once purified by the added details, is the same as the original model.

Tools availability and development The use and/or development of tools for synthesis, verification, simulation etc. can be based on the MoC and not on the specific model. Therefore, their techniques can be applied independently of the specific design.

Unfortunately there is a *trade-off* between all this properties. It is very likely that making an MoC very expressive would make verification very difficult or even impossible! One way of getting all these benefits together is to mix many simple MoCs in a rigorous way each dedicated to the specification of small parts of the same system and keep verification and synthesis separate for each part.

We should make
for each one of the p
using the example j
duced. We should s
models can have the
pressiveness but one
easier to use for a ce
plication.

5.4 Models of Computation for Embedded System Design

5.4.1 FSM

The Finite State Machine (FSM) MoC is widely used both in hardware and software design. In this MoC, a function is described in terms of states and transitions. When the system is in a given state, external events force the system to make transitions from one state to the next. The state can be thought of as a variable that keeps memory of what as happened so far and decides what can be executed next.

The specification of a function often contains an implicit reference to system states. For instance, consider the handshaking protocol between a transmitter and a receiver described in Example 5.3.3. The protocol can be informally specified as follows:

> The transmitter issues a request of sending a packet. *If a request has been sent*, and *only after an acknowledgement has been received*, the transmitter sends the packet and waits for a notification that the packet has been correctly received. *If a packet has been sent* and a notification does not arrive, the packet has to be retransmitted. If a notification arrives, the transmitter can *start over* with a new transmitting cycle.

The specification makes explicit reference to past actions taken by the transmitter. Thus, the transmitter sub-system must necessarily keep track of its *state*: idle, a request has been sent, a request has been acknowledged, a packet has been sent, a packet has been acknowledged and so on. The definition of the system states is usually done incrementally during the design of a system. Transitions from one state to another are triggered by events.

In this chapter we first introduce a classical definition of pure finite state machine. In this type of model, transitions are triggered by events and events are emitted when a transition is taken. There is not type associated to the inputs or outputs of a finite state machine, and no computation can be done when a transition is taken. A system is typically partitioned into sub-systems that communicate using inputs and outputs. We will define the composition of finite state machines and show the semantic problems arising from feedback connections.

The pure finite state machine model may not be expressive enough to model complex systems where even simple computations, such as incrementing the value of a variable, are common. The model can be extended to allow computation to be done when a transition is taken. Other limitations are the inability to express concurrent activities and hierarchy. Extensions that come with these two features exists and are nowadays available in commercial tools such as Stateflow from The Mathworks an Statemate from Telelogic.

Input/Output/State Description

The conceptual model of a finite state machine is graphically shown in Figure 5.11. It has a set of inputs, a set of outputs and internal states. The first type of FSM that is worth studying is usually referred to as conventional or pure. In this type of FSM, inputs and outputs are simply sets of symbols. Each input is not associated with a specific type and can be thought of a a boolean value. For instance, the FSM in Figure 5.11 has the set of inputs $I = \{i_1, \ldots, i_n\}$. Each input may be a Boolean value (i.e. can assume value 0 or 1), or it can be an event which is present (if the event occurred) or absent (if the event did not occur). In practice, however, the presence or absence of an event will also be encoded by a Boolean variable, making the distinction of the two case not relevant to our discussion. The formal definition that we will use is more abstract. It will only consider sets of symbols.

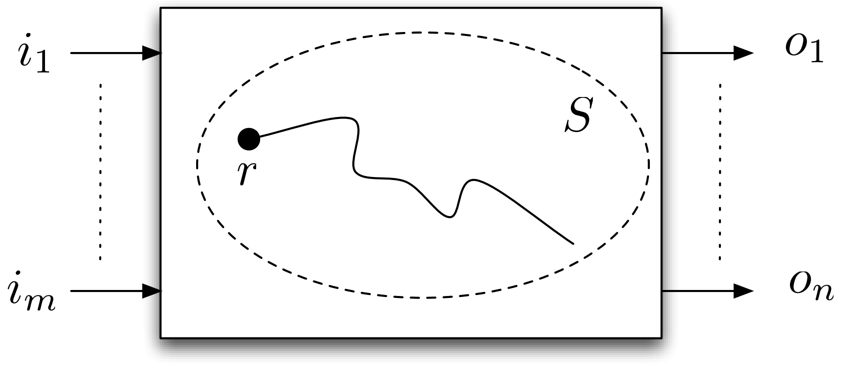


Figure 5.11: A graphical representation of the elements of a finite state machine model.

Definition 5.4.1 (Deterministic Finite State Machine) *An FSM is a tuple $(I, O, S, r, \delta, \lambda)$ where $I = \{i_1, ..., i_m\}$ is a finite set of inputs, $O = \{o_1, ..., o_n\}$ is a finite set of outputs, $S = \{s_1, ..., s_k\}$ is a finite set of states, $r \in S$ denotes the initial state, $\delta : 2^I \times S \rightarrow S$ is the state transition function and $\lambda : 2^I \times S \rightarrow 2^O$ is the output function.*

The operation of an FSM starts from the initial state r. Notice that the initial state is part of the specification that is different for different values of r. The transition function determines the sequence of states that an FSM walks through during an execution. It maps a state and a subset of the inputs to another state. Semantically, the transition function determines the next state of the FSM starting from the current state and the inputs. The output function determines the outputs of the FSM that result from a transition. More specifically, given the current state of the FSM and depending on the set of "active" inputs, a subset of the output can be "emitted" (thereby becoming enabled).

This formal description of an FSM does not include any notion of time. Often, there are preconceived ideas on the execution of an FSM that come from its hardware implementation. When an FSM is implemented in hardware, transitions from one state to the next are associated with the system clock that naturally establishes a relationship between the behaviour of an FSM and time. Definition 5.4.1 makes no reference to a triggering clock and neither to time.

A finite state machine is *completely specified* if the two functions δ and λ are defined for each state $s \in S$ and for any subset of the inputs $i \in 2^I$. Because λ and δ are functions, for each combination of current state and inputs there is a uniquely defined next state and set of outputs. This type of FSMs are called *deterministic* as opposed to *non-deterministic* FSMs where multiple next states may result from the same combination of current state and inputs (see Definition 5.4.3.

Example 5.4.2 (Seat Belt Controller) We want to specify the behaviour of a seat belt alarm controller. The informal specification can be given in natural language as follows:
If the driver turns on the key and does not fasten the seat belt within five seconds, then an alarm beeps for five seconds or until the driver fastens the seat belt, or until the driver turns off the key. There are clearly three states in this system. A rest state where the system remains until the driver turns on the key. An alarm state that is entered after the key has been turned on. The system return to the rest state if some countermeasures are taken by the driver: either the driver turns the key back off or puts the set belt on. If such countermeasures are not taken within five seconds, the alarm starts beeping. The FSM describing this system is shown in Figure 5.12.

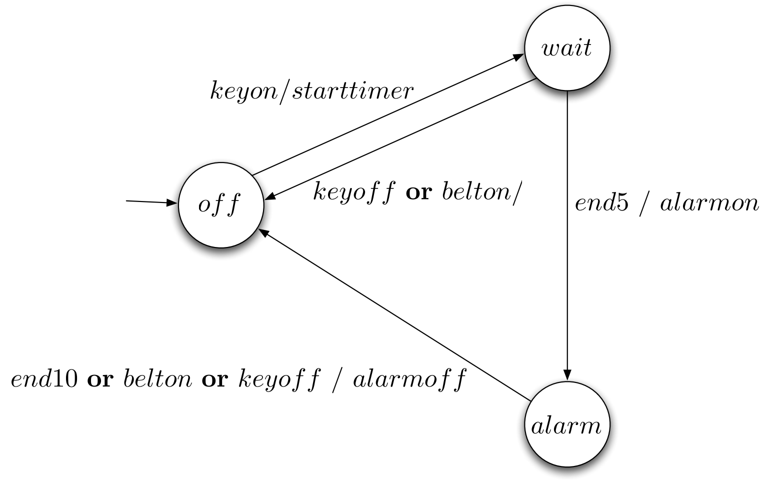

$wait$
$keyon/starttimer$
off
$keyoff$ **or** $belton/$
$end5\ /\ alarmon$
$end10$ **or** $belton$ **or** $keyoff\ /\ alarmoff$
$alarm$

Figure 5.12: State diagram of the belt controller FSM.

The graphical syntax used in this example has the following meaning. Circles represent states and are labelled with the state's name. Arcs represent transitions between states. Each arc is labelled with a pair $< In >\ /\ < Out >$ where In is a subset of the inputs and Out is a subset of the outputs. Notice that in Figure 5.12 we have used the more convenient keyword **or** called disjunction. The disjunction i_1 **or** i_2 represents three sets: $\{i_1\}$, $\{i_2\}$ and $\{i_1, i_2\}$. In the diagram of Figure 5.12 we did not explicitly described what happens under any possible input which would make the specification of the FSM incomplete. However, we followed the convention that self transitions from one state to itself are left implicit. For instance, when the state is off and the driver puts the belt on, the state does not change. The implicit self transitions occur where there is no condition that is satisfied. Considering the implicit transitions, the state machine is completely specified.

Following Definition 5.4.1, the seat belt controller states, inputs, outputs and initial state are defined by the following sets:

- $S = \{off, wait, alarm\}$
- $I = \{keyon, keyoff, belton, beltoff, end5, end10\}$
- $O = \{starttimer, alarmon, alarmoff\}$
- $r = \{off\}$

Consider the transition from off to $wait$. Its label says that the transition is enabled, and therefore can be "executed" or "taken", when the key is turned on. When this transition is taken, the output *starttimer* is "emitted", or is "enabled". In terms of the two functions δ and λ, it means the following:

$$\delta(\{keyon\}, off) = wait, \qquad \lambda(\{keyon\}, off) = \{starttimer\}$$

Similarly, the transition from *wait* to off is specified as follows:

$$\begin{array}{ll} \delta(keyoff\}, wait) = off & \lambda(\{keyoff\}, wait) = \emptyset \\ \delta(belton\}, wait) = off & \lambda(\{belton\}, wait) = \emptyset \\ \delta(\{keyOff, belton\}, wait) = off & \lambda(\{keyoff, belton\}, wait) = \emptyset \end{array}$$

A few comments should be made on Example 5.4.2. The reader may be mislead by the specification of the seat belt controller that includes the declaration of a real-time deadline between the key is turn on and the alarm starts beeping. We made explicit the existence of an external timer in Figure 5.12 that can be initialized by the *starttimer* output of the controller and that provides the $end5$ input. However, if the timer is off and the $end5$ event is emitted after 7.5 seconds, the FSM would still go through the same sequence of states. This example should convince the reader that the FSM does not have any notion of time attached to it. We also point out that for a given state and subset of inputs, the seat belt controller has a uniquely defined next state. This is the reason why this type of finite state machines are called *deterministic*: for the same input sequence, the FSM produces always the same output sequence.

There are cases where a non-deterministic description of an FSM may be useful. Two cases are worth mentioning: unknown behaviours and don't cares. In many cases, a full characterization of the environment of an embedded system is not available simply because the phenomena governing the environment evolution are not fully understood (or difficult to model). Even when the environment behavior may be perfectly characterized, its description may be very detailed and an abstraction of it could potentially simplify the verification of correctness of the embedded system specification. The abstraction may group states together. The abstraction process may result in two different states be reachable by one state for the same combination of inputs.

In other cases, assumptions on the possible sequence of inputs to a state machine may be available to the system modeller. For instance, in Example 5.3.3 we may know that a new input *in* will not be present unless *ack_out* is generated. We implicitly used other assumptions such as the mutual exclusion between the two inputs *nack* and *ack*. What should the next state be when both *ack* and *nack* are present at the input. This condition will never happen and we may define the next state to be any of the states in the state machine. This is a degree of freedom that can be leveraged in during the implementation into hardware or software as the logic that implement the next state function may be better optimized. At the same time, it is no longer true that the next state is uniquely defined by the combination of the inputs. This type of finite state machines are called non-deterministic.

Definition 5.4.3 (Non-Deterministic Finite State Machine) *A non-deterministic FSM is a tuple $(I, O, S, R, \delta, \lambda)$ where $I = \{i_1, ..., i_m\}$ is a finite set of inputs, $O = \{o_1, ..., o_n\}$ is a finite set of outputs, $S = \{s_1, ..., s_k\}$ is a finite set of states, $R \subseteq S$ denotes the set of initial states, $\delta \subseteq 2^I \times S \times S$ is the state transition relation and $\lambda \subseteq 2^I \times S \times 2^O$ is the output relation.*

Definition 5.4.3 is very similar to Definition 5.4.1 with the key difference that λ and δ are relations instead of functions. The other key difference is that the initial state is not required to be unique. Consider a non-deterministic FSM where $(\{i\}, s, s') \in \delta$ and $(\{i\}, s, s'') \in \delta$ for some input i and some states s,s',s''. This specification contains two valid next states s' and s'' for the same combination of inputs and present state. The next two examples show how non-determinism can be used.

Example 5.4.4 (Environment Assumptions) In Example 5.4.2, a reasonable assumption to make is that after *starttimer* has been emitted, the external timer will emit *end*5 before emitting *end*10. Therefore, we could change the δ function into a relation such that :

$$\begin{aligned} (\{end10\}, wait, off) &\in \delta \\ (\{end10\}, wait, alarm) &\in \delta \end{aligned}$$

Similarly we could change the λ function into a relation such that:

$$\begin{aligned} (\{end10\}, wait, \{starttimer\}) &\in \lambda \\ (\{end10\}, wait, \{alarmon\}) &\in \lambda \\ (\{end10\}, wait, \{alarmoff\}) &\in \lambda \\ (\{end10\}, wait, \{starttimer, alarmon\}) &\in \lambda \\ \ldots \end{aligned}$$

Even if the behavior of the entire system is not affected by this new definition of δ, it is possible to use the new transitions in the hardware and software synthesis algorithms to generate compact and efficient implementations. Intuitively, if the input is $\{end10\}$ we *don't care* what the next state is and we *don't care* what the output is either. Thus, we can decide both next state and output in such a way that the final implementation is simplified.

Similarly, if we knew that after *alarmon* has been emitted, the external beeper is no longer sensitive to the *alarmoff* signal, we could include the *alarmon* output in the transition between *alarm* and *wait*.

Example 5.4.5 (Unknown Behaviors) Sometimes, the behavior of a state machine is not known. The seat belt controller of Example 5.4.2 is embedded in an environment whose behavior is difficult to predict. For instance, to analysize or simulate the seat belt controller, a model of the driver is needed. However, it is difficult to predict the sequence of inputs that *any* driver would provide to the seat belt controller. Modeling one possible sequence of inputs would lead to an incomplete validation of the controller that is supposed to operate correctly under all possible driver behaviors. A non-deterministic specificaiton of the driver helps in this case. It answer to the modeling need where *the driver can do either action a_1 or action a_2* and end up in two different states.

The most general non-deterministic state machine for the driver model has one state only $S = \{s\}$, and a set of outputs $O = \{keyon, keyoff, belton, beltoff\}$. There is only one transition $(\emptyset, s, s) \in delta$ and the output relation contains the following elements:

$$\begin{aligned}
(\emptyset, s, \{keyon\}) &\in \lambda \\
(\emptyset, s, \{keyoff\}) &\in \lambda \\
(\emptyset, s, \{belton\}) &\in \lambda \\
(\emptyset, s, \{beltoff\}) &\in \lambda
\end{aligned}$$

This specificaiotion of the driver behavior allows us to verify correctness of the set belt controller under all possible inputs rather than relying on a selected subset of test cases. This is also an example of abstraction in the sense that a model for *any* driver has been abstracted and simplified into a one state model with any actions possible.

Because a function is a particular type of relation, one may think that non-deterministic FSMs are more expressive than deterministic ones. It has been proven [117] that this is not the case and that they are equivalent. Given a non-deterministic FSM, it is possible to construct a deterministic FSM that admits the same set of behaviors, albeit having a larger set of states.

The popularity of the FSM model among hardware and software designers is due to the simplicity of specification. The graphical syntax presented in Example 5.4.2 is very intuitive and easy to understand even by designers that are not fully aware of the rigorous, implicit Definitions 5.4.1 and 5.4.3. Besides the ease of use of this formalism, many tools are availalbe for verification and synthesis that make the use of this model very appealing.

Finite State Machine can be *automatically verified* against properties. A property is tipically a proposition expressed in a particular logic (see Section 5.6.2 for an example of logic). The property identifies a set of states or a computation path as a sequence of states. More generally, an interesting question is the following: *does it exist a sequence of inputs such that state s can be reached starting from r?*. This is known as the reachability problem which is decidable for FSMs. Being able to answer this question allows for instance to compute the set of all reachable states S of an FSM. Another question that can be answer is the safety question. Given a subset of states $B \subset S$ called *bad states*, the safety verification problem can be stated as follows: *does it exist a sequence of inputs such that one of the states in B can be reached starting from r?*. Typically, safety verification can be turned into a reachability problem by checking that $S \cap B = \emptyset$. Efficient algorithms for reachability analysis can be used to solve this problem. Verification is a powerful method that can check a model against any possible input. It is fundamentally different from simulation where the model is stimulated by test cases that represent a finite set of input sequences. The applicability of verification techniques is only limited by the complexity of the model and of the verification algorithms.

Finite state machines can be *automatically synthesized* into hardware and software. The possibility of applying synthesis techniques is fundamental since the equivalence of specification and implementation is guaranteed by construction (provided that the synthesis algorithm is correct). The verification problem need to be solved at the specification level only because the implementation is guaranteed to be "equivalent"[3]. Since the implementation contains far more details than the specification, the other advantage is that the verification problem is easier at the specification where "less" conditions need to be checked.

The FSM model has also some limitations. The number of states and transitions of the system can become unmanageable especially when the input set contains many elements. Additional constructs such as hierachy and composition can help making the description of a finite state machine model more compact. Further, the model that we present does not allow any sophisticated computation to be done either while in a state or when taking a transition. The Extended Finite State Machines (EFSM) model of computation extends the basic FSM model to provide additional features such as typed inputs and outputs, state variables, and aritmetic computations on them. A model that embodies all these extensions is StateCharts, originally presented in [76] and summarized in Section 5.4.1.

Composition of Finite State Machines

Complex systems have many states and many transitions among them. Capturing a system with a single FSM is not practical and not natural. Designers partition the functionality of a system into sub-systems, each having inputs and outputs. Subsystems are then interconnectet to provide the system functionality. Given a set of FSMs, one for each subsystem, and their interconnection, what is the behavior of the whole system? The behavior is described again by a FSM resulting from the *composition* of the FSMs of each sub-system.

Consider a set of FMs $\{M_1, \ldots, M_n\}$, where S_i is the set of states of M_i. The state of their composition is the vector of states $(s_1, \ldots, s_n)$ where $s_i \in S_i$. If the FSMs do not interact, and if each state in S_i is reachable, then each state in the cross product $S_i \times \ldots \times S_n$ is a reachable system state. In this case, if each FSM has k states, then the total number of state resulting from the composition is k^n which is exponential in the number of sub-systems. This explonential dependency of the number of states from the number of sub-systems is known as the state explosion problem.

When the sub-systems interact, some states in the cross-product are not rachable because of the dependencies that exist between inputs and outputs. To precisely define the behavior of the composition of FSMs, the *communication semantics* must be defined. Communication among FSMs is assumed to be *synchronous*, meaning that transitions are taken together in lock step. The system proceeds in steps. At each step, each FSM executes one of the enable transitions.

Example 5.4.6 (Seat Belt Controller and Timer) The seat belt controller has two inputs *end*5 and *end*10 coming from a timer. The timer is another sub-system shown in Figure 5.13. It is a finite state machine that walks through a chain of states counting from 1 to 10. It has three inputs: *start* coming from the seat belt controller, and *sec* coming from an external device that emits the *sec* trigger every second. We do not model the external device which is part of the environment.

The initial state of the timer FSM is 0. If a start command is received, the FSM moves through a chain of states, advancing state each time a new *sec* input is present. After five steps (corresponding to five seconds of the external timer) from

[3]We did not formalized the concept of equivalence. The term is used informally here.

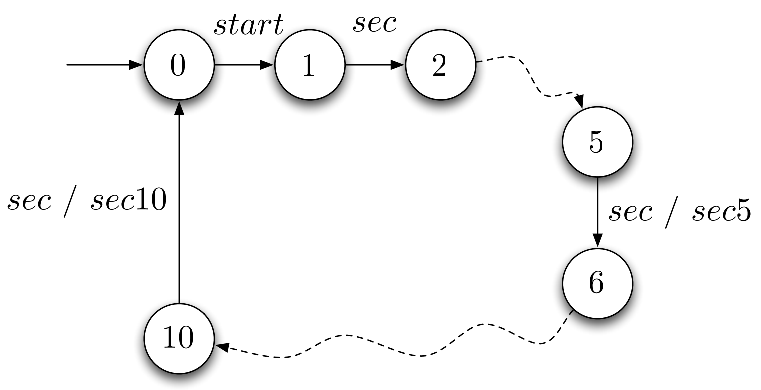


Figure 5.13: State diagram of the timer FSM.

the last *start* the output *sec*5 is emitted. Similarly, after ten seconds from the last *start* the output *sec*10 is emitted. Figure 5.14 shows the interconnection of the seat bel controlelr and the timer FSMs. Connections are depicted by dashed arrow as they are mathematical concepts rather than physical connections. We will see later that connections are constraints between inputs and outputs.

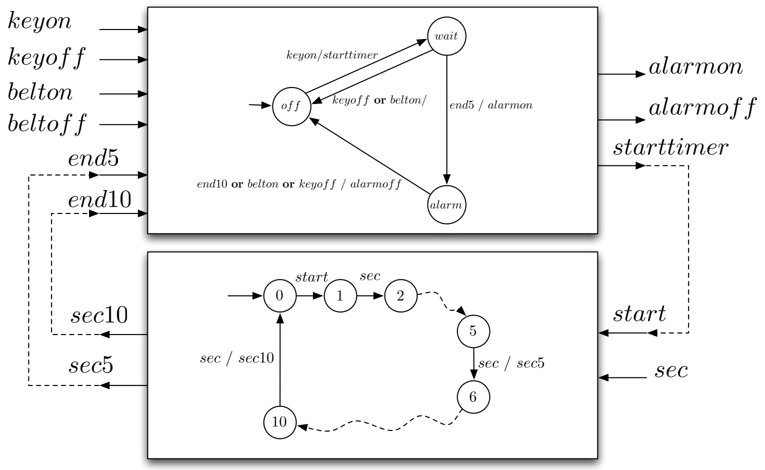


Figure 5.14: The seat belt controller and the timer together.

The composition of the two FSMs is another FSM $(I, O, S, r, \delta, lambda)$. The elements of the tuple are defined by the two FSM being composed. In this example, some example are given to build the intuitiont of how composition works. A formal definition of composition is given later in this section. The state of the composed system is a pair $(s_b, s_t) \in \{off, wait, alarm\} \times \{0, ..., 10\} = S$ where s_b is the state of the set belt controller and s_t the the state of the timer. The initial state of the system $r = (off, 0)$. Assume that the key is turn on. What should the next system state be? Figure 5.15 shows two possible transistions. Consider the transition from state $(off, 0)$ to state $(wait, 0)$. The transition is taken when *keyon* is the only present input. The output that is emitted is *starttimer*. This transition violates

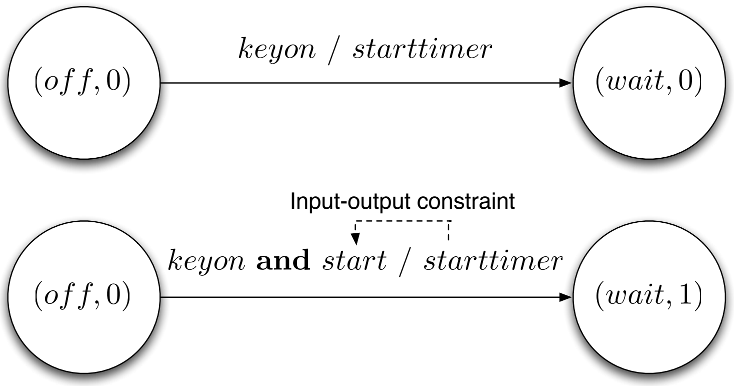


Figure 5.15: Examples of system-state transitions in the case of seat belt controller and timer.

the connection constraints shown in Figure 5.14. According to these constraints output *starttimer* must be equal to input *Start* meaning that they are either both present or both absent. Consider instead a transition from $(off, 0)$ to $(wait, 1)$. The transition is taken when both *keyon* and *start* are present. The output that is emitted is *starttimer*. This transition satisfies the constraint that output *starttimer* and input *start* be equal.

Having the reader acquired familiarity with FSMs in other contexts (related mainly to hardware design), the natural question that might arise is the following: *after the key is turned on, signal StartTimer is emitted by the seat belt controller. Is such signal immediately used by the conter FSM to make a transition from state* 0 *to state* 1*?*. In order to answer this question we need to give a precise definition of composition of finite state machines.

Let $M_1 = (I_1, O_1, r_1, \delta_1, \lambda_1)$ and $M_2 = (I_2, O_2, r_2, \delta_2, \lambda_2)$ be two finite state machines. A connection constraint is a relation $C \subset O_1 \cup O_2 \times I_1 \cup I_2$. For instance, if output $o_1 \in O_1$ is connected to input $i_2 \in I_2$, then $(o_1, i_2) \in C$. Being C a relation, an output can be connected to multiple inputs allowing the specification of multiple fan-out .

Should the inputs and outputs be disjoints, respectively?

Definition 5.4.7 (Composition of FSMs) *Given two FSMs M_1 and M_2 and a connection constraint C, their composition denoted as $M_1||M_2$ is another state machine $M = (I, O, S, r, \delta, \lambda)$ such that:*

$$I = I_1 \cup I_2$$
$$O = O_1 \cup O_2$$
$$S = S_1 \times S_2$$
$$r = r_1 \times r_2$$
$$\delta = \{(A_1 \cup A_2, (s_1, s_2), (s_1', s_2')) : (A_1, s_1, s_1') \in \delta_1 \wedge (A_2, s_2, s_2') \in \delta_2\}$$
$$\lambda_u = \{(A_1 \cup A_2, (s_1, s_2), B_1 \cup B_2) : (A_1, s_1, B_1) \in \lambda_1 \wedge (A_2, s_2, B_2) \in \lambda_2\}$$
$$\lambda = \{(A_1 \cup A_2, (s_1, s_2), B_1 \cup B_2) \in \lambda_u : \forall o \in B_1 \cup B_2,\ \forall i \in I_1 \cup I_2,\ (o, i) \in C \Rightarrow i \in A_1 \cup A_2\} \quad (5.1)$$

In Definition 5.4.7, Condition 5.1 imposes that outputs and inputs connected together behave consistently.

Definition 5.4.7 can lead to undefined behaviors. Consider a model of a well-behaved driver as in Figure 5.16.

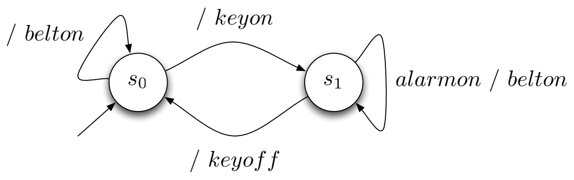


Figure 5.16: Finite state machine modeling a well-behaved driver.

A well-behaved driver either puts the belt on before turning the key on or she puts the belt on as soon as the alarm starts beeping. Consider the case where the driver turns the key on before putting the belt on. The system is in state $(s_1, wait)$. For the driver FSM, $(\{alarmon\}, s_1, s_1) \in \delta$ and $(\{alarmon\}, s_1, \{belton\}) \in \lambda$. The only possible next state for the entire system is $(s_1, alarm)$. Based on Definition 5.4.7, the composition should satisfy:

$$(\{end5, alarmon, belton\}, (s_1, wait), (s_1, alarm)) \in \delta$$

and for the output relation

$$(\{end5, alarmon, belton\}, (s_1, wait), \{alarmon, belton\}) \in \delta$$

Unfortunately, while in state *wait*, there is no transition defined in the seat belt controller with input set $\{end5, belton\}$, therefore the output of the seat belt controller is actually not defined.

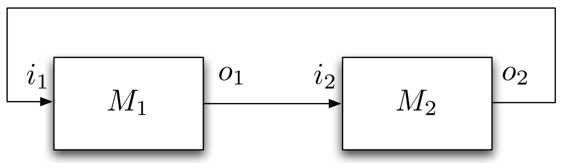


Figure 5.17: Two finite state machines in feedback connection

Figure 5.17 shows the general case of a feedback connection of two FSMs M_1 and M_2.

The standard solution to solve the feedback connection problem is to make the output independent from the input: $\lambda \subseteq S \times 2^O$. This new type of Finite state machines is known as of Moore's type (as opposed to Mealy machines).

Hierarchical FSMs: StateCharts

The FSM model of computation is suitable to capture functions that are control dominated. Modeling using FSMs is, however, rendered difficult by the complexity coming from the number of states and transitions. Consider an abstract model of the status of a thread (Figure **??**-a). A thread may be in the $sleep_1$ state where it does not need any CPU time to be executed. An input req_1 may resume the thread execution that enters a new state $ready_1$ where the it is waiting to be executed. When the CPU is availalbe, the thread receives a notification through input $exec_1$

Figure 5.18: Example of one and two threads FSMs.

Figure 5.19: Example of one and two threads FSMs.

and it transitions to state run_1 where it is running on the CPU. The thread goes back to the $sleep_1$ state after completing the execution. It emits output rel_1 to release the CPU and make it availalbe to other threads. Consider two parallel threads running on the same CPU. The state machine of these two parallel threads is shown in Figure 5.18-b. It has 9 states and many transitions. This FMs results from the composition of the two concurrent threads. Unfortuntately, the FSMs model of computation does not provide any syntactic of semantic support for expressing concurrent executions of finite state machines. Intuitively, we would like to simply describe the system as the *concurrent or parallel* composition of two threads, leaving the semantic of the composition to be defined by the model of computation.

While in the run_1 state, thread M_1 needs to execute some control actions. For instance, it may execute a thread corresponding to the seat belt controller. Moreover, while in the run_1 state, the thread may be preempted by an *abort* signal that sends the thread back to state $sleep_1$ (Figure 5.19-a). The finite state machine may get very complex as the nubmer of transitions induced by the abort signal equal the nubmer of states in the seatbelt controller. A more intuitive way of describing this thread is shown in Figure 5.19-b. It intuitively uses a construct tha defines a state machine inscripted into a state of a higher level state machine. This hierarchical construction is not possible in the FSM model of computation.

Another example of of use of these two constructurs (concurrency and hierarchy) comes from the composition of the seat belt controller and the timer (Figure 5.14). The seat belt controller has 3 states and the timer has 11 states. The composition of the two is a state machine with 33 states. However, not all of them are necessarily reachable. The state transition diagram of the composition is shown in Figure 5.20.

This figure should be completed, the conditions on all the transitions snould be checked.

To overcome the complexity encountered in capturing systems with FSMs, Harel proposed a visual formalism called Statecharts [77]. This formalism extends the FSM model with the notion of hierarchy, concurrency, and communication. The seat belt controller and timer example will drive the presentation of the Statecharts. The system can be decomposed as follows:

- *The timer FSM consists of a non-alarm zone (from 0 to 5) and an alarm zone (from 5 to 10).* This type of decomposition will make use of hierarchy. Alarm

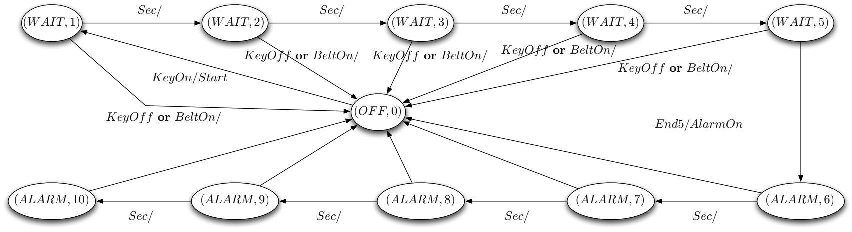


Figure 5.20: Result of the composition of the seat belt controller and the timer. The composition has 11 states and 20 transitions.

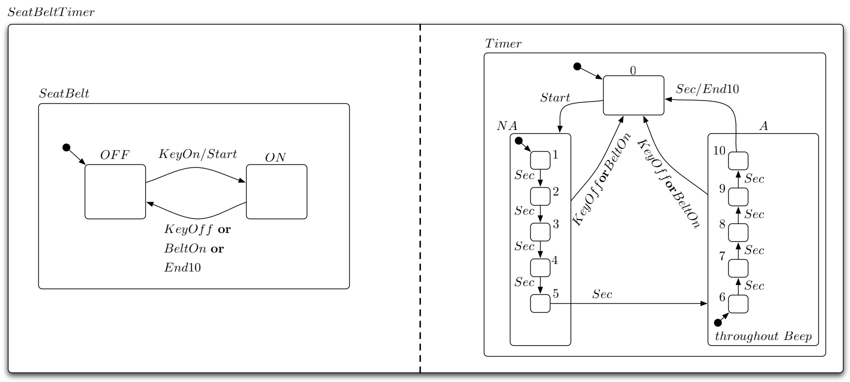


Figure 5.21: Statecharts description of the seat belt controller and the time.

and non-alarm zones will be refined into further states.

- *The timer FSM makes transitions independently from the seat belt controller.* This decomposition will make use of concurrency.
- *Whenever the seat belt is fasten, both the controller and the timer make a transition to their initial states (corresponding to system state $(off, 0)$).* Hierarchy will also be used to simplify the description of this type of transition.

The description of the combination of the seat belt controller and the timer using Statecharts is shown in Figure 5.21. This example only use some of the basic features of Statecharts. Advanced features will be shortly explained later in this section.

Each state of a Statechart is graphically represented by a box. States can contain other states in a hierarchical fashion. A *basic state* is a state that does not contain sub-states, i.e. it is a leaf of the hierarchy (e.g. state 0 in the *timer* state. The *root state* is a state without parents. In this example, *seatbelttimer* is the root state. The *timer* state contains three sub-states: state 0 is an idle state, state *na* is a non-alarm zone state, while *a* is an alarm zone state. The timer can only be in one of these three state, therefore state *timer* is called an *OR-State* meaning that its sub-states are related to each other by exclusive-or. State *a* and state *na* are also OR-states.

The entire system is the composition of the *seatbelt* state and the *timer* state. However, this type of composition does not have the same meaning as the OR-State one. The two states are concurrent, meaning that the system is in one of the sub-states of *seatbelt and* one of the sub-states of *timer*. The composition is denoted by a dashed line separating the sub-states and the parent state (the *seatbelttimer* state in this example) is called an *AND-State*.

Transitions are more expressive than in the FSMs model. A transition label is of the form $e[c]/a$ where:

- e is the event that triggers the transition.
- c is a guard condition. This condition must be true in order for the transition to be enable. If the guard condition is evaluates to false, the transition cannot be taken even if event e has occurred.

- a is an action that is carried out when the transition is taken. The action can be an event that in turn triggers other transitions.

Events and actions provide an infrastructure for interaction and communication among FSMs. The example shown in Figure 5.21 only used events and actions. A special symbol, a dot with an arrow pointing to a state, is used to mark the initial states. The initial state is off AND 0 for the $seatbelt$ AND $timer$ states, respectively. When the key is turned on, the transition from off to *on* is taken and event *start* is emitted. Notice that event *start* also triggers a transition from 0 to *na*. This is an example of broadcast communication of events. Broadcast communication of events is *synchronous* in the sense that the *start* event is visible *immediately*, as soon as emitted, by the entire model. Therefore, transition from 0 to *na* is taken together with transition from off to *on*.

After the key is turned on, the new state of the system is *on* and *na*. Notice that *na* is an OR-state that contains sub-states. The initial sate for *na* is 1, therefore the system goes into state *on* and 1. Another useful feature provided by Statechart is a way of modeling preemption. From any of the sub-states in *na*, if the key is turned off or the seat belt is fasten, the active state of the timer becomes 0. This feature simplifies the diagrams in terms of number of transitions.

Before, describing the semantics of a Statechart diagram, one more feature is worth describing. A state can be associated with an activity. For instance, the alarm is an activity that we denote $BEEP$. This activity must start when the alarm zone is entered and must stop when the alarm zone is exited. Statecharts allows the sepcification of an activity A in state s throuhg the notation $throughout A$ written insied s. This notation corresponds to generating an event $start(A)$ when the state is entered, and the event $stop(A)$ when the state is exited.

Should we add features like timeouts, connectors and history?

We explained the syntax and the intuitive meaning of the main features of Statecharts. The semantics of a Statecharts diagram is given by an algorithm that executes the model and generates a sequence of states. Unfortunately, there are many ways of defining the execution of a Statecharts model. In 1994, seven years after the work of Harel, Van Beeck summarized 22 different variants of Satecharts semantics [140]. Two years later, in 1996, Harel and Naamad published the semantics of STATEMATE [78], a tool that implemented Statecharts. STATEMATE was a product offered by i-Logix, Inc., that was later acquired by Telelogic which was then acquired by IBM. STATEMATE is still shipped today [2].

What follows is a description of the the semantics of Statecharts as implemented by STATEMATE. The execution of a Statecharts model is a sequence of snapshots of the system execution. A snapshot is called *status*. Each execution starts with a status that represents the initial state. A transition from one status to the next is called *step*. The definition of a step and the decisions that are taken during a step define the semantics. The basic principles adopted by the developer of STATEMATE are the following:

1. All changes that occur during a step can be sensed only in the next step. In particular, if an event is emitted during a step, it will be made visible to the rest of the system only in the next step.

2. Event stay active only for the duration of a step. They are not “remembered” in subsequent steps.

3. Computations done in a step are based on the status of the system at the beginning of the step.

4. A maximal set of non-conflicting transitions is always executed. This means that if a transition can be taken than it is taken.

These four principles are used to compute the next status starting from the preset one.

Need to find a way of simplifying the semantics and explain it.

5.4.2 Discrete Event

The model of time that we all share is the one of a real variable. Systems evolving in time are observable through physical quantities that are functions of time. These functions are defined in any point of the real line, i.e. for any value of time. Also, at least at the macroscopic level, these functions are continuous. However, in many applications, only a subset of points on the real line are the interesting ones. In these cases, the continuous time functions can be abstracted into a discrete set of points called *events*. Abstraction is done when the information contained in the time interval between two consecutive events can be either recovered, or it is simply uninteresting. The important information that is retained is the total ordering of the events. Given two events, it is always possible to order them depending on which one happened first.

Example 5.4.8 (Sampling) Consider for instance a continuos time signal modeled by a function $x : \mathbb{R} \to \mathbb{R}$. We denote this signal simply by $x(t)$. The Furier transform of $x(t)$ is $X(f)$ that describe the frequency spectrum of $x(t)$. Signal $x(t)$ is band-limited if there exists a frequency value B such that $X(f) = 0$ for all $|f| > B$. Constant B is called the bandwdith of $x(t)$. For instance, a sinusoid $x(t) = \sin(2\pi f_o t)$ is band-limited since $X(f) = \delta(f - f_0)$ [4].

The Nyquist-Shannon sampling theorem says that a band-limited signal $x(t)$ with bandwdith B can be completely represented by uniformly spaced samples taken with time interval:

$$T \leq \frac{1}{2B}$$

where $2B$ is also called the Nyquist rate. Therefore, it is sufficient to know the set of values $V = \{x(nT) : n \in \mathbb{Z}\}$ to reconstruct the entire waveform $x(t)$.

Another example of abstraction is digital circuit simulation. The interesting events in this context are the transitions from 0 to 1 and from 1 to 0. These transitions are abstractions of the corresponding ones from low to high voltage and from high to low voltage, respectively, that occur in at the output voltage of the transistors implementing the digital circuit. The actual voltage waveform is abstracted away because from the logic point of view the values of the voltage at all time are not relevant. This abstraction is of course possible only when the logical behavior of the system is not affected by the actual signal waveform.

Example 5.4.9 (Digital circuit) Consider the circuit in Figure 5.22. A pulsed voltage source is connected to the input of three cascaded inverters. Each inverter is implemented by a pair of MOS transistors, an NMOS and a PMOS. The supply voltage is $5V$.

The voltage source generates a square wave with period equal to $20ns$, swinging between $0V$ and $5V$. The instantaneous change in voltage at the input of the first inverter propagates with some delay. Moreover, the waveform generated at

[4] With δ we denote the dirac pulse.

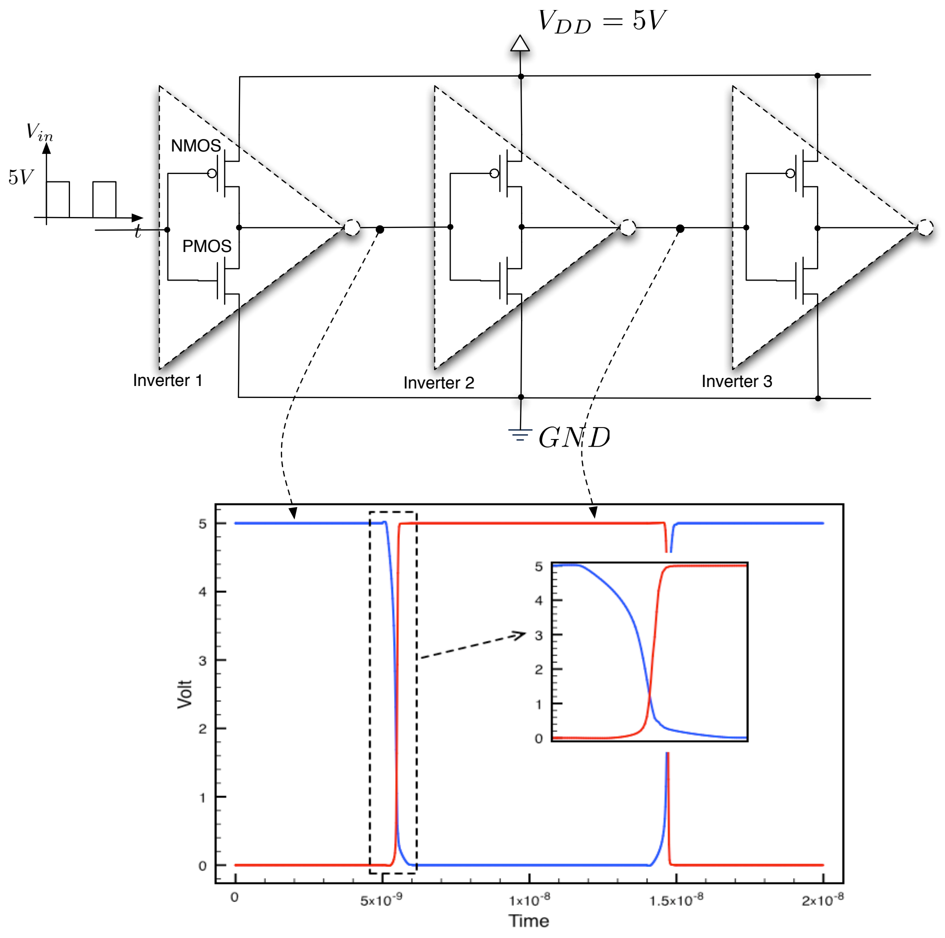


Figure 5.22: Cascade composition of three inverters and result of the SPICE simulation.

the output is continuous with a certain slope. Because transistors have input and output capacitances that need to be charged (and discharged) to change the input and output voltage, some time is needed before the that transition changes its region of operation. The output of the second inverter is not a square wave anymore. However, the second inverter switches its output voltage as a consequence of its input voltage crossing some threshold.

These circuit can be modeled using SPICE [**?**]. The result of the transient analysis are also shown in Figure 5.22. The sub-plot shows the actual voltage change at the output of the first and second inverters. The interval of time where the voltage is between the two extreme values $0V$ and $5V$ is negligible compared to the time where the voltage is equal to one of the two extreme values. Therefore, the switching can be completely abstracted as if it happened in zero time. In a digital circuit simulator, the two voltage levels are encoded by 0, for $0V$, and 1 for $5V$. Transitions from one level to the other happen instantaneously in a discrete set of points in time. These events retain the total order imposed by the continuos time nature of the real circuit.

The definition of the semantics of the discrete time model of computation requires the introduction of the notion of events and signals (see Section 5.7.2 for more details). An event is a pair $(v, t) \in V \times T$ where V is a set of *values* and T is a set of *tags*. For instance, the set of values could be the set of real numbers. If a signal is quantized, the set V contains a finite set of values in a given range. For discrete time systems, the set of tags is a coutable subset of real numbers. Recall that a countable set is a set where each element can be associated with a natural number (the element of the set can be counted).

The discrete event model of compoutation is a timed model, meaning that there is an exclicit notion of time that induces a *global order* of the events in the system. Given two events, $e_1 = (v_1, t_2)$ and $e_2 = (v_2, t_2)$, their ordering is induced by the ordering of the tags: $e_1 \leq e_2 \iff t_1 \leq t_2$. In discrete event systems, the set of tags is totally ordered meaning that tags can be alway compared to each other. Given two tags $t_1, t_2, \in T$, either $t_1 \leq t_2$ or $t_2 \leq t_1$. This means that given two events, it is always possible to establish which one happened first.

A signal is simply a set of events. Events can happen at any time asynchronously. In practical terms, an event in a discrete event system is a change in value of a signal. When an event appears at the input of a *block*, the block is *executed*. The execution consists in reading the values of the inputs, run an algorithm that implement the input-output function of the block, and generate other events on the output signals. The tags associated with the output events do not have to be equal to the tags of the input events. A block may introduce a delay by associating tags to the output events that are greater than the tags of the input events. This feature can be used to model the execution time of the block. However, the execution time may also be zero which has serious implications on the definition of the execution semantics of a discrete event system.

A simulation engine for discrete event systems can follow a very simple set of steps to produce a simulation trace (Algorithm 1. The algorithm taks as input the set of signals in the system S and the set of blocks B. Each block $b \in B$ is connected to a set of input signals $I(b) \subseteq S$ and a set of output signals $O(b) \subseteq S$. When a block runs, it generates a set of events $E = run(b)$. The algorithm maintains an event queue Q where events are ordered by their tags. The queue is initially empty. All sources, i.e. all blocks without input signals, run first and generate events that are placed in the queue. Then, the main loop starts. The simulation engine extracts the event with the least time stamp from the queue. The event belong to a signal s.

Algorithm 1 Discrete event simulation

```
Input: Set of signals $S$; set of blocks $B$
$Q \leftarrow \emptyset$
forall $b \in B$ such that $I(b) = \emptyset$ do
    $E \leftarrow$ run(b)
    add all events in $E$ to $Q$
while $Q \neq \emptyset$ do
    $e \leftarrow$ extractmin($Q$)
    $s \leftarrow$ getsignal($e$)
    forall $b \in B$ such that $s \in I(b)$ do
        $E \leftarrow$ run(b)
        add all events in $E$ to $Q$
```

All blocks that are "sensitive" to that event, meaning all blocks that have s as input signal, are executed. The execution will generate other events that are inserted in the queue. Although the algorithm seems simple, the simulation of a discrete event system must deal with many subtleties, some of which are presented next.

The system in Figure 5.23 has three blocks a, b and c. Block a is a source that generates two events $e_1 = (v_1, t)$ and $e_2 = (v_2, t)$ with two different values v_1 and v_2 but the same time stamp t. Funtion *extractmin* must make a decision between the two events. It is arbitrary to process e_1 before e_2 or vice versa. Since the simulator provides one execution trace only, a decision must be made. The two possible decisions provide two different execution traces of the same system. If e_1 is extracted first, block b is executed, whereas if e_2 is extracted first, block c is executed.

Suppose a simulator choses the first alternative, thereby executing b. Also, suppose that the execution of b happens in zero time (i.e. with not delay from the inputs to the outputs). Block b generates one event $e_3 = (v_3, t)$ that is placed in the queue. There are now two events, e_2 and e_3, in the queue. These two events are both connected to the input of c. Should c be executed once or twice? Algorithm 1 extracts one event at the time from the queue. Theefore, according to this algorithm, block c will be executed twice. However, the two events have the same tag and they should be seen (and "consumed") together by c. This is yet another decision that affects the simulation results because if c is executed twice, it will generate two output events, whereas if it is executed once, only one event will be generated. If the execution of b requires a certain time Δ, the the output events generatd by b will be delayed and c will be definitely executed twice.

Some discrete event simulator engines assume that the execution of a block always takes a minimum amount of time such that some of the problems we presented can be eliminated. The interested reader can find a more detailed discussion in the in depth chapter.

5.4.3 Process Networks

Data-flow networks first appeared at the end of the 50s as a model for parallel programming. In this new model, a program is described as a set of processes interacting asynchronously by echanging messages over queues. The intent of the model is to capture only data dependency among processes, thereby leaving the freedom of executing processes without dependencies in parallel. This is majoor advantage of languages that provide ways of expressing concurrency of actions.

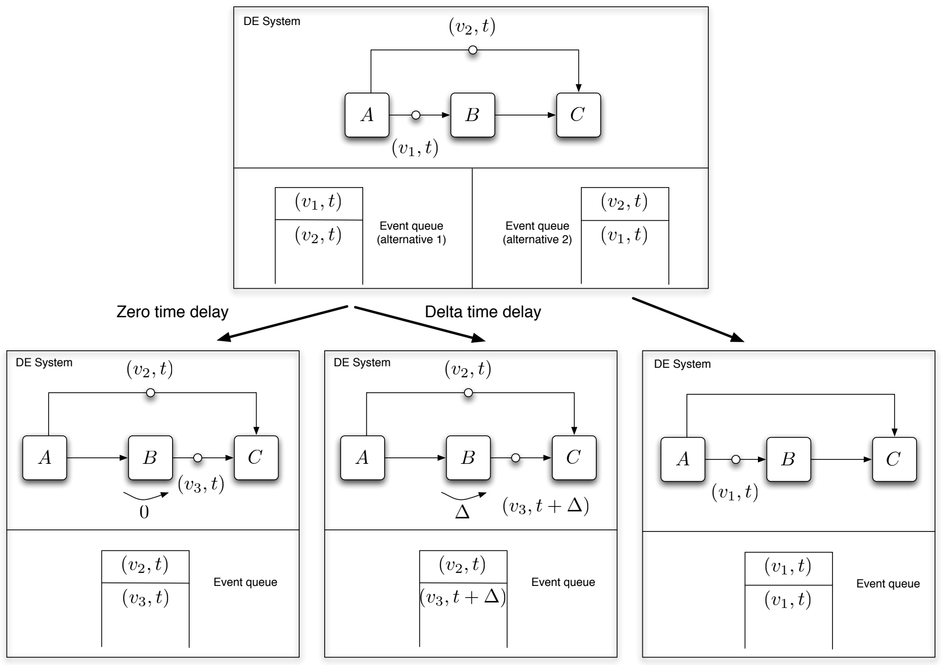


Figure 5.23: Discrete event simulation of a simple system with three blocks. Block A generates two events with the same time stamp t. The simulator has the choice to run B before C or vice versa. Many simulators avoid zero time delay by introducing a fixed reaction time for each block.

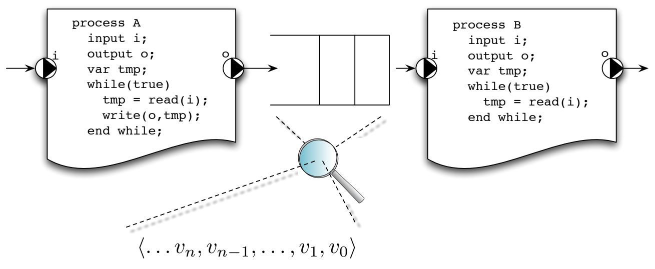
Programs described using function calls do not provide this important feature since all functions run in the same thread of execution.

In one of the early research work on this topic, Gilles Kahn [**?**] introduced a simple parallel programming language and provided also a formal description of its semantics. In this model, that today falls under the name of Kahn Process Networks (KPN), processes exchange information on First-Input-First-Output (FIFO) channels with unbounded capacity. In this section we present this model and its properties following the original definitions by Gilles Kahn, further developed by Edward A. Lee and Thomas Park [**?**].

Remark 5.4.10 (FIFO Channels and Communication Semantics) A First-In-First-Out unidirection communication channel can be thought of as a memory with to ports:

- One port provides a service to write data in the memory. This port is used by a process called *producer*.
- One port provides a service to read data from the memory. This port is used by a process called *consumer*.

The FIFO channel has the followning properties. The order in which data are extracted from a FIFO is the same in which they are inserted. Figure 5.24 shows and example of a system with two processes A (the producer) and B (the consumer) accessing a unidirectional FIFO.



```
process A
  input i;
  output o;
  var tmp;
  while(true)
    tmp = read(i);
    write(o,tmp);
  end while;
```

```
process B
  input i;
  output o;
  var tmp;
  while(true)
    tmp = read(i);
  end while;
```

Figure 5.24: Two concurrent processes communicating through a FIFO channels.

Process A has an input i from which it reads data that are then written in the FIFO. Process B is a sink that reads from the FIFO. The behavior of the two processes is specificed as a sequential program. Each process executed an infinite loop performing write and read operations (see the pseudo-code in Figure 5.24). Two natural questions arise:

1. *What happens upon a read request if the FIFO is empty?*
2. *What happens upon a write request if the FIFO is full?*

These questions are related to the communication semantics of the two processes A and B. Consider process B attempting to read from an empty FIFO. There are two possible cases:

- The data contained in the FIFO is absolutely necessary for the process to keep running. In this case the process must wait idle until some data are written in the FIFO.

- The process may have several tasks to execute, some of which may not depend on the content of the FIFO. Therefore, the process may decide to keep running by executing those tasks and come back to check whether the FIFO is still empty or not.

In the first case, process B attempting to read from an empty FIFO simply blocks until another process A writes enough data to let B continue its execution. This is called *blocking read* semantics. The second communication mechanism, instead, has a *non-blocking read* semantics. Notice that, to implement this semantics, the process must be able to check whether the FIFO is empty. Allowing a process to do so introduces some fundamental problems that will be covered later in this chapter.

To answer the second question, we need to distinguish between the case where the FIFO has a finite memory and the case where it is *unbounded.* A FIFO can be full only if it has finite memory. If the FIFO is unbounded, a process attempting to write will always succeed, therefore there is no need to block the writing operation. This is called *non-blocking write* semantics. Whereas, if the FIFO is bounded, a process that attempts to write can be blocked. This is called *blocking write* semantics. Even if the FIFO is bounded, a non-blocking write semantics can be implemented. In this case there are a few choices: the process may check whether the FIFO is full and decide to execute other actions waiting for the consumer process to read from the FIFO and release some space; or the process may simply overwrite the content of the FIFO.

A KPN can be represented as a directed simple graph where nodes are processes, called *actors*, and arcs represent communication channels (i.e. unbounded FIFOs). Actors exchange *tokens* whose type is abstract and we will never define explicitly when discussing the properties of KPNs. The concrete type of a token can be a simple integer, a matrix or an image. Let D denote the domain of tokens, meaning the set of values that tokens can take on. For example, for integer tokens $D = \mathbb{Z}$, while for images D can be the set of all images with size 640×480 pixels and a color depth of 8 bits. Figure 5.24 is an example of KPN. Consider an observer seating on the FIFO channel. She will observe a sequence of tokens flowing from the producer to the consumer. The sequence of values can be finite or infinite.

In defining the semantics of a KPN, the content of processes (i.e. the specifics of the program describing their execution) is completely hidden. Processes are characterized by properties based on which properties of the entire KPN program can be inferred. Little assumptions are necessary to obtain strong results about the behaviors of a KPN. It is assumed that a process has an internal thread and it can contain states. Because the output sequence of a process depends on the input sequence and also on the internal state, a process cannot be defined as a function from input tokens to output tokens. A process is defined as *a function from input sequences to output sequences.* The set of finite and infinite sequences of a set D is denoted by D^ω. A sequence is denoted as $X = [x_1, x_2, \ldots]$ where x_i is the i-th token of the sequence.

Example 5.4.11 (Infinite Impulse Response Filter Process.) Consider the following pseudocode describing the thread of a KPN process:

```
process IIR
  input x ;
  output y ;
  variable state initially 0 ;
  variable tmp ;
  while( true )
    tmp = read( x ) ;
    state = state + a * tmp ;
    write( y , state ) ;
  end while ;
end process
```

This process is an Infinite Impulse Response (IIR) filter that reads one token at the time from the input channel x, updates the *state* variable, that is initialized to 0, and writes the output y. Let us assume that the tokens are real numbers. Can we define a function $f : \mathbb{R} \to \mathbb{R}$ from the input to the output representing process IIR? The output value is determined by the update equation $state = state + a \cdot tmp$. Therefore, the output depends on the internal state as well, and cannot be directly computed only by knowing the value of the input. As a simple example, consider the two sequences $X_1 = [0.0, 1.0, 2.0, 3.0, 4.0 \ldots]$ and $X_2 = [0.0, 2.0, 4.0, \ldots]$, and assume $a = 0.5$. The output sequences corresponding to these two input sequences are $Y_1 = [0.0, 0.5, 1.5, 3.0, 5.0, \ldots]$ and $Y_2 = [0.0, 1.0, 3.0 \ldots]$, respectively. The output corresponding to the input value 4.0 is 5.0 and 3.0 respectively.

To take into account the effect of the internal state, the IIR fileter is rather defined as a function that maps input sequences to output sequences a s follows:

$$f : D^\omega \to D^\omega$$

For a given input sequence $X = [x_1, x_2, \ldots]$ the output sequence $Y = [y_1, y_2, \ldots]$ is defined as follows:

$$y_i = a \sum_{j=1}^{i} x_i$$

In general, a process has many inputs and many outputs, therefore a process is a functions from a tuple of input sequences to a tuple of output sequences. A process with n inputs and m outputs is a function defined as follows:

$$f : \underbrace{D^\omega \times \ldots \times D^\omega}_{n} \to \underbrace{D^\omega \times \ldots \times D^\omega}_{m}$$

This representation can also be viewed as a vector of m functions, one for each output sequence.

Consider a network of processes. It may be possible that several processes have enough tokens in its input queue to keep running. All these processes could run concurrently meaning without any precedence among them. However, the program will eventually implemented on a computing platform that may have limited concurrency. For instance, if the KPN program is implemented on a single processor, then processes cannot be executed concurrently. A natural question to ask is whether the order in which processes are executes matters to the final result of the computation. The most important property about this model of computation is that *given a unique input sequence to a KPN, the output sequence is uniquely defined independently from the order in which the processes are executes.* This property is called *determinacy.* Although each schedule gives the same result, they have different latency, code size and memory requirements [107]. The interested student may refer to Section 5.4.3 for a formal proof of the determinacy property.

Obviously, the assumption of FIFO channels with unbounded capacity makes a KPN program not amenable to hardware or software implementation since, in practice, FIFOs are always bounded. Therefore, it is important to be able to find a schedule of the processes such that the number of tokens in each FIFO in guaranteed to be bounded. In Section **??** we introduce some restrictions on the KPN model such that such schedule can be found efficiently. Example **??** clarifies this point and shows how the KPN model can be used to describe the functionality of a simple system.

Example 5.4.12 (Kahn Process Network Modeling) We will model a simple Automatic Gain Control (AGC) loop. This sub-system is part of the front-end of radio receivers to adjust the voltage range of a signal based on the knowledge of the amplitude of the transmitted signal. Figure 5.25 shows the entire systems that is the interconnection of five processes. A random number generator is the source of the system that generates random real numbers in the interval $[0, 2]$. A simple process that scales the input by a constant factor k_1 mimics the attenuation effect of the channel between the transmitter (i.e. the source) and the receiver. The other three blocks constitute the AGC. This is a classical control loop whose purpose is to drive the value of a variable to a reference. The AGC tries to recover the attenuation factor and eliminate its effect by multiplying the input signal by the inverse of k_1.

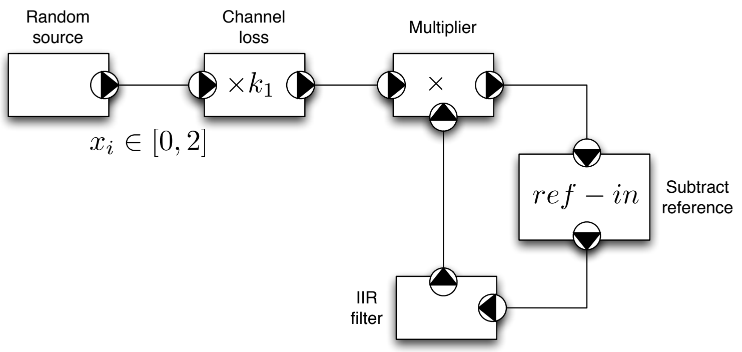
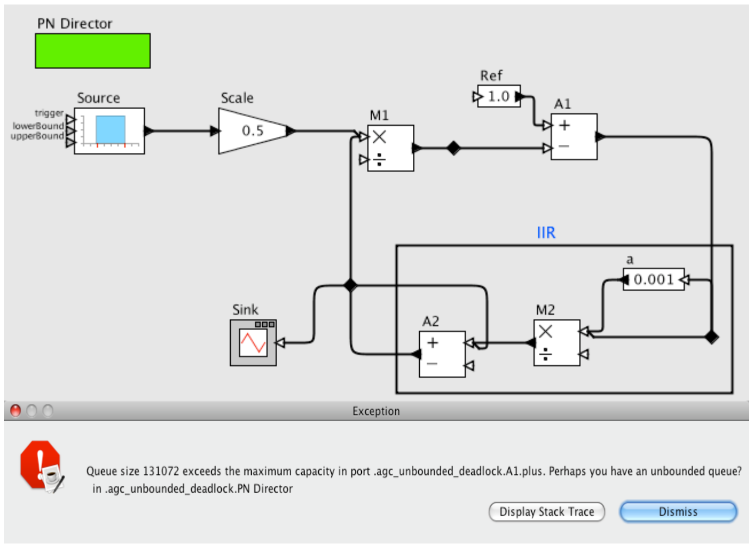


Figure 5.25: An automatic gain control loop

The AGC multiplies the incoming signal by the output of a filter whose input is the difference between the output of the AGC and the reference value. The reference value is set to an appropriate value that depends on the range of the random source. The effect of the loop is the following. If the gain at the output of the filter is too high, the output of the multiplier will be greater than the reference value. Therefore the difference between the reference and the output of the multiplier will be negative and the output of the filter will decrease. On the other hand, if the gain is too small, the input to the filter will be positive and the output of the filter will increase. Therefore, we simply need to set the reference value to the average value of the source which is 1.

There are some interesting details to notice. Suppose that each process proceeds by reading its inputs, doing some elaboration and then writing its outputs. The random source simply computes a random number and then writes that number in the output FIFO. While the specification of the system is correct, the simulation in a real environment may expose some difficulties. First of all, it is perfectly fine for the source to generate an infinite sequence of tokens and place the into its output FIFO even if no other process executes. Therefore, when using a real simulation environment, the source output queue can easily go unbounded. Second, when the system starts, each process attempts to read from its input buffer. Unfortunately, the multiplier does not have any input token from the filter which in turn does not have any input token from the Subtract Reference process. Therefore, no processing is executed by the AGC.

Figure 5.26 shows a model of the AGC loop in Ptolemy II. The source is an actor that generates random values in the interval $[0, 2]$ with uniform probability density function. As soon as the simulation engine is started, an error message is displayed. In this model, two actors only have sufficient tokens to run: the source and the constant reference. The two corresponding output buffer fill up instantaneously and an error message is displayed.



Queue size 131072 exceeds the maximum capacity in port .agc_unbounded_deadlock.A1.plus. Perhaps you have an unbounded queue?
in .agc_unbounded_deadlock.PN Director

Figure 5.26: Error message due to the presence of an unbounded buffer in the AGC model in Ptolemy II.

To fix this problem, few more components are needed. Figure 5.27 shows the modified model. Delay elements are introduced in the AGC loop and the sources are now controlled by input triggers (for instance, the `Source` actor is triggered by the output of the filter meaning the it will generate a new output for each new output produced by the filter). A delay element outputs a certain number of initial tokens and after that behaves as an identity process that passes its input to its output. The simulation result shows that the estimated gain send to the multiplier is equal to 2 which is the inverse of the channel loss which is 0.5 in this example.

Formal Semantics of Process Networks

Sequences of token can be ordered accoridn to a *prefix ordering*. A sequence X preceded a sequence Y, denoted $X \sqsubseteq Y$, if X constitutes an initial sub-sequence of (or it is equal to) Y, i.e. the sequence X is a prefix of the sequence Y. For example, $[x_1, x_2, x_3] \sqsubseteq [x_1, x_2, x_3, x_4]$ while $[x_1, x_2, x_3]$ and $[x_1, x_2, x_4]$ are incomparable. Problem **??** shows that the prefix ordering is a partial order on the set of all finite and infinite sequences. Let $\perp$ denote the empty sequence. Obviously, $\perp \sqsubseteq X$ for

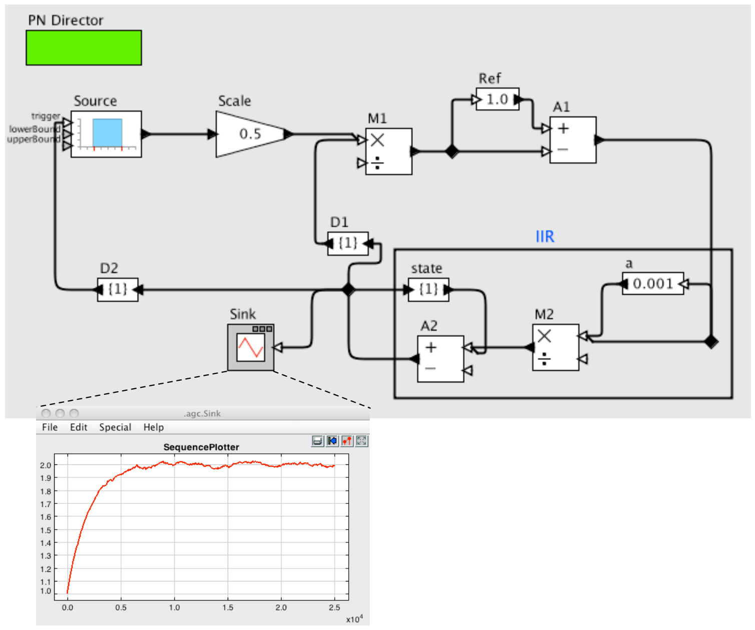


Figure 5.27: Simulation result of the correct model of the AGC loop in PTOLEMY II.

any sequence X.

Let S denote the set of all finite and infinite sequences of tokens. A subset $C = \{X_1, X_2, \dots, X_n\}$ of S is called a *chain* if any two elements of C are comparable. The reason why it is called a chain is because the element of C can be linearly ordered, i.e there exists a permutation σ of the indexes $1, \dots, n$ such that $X_{\sigma(1)} \sqsubseteq X_{\sigma(2)} \sqsubseteq \dots \sqsubseteq X_{\sigma(n)}$.

Given a subset $B = \{Y_1, \dots, Y_m\}$ of S, an upper bound of B is a sequence $Y \in S$ such that $Y_i \sqsubseteq Y$ for all $Y_i \in B$, meaning that an upper bound of a subset of S is an sequence in S that is "greater" that any sequence in B. Notice that the upper bound does not have to belong to B. If it does, then it is called the *maximum element* of B. Moreover, the upper bound is not unique. There is a set of upper bounds U of a set B that contains all those sequences that are greater than the sequences in B. If the set U has a minimum element, it is called the *least upper bound* of B and it is unique.

The set S of infinite an finite chains has an interesting property. *Every chain $C \subset S$ has a least upper bound.* Because of this property, S is called a *complete partial order* (CPO). The least upper bound of a chain C is denoted $lub(C)$ and it can be thought of as the limit of the chain.

Consider a process with p inputs and q outputs. It is a function $f : S^p \to S^q$ that mapts q-tuples of input sequences to q-tuples of output sequences. Tuple of sequences are also ordered by the ordering relation induced by the prefix ordering defined on sequences. Specifically, given two tuples $(X_1, \dots, X_k) \in S^k$ and $(Y_1, \dots, Y_k) \in S^k$:

$$(X_1, \dots, X_k) \sqsubseteq (Y_1, \dots, Y_k) \iff X_i \sqsubseteq Y_I, \ \forall i = 1, \dots k$$

This ordering allows to extend the definition of chains and of least upper bound of chains to tuples of sequences S^k.

Functions functions between tuples of sequences can be extended to functions between sets of tuples of sequences. Let $B \subseteq S^p$ be a set of tuple of sequences, then the set $f(B)$ is defined as follows:

$$f(B) = \{f(X) \in S^q | X \in B\}$$

A natural question to ask is the following. Given a chain $C \in S^p$, is $f(C) \in S^q$ also a chain? This is not true in general but it is a very desirable property for a process to have. The following definition states the property that a process must have in order for this to be true.

Definition 5.4.13 (Continuity) *A process $f : S^p \rightarrow S^q$ is continuous if and only if for all chains $C \in S^p$, $lub(F(C))$ exists and:*

$$F(lub(C)) = lub(F(C))$$

We can also define monotonicity of a process.

Definition 5.4.14 (Monotonicity) *A process $f : S^p \rightarrow S^q$ is monotonic if and only if for all pairs $X \in S^p$ and $X' \in S^p$ the following holds*

$$X \sqsubseteq X' \Rightarrow f(X) \sqsubseteq f(X')$$

Monotonicity implies that if the input sequence to a process is extended (i.e is made longer by adding tokens), the output sequence cannot shrink (i.e. must either stay the same or it must also extend with more tokens). It is a generalized notion of causality. It also means that a monotonic process transforms chains into chains.

To arrive at the definition of the semantics of a KPN, its behavior is written as a single fix point equation. Consider the KPN of Figure 5.28 (taken from the original paper published by Kahn [**?**]). A process $f : S^p \rightarrow S^q$ can be though of as a vector of q functions, one for each of the outputs. The function that computes the i-th output is denoted by $f(i) : S^q \rightarrow S$.

The KPN shown in Figure 5.28 can be described by the following set of equations:

$$\begin{aligned}
X_1 &= f_1(X_6, X_5) \\
X_2 &= f_2(2)(X_1) \\
X_3 &= f_2(1)(X_1) \\
X_4 &= f_4(2)(X_2) \\
X_5 &= f_3(1)(X_3, X_4) \\
X_6 &= I \\
X_7 &= f_3(2)(X_3, X_4) \\
X_8 &= f_4(1)(X_2)
\end{aligned}$$

The previous system of equations can be also written in a more compact form as follows:

$$\mathbf{X} = \mathbf{F}(I, \mathbf{X})$$

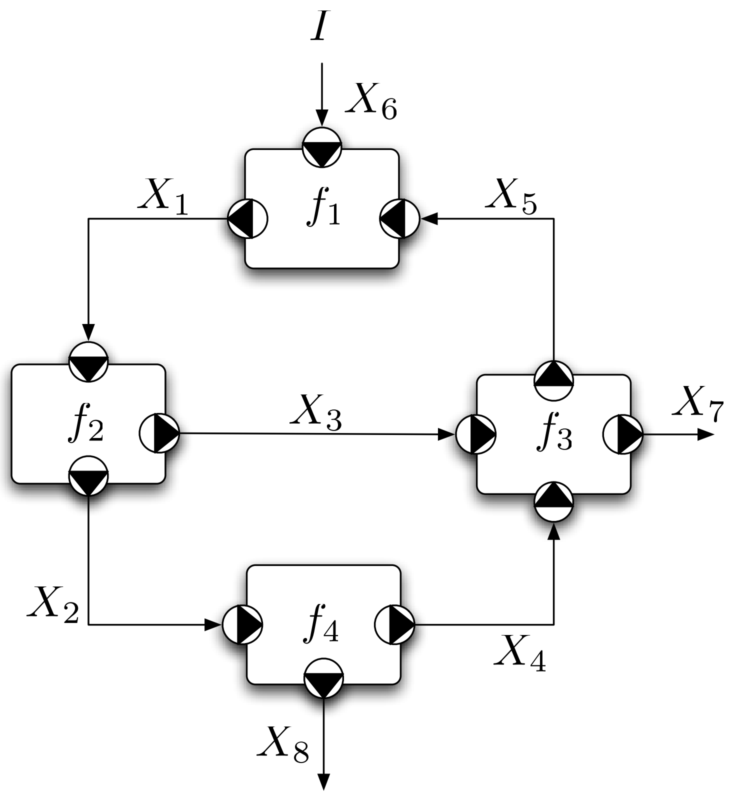


Figure 5.28: A KPN with four processes f_1,f_2,f_3 and f_4.

The behavior of the system is the solution of this equation. Notice that there can be many solutions to this equations. The behavior of the system is defined to be the least element of the set of solutions. The following theorem clarifies the reason why it is important that all processes are continuous.

Theorem 5.4.15 (Fix-Point theorem) *If F is continuous, the following equation:*

$$\mathbf{X} = \mathbf{F}(\mathbf{X})$$

has a unique fix point solution $X^* = lub(\{F^n(I, \perp) : n \geq 0\})$.

5.4.4 Data Flow

The data-flow model of computation is very similar to Kahn process networks. In fact, a data flow network is a set of processes that communicate through unbounded FIFO channels. The communication semantics is also the same, namely non-blocking write and blocking read. The fundamental difference between the two models consists in the restrictions that are imposed on the processes.

Data flow processes are called *actors*. The execution of an actor is a sequence of *firings*. Each firing consumes a certain number of tokens from the inputs and produces a certain number of tokens on the outputs. Actors can fire only if there are enough tokens in its input queues. Specifically, a set of rules, called *firing rules*, are associated to each actor and define the conditions under which the actor can fire.

Example 5.4.16 (A simple data-flow example) Consider a two-input adder like the one in Example 5.3.4. The two inputs are the operands of the addition operation, while the output is the result of adding the inputs. The intuitive execution of the adder is the following: read one token (e.g. a fix point number) from one input, read one token from the other input, compute the sum, write the result to the output. An alternative (and still correct) execution is the following: read tokens v_1 and v_2 from one input, read tokens v_1' and v_2' from the other input, compute the sums $o_1 = v_1 + v_1'$ and $o_2 = v_2 + v_2'$ and write the two results o_1 and o_2 to the output. The two execution ultimately produce the same output stream. There are many other execution that are correct according to the intuition of what the adder actor is supposed to do. The *firing rule* associated with the adder determines when the adder performs the addition, i.e. under what condition is the adder allowed to sum the inputs and generate the output value. The following statement is an example of firing rule:

> The adder is ready to fire when there is at least one token in each input queue.

This statement resembles the concept of prefix of a sequence. As long as the sequence of tokens defined by the firing rule is a prefix of the sequence of tokens that are stored in the input FIFOs, the actor can fire. This concept is formalized in this section.

A firing rule can be formally defined as a prefix of the input sequence that must be matched in order for the actor to be enabled to fire. An actor with p inputs can have a set of N firing rules:

$$\mathcal{R} = \{\mathbf{r}_1, \ldots, \mathbf{r}_N\}$$

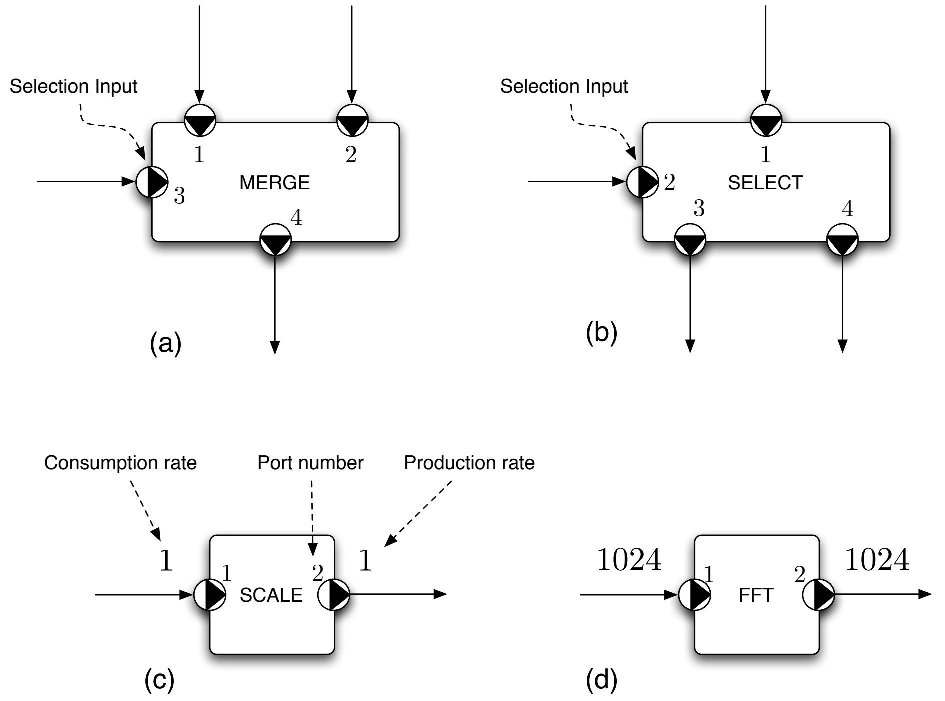


Figure 5.29: Examples of three different data-flow actors: (a) a merge actor that copies one of the inputs to the output depending on the Boolean value of a third input, (b) a select actor that send the input to one of the outputs depending on the Boolean value of a second input, (c) a simple actor that generates one output token for each input token, (d) a simple actor that generates 1024 output tokens for each 1024 input tokens. actor that

where each element of the set $\mathbf{r}_i$ is a tuple of p elements, one for each input as follows:

$$\mathbf{r}_i = (r_{i,1}, \ldots, r_{i,p})$$

Each element of $r_{i,j}$ defines the prefix of a sequence. When the firing rules are a prefix of the tokens in the inputs queues of an actor then the actor can fire.

Firing rules are defined by sequences of values. Consider the adder case with integer inputs. The number of firing rules is infinite because each sequence with one integer is a valid firing rule. In these cases, a special symbol $*$ is used to denote *any value*. A sequence $[*]$ is a prefix of any sequence with at least one token, and similarly $[*, *]$ is a prefix of any sequence with at least two tokens. Let $\tilde{x_j}$ be the sequence tokens in the j-th input queue of an actor. Then, firing rule $\mathbf{r}_i$ is enabled if:

$$r_{i,j} \sqsubseteq \tilde{x_j}, \quad \forall j = 1, \ldots, p$$

Figure 5.29 shows four different actors. A merge actor has three inputs and one output. Two of the three inputs are streams of data while the third one is a selection input that is used to decide which one of the two data inputs is passed to the output. The selection input accepts Boolean tokens (i.e. tokens whose value can only be *true* (T) or *false* (F)). If the selection token is true, then one token from

We need to be explicit about the restrictions on the actors (the must be functional) and about the conditions under which actors are monotonic In the original paper of Lee and Parks, the select and merge actors have different definitions. Maybe we want to rename them input 1 is copied to the output; otherwise if the selection token is false, one token from input 2 is copied to the output. The firing rules for the merge actor can be written as follows:

$$\begin{aligned}\mathbf{r}_1 &= ([*], \perp, [T]) \\ \mathbf{r}_2 &= (\perp, [*], [F])\end{aligned}$$

Notice that, according to these two firing rules, if the token written in the FIFO connected to the selection input are not Boolean, the merge actor will never fire.

The *select* actor has two inputs: a data input and a selection input. If the selection input is true, then one input token is copied to output 3, otherwise to output 4. The select actor has the following firing rules:

$$\begin{aligned}\mathbf{r}_1 &= ([*], [T]) \\ \mathbf{r}_2 &= ([*], [F])\end{aligned}$$

For the select and merge actors, the consumption and production rates (i.e. the number of consumed and produced tokens) can vary from one firing to the other. Each time the select actor fires, it consumes one token from input 1 and one token from input 2. If the selection is true, it will produce one token on output 3 and none on output 4, otherwise it will produce one token on output 4 and none on output 3. Similarly, the merge actor has a different consumption pattern depending on the value of the selection input. Specifically, it consumes one token from input 1 and one token from input 3 if the selection is true, and one token from input 2 and one from input 3 if the selection is false. The merge actor always produces one token.

When an actor has fixed production and consumption rates, the number of consumed and produced tokens for each firing is written on its input and output edges (close to the corresponding port). An example of such actors is shown in Figure 5.29(b) and Figure 5.29(c). For example, the *scale* actor shown in figure always consumes one token from input 1 and produces one token on output 2. The 1024 point *Fast Fourier Transform* (FFT) actor in figure, always consume 1024 tokens at the input and produces 1024 tokens at the output. If the consumption and production rates are known and fixed for all actors, then we call the data-flow program a *Static Data Flow* (SDF).

Static Data-Flow

A static data-flow graph is a special type of process network where each actor has a fixed consumption and production rate associated to each port. This restriction allows to prove many properties and to develop optimization techniques to statically schedule the data flow execution. Given a data flow network, it is possible to compute a schedule off-line. A schedule defines the sequence in which actors should be run and guarantees that an actor is run only when it is enabled to run (i.e. there are enough input tokens in its input queues). The ability to compute a schedule off-line brings several advantages:

- At run-time, actors are fired according to the schedule. The schedule guarantees that actors are fired only when they are ready to be fired (i.e. when there are enough tokens in their input queues). Therefore, a static schedule avoids the overhead of checking whether an actor can run and deciding which one to execute among the enabled ones.

- It is possible to compute the size of each FIFO channel off-line. The static schedule and the production and consumption rates of the actors allow to compute the maximum number of tokens to be stored for each FIFO.

In this section we cover the basic techniques used to statically schedule a SDF network. The scheduling technique uses only information on the *process graph*, meaning the actors, their interconnection and the number of produced and consumed tokens.

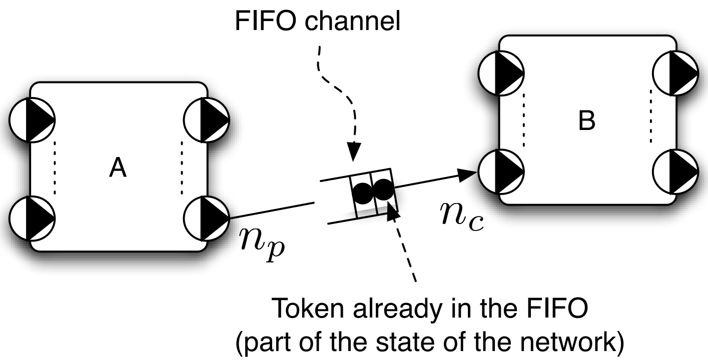


Figure 5.30: Two data flow actors A and B that communicate over a FIFO channel. Upon firing, A produces n_p tokens and writes them in the FIFO, while B consumes n_c tokens from the FIFO.

Figure 5.30 shows two actors A and B communicating over a FIFO channel. As shown in figure, at each point of an execution of a SDF network, a certain number of tokens is available in the channels (i.e. tokens that have been produced by not yet consumed). The vector containing the number of tokens in each FIFO of the SDF is the *network state*. A schedule should have the following two properties:

- A schedule should be *admissible*, meaning that it should fire an actor only when it is ready to fire;
- A schedule should be *periodic*, meaning that it should be able to bring the network back to its initial state firing each actor at least once.

A schedule S that satisfies these two properties is called a *valid schedule.* A SDF network admits many valid schedules. Valid schedules differ in the memory size needed for their execution, the code size and other metrics that can be seen as cost function. Selecting one valid schedule among the many possible ones becomes an interesting optimization problem.

A *firing vector* $\mathbf{v}_S$ is a vector that contains one element for each actor in the SDF network. The firing vector characterizes the schedule in terms of the number of times each actor is executed in the schedule. For instance, if actor A is executed 5 times, then $\mathbf{v}_S(A) = 5$. A valid schedule should be periodic. Thus, after all actors have been fired, each FIFO must have exactly the same number of tokens that it had at the beginning of the schedule execution. It means that the execution of a schedule can neither create nor consume extra tokens. Therefore, for each FIFO channel between actor A_i and actor A_j in the process graph, the total number of tokens produced by the A_i must be equal to the total number of tokens consumed by A_j. Consider the example shown in Figure 5.30. For a given schedule S and firing vector $\mathbf{v}_S$, the total number of tokens produced by actor A is $\mathbf{v}_S(A) \cdot n_p$ while the total number of tokens consumed by B is $\mathbf{v}_S(B) \cdot n_c$. The following *balance equation* must be satisfied by any valid schedule:

$$\mathbf{v}_S(A) \cdot n_p = \mathbf{v}_S(B) \cdot n_c \Rightarrow \mathbf{v}_S(A) \cdot n_p - \mathbf{v}_S(B) \cdot n_c = 0$$

The balance equation can be written for each FIFO channel in the process graph leading to a system of equations that can be solved statically.

Example 5.4.17 Figure 5.31(a) shows an example of process graph with three actors A, B and C. The balance equations for these graph are the following:

1. Output 2 of A to input 1 of B: $3\mathbf{v}_S(A) - \mathbf{v}_S(B) = 0$
2. Output 2 of B to input 1 of C: $\mathbf{v}_S(B) - \mathbf{v}_S(C) = 0$
3. Output 3 of C to input 1 of A: $\mathbf{v}_S(C) - 2\mathbf{v}_S(A) = 0$
4. Output 3 of A to input 2 of C: $2\mathbf{v}_S(A) - \mathbf{v}_S(C) = 0$

This set of equations can be written in matrix form. Define the matrix M as follows:

$$M = \begin{bmatrix} 3 & -1 & 0 \\ 0 & 1 & -1 \\ -2 & 0 & 1 \\ 2 & 0 & -1 \end{bmatrix}$$

Let $V = \{v_1, \ldots, v_n\}$ be the set of actors and $E = \{e_1, \ldots, e_m\}$ be the set of FIFOs of a SDF network. Then the entry $M_{i,j}$ of the matrix is equal to n_p if actor v_i produces n_p token on FIFO e_j, $-n_c$ if actor v_i consumes n_c tokens from FIFO e_j and zero otherwise. Thus, the set of balance equations can be written as follows:

$$M\mathbf{v}_S = 0$$

This is a linear system of m equations in n unknowns, where m is the number of channels and n is the number of actors. If this system has no solution, then a periodic schedule does not exist. In this example, the columns of M are independent, therefore a non-zero vector $\mathbf{v}_S$ that satisfies the balance equations cannot be found. An intuition of why this is the case is that in order to maintain the balance on the channels between A and C, C must be executed twice the times than A. To maintain the balance on the channel between B and C, these two actors must be executed the same number of times, implying that B must be executed twice the times than A. But this last condition does not satisfy the balance equation on the channel between A and B.

Consider the process graph shown in Figure 5.31(b). It is the same as in Figure 5.31(a) but the number of tokens produced by the A on output port 2 is now equal to 2. The balance equation is the following:

$$\begin{bmatrix} 3 & -1 & 0 \\ 0 & 1 & -1 \\ -2 & 0 & 1 \\ 2 & 0 & -1 \end{bmatrix} \mathbf{v}_S = 0$$

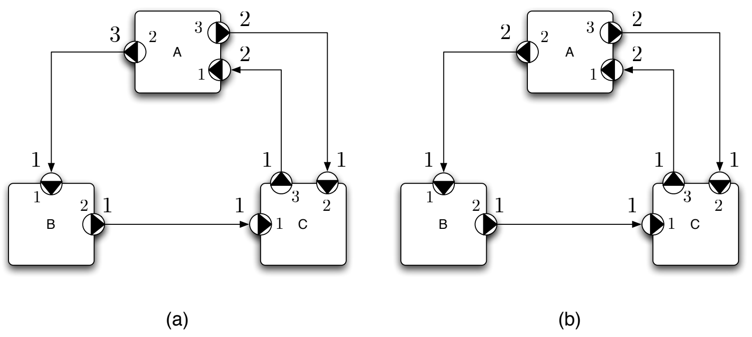


Figure 5.31: An SDF network with three actors A, B and C.

This system has an infinite number of solutions. Any multiple of the vector $\mathbf{v}_S = [1\ 2\ 2]^T$ is a valid firing vector. The firing vector indicates that actor B and C must be executed the same number of times that is twice the number of times that A is executed. Three valid schedules are $ABCBC$, $ABBCC$ and $ABABBCBCCC$. The first two schedules are called *minimal*, while the last one is non-minimal.

Example 5.4.17 gives us some hints on the existence of a valid schedule. First of all we notice that a connected graph with n vertexes must have at least $n-1$ edges. Therefore, the rank of the matrix M of a connected process graph is at least $n-1$. Also, the columns of matrix M cannot be all independent, otherwise a solution to the system cannot be found. Lee [?] proved the following theorem:

Theorem 5.4.18 *A connect SDF graph with n actors has a periodic schedule if and only if its topology matrix M has rank $n-1$. If M has rank $n-1$, then there exists a unique smallest integer solution q to $Mq = 0$.*

Although this theorem provides an existence result for a period schedule, it does not say anything about the admissibility of the schedule. Consider the example of Figure 5.31(b). A period schedule exists and we were also able to write one in Example 5.4.17. A minimal schedule is $ABCBC$. Unfortunately, this schedule is not admissible because A needs two tokens on input 1 to fire, but in the initial state there are no tokens in that FIFO channel. The admissibility of a schedule depends on the number of initial tokens that are present in the FIFO channels. In the example shown in Figure 5.31(b), adding two initial tokens in the channel between B and A makes the schedule admissible.

Need to add an example on how to fix a period, non-admissible schedule. Need to add boolean data flow. Need to add code and memory minimization.

5.4.5 Petri Nets

The Petri-net model of computation was developed by Carl Adam Petri during his Ph.D. [113] and published in English as a technical report in 1966 [114]. This model of computation was developed to model distributed computation, manufacturing processes, control processes, communication networks, transportation networks and in general systems that comprise a set of tasks depending on each other.

A Petri Net (PN) is a *bipartite weighted directed graph* with two set of vertexes: *places* and *transitions*. Transitions, that are graphically represented by bars or boxes, model actions or tasks. Places, that are graphically represented by circles, represent storage space used to store tokens that are produced by transitions. The arcs of the directed graph can only connect transitions to places or places to transitions, and are labeled with weights. The weight on an arc denotes the number of produced tokens (weights equal to one are omitted in the graphical representation of a PN).

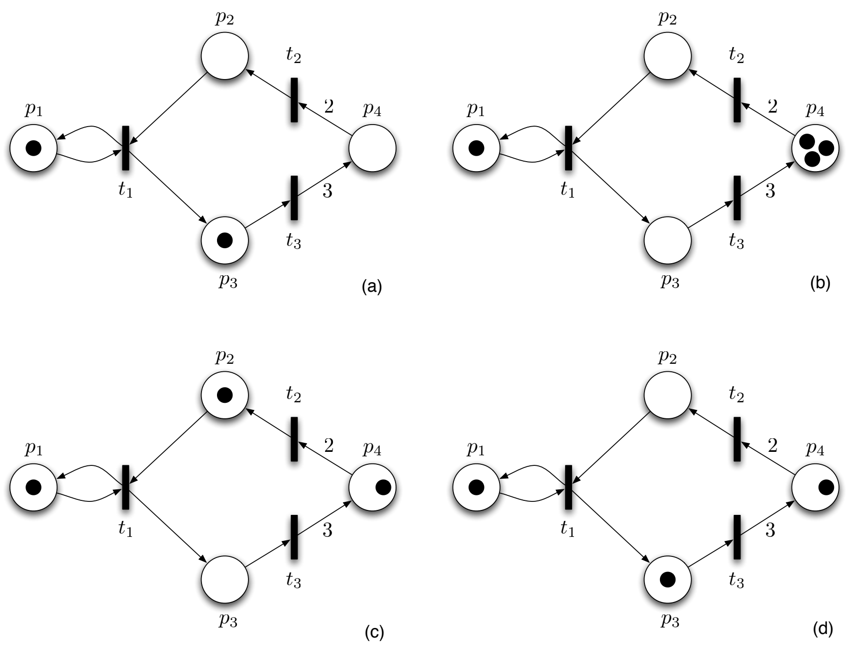


Figure 5.32: (a) Example of a Petri-Net with four places p_1, p_2, p_3 and p_4 and three transitions t_1, t_2 and t_3; (b) the state of the Petri-Net after transition t_3 has fired; (c) the state of the Petri-Net after transitions t_3 and t_2 have fired; (d) the state of the Petri-Net after transitions t_3, t_2 and t_1 have fired.

The example in Figure 5.32(a) shows the graphical notation used to describe a Petri-Net. This Petri-Net has four places, p_1 through p_4 and three transitions, t_1 through t_3. Black dots are used to denote tokens. The number of dots in each place is the *state* of the Petri-Net. Figure 5.32 shows a partial execution of the PN. Intuitively, the number of tokens in the input places of a transition is a precondition for the transition to be enabled. For instance, transition t_1 is enabled when there is at least one token in its input places p_1 and p_2. Similarly, transition t_3 is enabled when there is at least one token in place p_3, and transition t_2 is enabled when there are at least two tokens in place p_4. When a transition is enabled, then it can fire consuming tokens from the input places and producing tokens on the output places. For instance, upon firing, t_3 produces three tokens that are held in place p_4, and t_1 produces one token on place p_1 and one on place p_3. Consider the state of the Petri-Net in Figure 5.32(a). Transition t_3 is the only one that is enabled and that can fire. When it fires, it modifies the state of the Petri-Net into the one shown in Figure 5.32(b). Now, transition t_2 is enabled and can fire resulting in the state of Figure 5.32(c). Finally, transition t_1 can fire changing the state into the one shown in Figure 5.32(d). Notice that after firing all the transitions once, the state of the Petri-Net is not equal to the initial one. Later on in this section we will find a periodic schedule for a Petri-Net.

We can now introduce more precise definition of Petri-Nets. A *marking* of a Petri-Net is a vector M with m elements, where m is the number of places of the Petri-Net. The p-th element of the vector is the number of tokens in place p and is denoted $M(p)$. For instance, the initial marking in the example of Figure 5.32(a) is $M_0 = (1, 0, 1, 0)^T$. After transition t_3 fires, the new marking becomes $M_1 = (1, 0, 0, 3)^T$.

The formal definition of a Petri-Net is taken from an excellent survey written by Tadao Murata in 1989 [106].

We should add a comparison: input places=preconditions, transitions=events, output places=postconditions

Definition 5.4.19 (Petri-Net) *A Petri-Net is a* 5*-tuple* $PN = (P, T, F, W, M_0)$ *where:*

- $P = \{p_1, \ldots, p_m\}$ *is a set of places*
- $T = \{t_1, \ldots, t_n\}$ *is a set of transitions*
- $F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs*
- $W : F \to \mathbb{N}$ *is a weight function that associates to each arc a natural number called weight*
- $M_0 : P \to \mathbb{N} \cup \{0\}$ *is an initial marking that associates to each place a natural number denoting the number of tokens in that place*
- $P \cap T = \emptyset$, $P \cup T \neq \emptyset$

A Petri-Net structure is a tuple $N = (P, T, F, W)$ *without any specific initial marking. A Petri-Net with a given initial marking* M_0 *is denoted by a pair* (N, M_0).

Example 5.4.20 The Petri-Net in Figure 5.32(a) is the 5-tuple $PN = (P, T, F, W, M_0)$ where:

- $P = \{p_1, p_2, p_3, p_4\}$ is the set of the four places
- $T = \{t_1, t_2, t_3\}$ is the set of the three transitions
- $F = \{(p_1, t_1), (p_3, t_3), (p_4, t_2), (p_2, t_1), (t_1, p_2), (t_3, p_4), (t_2, p_2), (t_1, p_1)\}$ is the set of all arcs
- the weight function is defined as follows:

$$\begin{aligned}
&W(t_3, p_4) = 3\\
&W(p_4, t_2) = 2\\
&W(p_1, t_1) = W(p_3, t_3) = W(p_2, t_1) =\\
&= W(t_1, p_2) = W(t_2, p_2) = W(t_1, p_1) = 1
\end{aligned}$$

- the initial marking is such that $M_0(p_1) = 1$, $M_0(p_2) = 0$, $M_0(p_3) = 1$,$M_0(p_4) = 0$.

The behavior of a PN, i.e. the set of possible sequences of markings that can be generated by a PN model, depends on the *transition firing rule.* This rule defines when a transition is ready to fire as follows:

1. *Pre-condition*: A transition t is enabled if there are enough tokens in its input places, i.e. for each input place p of t the following condition holds:

$$M(p) \geq W(p, t)$$

2. *Execution*: An enabled transition may or may not fire.
3. *Post-condition*: A firing of an enabled transition changes the marking (state) of the Petri-Net by removing $W(p,t)$ tokens from each input place p and adding $W(t,p')$ tokens to each output place p'.

For instance, consider the case of Figure 5.32(a) for which a formal definition of the Petri-Net is given in Example 5.4.20. The input places to transition t_1 are p_1 and p_2. To check if t_1 is enabled, means to check that $M_0(p_1) \geq W(p_1,t_1)$ and $M_0(p_2) \geq W(p_2,t_1)$. Since the first condition is satisfied but the second is not, transition t_1 is not enabled.

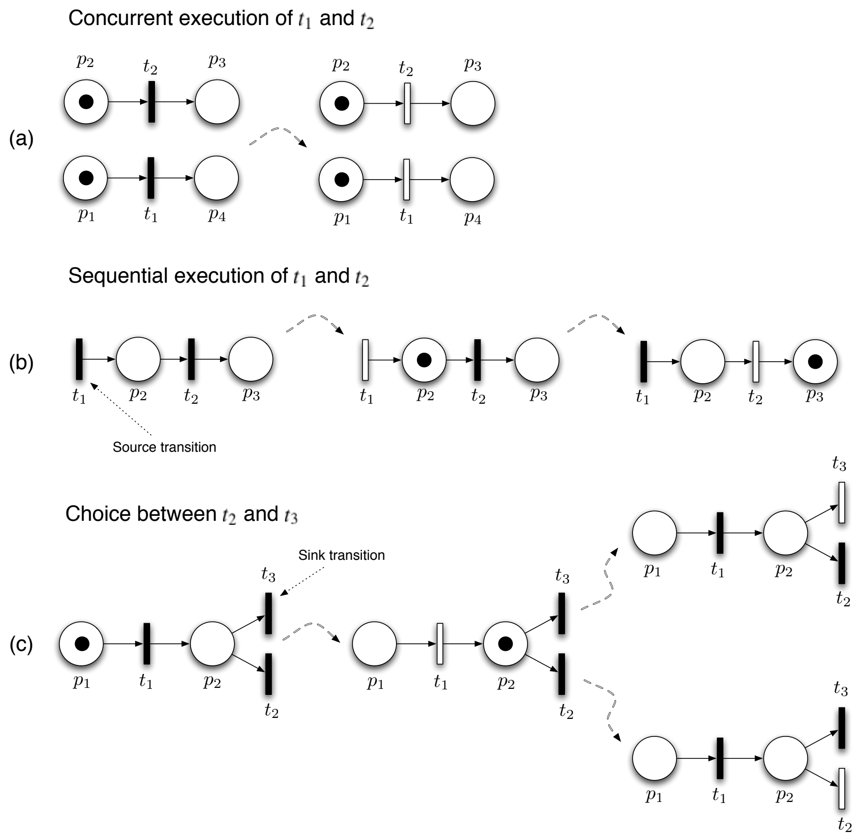


Figure 5.33: TBD

The Petri-Net model of computation can be used to capture *concurrency* (i.e. concurrent execution of transitions), *causality* (i.e. dependency among transitions) and *choice*. Figure 5.33 shows how these three features are modeled in the PN MoC. Consider the Petri-Net in Figure 5.33(a). This example can be considered an abstract model of the system of Example 5.7.5 where the input and output places represent the stack of incoming and outgoing mails, and the transitions are the stamping actions. A difference exists, though, between this model and the example because the tokens in a place are not ordered. Transitions t_1 and t_2 are both enabled because there is one token in each input place. Therefore they may or many not fire. However there is no restriction on which one has to be executed first. In fact, the

Petri-Net model of computation is un-timed, meaning that events (i.e. the firing of a transition) are only partially ordered. The two transitions can be executed concurrently moving both tokens from the input places to the output places.

Consider now the Petri-Net in Figure 5.33(b). Transition t_1 does not have any input place. These special kind of transitions are called *source transitions* and are always enabled. Transition t_2 cannot fire unless there is a token in place p_2. Therefore, transition t_2 will fire after transition t_1, capturing the sequential execution of the two actions associated with the two transitions.

Finally, consider the Petri-Net in Figure 5.33(c). Transition t_3 and t_4 are two special transitions with no output places. These special kind of transitions are called *sink transitions* and, upon firing, do not generate any token. After transition t_1 fires, a token is present in place p_2. Both transitions t_2 and t_3 are enabled. This condition is called *choice*. There are two possible executions of the Petri-Net: t_2 fires and consumes the token, in which case t_3 is not enabled anymore, or t_3 fires and consumes the token, in which case t_2 is not enabled anymore.

Remark 5.4.21 (On the meaning of a token)

Here we should clarify that a token does not have a value. A choice is an abstraction of a real choice. For verification and scheduling, both branches of the choice are important and must be considered.

Example 5.4.22 (Petri-Net model of a producer-consumer system)

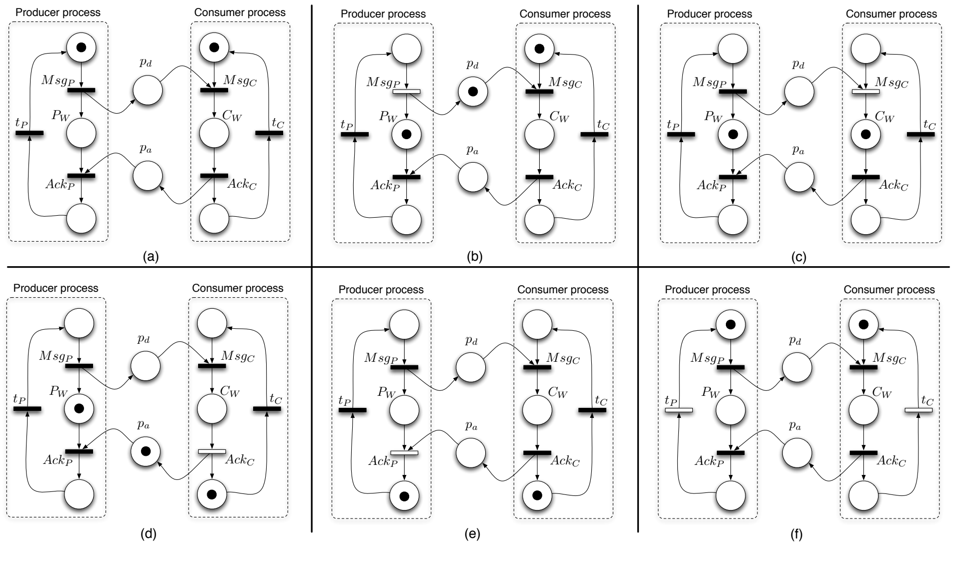


Figure 5.34: Execution of a simple producer-consumer example

This example is a model of a simple system with one producer and one consumer. Figure 5.34(a) shows the PN model of the producer process, consumer process and a buffer between the two. Initially, the producer is ready to generate a data to be sent to the consumer which, instead, is waiting for a data to arrive. When transition Msg_P fires, a token is placed in the data buffer p_d (Figure 5.34(b)) and a token is also placed in place P_W. Notice that a token in P_W encodes a state of the process that is waiting for an acknowledge. The consumer can now receive the token because transition Msg_C is enable. After firing transition Msg_C, a token is placed in C_W which indicates that a datum has been received (Figure 5.34(c)). The only enabled transitions now is Ack_C which sends an acknowledge to the producer process (Figure 5.34(d)). The producer, that was waiting for the acknowledge, can receive the acknowledge token and go back to the initial marking of Figure 5.34(a).

use the unbounded buffer example as one of the problems.

Properties of Petri-Nets

What are the interesting properties that we would like to be able to assert about a Petri-Net model? To answer this question we should understand what places and transitions model. Places are used to model the pre-conditions and post-conditions of events that are modeled by transitions. Places capture a sort of distributed state of the system. As in any system with states, an interesting question is wether a particular state, considered unsafe, is reachable from an initial state. Places also capture storage space for tokens. Eventually, a place will be implemented by a memory element. Therefore, another interesting question is wether the execution of a Petri-Net can accumulate an infinite number of tokens in a place.

The properties of Petri-Nets are divided into two classes: *behavioral* properties and *structural* properties. Behavioral properties are related to the evolution of the state of a Petri-Net and *depend on the initial marking.* Structural propetries depend only on the structure of the Petri-Net. Structural properties are verified independently from the initial marking and are, therefore, more general than behavioural properties. Unfortunately, such properties are very restrictive and are satisfied only in very few cases.

Interesting behavioural properties are:

Reachability . The firing of a transition in a Petri-Net changes the marking M. It is similar to a transition from one state to another in a finite state machine. A sequence of n firings results in a sequence of n markings form M_0 to M_n. In this case, marking M_n is *reachable* from M_0. The set of all reachable markings starting from M_0 is denoted by $R(N, M_0)$. The rechability problem for Petri nets is the problem of finding if a marking M_n belongs to the set $R(N, M_0)$ for a given initial marking M_0. The reachability problem is decidable [95].

Boundedness . The places of a Petri net represent buffers in real systems. Therefore, it is important that the maximum number of token that can accumulate in a place does not exceed a finite upper bound (i.e. the number of tokes does not go unbounded). A Petri net (N, M_0) is said to be k-bounded if for any reachable marking $M_n \in R(N, M_0)$, the number of tokens in each place does not exceed a finite number k, i.e $M_n(p) \leq k$, for all $p \in P$. A Petri net is said to be safe it is 1-bounded, i.e. the maximum number of tokens in each place for any reachable marking is equal to one.

Reversability . A Petri net is reversable if, for each marking M reachable from the initial marking M_0, M_0 is reachable from M. This property will be exploited later in the analysis method section to find a cyclic schedule of the transitions of a Petri Net.

Liveness . Each transition of a Petri net represents a task or action that is executed when the preconditions are satisfied, meaning that there are enough tokens in the input places. Of course, we don't want a transition to be idle all the time, otherwise it would represent a task that is never executed. A Petri net is said to be live if, no matter what marking has been reached, any transition in the Petri net can eventually be fired. Liveness implies deadlock-freedom but the converse is not true.

Should we add the definition of L0, L1 ... live?

Conservation . A Petri net is conservative if the sum of the number of tokens in each place is constant for all reachable markings.

In the description of the behavioral properties, we make explicit reference to the initial marking. Similar properties can be defined independently from the initial marking by looking only at the structure of a PN. There properties are more restrictive than their behavioral conterpart. For instance, structural boundedness implies behavioral boundedness for any finite initial marking, but the converse is not true in general (i.e. a PN (N, M_0) may be bounded but N may not be structurally bounded). Some interesting structural properties are the following:

Boundedness . A PN N is structurally bounded if it is bounded for any initial marking M_0.

Liveness . A PN N is structurally live is there exists an initial marking M_0 such that (N, M_0) is live.

Consistency . A PN N is consistent if there exists an initial marking M_0 such that it is possible to bring the PN back to M_0 by firing every transition at least once.

Analysis Methods

In this section we show some analysis techniques used to check structural and behaviroural properties of Petri-Nets. Analysis techniques can be divided into two classes: *structural* techniques and *state space* techniques. Structural techniques use only the structure (P, T, F, W) of a PN and infer properties that are valid independently from the sequence of markings. These methods are usually very efficient by offer only necessary or sufficient conditions. State space analysis techniques, instead, are mainly concerned with the states that can reached starting from a particular marking M_0. These techniques are more complex than the structural ones but they can provide necessary and sufficient conditions.

Incidence Matrix . The incidence matrix A of a Petri-Net structure (P, T, F, W) corresponds to the node-edge incidence matrix of a graph. Each row of the matrix refers to a place $p \in P$ and each column to a transition $t \in T$. If a place p is a predecessor of a transition t, then $A(p, t) = -W(p, t)$, meaning that when transition t fires consumes $W(p, t)$ tokens from p. If a place p is a successor of a transition t, then $A(p, t) = W(t, p)$, meaning that when transition t fires produces $W(t, p)$ tokens on place p. The entry $A(p, t)$ is equal to 0 when neither (p, t) nor (t, p) belong to F (i.e. when the place and the transitions are not connected).

The incidence matrix of the Petri-Net shown in Figure 5.35 is the following:

$$A = \left[\begin{array}{cccc} & t_1 & t_2 & t_3 \\ p_1 & 1 & 0 & 0 \\ p_2 & 1 & 1 & -1 \\ p_3 & 0 & -1 & 1 \end{array} \right]$$

The incidence matrix can be used to quickly check when a marking M is **not** reachable from a marking M_0. The concept of repetition or firing vecotr has been already introduced for data flow models (see Section 5.4.4). As in the case of SDF, a schedule for a Petri-Net must first be defined. A schedule is a sequence of firings of

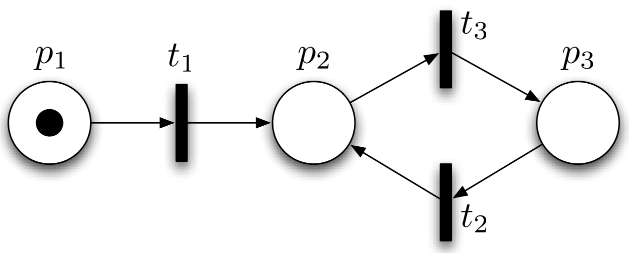


Figure 5.35: Example of a Petri-Net for which a periodic schedule exists.

transitions, that corresponds to a sequence of markings of the Petri-Net. Formally, a *firing sequence* is denoted by:

$$\sigma = M_0\ t_1\ M_n \ldots t_n M_n$$

of simply by $\sigma = t_1\ \ldots\ t_n$. For instance, a firing sequence of the Petri-Net in Figure 5.35 is the following:

$$\sigma = (1,0,0)\ t_1\ (0,1,0)\ t_3\ (0,0,1)\ t_2(0,1,0)$$

In this case, marking $(0,0,1)$ is reachable from $(1,0,0)$.

A firing vecotr v has one element for each transition such that $v(t)$ represents the number of times that a transition t fires in a firing sequence. At the end of a firing sequence, the total number of tokens in a place p is the sum of all the tokens produced by the transitions of which p is a successor, minus the sum of all the tokens consumed by the transitions of which p is s predecessor. Therefore, it can be computed by the following expression:

$$\sum_{t:(t,p)\in F} W(t,p)v(t) - \sum_{t:(p,t)\in F} W(p,t)v(t) = \sum_{t\in T} A(p,t)v(t)$$

This sum corresponds to the element-wise product of the row of the incidence matrix corresponding to p times the firing vector. Thus, given a firing vector v and an initial marking M_0, the marking M reached after all transitions have been fired can be computed as follows:

$$M = M_0 + Av$$

The underlying assumption is that the transitions can be fired, i.e. it is always possible to find a firing sequence that fires each transitions exactly the nubmer of times as specified by the firing vector. For instance, consider computing a firing vector for the Petri-Net in Figure 5.35 such that marking $M = (0,0,1)$ is reached. The firing vector is a solution to the linear system of equations $M = M_0 + Av$ that can be written as follows:

$$\begin{bmatrix} -1 & 0 & 0 \\ 1 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} v(t_1) \\ v(t_2) \\ v(t_3) \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ -1 \end{bmatrix}$$

The solution of this system is $v(t_1) = 1$ and $v(t_3) = v(t_2) + 1$ that corresponds to a space of solutions because the rank of the matrix is one less its dimension. Thus, any firing vector of the form $(1, k, k+1)^T$ satisfies the system of equations. In this simple case, it is possible to verify by hand that such firing vector correspond to a firing sequence where each transition can actually fire. One such sequence is $t_1\ t_3\ t_2\ t_3\ t_2\ t_3$.

If a marking M is reachable from the initial marking M_0 then there there exists a solution v to the state equation $M = M_0 + Av$. The converse is usually not true, meaning that if the state equation has a solution, M is not necessarily reachable from M_0. For instance, consider a Petri-Net with the same structure of the one in Figure 5.35 but with a different initial marking, namely $M_0 = (0, 0, 0)$. Is $M = (0, 0, 1)$ reachable? We may attempt to solve the state equation that becomes the following:

$$\begin{bmatrix} -1 & 0 & 0 \\ 1 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} v(t_1) \\ v(t_2) \\ v(t_3) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$$

The solution of this system of equation is $v(t_1) = 0$, $v(t_2) = v(t_3)$ meaning that any firing vector of the form $(0, k, k)$ should lead to marking M. However, M is actually not reachable because if the initial marking is $(0, 0, 0)$ no transition can be fired.

This algebraic method provides a necessary condition for reachability. It is very fast because it only requires to solve a linear system of equations. Therefore, this method can be used to check before the reachability tree (see Section 5.4.5) is constructed to rule out unreachable states.

Invariants The state equation can be used to find schedules for Petri-Nets and to check for structural boundedness.

Similarly to SDF models, a cyclic schedule of a Petri-Net (N, M_0) is a firing sequence that brings the Petri-Net back to its initial marking M_0. If a cyclic schedule exists, then there must exist a non-zero firing vector that satisfies the equation $M_0 = M_0 + Av$, i.e. $Av = 0$. The solutions to this equation are called *T-invariants*. For example, the T-invariant of the Petri-Net in Figure 5.35 are all the firing vectors of the form $(0, k, k)^T$.

Another important property is conservativeness that implies structural boundedness. A Petri-Net is said to be partially conservative if there exists a positive integer $y(p)$ for each place p such that the weighted sum of tokens $y^T M$ is constant for all reachable markings from any initial marking M_0. The state equation can be used to prove conservativeness as follows. Multiplying each side of the state equation by y^T we obtain:

$$y^T M = y^T M_0 + y^T Av$$

Since the weighted sum of the tokens in the places must be constant the following must hold :

$$y^T M = y^T M_0 \Rightarrow y^T Av = 0\ \forall v \Rightarrow y^T A = 0$$

Coverability Tree The coverability tree belongs to the class of state space analysis methods. Each node of the coverability tree represents a marking. There is an edge between two markings (i.e. between two node of the tree) if and only if the child marking can be reached from the parent marking by firing one transition. Figure 5.36 shows an example of such a tree for the Petri-Net of Figure 5.35. The root of the tree is the initial marking M_0 and each edge is labeled by a transition that, when fires, changes the marking into the one that is written in the landing node. Even if it may seem that this tree has an infinite number of nodes, this example suggests that some nodes may be discovered multiple times. Whenever a node is

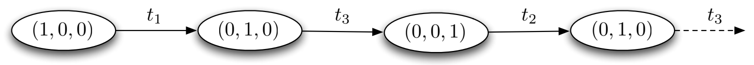


Figure 5.36: Example of a tree representing the marking evolution of a firing sequence of the Petri-Net of Figure 5.35.

discovered again, the entire tree rooted at that node has been certainly discovered, and there is no need to continue exploring it. Even if we stop the construction of the tree when new nodes cannot be found, the coverability tree may still grow unbounded. This is the case for unbounded Petri-Nets (see Problem).

The coverability tree can be made bounded in size with the following checking if the number of tokens in a place can go unbounded and using a special symbol in to indicate this condition. The algorithm to construct the coverability tree of a Petri-Net was originally developed by Karp and Miller [89]

Algorithm 2 Generation of the Coverability Tree

```
Input: A Petri-Net $(N, M_0)$
Output: The coverability tree
Label $M_0$ as the root of the tree and tag it new   while new markings exist do
    Select a new marking $M$   if $M$ is identical to a marking on the path from $M_0$
    to $M$ then
        Tag $M$ old
    else
        if No transitions are enabled at $M$ then
            Tag $M$ dead-end
        forall Enabled transitions $t$ at $M$ do
            Compute $M'$ obtained by firing $t$   On the path from the root to $M$, if
            there exists $M''$ such that $M'(p) \geq M''(p)$ for each $p \in P$ and $M' \neq M''$,
            replace $M'(p)$ by $\omega$ for each $p$ such that $M'(p) > M''(p)$   Insert $M'$ in
            the tree and an arc from $M$ to $M'$ with label $t$; tag $M'$ new
```

Algorithm 2 sketches the procedure to build the coverability tree. Each node of the tree is a marking of the Petri-Net that has a tag associated with it. The tag can be **new**, **old** or **dead-end**. The algorithm maintains a list of **new** markings that is initialized with the root M_0. A new marking might have been already found, in which case it should be in the tree on the path from the root to the current marking. If this is the case, the marking is tagged **old** and will not be considered anymore. If a marking is not old, then the algorithm starts exploring all enabled transitions. If there are no enabled transitions, than the marking is a **dead-end** meaning that the Petri-Net cannot progress further. Otherwise, for each enabled transition t at M, the algorithm computes the new marking M' obtained by firing t. Suppose there is another marking M'' on the path from the root to M' such that the number of tokens in each place in M'' is less that or equal to the number of tokens in each place in M' (and the two markings are different). Then, it would be possible to fire the same sequence of transitions that reached M' from M'' obtaining another marking M''' with even more tokens in each place. Therefore, those places are unbounded and this is denoted by writing an ω in each of them.

The coverability tree can solve the reachability problem for bounded Petri-Net and can verify if a Petri-Net is bounded. However, the ω sign does not tell us

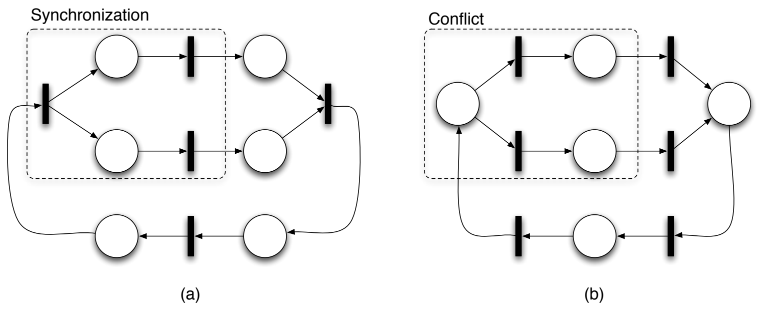


Figure 5.37: Example of (a) a marked graph and (b) a state machine.

anything about the reachability of a particular state (see Problem **??**), therefore this technique cannot solve the reachability problem in general.

Slide 72 of lecture notes.

Concuding Remarks

Special classes of Petri-Nets

The Petri-Net model of computation is very expressive. In fact, transitions are like data flow actors that allow to capture streaming computation. At the same time, a marking is a distributed state which makes this model of computation also able to capture state based systems. This expressiveness prevents us to develop methods that can provide strong verification results. In this section we show how, by putting restriction on the structure of a Petri-Net, we can find a compromise between expressiveness and complexity of analysis mthods.

The first class of Petri-Nets that we introduce is the so called *Marked Graphs* (MG). In a marked graph, each place has exactly one input transition and one output transition. Figure 5.37(a) is an example of MG. It is clear that MGs can model concurrency, causality and synchronization but they cannot model choice.

The second class of Petri-Nets is *state machines* (SM). In a state machine, each transition has exactly one input place and one output place. State machines are dual of marked graphs in the sense that given a MG, if each transition is replaced by a place and each place by a transition the resulting Petri-Net is a SM. Figure 5.37(b) shows an example of SM. SMs can model conflicts but they cannot model synchronization.

A third class of Petri-Nets that embodies the features of MG and SM is the class of *free-choice* Petri-Nets (FCPN). This class of Petri-Nets was introduced by Hack in 1972 [74] in his master thesis at the MIT. In a FCPN each transition after a choice has exactly one input place.

Figure 5.38(a) shows a FCPN. The two transitions t_2 and t_3 after the choice have only one predecessor place p_1. In this Petri-Net the choice of firing t_2 or t_3 only depends on the choice that is taken locally when p_1 holds at least one token. Figure 5.38(b) shows the case where a choice is not free because firing t_3 or t_2 depends also on the number of tokens in place p_4. The meaning of a choice being free is that the outcome of a choice depends only on the value of the tokens (that is abstracted non-deterministically) rather than on its arrival time. This is true for the Petri-Net in Figure 5.38(a) but it is not true for the one in Figure 5.38(b) because the decision betweeen t_2 and t_3 is conditioned on the arrival of a token in

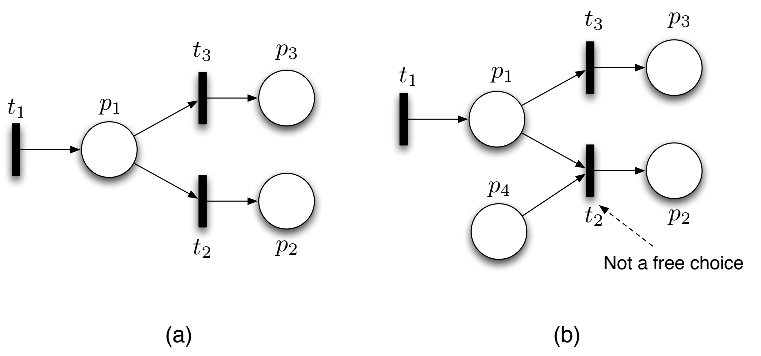


Figure 5.38: An example of Free-Choice Petri-Net (a) and a Petri-Net where the choice is not free (b).

p_4 prior to the choice.

FCPN have been extensively studied by Best [17] and Desel and Esperanza [43]. They can express concurrency, causality, synchronization and conflicts (but limited to free-choice). A very strong structural theory has been developed for FCPN. Necessary and sufficient conditions for liveness and safeness can be derived only based on the structure of the Petri-Net. The approach to derive these conditions is based on the decomposition of a FCPN in marked graphs and state machines.

Marked Graph (and State Machine) decomposition of a FCPN Consider a free choice Petri-Net. Each choice is represented by a place with multiple output transitions. Define an *allocation* to be a function $A : P \rightarrow T$ that establishes which transition to fire among the conflicting ones at the output of a place. Once an allocation has been decided, only one output transition for each place can fire. Thus, unused transitions can be removed from the Petri-Net. The result is obviously a conflict-free Petri-Net, i.e. a Marked Graph.

Algorithm 3 Marked Graph decomposition of a Petri-Net

```
Input: A Free-Choice Petri-Net structure $(P, T, F, W)$
Output: A set of Marked Graphs $MG$
$MG \leftarrow \emptyset$    forall Allocations $A : P \rightarrow T$ do
    $Tmp \leftarrow (P, T, F, W)$   Delete all unallocated transitions from $Tmp$, i.e. $t$ such
    that $(p, t) \in F$ and $A(p) \neq t$    repeat
        Delete all places form $Tmp$ that have all input transitions already deleted
        Delete all transitions from $Tmp$ that have at least one input place already
        deleted
    until Neither transitions nor places have been removed form $Tmp$ ;
    $MG \leftarrow MG \cup \{Tmp\}$
return $MG$
```

Algorithm 3 computes the marked graphs obtained by selecting all possible allocations. An example of the execution of the algorithm is shown in Figure 5.39. The FCPN shows in this example has only one choice which is represented by place p_1 and its two output transitions t_1 and t_2. Thus, there are two possible allocations corresponding to the two choices where t_1 fires or t_2 fires. For each allocation, for instance the one that selects t_1, the non-allocated transitions are

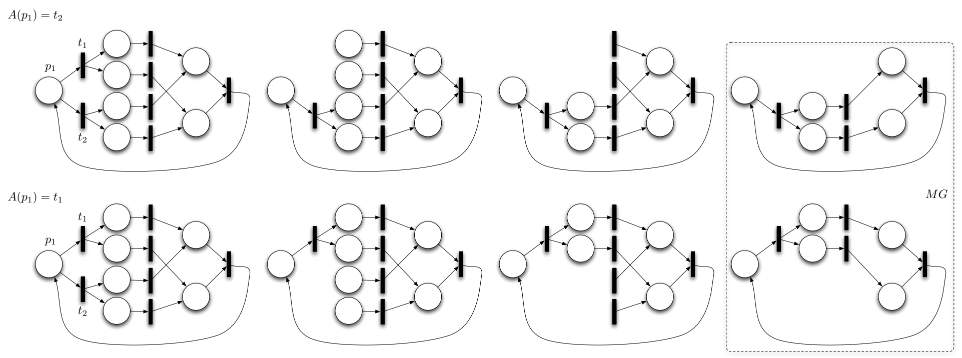


Figure 5.39: The pictorial representation of the execution of Algorithm 3.

deleted from the Petri-Net structure. Deleting transitions will leave some places with no input transitions. The algorithm continues by deleting those places as well. As a consequence of deleting places, some transitions will remain without output places. These transitions are also removed from the Petri-Net. The algorithm continues until neither transitions nor places can be deleted.

For each allocation, a marked graph is generated and stored into a set. When the algorithm terminates, it returns the set of all marked graphs. The following theorem gives necessary and sufficient conditions for the liveness and safeness of a FCPN

Theorem 5.4.23 *Let N be a FCPN, then N has a safe and live initial marking if and only if*

- *Every Marked Graph reduction is strongly connected and non empty, and the set of all reductions covers N, or*
- *Every State Machine reduction is strongly connected and non empty, and the set of all reductions covers N.*

5.4.6 Continuous Time

The continuous time model of computation has been implicitly used in several engineering discipline such as automatic control and circuit analysis and design. In the continuous time model of computation, the inputs and outputs of a process are waveforms. A waveform is a function of time $f : \mathbb{R} \rightarrow \mathbb{R}$ that associates a value to each time instance $t \in \mathbb{R}$. For instance, $\sin(\omega t)$ is a sinusoid with angular frequency ω.

A continuous time block, or process, can be described by a set of equations that define the output waveforms depending on the input waveforms. Simple examples of these equations are the one defining a process that operates the sum of two waveforms or a process that simply scales its input by a constant factor (Figure 5.40. The SCALE process is defined by the following equation:

$$y(t) = k \cdot u(t), \quad \forall t \in \mathbb{R}$$

where k is the scaling factor. The SUM process is defined by the following equation:

$$y(t) = u_1(t) + u_2(t), \quad \forall t \in \mathbb{R}$$

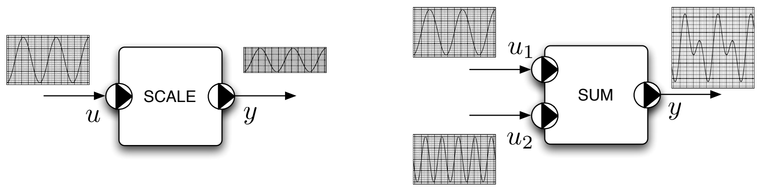


Figure 5.40: Examples of a continuous time process that scales the input by a constant factor (process SCALE), and a process that sums the inputs (process SUM).

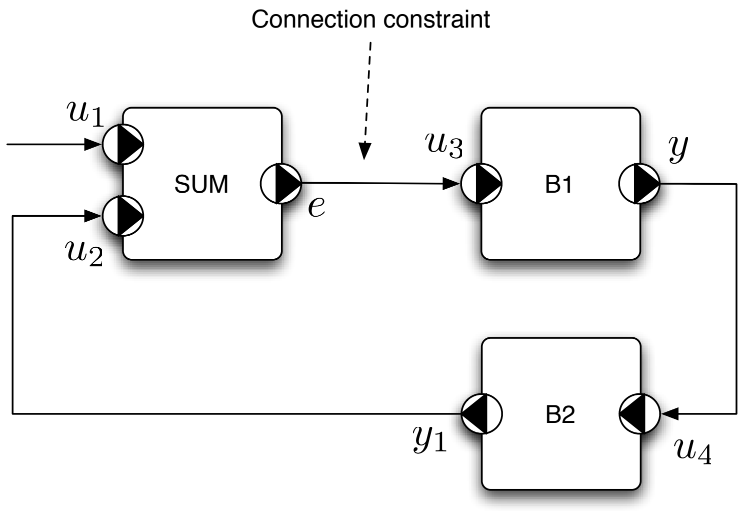


Figure 5.41: Feedback connection of continuous time systems.

Processes can be connected together by defining connection constraints. Pictorially, these constraints are represented by arrows connecting the outputs of some processes to the inputs of others as in Figure 5.41. In this figure, each arrow corresponds to one equation that imposes the output waveform of one process to be equal to the input waveform of another process. For instance, the connection constraint pointed in Figure 5.41 corresponds to the equation:

$$u_3(t) = e(t), \quad \forall t \; in \mathbb{R}$$

Thus, a continuous time system corresponds to a system of equations. Some of these equations describe the behavior of each process and some of the are connection constraints. Assume, for instance, that the two processes B1 and B2 in Figure 5.41 are described by equations $y(t) = f_1(u_3(t))$ and $y_1(t) = f_2(u_4(t))$, respectively. Then the entire system corresponds to the following set of equations:

$$\begin{aligned} e(t) &= u_1(t) + u_2(t) \\ y(t) &= f_1(u_3(t)) \\ y_1(t) &= f_2(u_4(t)) \\ u_3(t) &= e(t) \\ u_4(t) &= y(t) \\ u2(t) &= y_1(t) \end{aligned}$$

Computing the behavior of the entire system for a given input waveform $u_1(t)$ corresponds to computing all the waveforms of the system, this means solving the system of equations. Depending on the functions f_1 and f_2, a closed form solution may or may not be easy to find. Therefore, simulation of continuous time systems is a very important verification tool.

Example 5.4.24 Room temperature control with a pid loop.

Example 5.4.25 Power circuit

Example 5.4.26 Electronic circuit.

5.4.7 Hybrid Systems

The first example of heterogeneous model of computation that we discuss is a particular composition of discrete event and continuous time systems. This type of models are known as hybrid systems. A hybrid system has a continuous evolution and occasional jumps. It is difficult to find a direct example in nature of a hybrid system since our perception of natural phenomena is continuous. However, in many cases, the evolution of a system appears to us as having instantaneous jumps. A classical example is a bouncing ball. Consider a ball that falls from a certain height. We observe the ball accelerating until it hits the ground. After the collision takes place, the velocity instantaneously changes direction and the ball moves up with a decreasing velocity. If we observe the collision at a finer time scale, we realize that the change in velocity is actually continuous. From a macroscopic point of view, though, we are not interested in the entire process. In particular, we are not interested in what happens in the short interval of time where the ball is in contact with the ground. We abstract this interval and we simply say that the velocity changes instantaneously. This abstraction has very serious implications that we will discuss later in this section.

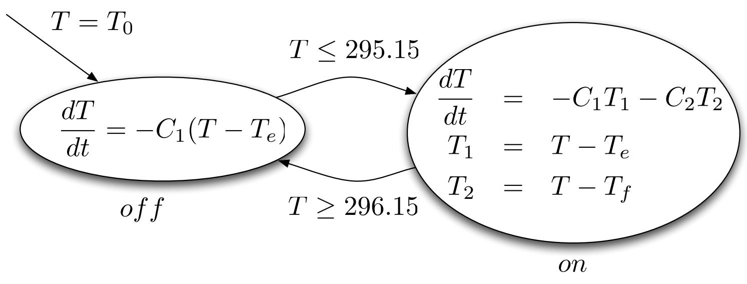


Figure 5.42: The hybrid model of temperature conditioned by a thermostat.

Other examples of hybrid systems are mechanical systems controller by electronic systems. Physical quantities like pressure, temperature, velocity or acceleration are measured by sensors. The electric signal generated by the sensors is converted to digital values and processed by the electronic systems which then sends commands to actuators that are able to influence the temporal evolution of the physical quantities. While the mechanical system is governed by physical laws that are continuous in time, the electronic system is driven by a clock which makes it discrete in time.

As an example, consider the time evolution of the temperature of an environment that is controlled by a thermostat. The thermostat is a sensing device that measures temperature and commands an actuator like a heater. The heater can be turned on or off. The temperature of the room, is different depending on the state of the thermostat. If the heater is on, the temperature increases as a function of the difference between the heater and room temperatures. If the heater is off, the temperature decreases. The equations that regulate the temporal evolution of temperature are very similar to the equation of the voltage across a capacitor that is connected to a voltage source by a resistor. Figure 5.42 shows a first model based on intuition. Intuitively, we have modeled the discrete behavior of the thermostat with a finite state machine that has two states: *on* and off. In each state of the finite state machine we have written the equations that describe the room temperature. Switching from one state to the other is regulated by transitions that are guarded by a condition on the temperature itself.

The initial state is the off. Since the hybrid system also has continuous time variables as state (temperature in this case), we also have to define the initial condition on this variables. In this case, the temperature T is set to an initial value T_0. In the off state, the room exchanges heat with the environment, thus the temperature tend to the temperature of the environment T_e. When temperature drops below 22 degrees C, the thermostat turns the heater on. In this state the equations change. There is still a contribution due to the heat exchange with the environment but also the heater warms up the room with an equivalent temperature T_f. When the temperature reaches 23 degrees C, the thermostat turns the heater off. Notice that the assumption is that as soon as the guard condition is enabled (meaning it evaluates to true), the hybrid system makes a transition from one state to the other. This type of semantics of the transition is called *triggering semantics*. A different semantics says that if the guard condition is enabled, then the transition may be taken but it doesn't have to be taken. With this semantics, it is not guaranteed that the temperature stays between 22 and 23 degrees.

The language used to describe hybrid systems has two more features that are

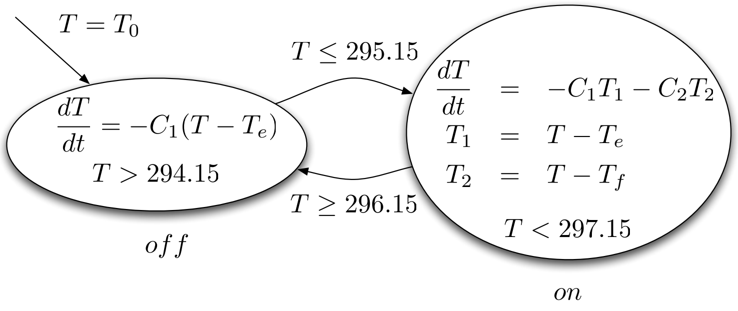


Figure 5.43: The hybrid model of temperature conditioned by a thermostat. This model includes invariant conditions.

called *invariants* and *resets*. An invariant is a condition associated to a state. If the condition is true, then the system can stay in that state, otherwise it has to switch to another state. This means that when the hybrid system is in one state and the invariant becomes false, there must be at least one transition from that state with an enabled guard condition. Figure 5.43 shows the same model of the thermostat that also includes invariant conditions. In this model, the switching from off to on state happens between 21 and 22 degrees, and the switching from off to onn happens between 23 and 24 degrees.

Another useful feature is the possibility of resetting the value of state variables when a transition is taken. The bouncing ball is one example where this feature is useful. After hitting the ground, the velocity of the ball has to change sign. Moreover, because some energy is lost in the impact, the magnitude of the velocity is also scaled by a constant factor. Figure 5.44 shows the hybrid system model of the bouncing ball. In this model, y denotes the vertical position of the ball, v the vertical velocity, g is the gravitational acceleration, y_0 is the starting quota and ϵ is a loss factor. The initial condition is that the ball starts from a positive height y_0 and initial velocity v_0. Subject to gravity, the ball will fall and when it touches the ground, we reset its velocity. The reset condition is written as follows:

$$\begin{aligned} y' &= y \\ \dot{y}' = -\epsilon\dot{y}' \end{aligned}$$

where primed variables denotes the values after the transition while non-primed variables denote the values before the transition. The first reset condition guarantees continuity of the position variables. The second reset condition says that the velocity after the collision is equal to the velocity before the collision scaled by a factor ϵ and with opposite direction.

While this informal description seems rather simple, the precise definition of the evolution of a hybrid system is quite complex. Early work on formal models for hybrid systems includes *phase transition systems* [7] and *hybrid automata* [103]. These somewhat simple models were further generalized with the introduction of compositionality of parallel hybrid components in *hybrid I/O automata* [102] and *hybrid modules* [8]. In the sequel, we follow the classic work of Lygeros et al. [101] to formally describe a hybrid system as used in the control literature.

We consider subclasses of continuous dynamical systems over certain vector fields X, U and V for the continuous state, the input and disturbance, respectively. For

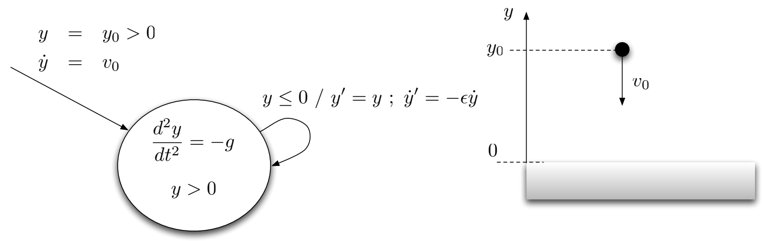


Figure 5.44: The hybrid system model of a bounding ball.

this purpose, we denote with $\mathcal{U}_C$ the class of measurable input functions $u : \mathbb{R} \to U$, and with $\mathcal{U}_d$ the class of measurable disturbance functions $\delta : \mathbb{R} \to V$. We use the symbol $\mathbf{S}_C(X, U, V)$ to denote the class of continuous time dynamical systems defined by the equation

$$\dot{x}(t) = f(x(t), u(t), \delta(t))$$

where $t \in \mathbb{R}$, $x(t) \in X$ and f is a function such that for all $u \in \mathcal{U}_C$ and for all $\delta \in \mathcal{U}_d$, the solution $x(t)$ exists and is unique for a given initial condition. A hybrid system can then be defined as follows

Definition 5.4.27 (Hybrid System) *A continuous time hybrid system is a tuple* $\mathcal{H} = (\mathbf{Q}, \mathbf{U}_D, E, X, U, V, \mathcal{S}, Inv, R, G)$ *where:*

- $\mathbf{Q}$ *is a set of states;*
- $\mathbf{U}_D$ *is a set of discrete inputs;*
- $E \subset \mathbf{Q} \times \mathbf{U}_D \times \mathbf{Q}$ *is a set of discrete transitions;*
- X, U *and* V *are the continuous state, the input and the disturbance, respectively;*
- $\mathcal{S} : \mathbf{Q} \to \mathbf{S}_C(X, U, V)$ *is a mapping associating to each discrete state a continuous time dynamical system;*
- $Inv : \mathbf{Q} \to 2^{X \times \mathbf{U}_D \times U \times V}$ *is a mapping called* invariant*;*
- $R : E \times X \times U \times V \to 2^X$ *is the reset mapping;*
- $G : E \to 2^{X \times U \times V}$ *is a mapping called* guard*.*

Note that we can similarly define *discrete time* hybrid systems by simply replacing $\mathbb{R}$ with $\mathbb{Z}$ for the independent variable, and by considering classes of discrete dynamical systems underlying each state. The triple $(\mathbf{Q}, \mathbf{U}_D, E)$ can be viewed as an automaton having state set $\mathbf{Q}$, inputs $\mathbf{U}_D$ and transitions defined by E. This automaton characterizes the structure of the discrete transitions. Transitions may occur because of a discrete input event from $\mathbf{U}_D$, or because the invariant in Inv is not satisfied. The mapping $\mathcal{S}$ provides the association between the continuous time definition of the dynamical system in terms of differential equations and the discrete behavior in terms of states. The mapping R provides the initial conditions for the dynamical system upon entering a state.

The transition and dynamical structure of a hybrid system determines a set of *executions*. These are essentially functions over time for the evolution of the continuous state, as the system transitions through its discrete structure. To highlight the discrete structure, we introduce the concept of a hybrid time basis for the temporal evolution of the system, following [101].

Definition 5.4.28 (Hybrid Time Basis) *A hybrid time basis τ is a finite or an infinite sequence of intervals*

$$I_j = \{t \in \mathbb{R} : t_j \le t \le t'_j\}, \quad j \ge 0$$

where $t_j \le t'_j$ and $t'_j = t_{j+1}$.

Let $\mathcal{T}$ be the set of all hybrid time bases. An execution of a hybrid system can then be defined as follows.

Definition 5.4.29 (Hybrid System Execution) *An execution χ of a hybrid system $\mathcal{H}$, with initial state $\widehat{q} \in \mathbf{Q}$ and initial condition $x_0 \in X$, is a collection $\chi = (\widehat{q}, x_0, \tau, \sigma, q, u, \delta, \xi)$ where $\tau \in \mathcal{T}$, $\sigma : \tau \to \mathbf{U}_D$, $q : \tau \to \mathbf{Q}$, $u \in \mathcal{U}_C$, $\delta \in \mathcal{U}_d$ and $\xi : \mathbb{R} \times \mathbb{N} \to X$ satisfying:*

1. *Discrete evolution:*
   - $q(I_0) = \widehat{q}$;
   - *for all j, $e_j = (q(I_j), \sigma(I_{j+1}), q(I_{j+1})) \in E$;*
2. *Continuous evolution: the function ξ satisfies the conditions*
   - $\xi(t_0, 0) = x_0$;
   - *for all j and for all $t \in I_j$,*

     $$\xi(t, j) = x(t)$$

     where $x(t)$ is the solution at time t of the dynamical system $\mathcal{S}(q(I_j))$, with initial condition $x(t_j) = \xi(t_j, j)$, given the input function $u \in \mathcal{U}_C$ and disturbance function $\delta \in \mathcal{U}_d$;
   - *for all j, $\xi(t_{j+1}, j+1) \in R\left(e_j, \xi(t'_j, j), u(t'_j), v(t'_j)\right)$*
   - *for all j and for all $t \in \left[t_j, t'_j\right]$,*

     $$(\xi(t, j), \sigma(I_j), u(t), v(t)) \in Inv\,(q(I_j))$$

   - *if τ is a finite sequence of length $L+1$, and $t'_j \neq t'_L$, then*

     $$\left(\xi(t'_j, j), u(t'_j), v(t'_j)\right) \in G\,(e_j)$$

We say that the behavior of a hybrid system consists of all the executions that satisfy Definition 5.4.29. The constraint on discrete evolution ensures that the system transitions through the discrete states according to its transition relation E. The constraints on the continuous evolution, on the other hand, require that the execution satisfies the dynamical system for each of the states, and that it satisfies the invariant condition. Note that when the invariant condition is about to be violated, the system must take a transition to another state where the condition is satisfied. This implies the presence of an appropriate discrete input. Because a system may not determine its own inputs, this definition allows for executions with blocking behavior. When this is undesired, the system must be structured appropriately to allow transitions under any possible input in order to satisfy the invariant.

Note also that the same input may induce different valid executions. This is possible because two or more trajectories in the state machine may satisfy the same constraints. When this is the case, the system is non-deterministic. Non-determinism is important when specifying incomplete systems, or to model choice or don't care situations. However, when describing implementations, it is convenient to have a deterministic specification. In this case, one can establish priorities among the transitions to make sure that the behavior of the system under a certain input is always well defined. Failure to take all cases of this kind into account is often the cause of the inconsistencies and ambiguities in models for hybrid systems.

Definition 5.4.30 *A hybrid system execution is said to be (i) trivial if $\tau = \{I_0\}$ and $t_0 = t'_0$; (ii) finite if τ is a finite sequence; (iii) infinite if τ is an infinite sequence and $\sum_{j=0}^{\infty} t'_j - t_j = \infty$; (iv) Zeno, if τ is infinite but $\sum_{j=0}^{\infty} t'_j - t_j < \infty$.*

The Zeno behavior of a system is a rather interesting phenomenon resulting from the abstraction of certain dynamics. The bouncing ball is a classical example of such behavior. Let us compute the time interval between two bounces. We use an index k ranging from 0 to infinity to denote the interval number. In the k-th interval, the initial velocity is v_k and the initial position is equal to zero. To compute the next time the ball hits the ground, we need to solve the following equation

$$-gt^2 + v_k t = 0$$

which has two solutions: $t = 0$ (the trivial solution) and $t = 2v_k/g$. The velocity at the next impact is still equal to $-v_k$ by symmetry. Therefore, the initial velocity of the $k+1$-th interval is $v_{k+1} = \epsilon v_k$. The total length of the hybrid system execution can them be computed as follows:

$$t_0 + \frac{2v_0}{g} \sum_{i=1}^{\infty} \epsilon^i$$

Since $\epsilon \leq 1$, the sum is finite, meaning that the execution is Zeno. This is the consequence of the abstraction that we have made in considering an instantaneous bounce.

Example 5.4.31 Obstacle avoidance.

Example 5.4.32 Power systems with contactors.

5.5 Composing Models of Computation

In the course of this chapter, we presented many modeling languages to capture the behavior of systems. Each language is suitable for modeling systems belonging to a certain class. For instance, Finite State Machines are used to model control dominated systems while dataflow languages are used to model signal processing applications. Moreover, each MoC also comes with a set of tools that enable formal verification and synthesis.

A complex embedded systems seldom fits in only one application domain. A simple vending machine, discussed in Section 5.2.1, can be conveniently described by a finite state machine that, however, only capture its logical behavior. An output event that indicates that the machine is serving a coffe is actually a command to an electromechanical system that is in charge of dispensing a cup and filling it up with requested product. This is not a unusual situation if one considers that an embedded system interacts with a physical system through sensors and actuators. Many interesting behaviros that are worth analyzing arise from this very same interaction, which requires to model both the logical behavior and the embedded system and the physical behavrior of the environment. The use of hybrid systems (Section 5.4.7) to model the interaction between the contious and the discrete time parts of an embedded systems is one example of heterogenous modeling.

Heterogeneity does not only arise when considering the combination of a continuos time plant[5] and a digital controller. Within an embedded system, it is common to find several application domains with different modeling needs such as control, analog and digital signal conditioning and processing, human-machine interface software, network protocol stack etc. All these sub-systems need to interact to deliver the required functionality. For instance, the analog signal received by a transceiver is first processed by an analog front-end. Then, the signal is converted into a digital representation and it is processed by a digital signal processor that recovers the transmitted packets. The header of the packet is analyzed to generate signals for a state machine that sends commands to the rest of the protocol stack software to determine the content of the payload and deliver it to the application software.

In this section we will discuss the definition of the interaction among sub-systems that are described using semantically heterogenous languages. We will discuss three different ways of defining such interaction. A method that requires to refine all sub-systems into a common model of computation, thereby obtaining an homogenous system. A method that composes models hierarchically where each level of the hierarchy is a homogenous system. A method that provides the special constructs, meaning special interfaces, to define the direct interactions among components described using different MoCs.

Example 5.5.1 (A simple Wifi model) To understand the difficulties a designer faces when composing models with different semantics, consider the following simple description of the basedband processing provided by the IEEE 802.11[6] standard. In this example, we look at the processing performed by the physical layer which receives a packet to send from the MAC layer. The packet to be sent is called MAC Packet Data Unit (MPDU). The structure of the physical level packet, called PPDU, is shown in Figure 5.45.

The packet starts with a preamble of 144 bits. The preamble is divided into a series of bits used for synchronization (SYNC) and a Start Frame Delimiter (SFD). The frame is also divided into two parts: a header that defines the properties of the packet (e.g. data rate, length and service type), and the MPDU.

The operations carried out buy the physical layer are a mix of control and computation. The heterogeneous nature of the operations is clear by reading the specification provided by the standard. The preamble and the header of the packet have to be transmitted using a data rate of 1 Mb/s and DBPSK modulation. The MPDU can be transmitted using different data rates and a different type of modulation. Moreover, the interactions between the physical layer and the MAC layer is determined by a handshaking procedure that is clearly depedent on the state of the two layer. On the other hand, the stream of bits to be transmitted undergoes a sequence of elaborations that are data-processing oriented.

Figure 5.46 shows one possible block diagram that describes the structure of the physical layer of a transmitter. We omit the processing required after the mapping of bits on a constellation for the sake of simplicity. Each block of the data processing

[5]By plant we mean the physical system controlled by the embedded system.
[6]Reference to the standard needed here.

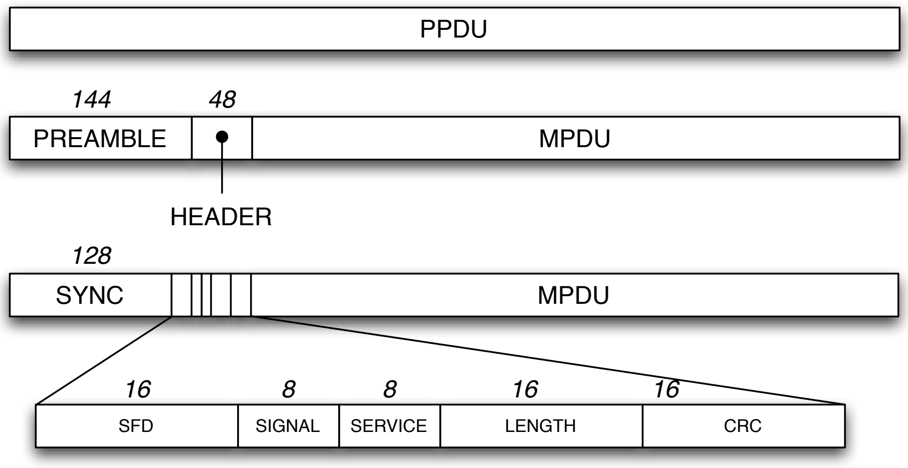


Figure 5.45: The structure of a physical layer packet data unit.

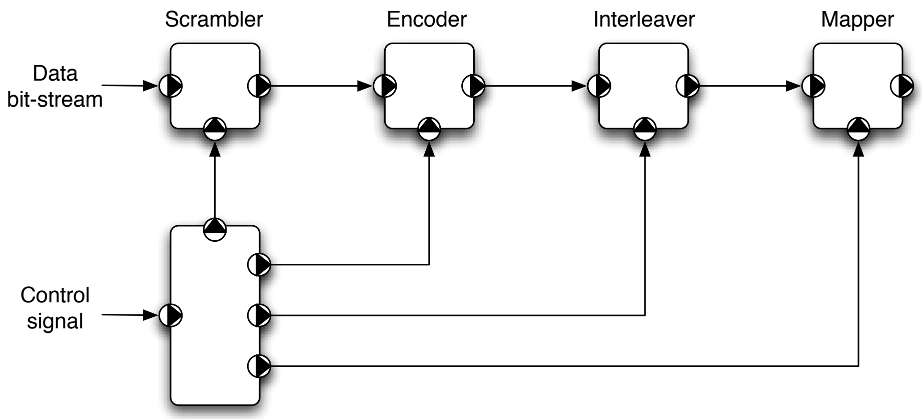


Figure 5.46: Block diagram of an 802.11 transmitter.

performs a specific task:

- The *scambler* takes care of adding enough diversity in the transmitted bits. Intuitively, if the sequence of trasmitted bits is constant, it would be very difficult for the receiver to synchronize its clock to march the phase of the trasmitted signal. The scambler makes sure that such patological situation does not happen. Only the preamble had to be processed by the scrambler.
- The *encoder* encodes the bits according to a scheme used for error recovery. Convolutional codes are used in the 802.11 protocol.
- The *interleaver* performs and interleaving of the trasmitted bits. Interleaving is need to avoid that erros accourring in burts will not be corrected. In fact, the error correction is only capable of correcting one erroneous bits every n bits (where n depends on the encoding scheme). If bits are interleaved and a burst of erros occurs, only a few bits for each codeword will actually be erroneous, thereby enhancing the overall performance of the receiver.
- The *mapper* maps a sequence of bits to one point of a bidimensional constellation.

The details of each block are not essential to our discussion. It is sufficient to understand that the preamble of the PPDU has to be scrambled and trasmitted at 1 Mb/s with DBPSK modulation while the MPDU must not be scrambled and can be trasmitted at a different rate, also corresponding ot a different modulation scheme. The controller block is a state machine that controls the operation of each of the data processing block.

The simple interaction that is interesting to analyze is the one between the controller and the scrambler.

5.5.1 The method of common refinement

5.5.2 Hierarchical composition of heterogeneous models of computation

Hybrid systems are a good example of this approach. Each state of a finite state machine represents a continuous time system described by differential (and possibly algebraic) equations. In general, this concept can be exploited to compose the finite state machine model with other models of computations, such as data flow or discrete event. Conversely, a finite state machine model can be used to define the input-output relation of a data flow actor. This way of composing hetorogeneous models of computation is the one used by Ptolemy II [**?**, **?**].

An illustrative example is shown in Figure 5.47. The top level model is a static data flow with three actors A, B and C. Actors A and B consume one token and produce one token, while actor C consumes two tokens on each input and produces one token. A periodic schedule for this data flow model is ABABC. Actor A and actor C are refined into two finite state machines D and E, respectively. State b of D is refined into another finite state machine F, while state h of E is refined into a data flow model containing two actors, G and H. State g of E is refined into another date flow model containing three actors: I, J and K. Intuitively, when actor A fires, state machine D is executed. Similarly, when actor E fires, state machine E is executed. If D is in state b and it is executed, then state machine F is also executed. If E is in state h, the entire data flow model that refines h is also executed.

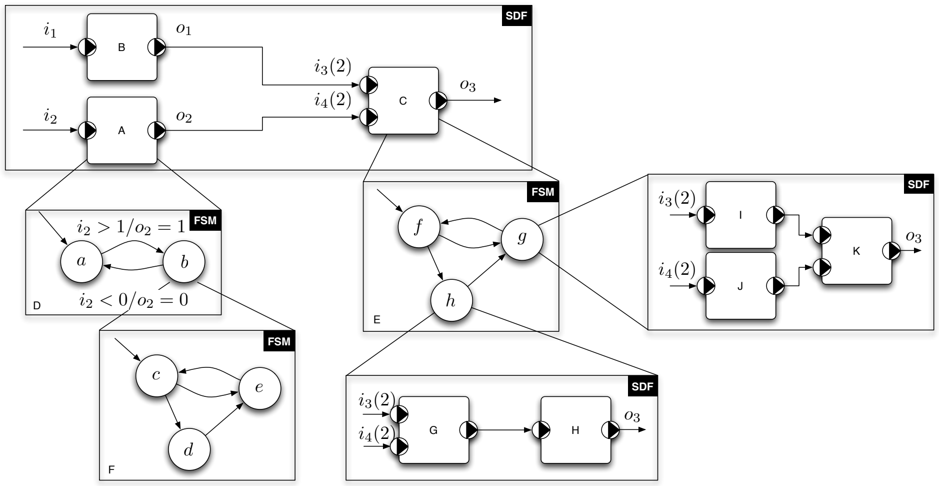


Figure 5.47: An example of hierarchical composition of different models of computation.

This simple example shows many of the subtleties that arise when models are composed hierarchically. Actor A has a static toke production and consumption rate. This property must be maintained by any refinement of the actor. Consider the state machine D in state a. When actor A fires, if the value of the input is greater than 1 then the state machines makes a transition to the state b and emits an output equal to 1. If the value of the input is less than 1, the state machine remains in state a and the output is not emitted (i.e. it is absent). Since one token has to be emitted for each reaction of the finite state machine, we must encode the absence of a token into a value in the data flow model. This value can be a default value or it can be a special value that is added to the set of possible values of the signals in the top level data flow model.

Consider the case where D is in state b, actor A is fired and the value of input i_2 is less than 0. State machine D is executed and the transition back to state a is enabled. However, state b is refined into another state machine that should also be executed. Therefore, the refinement (i.e. F) is executed first and the the transition in D is taken. Since the refinement F is executed, it may also take a transition[7] and emit the output o_2 that is then emitted, and equal to 0, by D. If the values emitted by D and F are the same, then the output is uniquely defined. If the two values are different, a rule must be defined to combine the two values. Notice that the choice of executing the refinement first and the transition of the higher level FSM after is one possible choice. Another choice is to always execute the transition of the higher level FSM first and the refinement of the arrival state after. However, the problem of resolving the possible incompatibility of the values emitted by the two FSMs is still to be addressed.

FSM E has two states refined into two different data flow models. If the state machine is in state h and actor C is fired, then the refinement of state h is first

[7]In this example we have not specified the guard conditions and the output actions of each transition in the finite state machines since they are not essential to understand the problems when composing models hierarchically.

executed. Since the state machine is supposed to execute in a finite amount of time (where time is an abstract concept at this abstraction level), the data flow model that refines a state must also run for a finite amount of time. This condition requires to define an iteration for each model that refines a state of state machine. The case presented in the example is particularly simple to handle. Since the refinement of state h is a static data flow, it is possible to compute a periodic schedule which implicitly defines an iteration of the model. In this simple case, an iteration is the successive firing of G and H. For the refinement of state g, one iteration is the firing of I, J and K. When the state machine is executed, one iteration of the refinement of the current state is executed first and then the state machine can possibly make a transition. Notice that there is a consistency constraint that the SDF refinements must satisfy. Since actor C consumes two tokens from each input and produces one token on its output, each refinement must also consume two tokens from i_3 and i_4 and produce one token on o_3, as shown in Figure 5.47.

5.5.3 Direct composition of heterogeneous models

The hierarchical composition of heterogenoeus models, despite the subtlelties that the designer should be aware of, simplyfies the definition of the interactions among heterogenous components. The simplication comes from the homogenous nature of the models at the same level of the hierarchy. Thus, the change of model appears only at the inner boundary of a component and the interaction between two models is determined by the way in which the inner model is controlled by the outer model.

This approach, though, has some limitations. Some components are naturally described by heterogenous interfaces. One simple example is an analog-to-digital converter that has two ports: an input port that accepts continuous time waveforms and an output port that esposes a discrite time representation of the input. In principle, we could represent this component as being a continous time one by representing the exact waveform of the discrete time output (i.e. by representing the output as a waveform with sharp transitions as in a sample-and-hold model). However, this approach corresponds to cosidering the analog to digital converter as a continuos time componet, thus giving up the potential benefit that a discrite time abstraction could bring to the table.

Perhaps, the most important limitation is in the structural difference between the model and the real system. Example 5.5.2 shows a simple system that explains these potential discrepancy.

Example 5.5.2 (A simple wifi transceiver)

A more natural way of describing a heterogenous embedded system would be to allow the direct connection of components described using different MoCs. These type of connections, however, are not easy to define. For instance, Example **??** shows a system where finite state machines and dataflow actors are mixed at the same level of the hierachy.

Example 5.5.3 (Digital receiver)

As described by Example **??**, the definition of the interfaces between the different components is not uniquely defined. The way in which events are exhanged between a finite state machine block and a dataflow actor depends on the design intent. Fixing one particular translation would limit the expressiveness of the heterogenous model and, consequently, the type of designs that can be captured.

There are few rigorous approaches to provide designer with a language for the description of the interaction between heterogenous components. Languages such as Simulink, VHDL-AMS and Modelica, all define the concepts of discrete and contnuos time variables that can be exposed through ports. However, these environment do not really allow modeling heterotenous systems, but they are rather specialized to deal with discrete events and continuous waveforms.

Among the academic approaches, we describe two here: the ModHel'X and the MetropolisII ones.

ModHel'X

Metropolis II

5.5.4 Interface Automata

5.6 Analysis and Verification

The advantage of using formal models to capture the specification of a system reveals itself in the possibility of using analysis tools. Perhaps, the most common analysis tool is simulation. The simulation of a model can give many insights into its behavior. The design selects a certain sequence of inputs, also called stimuli, under which she expects the system to behave in a certain way and produce a sequence of outputs. If the behavior is not the one expected by the designer, the model undergoes a debugging phase which can also be facilitated by tools. Finding errors through simulation requires the designer to select the input sequences in such a way that all corner cases are covered. This task is particularly difficult for complex systems. Therefore, bugs can go undetected in the final product.

Formal methods, such as model checking and theorem proving, provide a way of verifying that a system satisfies certain properties independently on what are the stimuli. Model checking is used to verify properties of finite state systems. This technique takes two inputs: the model of the system to verify and a set of properties described using formulae defined on the state space. The result is either that the system satisfies the properties or, most interestingly, a counterexample that for some input sequence the properties are not satisfied. The computational complexity of this technique is exponential in the size of the system, where by size we mean the number of states. Several improvement have been devised to make practical use of model checking such as the use of ordered binary decision diagrams (OBDD) [25] to represent transition functions and the use of composition and abstraction. Compositional reasoning aims at inferring properties on the entire system by looking at the properties satisfied by each component. Abstraction consists in hiding some of the information of the current system and derive another system with a smaller state space. For instance, assume that the state variables $x_1, \ldots, x_n$ are of integer type and that variable x_i takes on value from a domain D_i. The the state space is the cross product $D_1 \times \ldots \times D_n$. Depending on the size of each domain, the state space may be too big to be exhaustively searched buy model checking tools. However, we main abstract these system by abstracting the real value of the variables ad only exposing their sign. Of course, properties that depend on the actual value of the variables cannot be checked.

Theorem proving has the advantage of being able to verify infinite sate space models. In theorem proving, a proof can be constructued by starting from axioms and applying inference rules. An inferecen rule has premises (that are propositions) and a conclusion. The inference rule has the following meaning: if all the premises are provable (withing the proof system), then the conclusion holds. Given a proof system, proving a theorem means to find a chain of inference such that starting from the axioms the desired conclusion can be found. Even if theorem proving technique are very powerful, they cannot be fuly automated. Theorem provers are rather tools that help humans in building the proof of a therem. Conversely, model checking can be fully automated.

5.6.1 Simulation

5.6.2 Formal Methods

Model Checking of Finite State Systems

Model checking was jointly proposed by Clarke and Emerson [35] and Queille and Sifakis [116] in 1982. Model checking is a technique to verify that a finite state system satisfies a property expressed as a formula in some logic, which is also called specification. If the proterty is satified, it means that the finite state system is a model for the given specification.

Model checkers accept the description of the finite state system in a formal language that is then interpreted as a finite state transition system. Following the approach of Clarke and Emerson, we will use Kripke structures defined as follows.

Definition 5.6.1 *A Kripke structure is a tuple $M(AP, S, S_0, R, L)$ where:*

- *AP is a set of atomic propositions,*
- *S is a finite set of states,*
- *$S_0 \subseteq S$ is a set of initial states,*
- *$R \subseteq S \times S$ is a transition relation such that for each s tate $s \in S$ there exists a sate $s' \in S$ such that $(s, s') \in R$,*
- *$L : S \rightarrow 2^{AP}$ is s function that labels each state with the set of proposition that are true in that states.*

To undestand the meaning of this structure and the way in which automatic model checking works, we use a simple example take from [34]. Consider two sequential processes P_1 and P_2 as shown in Figure 5.48. Each process has a program counter that represents the address of the current instruction beigh executed. The two processes can executed independently from each other but when they reach a block of instruction called *critical sections*, and denoted by C_1 and C_2 in Figure 5.48. Only one of the two processes can enter the critical section. If one of them, say P_1 is executing C_1, then process P_2 cannot execute C_2. This corresponds, for instance, to accessing the same variable in memory. Before process P_i tries to access the critical section, it attempts to do so by executing a “trying” instruction T_i. If the execution is succesful, the process access the critical section.

Any implementation of this system must satisfy two important properties:

- the two processes P_1 and P_2 never enter the critical sections C_1 and C_2 concurrently (*mutual exclusion*).
- Each process is eventually granted the access to its critical section (*absence of starvetion*). A system where only one proceess can enter its critical section, althoguh it prevents the simultaneous access, is not correct.

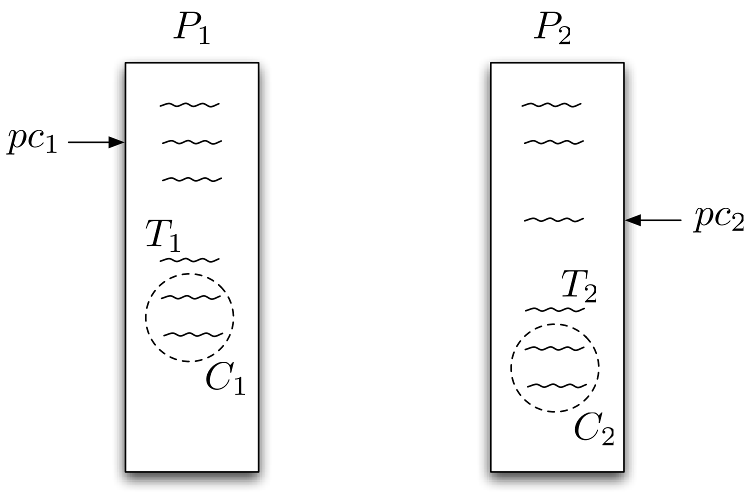


Figure 5.48: Example of two sequential processes P_1 and P_2 with critical sections C_1 and C_2, respectively.

A finite state space model for this example can be built by observing the values of the two program counter. Each pair of values of the program counters represents a new state. Mutual exclusion can be easily identified with a set of states, i.e. the set of all states where the two program counters are not in the critical sections. Absence of starvation is not so obvious to define as a set of states. This property requires to look at sequences of states or computation path. Absence of starvation requires that, along a computation path, access to the critical section is eventually granted to both processes. A path in M is an infinite sequence of states $(s_0, s_1, \ldots)$ such that each pair $(s_i, s_{i+1}) \in R$, meaning that there is a transition in M for each pair of adjacent states in the path. When a formula f is true in a state s_0[8] of a Kripke structre M, we use the notation $M, s_0 \models f$.

These properties are expressed by formulas in a logic called Computational Tree Logic (CTL). The syntax of this logic is defined by the following rules:

- AP is an underlying set of atomic propositions.
- Each atomic proposition $p \in AP$ is a CTL formula.
- If f_1 ad f_2 are two CTL formulas, then $\neg f_1$, $f_1 \wedge f_2$, AXf_2, EXf_2, $A[f_1\ U\ f_2]$ and $E[f_1\ U\ f_2]$ are also CTL formulas.

We define the semantics of CTL formulas later in this section, but we give an intuition of the meaning of the path quantifies A and E and of the *until* connective U. The quantifier A stands for *for all paths* while E stands for *for some path* (or *there exists a path*). X is the next step operator. Intuitively, the formula AXf_1 is true in a state s if for all paths, f_1 is true in the successor state of s. The formula $A[f_1\ U\ f_2]$ is true in a state s_0 if for all paths $(s_0, s_1, \ldots)$, f_1 holds until a state a reached where f_2 holds. Formally, the semantics of CTL is defined in terms of a

[8]We use the symbol s_0 to denote the first state of a path. The reader should not think to an initial state only.

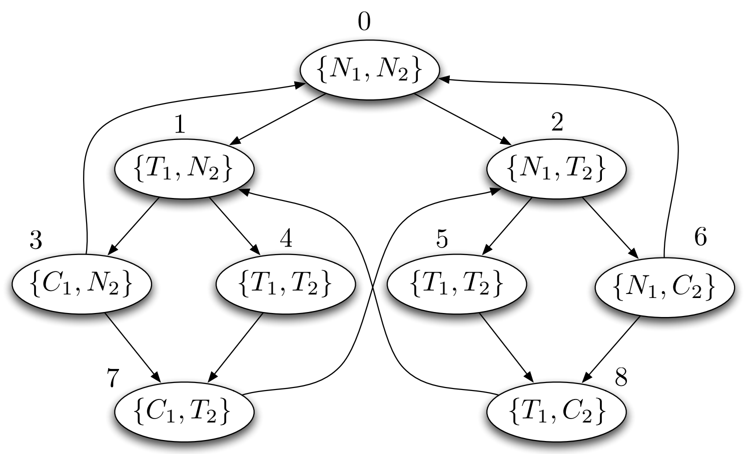


Figure 5.49: A finite transition system for the two-process example.

Kripke structure M. The relation $\models$ is defined inductively as follows:

$$\begin{aligned}
s_0 \models p &\iff p \in L(s_0)\\
s_0 \models \neg f &\iff not(s_0 \models f)\\
s_0 \models f_1 \wedge f_2 &\iff s_0 \models f_1 \ and\ s_0 \models f_2\\
s_0 \models AXf_1 &\iff for\ all\ t\ such\ that\ (s_0, t) \in R,\ t \models f_1\\
s_0 \models EXf_1 &\iff for\ some\ t\ such\ that\ (s_0, t) \in R,\ t \models f_1\\
s_0 \models A[f_1\ U\ f_2] &\iff for\ all\ paths\ (s_0, s_1, \ldots)\ \exists\, i \geq 0\ such\ that\\
&\qquad s_i \models f_2\ and\ for\ all\ 0 \leq j < i\ s_j \models f_1\\
s_0 \models E[f_1\ U\ f_2] &\iff for\ some\ path\ (s_0, s_1, \ldots)\ \exists\, i \geq 0\ such\ that\\
&\qquad s_i \models f_2\ and\ for\ all\ 0 \leq j < i\ s_j \models f_1
\end{aligned}$$

Other fomulas can be defined in terms of the ones defined above. Consider the formula AFf that is true in a state s_0 if for all paths starting at s_0, f is true in some future state along the path. This formula can be written as $A[true\ U\ f]$ because the atomic proposition *true* is trivially true in all states. Similarly, EGf that is true in a state s_0 if if there exists a path such that f is true in all states of the path can be written as $\neg AF \neg f$, meaning that *it is not true that for all paths f never holds.*

Using the values of the program counters to define the state of the system is not a convenient choice in our case. The number of states of this model is the square of the number of possible values that a program counter can take on, which can be potentially very large. Since we are not interested in the detailed execution of each instruction of the two processes, we can abstract the domain of values of the program counters into a simple three-value domain where the values are N, indicating that the program counter in not ina critical region, C indicating that the program counter is in the critical region and T, indicating that the program counter points at the instruction that tries to access the critical region.

Figure **??** shows a finite transition graph of the two-process example. Each state corresponds to a particular combination of abstract values of the program counters. In the fugure we have use the values also as atomic proposition and we have labelled each state with the set of atomic proposition that are true in that state. For instance, label $\{N_1, N_2\}$ is a set of two atomic propositions where N_i means *process i is in a non critical region*. The state transition graph represents a specific implementation where only one process at the time is allowed to advance in its program. We now express the two properties that we are interested in using CTL formulas:

- mutual exclusion : $\neg EF(C_1 \wedge C_2)$ (there is no path that eventually leads to a state where both processes are in the critical section).
- Absence of starvation for process P_1: AFC_1 (for all path, eventually process P_1 is allowed to access its critical section). A similar formula can be written for process P_2.

It is easy to verify that the mutual exclusion formula is true in every state.

The rest of this section defines the verification problem and gives an algorithm to solve it. Given a Kripke structure $M(S, S_0, R, L)$, it is possible to define a *lattice* of predicates over S, $P(S)$. To do this, given a predicate Z (i.e. a formula defined on the state space S), let $S[Z]$ be the set of states that satisfy Z. For instance, consider the example in Figure 5.49 and the predicate C_1. The set of states that satisfies C_1 is $S[C_1] = \{3, 7\}$. Predicates can be ordered by set inclusion. Given two predicates Z_1 and Z_2, $Z_1 \leq Z_2$ if and only if $S[Z_1] \subseteq S[Z_2]$. For instance, the predicate $C_1 \vee C_2$ is such that $S[C_1 \vee C_2] = \{3, 7, 6, 8\}$, thus $C_1 \leq C_1 \vee C_2$. In particular, $S[false] = \emptyset$ and $S[true] = S$, meaning that $false$ is the least element of the lattice and $true$ is the greatest element.

A *predicate transformer* is a function $\tau : P(S) \rightarrow P(S)$ that mas predicates into predicates. A predicate transformer is monotonic if $Z_1 \leq Z_2$ implies $\tau(Z_1) \leq \tau(Z_2)$. Monotonicity is very important because the set of states satisfying a formula can be defined as the least or greatest fix point of a predicate transformer that is guaranteed to exist if it is monotonic. We need to show that, given a CTL formula, computing the set of states that satisfy the formula is equivalent to computing the fix point of a predicate transformer. Consider the formula $A[f_1\ U\ f_2]$. The set of states that satisfy this formula is the union of two sets:

- the states where f_2 is satisfied;
- the states where f_1 is satisfied and, for all successor states $A[f_1\ U\ f_2]$ is satisfied

Therefore, the solution of the following equation is the set of states that satifies the predicate:

$$A[f_1\ U\ f_2] = f_2 \vee (f_1 \wedge AX(A[f_1\ U\ f_2]))$$

Consider the folliwng predicate transformer

$$\tau(Z) = f_2 \vee (f_1 \wedge AXZ)$$

It can be shown that this predicate transformer is monotonic. The least fixed point of this predicate is the set of states that satisfy $A[f_1\ U\ f_2]$. Algorithm 4 computes the least fixed point of an input predicate transformer τ.

Consider now the property that we want to verify that is $AFC_1 = A[true\ U\ C_1]$. The execution of the Algorithm 4 is shown in Table 5.2. Predicate Q is initialized to $false$. The first step of the algorithm computes the following predicate:

$$\tau(false) = C_1 \vee (true \wedge AXfalse) = C_1 \vee false = C_1$$

Algorithm 4 Least fixed point computation

```
Input: $\tau$ predicate transformer
Output: Predicate $Q$
$Q \leftarrow false$
$Q \leftarrow \tau(Q)$
while $Q \neq Q'$ do
    $Q \leftarrow Q'$
    $Q' \leftarrow \tau(Q)$
```

Step	Q	$S[Q]$
0	$false$	$\emptyset$
1	$C_1 \vee (true \wedge AXfalse) = C_1$	$\{3, 7\}$
2	$C_1 \vee (true \wedge AXC_1) = C_1 \vee AXC_1$	$\{3, 7, 4\}$
3	$C_1 \vee AX(C_1 \vee AXC_1))$	$\{3, 7, 4, 1\}$

Table 5.2: Values of the predicate Q and set of states $S[Q]$ at each step of Algorithm 4.

The set of states that satisfy this predicate are 3 and 7. The second step of the algorithm computes $\tau(\tau(false))$ and so on. Notice that each time the predicate transformer is applied, the algorithm increases the length of the computation path that has to be looked at. The predicate that is returned corresponds to the set of states $S[Q] = \{3, 7, 4, 1\}$. The result means that if the initial state is one of among $S[Q]$, precess P_1 eventually enters the critical section independently from the computation path. This is also easy to verify by inspection in this simple example. In particular, state 0 does not satisfy this property.

The major problem of model checking is its complexity that depends on the number of sates. This technique can be used only for systems with a limited state space. A major breakthrough came later in when McMillan used OBDD to represent transition functions []. This technique is called *symbolic model checking* and it is conceptually similar to model checking. The speed improvement is achieved by using OBDD and defining the operators $\vee$, $\wedge$ and the predicate transformer directly on the OBDDs.

Verification of Hybrid System Models

5.7 In Depth

5.7.1 Abstract Syntax and Abstract Semantics

5.7.2 Denotational Frameworks

Tagged Signal Model

In order to compare different models of computation, we need to introduce a mathematical tool able to capture the fundamental properties of each model. The description of a model can be given as a set of functions representing the possible input-output relations of the agents of that model. This type of interpretation of a model is called *denotational semantics*. A different way of describing a model is by giving an algorithm that dictates the rules to execute the agents in a system. The result of algorithm is the behavior of the systems. This type of interpretation of a model is called *operational semantics*.

We use the Tagged Signal Model (TSM) as a denotational framework to describe and compare models of computation. In TSM, a *process*, that is an agent like the adder in Figure 5.10, is a set of *behaviors*. Intuitively a behavior is the input-output relation of the process. Inputs and outputs are called *signals*. A signal is a set of *events* where an event capture what happens on that signal at a specific "time". We will see that the notion of time is abstracted into something else called *tag* that can potentially capture more sophisticated "time" structures.

In the exposition of the TSM we follow a reverse order by defining the events first, then the signals and then the behaviors.

Events and Signals There are many examples of signals. In electronics, we learn what digital and analog signal are. We also learn that an analog signal can be sampled and elaborated digitally. Figure 5.50 show the process of sampling a signal to obtain a "discrete" version of it. The sampled signal is discrete both in time and in its values.

The original sinusoid is a continuous time signal that is well described by a function of time:

$$f(t) = sin(2\pi t)$$

In this case we only represented one period of the sinusoid. The function $f(t)$ associates one value of the signal to each value of the time. To sample the sinusoid, we decide a sampling step $\Delta t = 0.1$ and only take one sample of the signal every Δt seconds. Therefore, the new *sampled* signal can be represented by a sequence of values now:

$$f(0),\ f(0.1),\ f(0.2), ...$$

Each of the values if obviously associated with a time stamp. The same applies also to the continuous case but it is difficult to represent the continuous waveform with a sequence because the time variable t is a real number.

One sample of the signal can be represented by a pair (v, t) where v is the value and t is the times. In our case we have $(f(t), t),\ t = k\Delta t$. The sampled signal f, therefore, can be easily represented by the set of such samples.

Remark 5.7.1 (Sequences and sets) We changed the representation of the signal from sequence to set. This is a very important step. Notice that the purpose of the sequence is to maintain an order among the values of the samples signal. Since we added the time information in the pairs that define the signal, it is not necessary

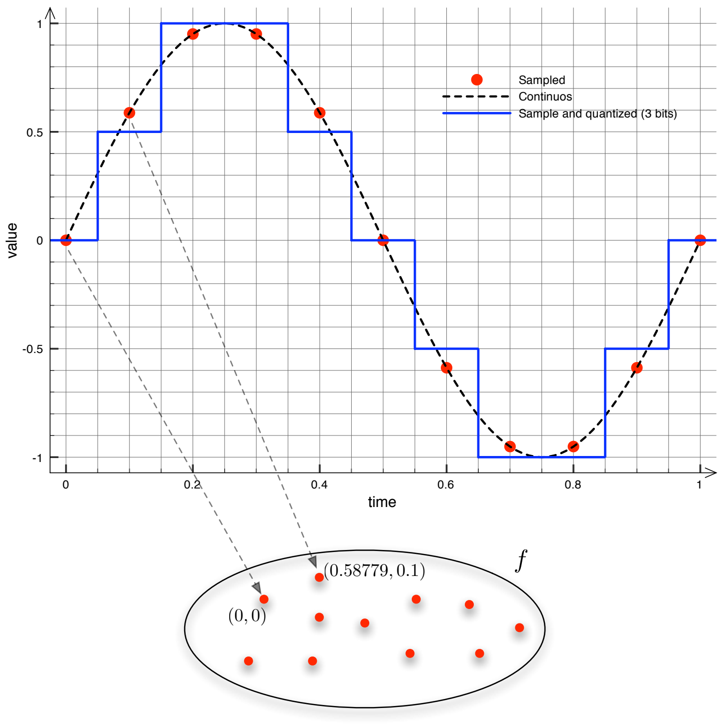


Figure 5.50: One period of a sinusoid

anymore to maintain such order by using a sequence because given two samples, it is always possible to order them by looking at the time.

Signals can be generalized using the more abstraction concept of *events*.

Definition 5.7.2 (Event) *Let V be a set of values and T a set of tags. An event e is an element of the set $V \times T$.*

A signal is a set of events. Given the set V and T, a signal does not have to contain all possible events.

Definition 5.7.3 (Signal) *Let V be a set of values and T a set of tags. A signal s is an element of the power set $2^{V \times T}$.*

Example 5.7.4 In our example of the sampled sinusoid, the set of values is the set of real numbers: $V = \mathbb{R}$. The set of tags is the set is the set of time instants equally spaced by $\Delta t = 0.1s$: $T = \{k \cdot 0.1 \ s.t. \ k \in \mathbb{Z}\}$.

The signal f is the set of samples of the sinusoid $f(t)$, therefore:

$$f = \{(v,t) \in V \times T \ s.t. \ v = f(t)\}$$

The set of events belonging to the signal f is a subset of $V \times T$. For instance, the pair $(0, 0.1)$ is an event that does not belong to the signal.

After sampling, the signal has to be represented with a finite number of bits. If we use for instance 3 bits to quantize the signal, then the set of possible values is restricted to $V' = \{-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5\}$.

We introduced the notion of tags using the sinusoid examples. In this example the set of tags are real numbers that can be ordered: given two tags t_1 and t_2 it is always possible to compare them and establish with one is the smallest (using the relation $\leq$ that is defined on the real numbers). The tag set, in this case, is said to be *totally ordered.*

The tag set, though, can be more general than time. Indeed, the set of tags could be only *partially ordered* meaning that some tags could be *not comparable.* For instance, if the tag set contains colors, i.e. $T = \{read, yellow, gree\}$, then it is not possible to compare them and establish whether *red* is less than *yellow* [9].

Example 5.7.5 (Partially ordered tags) Two independent offices, that handle two different kind of mails, receive piles of mails in the morning (Figure 5.51). The order of the mail in stack is the order in which they arrived at the office. The mail must be stamped in the same order and placed in an outbox to be delivered. Can the office employee tell exactly when the mail where stack in the pile? Obviously, since the pile is received in the morning and not in real time, the employee cannot answer this question. Nevertheless, she can tell which one came first. Definitely, the pile is ordered. We can imagine that each mail is an event where the value is the mail itself (the value is not important for our discussion), and the tag is just drawn from a set of tags.

The set is only partially ordered. For instance, we can say that: $t_3 \leq t_2 \leq t_1$ and that $t_9 \leq t_8 \leq t_7 \leq t_6$. Can we establish an order among the tags t_3 and t_9? In order words, can be say if mail v_9 arrived before mail v_3? Such order cannot be

[9] It is possible to define an order among colors by using their RGB values. The RGB code of a color is a triple of integers that can be ordered lexicographically. In this example we are using labels for the colors for which an natural ordering relation does not exist.

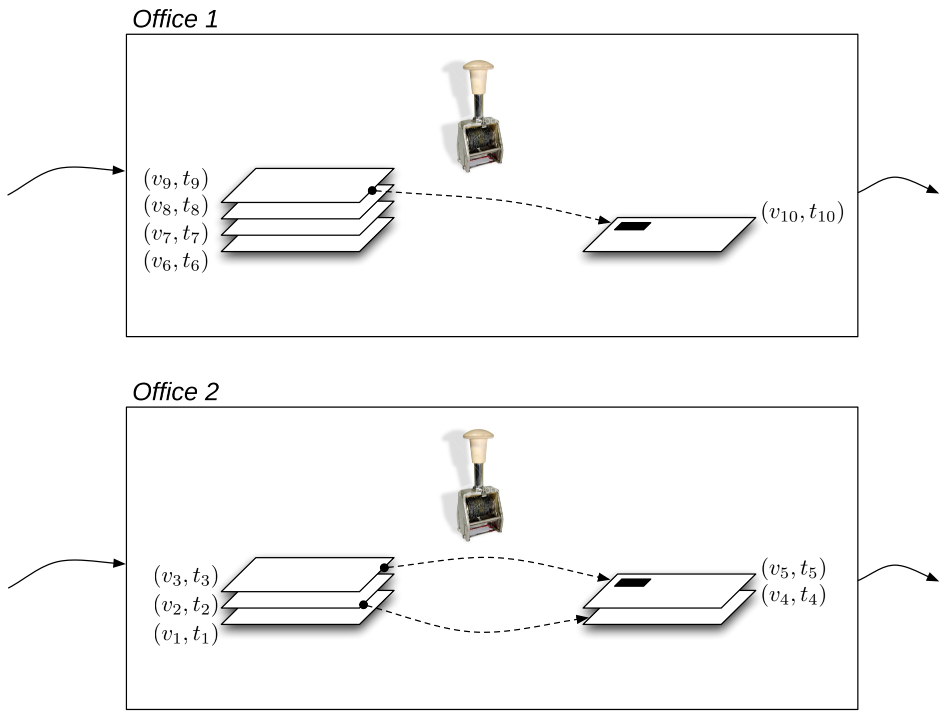


Figure 5.51: Example of two independent offices that receive a stack of mails to stamp in the morning.

established by the two employees that receive the two piles. The two tags are said to be *not comparable*.

This example also shows how the tags can capture the causality among events. Each mail is stamped by the employee that takes a certain amount of time to do her job. Independently from the real time when the mail is placed in the outbox, we can assert that its tag must be greater than the corresponding tag that the same mail has in the inbox pile. Thus we can say that $t_3 \leq t_5$. Moreover, since the mails are handled in order, we can say that $t_2 \leq t_4$.

If we only look at the outboxes of the two offices, it is also true that we cannot compare t_{10} and t_5 because we don't know where the mails where stamped relative to each other.

The set of tags $T = \{t_1, \ldots, t_{10}\}$ of this example is a partially ordered set.

Behaviours, Processes and Composition A component in a system has inputs and outputs that connect to other components. In this section we formalize the notion of components and composition of them. We start with an example and where we introduce informally most of the concepts explained in the rest of this section.

Example 5.7.6 (Adder) The adder in Figure 5.10 is a component of the FIR filter. It has two inputs and one output. Intuitively, it reads one value from each of the inputs, and generate one output that is the sum of the two values from the inputs. Instead of looking at the single execution of this component, we can use the notion of signals to denote the entire behavior of the adder. Given two input sequences :

$$\begin{aligned} in_1 &= \langle 8\ 5\ 9\ 2\ \ldots \rangle \\ in_2 &= \langle 1\ 0\ 4\ 1\ \ldots \rangle \end{aligned}$$

the output sequence is :

$$out = \langle 9\ 5\ 13\ 3\ \ldots \rangle$$

This is one possible combination of input-output sequences that is compliant with the functionality of the adder. The tuple of the three sequences together is called a *behavior*, i.e. a behavior is a tuple of signals. Another valid tuple of sequences for the adder is the following:

$$\begin{aligned} in_1 &= \langle 8\ 1\ 9\ 2\ \ldots \rangle \\ in_2 &= \langle 0\ 1\ 4\ 1\ \ldots \rangle \\ out_3 &= \langle 8\ 2\ 13\ 3\ \ldots \rangle \end{aligned}$$

In the TSM denotational framework a component, like the adder, is called a *process*, i.e. a process is simply defined as set of behaviors.

For a given set of values V and a give set of tags T, let $S = 2^{V \times T}$ denote the set of all signals. A tuple of N signals $(s_1, \ldots, s_n)$ is a element of the set:

$$\underbrace{S \times \ldots \times S}_{N\ times} = S^N$$

Definition 5.7.7 (Process) *A process P is a subset of S^N for some N.*

Remark 5.7.8 (Inputs and Outputs) In the definition of a process, the role of N is pretty clear. A process $P \subseteq S^N$ has N *ports* that are not partitioned in inputs and output ports. Moreover, ports are not labelled by names but they are indexed by their position in the tuple. It is possible to define a partition of the inputs and the outputs. For instance, in the adder example, process *Add* is a subset of S^3. Assume that the first two element of the tuple are inputs and the last one is the output. Then the process can be also seen as a relation among the input and output signals. In fact a relation between two sets A and B is a subset of $A \times B$ and process *Add* is a subset of $S^2 \times S$.

Definition 5.7.9 (Behavior) *A behavior $s \in S$ is said to satisfy a process P if and only if $s \in P$.*

So far, we have defined processes. A system is the result of the composition of may processes. We have not defined composition yet, but the meaning of this operation is pretty intuitive. Composing two processes means that they will have some signals in common. Therefore, the composition of two processes P_1 and P_2 is another process defined by their intersection $P_1 \cap P_2$. To take the intersection, though, the two processes must have the same number of signals (i.e. each process must be a subset of the same set S^N for some N).

Definition 5.7.10 (Composition) *Given a set of processes $\mathbf{P} = \{P_1, ..., P_k\}$ with $P_i \in S^N$, the composition of these processes is a processes $Q \in S^N$ such that*

$$Q = \cap_{P_i \in \mathbf{P}} P_i$$

Example 5.7.11 (Parallel processes) Figure 5.52 shows a process Q that is comprised of two independent processes P_1 and P_2. Notice that process Q has 8 signals $s_1, \ldots, s_8$. Four of them are defined by that behaviours of P_1 and the other four by the behaviours of P_2. Since the two processes are independent, we can define Q as $P_1 \times P_2$. Connecting two signals means imposing their equality. For instance, connecting s_2 and s_5 means imposing that $s_2 = s_5$. A connection, therefore, is simply a process defined by the equality of some of its signals. Process $C_{2,5}$ is the connection of s_2 and s_5, more precisely:

$$C_{2,5} = \{\mathbf{s} = (s_1, \ldots, s_8) \in S^8 | s_2 = s_5\}$$

The intersection $Q \cap C_{2,5}$ is also shows in Figure **??**.

Classification of MoCs Models of computations can be compared using the tagged signal model. Each model distinguishes itself from the others by the way in which events are related to each other. There are two classes of MoCs: *timed* and *untimed*.

A timed model of computation is a model where the set of tags T is totally ordered. It means that given any two members of t and t' of the set T, either $t < t'$ or $t' < t$. Thus, given any two events in a system, it is always possible to order them according to which one happened first. For timed models of computation, a tag is also called a *time stamp.*

Timed models are natural to us because we perceive *time* as being totally ordered. For any two events in time, we are able to order them and say which one happened first. Moreover, we can state the length of time that elapsed between the two events because we have a physical notion of time that we assume to be the same

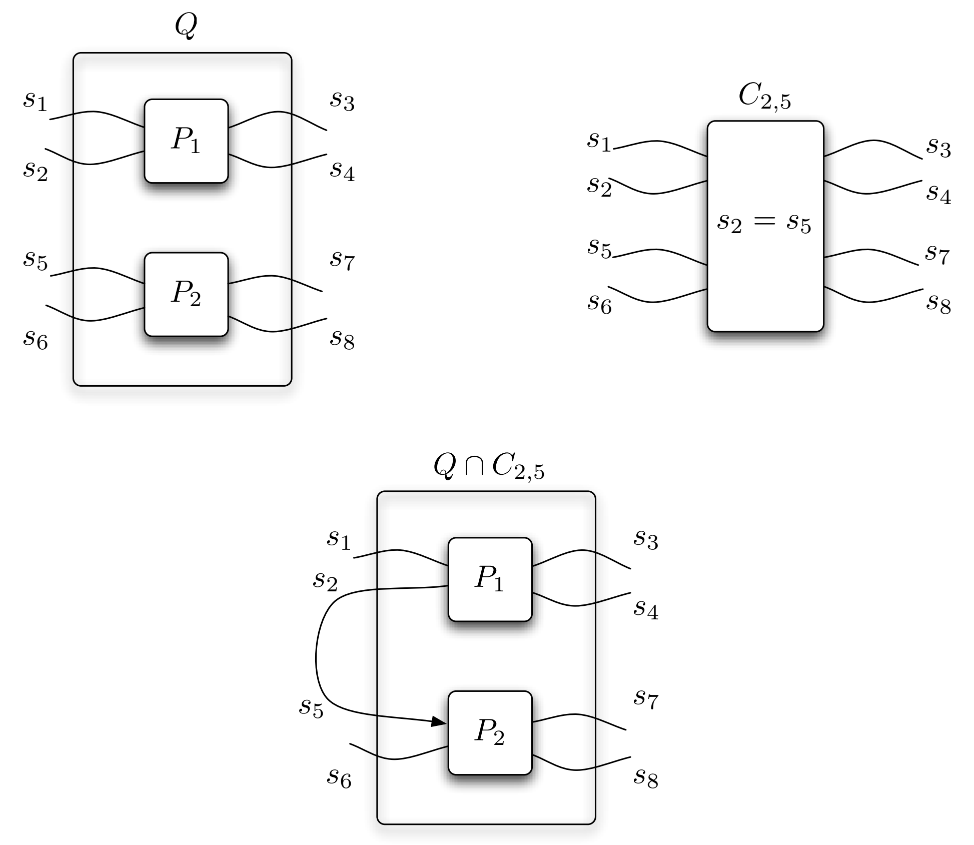


Figure 5.52: A process Q composed of two parallel processes P_1 and P_2, a connection $C_{2,5}$ between two signals s_2 and s_5 and the result of their composition.

everywhere [10]. In the physical world, time is a real number, meaning that $T = \mathbb{R}$. Given two members of T, say t and t', the length of time that elapse between them is $|t - t'|$. In general, when a notion of distance, or better a metric, can be defined on the elements of T, we refer to a tag system as *metric time*.

Remark 5.7.12 (Metric) Given a set T, a metric is a function $d : T \times T \to \mathbb{R}$ such that, for all $t, t', t'' \in T$ the following properties are satsfied:

- $d(t, t') \geq 0$ *(Non-negativity)*
- $d(t, t') = 0 \iff t = t'$ *(Identity of indiscernibles).*
- $d(t, t') = d(t', t)$ *(Symmetry).*
- $d(t, t'') \leq d(t, t') + d(t', t'')$ *(Subadditivity or triangle inequality).*

The *continuous time* model of computation is a metric timed model of computation where the set of tags is the set of real number $\mathbb{R}$ (see Section **??** for a more precise definition). A signal in a continuous time model of computation is a waveform defined by a function of time. For instance, a sinusoid $\sin(2\pi f_o t)$ with frequency f_0 represents a continuous time signal.

Another timed model of computation, that will be described in more details in Section **??**, is the *discrete event*. In the discrete event model of computation, the set of tags T has an interesting structure. It is totally ordered and it is countable meaning that the set of tags can be enumerated by the integer numbers. This model is the one that is adopted in many simulators for logic circuits.

When the set of tags T of a model of computation is only *partially ordered*, the model is said to be *untimed*. Example 5.7.5 introduced already the notion of partially ordered event.

There are a number of reasons to relax the total ordering of the tags. Total order is used to capture dependency and synchronization. In some cases, there are activities that can be executed concurrently without forcing any relationship between the events that belong to the inputs and output of different processes.

Agent Algebras

Formal Methods for Verification

5.7.3 Operational Description

Languages for Embedded and Hybrid Systems

5.7.4 Mapping between Models of Computation

Notions of Equivalence

Concurrency and Coordination

De-synchronization

5.8 Tools for Heterogeneous Specification

5.8.1 Ptolemy II

The Ptolemy II [22, 23, 24] software framework is a modeling, analysis and design environment for heterogeneous and real-time embedded systems. Ptolemy II is

[10]We neglect relativistic effect.

part of the Ptolemy project conducted at the University of California at Berkeley. The predecessors of this framework are Gabriel [18], that focused on SDF models and code generation for digital signal processors, and Ptolemy Classic [26], that presented many extensions including boolean and dynamic data flow, multidimensional dataflow and process networks.

Ptolemy II offers a graphical environment for design capture. The user describes a model using a graphical language that is based on a solid *abstract syntax* which is common to any model independently of its semantics. The abstract syntax defines *entities*, ports, relations and links. Ultimately, entities will contain a thread of execution and will exchange data using links and relations. The communication semantics is determined by the way in which data are exchanged and it is determined by an object called *receiver* that is contained by each input port of an entity. The concurrency model, the order of execution and the state update of the actors are defined by the *director* of a model.

Ptolemy II provides many interesting features for modeling heterogeneous systems, debugging and generating code from a model. Moreover, polymorphism, inheritance and type inference are also provided by the framework.

Abstract Syntax

The abstract syntax of Ptolemy II is based on *clustered graphs*. These clustered graphs are uninterpreted in the sense that the vertexes, the edges and the clusters are not bound to a specific interpretation. For example, they can be used to capture a dataflow model where vertexes are actors and edges are connections between actors; they can also be used to describe a finite state machine where vertexes are states and edges are transitions.

A configuration of a clustered graph is called a *topology* which is a collection of *entities* and *relations*. Figure 5.53 shows a particular topology with three entities and one relation. Entities have ports that are connected to relations by links. A *connection* between two ports involves the two links that connect the ports to a relation. As shown in figure, many ports can be connected to a relation. The meaning of such a connection is defined by the semantics associated with the relation.

To support hierarchy, the abstract syntax defines *component entities* and *composite entities*. A component entity is a leaf of the hierarchy. A composite entity can contain component entities, composite entities and relations, namely another hierarchical topology. Figure 5.54 shows an example of a hierarchical topology. Entity E_5 is composite. It contains two entities, E_1 and E_2, and one relation R_1. A port of a component entity contains a list of links that are connected to the port. A port of composite entity contains two list of links: the links that are contained by the entity and the outside links. A port of compositey entity is used to hide the internal ports.

Example 5.8.1 (Use of the abstract syntax) The flexibility and expressiveness of the abstract syntax allows to capture different concept in the same environment. Figure 5.55 shows a dataflow model and a StateCharts model. In drawing these models we have used the same notation introduced in previous sections of this chapter.

Dataflow actors naturally map to entities. Dataflow actors communicate over point-to-point unidirectional channels that map to connections. Because channels are unidirectional, the explicit representation of the relation can be omitted. However, the rules of the abstract syntax force the use of a relation even in the case of a point-to-point connection. In Section 5.4.4 and 5.4.4, the definition of the dataflow

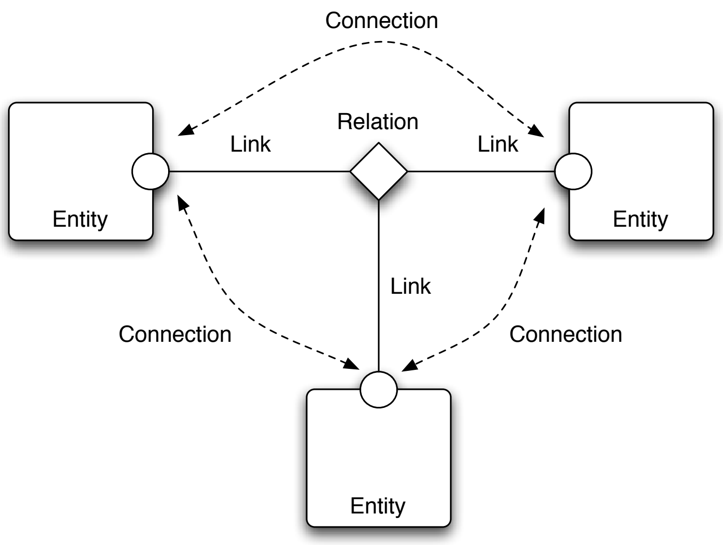


Figure 5.53: The elements of a topology.

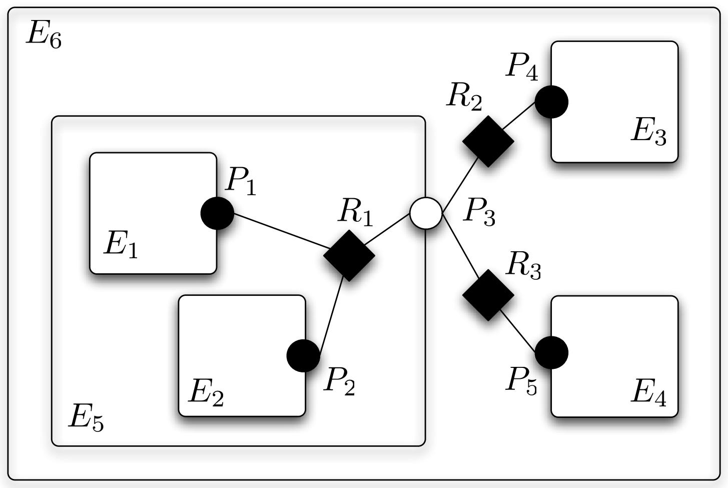


Figure 5.54: Hierarchical topology.

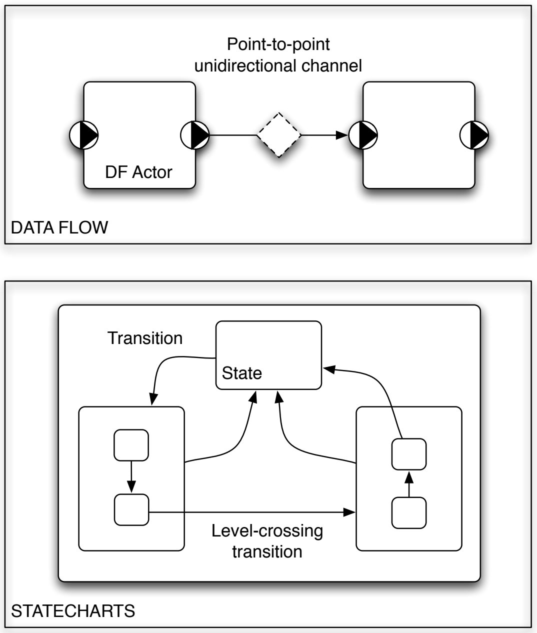


Figure 5.55: Dataflow and Statecharts models can both be captured by the abstract syntax.

model of computation did not include hierarchy. The abstract syntax allows the use of hierarchy in a very intuitive manner.

StateCharts models can also be represented[11]. States map to entities and transitions to connections. The constructs that the abstract syntax provide to describe hierarchical models is fundamental for StateCharts. In this case, connections can span multiple hierarchy levels. These type of connections are called level-crossing connections and are supported by the abstract syntax but not used in Ptolemy II.

Abstract Semantics

Executable entities, meaning entities that encapsulate a thread of control, are called *actors* in Ptolemy II. Actors also provide interfaces to communicate with other actors. There are no restrictions on the order in which the actors are executed and neither on the way in which data are exchanged between actors. The execution order is decided by the *domain* in which the actors are used and the communication semantics is implemented by special objects called receiver that is contained in input ports. The abstract semantics provides a framework for concurrent execution that is neutral about the model of computation.

Actors communicate through message passing. The input ports of an actor contain a special object called receiver that implements an interface for communication. Figure 5.53 shows a connection of three entities through a relation. This means that in input port can receive messages from multiple sources. Thus, a receiver can re-

[11]In the StateCharts diagram of Figure 5.55 we have avoided the explicit representation of relations.

ceive messages from multiple *channels*. The interface that a receiver implements has two methods: a `put` method and a `get` method. The `put` method is called by the originator of a message to deliver the message to the input port of an actor. The `get` method is called by the receiving actor to retrieve the message from the input port. Depending on the implementation of these two methods, the communication semantics changes. The following list of receivers are implemented and ready to use in Ptolemy II:

- *Mailbox*. This is a simple one-place buffer.
- *Asynchronous Message Passing*. This receiver implement a FIFO queue and it is typically used in models of computation like process network.
- *Rendezvous*. In the implementation of this receiver, the `put` and `get` methods are blocking. When a source actor calls the `put` method, the receiver blocks the source's thread of control until the target actor calls the `get` method, and vice versa.
- *Discrete event*. Each event sent to the receiver is stored in a calendar queue and processed by the discrete event simulation engine.

The concurrent execution and scheduling of actors depends on the domain or model of computation underlying a model. The approach followed by Ptolemy II is to provide a set of functions whose implementation and order of invocation defines the semantics of the domain. Each actor implements an interface that comprises eight methods: `preinitilize`, `initialize`, `prefire`, `fire`, `postfire`, `stopFire`, `wrapup` and `terminate`. An *iteration* consists of one invocation of `prefire`, any number of invocations of `fire` and one invocation of `postfire`. An *execution* consists of one invocation of `preinitialize`, followed by one invocation of `initialize`, any number of iterations followed by `wrapup`. These methods are typically called to perform different functions related to an actor. Among them, the following methods are important to the definition of the semantics of a model:

- `preinitialize`. This method is invoked exactly once at the beginning of an execution and before type resolution. It is used to instantiate the receivers on each input port and defines the type of the ports.
- `initialize`. This method is invoked exactly once after type resolution and can be invoked again to restart an execution. It is used to create and initialize internal variables and to generate events before the simulation starts.
- `prefire`. This methods can be called multiple times during an execution but only once for each iteration. It is used to check whether an actor is ready to be fired. If an actor is not ready to fire, then `prefire` is called again after the a change in the system state has been detected by the simulation engine.
- `fire`. This method implements the function performed by the actor. It can be called multiple times during an iteration until some convergence criteria are met. Thus, this method should not change directly the internal state of the actor which should be updated by the `postfire` method.
- `postfire`. This method is called exactly once after an iteration and it is typically used to update the internal state of an actor. This method is also used in the hierarchical composition of heterogeneous models of computation. In these case, each model should have a notion of finite execution and the `postfire` method is used to check whether an actor has ended its "mission".

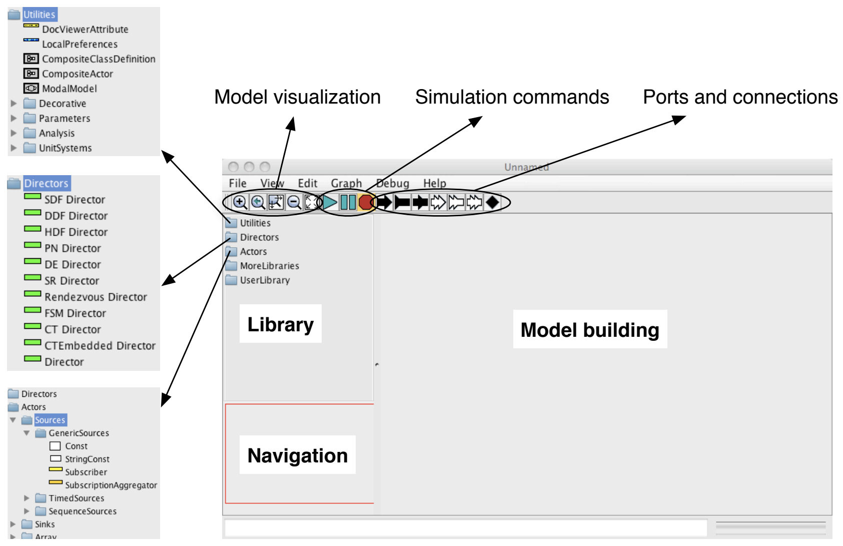


Figure 5.56: The Verginl graphical user interface of Ptolemy II

In a hierarchical model, a manager governs the execution of the top-level model. Each composite entity has its own local director that governs the execution of the entities at that level.

Example 5.8.2 (Static dataflow (SDF) domain) Each dataflow actors in this domain implement a simple `prefire` method that returns a boolean value `false` is there are not enough tokens in the input ports. If the are enough tokens to fire the actor, then `prefire` returns `true`. The receiver used in this domain is a simple FIFO queue which implements a method to check how many tokens are in the FIFO.

The director of the SDF domain uses a scheduler that solves the balance equations (see Section 5.4.4) and computes a schedule for the dataflow network. The computation of the schedule is performed during initialization of the director. The fire method of the director calls the `prefire`, `fire` and `postfire` of each actor according to the schedule.

The Vergil Graphical Inteface

Ptolemy II provides a graphical interface to model heterogeneous systems. The graphical interface is called Vergil and serves not only as a design entry but also as a framework to integrate several tools like for instance code generation.

Figure 5.56 shows the main windows of the Vergil graphical user interface. The main windows has three panes that are the library, navigation and model-building pane. The library pane contains all the components provide by the framework such as the utilities library, the director library and the actor library. The utilities library provides a set of general components such as `CompositeActor` that is used to implement hierarchical entities. The director library provides the local directors that implement the rules governing a model of computation. The actor library provides a variety of different actors such as sources, sinks, mathematical functions and input-output functions (e.g. functions to plot and visualize data). The top toolbar provides command to adjust the model visualization, control simulation and instantiate ports and connections.

Additional Features

Ptolemy II offer a reach set of features to embedded system designers. Modeling is not only simplified by the support of hierarchy but also by a powerful type system that is able to infer the type of the data exchanged by the actors. The user can explicitly specify the type associated with the ports of an actor. Ptolemy can propagate the type information to the rest of the model notifying the user of any inconsistency that may appear during type resolution.

Actors in Ptolemy II are polymorphic. Data exchanged by actors are encapsulated into an abstract data type called *token.* This allows the user to define the operation of an actor depending on the actual token type. Therefore the same actor can be used in different context independently from the data type. For instance, an actor performing the sum of the inputs can be used to sum integer numbers, floating point numbers or string.

Another interesting feature is the possibility of defining and *actor class* that get instantiated in different models. Moreover, inheritance is also supported. These two features are essential to provide an object oriented language. An actor class derived from a base class inherits all the actors contained in the based class.

Finally, the framework provides strong support for debugging. The user can step through the execution of a model and "listen" to the director or to any actor to verify that the behavior of the prefire, fire and post fire methods is the expected one. The plotting capabilities of Ptolemy II, provided by the `PtPlot` package is a distinguishing feature of this environment.

5.8.2 Mathworks

5.8.3 Metropolis

5.8.4 Labview

5.8.5 Modelica

Modelica is an object-oriented language for hierarchical physical modeling [62, 136] targeting efficient simulation. One of its most important features is *non-causal modeling.* In this modeling paradigm, users do not specify the relationship between input and output signals directly (in terms of a function), but rather they define variables and the equations that they must satisfy. Modelica provides a formal type system for this modeling effort. Two commercial modeling and simulation environments for Modelica are currently available: Dymola [51] (Dynamic Modeling Laboratory) marketed by Dynasim AB and MathModelica, a simulation environment integrated into Mathematica and Microsoft Visio, marketed by MathCore Engineering.

Modelica **Syntax**

The syntax of the Modelica language is described in [13]. Readers familiar with object-oriented programming will find some similarities with Java and C++. However, there are also fundamental differences since Modelica is oriented to mathematical programming. This section describes the syntactic statements of the language and gives some intuition on how they can be used in the context of hybrid systems. This, of course, is not a complete reference but only a selection of the basic constructs of the language. A complete reference can be found in [13]. The book by Tiller [136] is an introduction to the language and provides also the necessary background to develop MODELICA models for various physical systems.

MODELICA is a typed language. It provides some primitive types like Integer, String, Boolean and Real. As in C++ and Java, it is possible to build more complicated data types by defining classes. There are many types of classes: records, types, connectors, models, blocks, packages and functions. Classes, as well as models, have fields (variables they act on) and methods. [12] In MODELICA, class methods are represented by equation and algorithms sections. An equation is syntactically defined as <expression = expression> and an equation section may contain a set of equations. The syntax supports the ability to describe a model as a set of equations on variables (non-causal modeling), as opposed to a method of computing output values by operating on input values. In non-causal modeling there is no distinction between input and output variables; instead, variables are involved in equations that must be satisfied. The Algorithm sections are simply sequential blocks of statements and are closer to JAVA or C++ programming from a syntactic and semantic viewpoints. MODELICA also allows the users to specify causal models by defining *functions*. A function is a special class that can have inputs, outputs, and an algorithm section which specifies the model behavior.

Before going into the details of variable declaration, it is important to introduce the notion of *variability* of variables. A variable can be continuous-time, discrete-time, a parameter or a constant depending on the modifier used in its instantiation. The MODELICA variability modifiers are discrete, parameter and constant (if no modifier is specified then the variable is assumed to be continuous). The meaning is self-explanatory; the formal semantics is given in Section 5.8.5.

MODELICA also defines a connect operator that takes two variable references as parameters. Connections are like other equations. In fact, connect statements are translated into particular equations that involve the required variables. Variables must be of the same type (either continuous-time or discrete-time). The connect statement is a convenient shortcut for the users who could write their own set of equations to relate variables that are "connected".

MODELICA is a typed system. Users of the language can extend the predefined type set by defining new, and more complex, types. The MODELICA syntax supports the following classes: [13]

- **record**: it is just an aggregation of types without any method definition. In particular, no equations are allowed in the definition or in any of its components, and they may not be used in connections. A record is a heterogeneous set of typed fields.
- **type**: it may only be an extension to the predefined types, records, or array of type. It is like a typedef in C++.
- **connector**: it is a special type for variables that are involved in a connection equation. Connectors are specifically used to connect models. No equations are allowed in their definition or in any of their components.
- **model**: it describes the behavior of a physical system by means of equations. It may not be used in connections.

[12] C++ or JAVA programmers are used to this terminology, where methods are functions that are part of a class definition.

[13] Some of the constructs mentioned below are explained in Section 5.8.5

- **block**: it describes an input-output relation. It has fixed causality. Each component of an interface must either have causality equal to input or output. It can not be used in connections.
- **package**: it may only contain declarations of classes and constants.
- **function**: it has the same restrictions as for blocks. Additional restrictions are: no equations, at most one algorithm section. Calling a function requires either an algorithm section or an external function interface which is a way of invoking a function described in a different language (for instance C). A function can not contain calls to the MODELICA built-in operators der, initial, terminal, sample, pre, edge, change, reinit, delay, and cardinality whose meaning is explained in Section 5.8.5.

Inheritance is allowed through the keyword extends like in JAVA. A class can extend another class thereby inheriting its parent class fields, equations, and algorithms. A class can be defined as partial, i.e. it cannot be instantiated directly but it has to be extended first. The MODELICA language provides control statements and loops. There are two basic control statements (if and when) and two loop statements (while and for).

```
if expression then
  equation/algorithm
else
  equation/algorithm
end if
```

For instance, an expression can check the values of a continuous variable. Depending on the result of the Boolean expression, a different set of equations is chosen. It is not possible to mix equations and algorithms. If one branch has a model described by equations, so has to have the other branch. Also the number of equations has to match. The syntax of the for statement is as follows:

```
for IDENT in expression loop
  { equation/algorithm; }
end for
```

IDENT is a valid MODELICA identifier. A for loop can be used to generate a vector of equations, for instance. It is not possible to mix equations and algorithms. The while statement syntax is as follows:

```
while expression loop
{ equation/algorithm; }
end while
```

A while loop has the same meaning as in many programming languages. The body of the while statement is active as long as the expression evaluates to true. Finally, the when statement has the form:

```
when expression then
  { equation/algorithm; }
end when
when expression then
  { equation/algorithm; }
else when expression then
  { equation/algorithm; }
end when
```

The body of a when statement is active when the expression changes from false to true. Real variables assigned in a when clause must be discrete time. Also, equations in a when clause must be of the form $v = expression$, where v is a variable. Expressions use relation operators like $\leq, \geq, ==, ...$ on continuous time variables, but can be any other valid expression whose result is a Boolean.

Modelica **Semantics**

The Modelica language distinguishes between discrete-time and continuous-time variables. Continuous-time variables are the only ones that can have a non-zero derivative. Modelica has a predefined operator der(v) that indicates the time derivative of the continuous variable v. When v is a discrete time variable (specified by using the discrete modifier at instantiation time) the derivative operator should not be used even if we can informally say that its derivative is always zero and changes only at *event instants* (see below). Parameter and constant variables remain constant during transient analysis.

The second distinction to point out is between the algorithm and the equation sections. Both are used to describe the behavior of a model. An equation section contains a set of equations that must be satisfied. Equations are all concurrent and the order in which they are written is immaterial. Furthermore, an equation does not distinguish between input and output variables. For instance, an equation could be $i_1(t) + i_2(t) = 0$ which does not specify if i_1 is used to compute i_2 or vice-versa. The value of i_1 and i_2, at a specific time t_0, is set in such a way that all the equations of the model are satisfied. An algorithm section is a block of sequential statements. Here, order matters. In an algorithm section, the user should use the assignment operator := instead of the equality operator =. Only one variable reference can be used as left operand. The value of the variable to the left of the assignment operator is computed using the values of the variables to the right of it.

Causal models in Modelica are described using functions. A function is a particular class that has input and output variables. A function has exactly one algorithm section that specifies the input/output behavior of the function. Non-causal models are described by means of equation sections defined in classes or models. Statements like if then else and for are quite intuitive. In the case of if clauses in equation sections, if the switching condition contains also variables that are not constants or parameters then the else branch cannot be omitted, otherwise the behavior will not be defined when a false expression is evaluated.

The when clause deserves particular attention. When the switching expression (see Section 5.8.5) evaluates to true the body of the when clause is active. The switching expression is considered a discrete-time predicate. If the body of the when clause is not active, all the variables assigned in the body should be held constant to their values at the last event instant. Hence, if the when clause is in an equation section, each equality operator must have only one component instance on the left-hand side (otherwise it is not clear which variable should be held). Such component instance is the one whose value is held while the switching expression evaluates to false. This condition can be checked by a syntax checker.

Finally, a connect statement is an alternative way of expressing certain equations. A connect statement can generate two kinds of equations depending on the nature of the variables that are passed as arguments. In the first case, the variables $v_1, \ldots, v_n$ are declared *flows* at instantiation time (using the flow modifier) and the connection generates the equation $v_1 + \ldots + v_n = 0$. Otherwise, the connection generates the equation $v_1 = ... = v_n$.

Equivalent Mathematical Description of a MODELICA Program. A program written in the MODELICA language can be interpreted by defining a one-to-one mapping between the program and a system of Differential Algebraic Equations (DAE). The first step is to translate a hierarchical MODELICA model into a flat set of MODELICA statements, consisting of the set of equation and algorithm sections of all the used components. The resulting system of equations looks like the following:

$$c := f_c(rel(v)) \tag{5.2}$$

$$m := f_m(v, c) \tag{5.3}$$

$$0 := f_x(v, c) \tag{5.4}$$

where $v := [\dot{x}; x; y; t; m; pre(m); p]$. Here, p is the set of parameters and constant variables, m is the set of discrete event variables, $pre(m)$ is the value of discrete events variables immediately before the current event occurred, x and y are continuous variables, $rel(v)$ is the set of relations on variables in v and c is the set of expressions in if statements (including expressions coming from the conversion of when statements into if). The variables x and y are distinguished because x variables appear differentiated while y variables do not. A DAE solver will iterate in the following way:

- Equation 5.4 is solved by assuming c and m constants, meaning that the system of equations is a continuous system of continuous variables;
- during integration of Equation 5.4, the conditions in Equation 5.2 are monitored. If a condition changes its status, an event is triggered at that specific time and the integration is halted.
- at the event instant, Equation 5.3 is a mixed set of algebraic equations which is solved for the Real, Boolean and Integer unknowns;
- after the event is processed, the integration is restarted with Equation 5.4.

Examples

We first describe the full wave rectifier example, which shows the usefulness of object orientation and non-causal modeling. The variables are currents through and voltages across each component, whose types are defined as follows:

```
type Voltage = Real;
type Current = Real;
```

Each component in a circuit has pins to connect to other components. A pin is characterized by a voltage (with respect to a reference voltage) and an input current. A pin is defined as follows:

```
connector Pin
   Voltage v;
   flow Current i;
end Pin;
```

The connector keyword is used to specify that pins are used in connection statements. The flow keyword is used to declare that the variable i is a flow, i.e. the sum of all *Current* fields of *Pins* in a connection must be equal to zero. A generic two-pin component can be described in the following way [63]:

```
partial class TwoPin
```

```
  Pin p, n;
  Voltage v;
  Current i;
  equation
    v = p.v - n.v;
    0 = p.i + n.i;
    i = p.i;
end TwoPin;
```

This class defines a positive and a negative pin. Kirchoff's equations for voltage and current are declared in the `equation` section. This class is partial and we extend it to specify two pins components like resistors and capacitors. A capacitor for instance can be described as follows:

```
class Capacitor
  extends TwoPin;
  parameter Real C(unit="F") "Capacitance";
  equation
    C * der(v) = i;
end Capacitor;
```

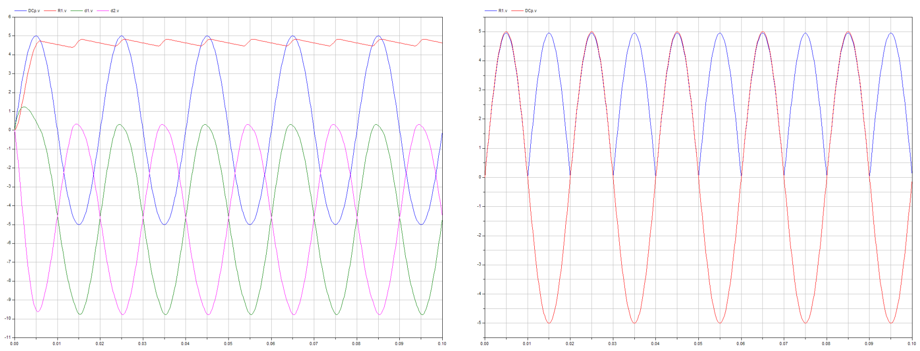
In the `equation` section, we need only declare the component constituent equation since the other equations are inherited from a two-pin component. A parameter is used for the value of capacitance. A diode is modeled as a component with two regions of operation: reverse bias for $v < 0$ and forward bias for $v \geq 0$:

```
class Diode
  extends TwoPin;
  equation
    if v ≥ 0 then i = v / 0.1;
    else i = -1e-15;
    end if;
end Diode;
```

In the forward-bias region, the diode is a resistor with a very small resistance while in reverse bias it is basically an open circuit (only a small reverse current flows through it). Each component can be instantiated and interconnected with others to build a netlist as in the following example:

```
class circuit
  Resistor R1(R = 10); Capacitor C1(C = 0.01);
  Vsin DCp(VA = 5); Vsin DCn(VA = 5);
  Diode d1; Diode d2;
  Ground G;
  equation
    connect( DCp.p, d1.p ); connect( d1.n , R1.p );
    connect( d1.n , C1.p ); connect( DCp.n, G.gpin );
    connect( DCn.p, G.gpin ); connect( DCn.n, d2.p );
    connect( d2.n , R1.p ); connect( C1.n, G.gpin );
    connect( R1.n, G.gpin );
end circuit;
```

where `Vsin` is the sinusoidal voltage source and `Ground` is a component that is used to fix the voltage of a node to $0V$. Figure 5.57 shows the simulation result for the two different types of load. The waveforms were obtained by simulating the MODELICA models with DYMOLA. DYMOLA is able to solve the algebraic loop by performing a symbolic manipulation.



(a) (b)

Figure 5.57: Dymola simulation results of the Modelica rectifier example: (a) for an RC load and (b) for a pure resistive load

5.8.6 Esterel, Signal, Lustre

5.9 Problems

Problem 5.9.1 Define process *SMALL* of the vending machine presented in Section **??**. Notice that a customer is allowed to insert one or two dollars.

Problem 5.9.2