

Chapter 2

Modeling and Specification of SoC Designs

2.1 Introduction

As a system-level specification, SystemC transaction level modeling (TLM) [1] establishes a standard to enable fast simulation and easy model interoperability for hardware/software co-design. It mainly focuses on communication between different functional components of a system and data processing in each component. Although UML is being used as a de facto software modeling tool, UML Profile for SoC [2] has been proposed as an extension of UML 2.X to enable SoC modeling. It can be used to capture the system behavior for both SoC software and hardware components [3–5]. However, both SystemC TLM and UML diagrams are not formal enough for automatic analysis, especially for the validation using model checking techniques [6]. The inherent ambiguity, incompleteness, and contradiction in specifications can lead to different interpretations. Therefore, it is necessary to formalize the semantics of such SoC specifications.

This chapter focuses on the formal modeling of the two widely used SoC specifications. It describes how to automatically extract the formal models from specifications. Such formal models can be used for automated generation of directed tests (see the details in Chap. 3). This chapter is organized as follows. Section 2.2 presents both graph and finite state machine (FSM)-based modeling of systems. Section 2.3 describes the formal modeling of SystemC TLM designs. Section 2.4 describes formal modeling techniques of UML activity diagrams. Finally, Sect. 2.5 summarizes the chapter.

2.2 Modeling of Complex Systems

Modeling plays a central role in design automation of SoC architectures. The formal modeling can not only help designers accurately describe the syntax and semantics of a design, but can also enable the automatic analysis using corresponding tools.

This section presents two widely used formal models: the graph model for structure modeling and the FSM model for behavior modeling. The combination of both models can be used to capture the high-level abstraction of various complex SoC designs.

2.2.1 Graph-Based Modeling

In system level, there is no explicit boundary between software and hardware designs. A system can be considered as an interconnection between different functional components (e.g., tasks, modules, etc). Such structural information can be captured as a *graph model*.

Definition 2.1 For a system-level design, its graph model G is a directed graph denoted by the two-tuple (V, E) , where

- V is a set of nodes indicating different kinds of functional components.
- E is a set of edges indicating the communication channels between the components. ■

Consider $G = (V, E)$ as a graph model of a pipelined processor, which is derived from the architecture description language (ADL) specification [7]. In this graph model, the node notation V denotes two types of components in the processor: *units* (e.g., fetch, decode, etc) and *storages* (e.g. register file, memory, etc.). The edge notation E indicates two types of edges: *pipeline edges* and *data transfer edges*. A pipeline edge transfers an instruction (operation) from a parent unit to a child unit. A data-transfer edge transfers data between units and storages. This graph model is similar to the pipeline-level block diagram available in a typical architecture manual.

For illustration, consider a simplified version of a MIPS processor [8]. Fig. 2.1 shows the graph model of the processor. In this figure, rectangular boxes denote units, dashed rectangles are storages, bold edges are instruction-transfer (pipeline) edges, and dashed edges are data-transfer edges. A path from a root node (e.g., Fetch) to a leaf node (e.g., WriteBack) consisting of units and pipeline edges is called a *pipeline path*. For example, one of the pipeline paths is *Fetch, Decode, IALU, MEM, WriteBack*. A path from a unit to main memory or register file consisting of storages and data-transfer edges is called a *data-transfer path*. For example, *MEM, DataMemory, MainMemory* is a data-transfer path.

2.2.2 FSM-Based Behavior Modeling

Although the graph model can be used to describe the structural information, it is not suitable to capture behavioral details. FSM is widely used for describing the internal behavior of software/hardware components. For example, in hardware design, FSM

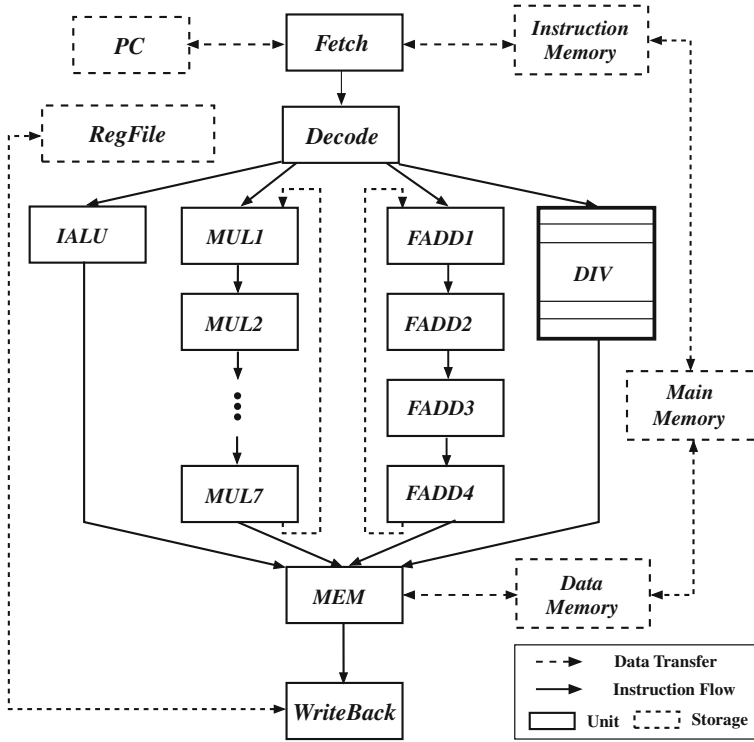


Fig. 2.1 Graph model of the MIPS processor

can be used to describe the state change of registers. In software design, FSM can be used to describe the execution of a piece of sequential code. FSM models can be derived from system-level specifications (e.g. ADL [7], SystemC TLM, UML).

Definition 2.2 A finite state machine M is a seven-tuple $(I, O, S, \delta, \lambda, s_i, s_F)$ where

- I is a finite set of inputs.
- O is a finite set of outputs.
- S is a finite set of states.
- δ is the state transition function $\delta : S \times I \rightarrow S$.
- λ is the output function $\lambda : S \times I \rightarrow O$.
- s_i is an initial state, an element of S .
- s_F is the set of final states, a subset of S .

■

When the model M is in the state s ($s \in S$) and receives an input a ($a \in I$), it moves to the next state specified by $\delta(s, a)$ and produces an output given by $\lambda(s, a)$. For an initial state s_1 , an input sequence $x = a_1, \dots, a_k$ takes the M successively to states $s_{i+1} = \delta(s_i, a_i)$, $i = 1, \dots, k$ with the final state s_{k+1} .

It is important to note that multiple smaller FSMs can be composed to form a larger FSM. In other words, each functional unit can be modeled as an FSM, and the composition of all the functional units is also an FSM [9]. As an example in the composite FSM model of a pipelined processor, a state indicates the value stored in all the registers of the processor. Assuming that each input a_i corresponds to instruction(s) fetched from instruction cache (or memory), the input instruction sequence $x = a_1, \dots, a_k$ can be used as a test program to exercise the path consisting of the states and the state transitions from s_1 to s_{k+1} .

2.3 Specification Using SystemC TLMs

As a framework built on C++, SystemC [10] mimics the hardware description languages such as VHDL and Verilog. With an event-driven simulation kernel, SystemC can be used to simulate the behavior of concurrent processes which can communicate with each other using procedure calls or other mechanisms offered by the SystemC library. Generally, SystemC is often associated with TLM [1, 11], because SystemC TLM provides a wrapper to facilitate the process of communication modeling. Since SystemC TLM provides a rapid prototyping platform for the architecture exploration and hardware/software integration [12], it is widely used to enable early exploration for both hardware and software designs. It can reduce the overall design and validation efforts of complex SoC architectures.

To enable automated analysis, various researchers have tried to extract formal representations from SystemC TLM specifications. Abdi et al. [13] introduced *Model Algebra*, a formalism for representing SoC designs at system level. The work by Kroening et al. [14] formalized the semantics of SystemC by means of labeled Kripke structures. Moy et al. [15] provided a compiler front-end that can extract architecture and synchronization information from SystemC TLM designs using HPIOM. Karlsson et al. [16] translated SystemC models into a Petri-Net based representation PRES+. This model can be used for model checking of properties expressed in a timed temporal logic. Habibi et al. [17] proposed a method that adopts the formal model AsmL. A state machine generated from AsmL can be verified, and then can be translated into both SystemC code and properties for low-level validation. All these modeling techniques focus on the formal modeling of SystemC specifications. This section discusses how to extract the formal models from SystemC TLM specifications to enable automated test generation.

2.3.1 Modeling of SystemC TLM Designs

As a system-level specification, SystemC TLM emphasizes the functionality of the data transfers instead of actual implementation. A SystemC TLM design interconnects a set of processes communicating with each other using transactions data tokens

(i.e., C++ objects). The initial process starts a communication, and the target process passively responds to the communication. Similar to the producer/consumer models, each process does the following tasks: consuming data, processing data, and producing data.

Since SystemC is based on C++, it supports various programming constructs (e.g., template, inheritance, etc.). Although the concept of some TLM components (signals, ports, etc.) is easy, their C++ implementation details are really complex. Therefore, it is difficult to directly translate their behaviors to enable automated validation. In this chapter, abstraction of certain SystemC components is used to hide the implementation details using the predefined SMV constructs. The underlying complex SystemC scheduler aggravates the modeling complexity. For SystemC TLM, to mimic the parallel execution of processes, the SystemC scheduler activates the *ready-to-run* processes in a “non-deterministic” way. Depending on the target model, translation of SystemC scheduler may or may not be required. For example, while translating SystemC TLM specification into SMV model, it is not necessary to model the SystemC scheduler explicitly since SMV inherently supports parallel execution.

For TLM, two most important factors are the *transaction data token* and the *transaction flow*. So the extracted formal model of TLM specifications should reflect both information. The extracted models should not only guide the generation of SMV specification, but should also enable automatic generation of properties and tests. Definition 2.3 provides the formal model of SystemC TLM designs.

Definition 2.3 The **formal model** of a SystemC TLM design is an eight-tuple $(\Sigma, P, T, A, E, M, I, F)$ where

- Σ is a set of transaction data tokens.
- $P = \{p_1, p_2, \dots, p_m\}$ is a set of places.
- $T = \{t_1, t_2, \dots, t_n\}$ is a set of transitions.
- $A \subseteq \{P \times T\} \cup \{T \times P\}$ is a set of arcs between places and transitions.
- $E = \{e_1, e_2, \dots, e_k\}$ is a set of arc expressions. The mapping $Expression(a_i) = e_i$ ($a_i \in A, 1 \leq i \leq k$) provides the enable condition e_i for a_i . A token can pass arc a_i only when e_i is true.
- $M : 2^{P \times \Sigma} \times T \rightarrow 2^{P \times \Sigma}$ is a function that describes the internal operations on input transaction data and output transaction data of a transition.
- $I \in 2^{P \times \Sigma}$ specifies the initial state.
- $F \subseteq 2^{P \times \Sigma}$ specifies the final states. ■

Graph model can be used as an immediate form to capture the execution as well as interconnection of processes.

Figure 2.2a shows an interconnection of six modules. Each arrow indicates a port binding between two modules. Figure 2.2b shows the graph representation of its corresponding formal model. In the graph model, each circle (node) is called a *place* that is used to indicate the input or output buffer of a module. It can temporarily hold the transaction data for later processing. The edges (vertical bars with incoming and outgoing arrow lines) are *transitions*, which are used to indicate modules that contain processes to manipulate input and output transaction data tokens. The places

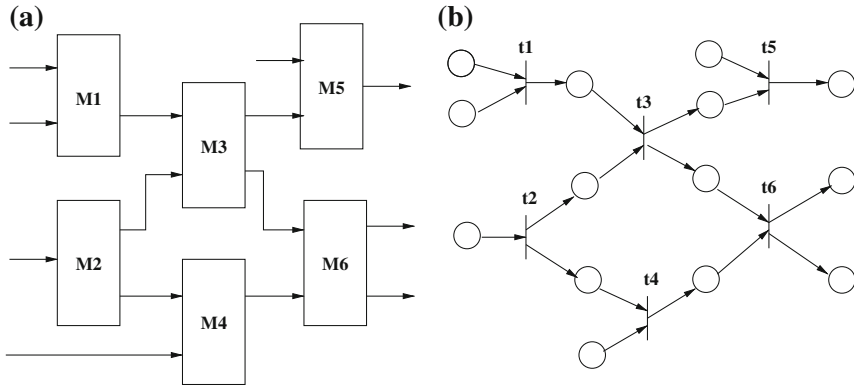


Fig. 2.2 Mapping from a SystemC structure to corresponding graph model. **a** Interconnection of modules. **b** Graph model of the module interconnections

without incoming arcs are *initial places* which start a transition. The places without outgoing arcs are *target places*. A transaction data token flows from the initial places to the target places and token values may change in transitions when necessary. The internal logic of a transition determines the flow of the transaction. It is a piece of code which can be modeled by an FSM model.

2.3.2 Transformation from SystemC TLM to SMV

As a popular model checking specification, SMV [18] is widely adopted to describe both the structure and behavior information of SystemC TLMs. This is because of the following reasons. First, the underlying semantics of SMV is similar to the semantics of SystemC scheduler. So we can mimic most TLM behaviors using SMV without modeling complex scheduler behavior. Second, SMV and TLM have the similar structure hierarchy. Each processing unit encapsulated by a TLM module corresponds to an SMV module. The interconnections (e.g. channels, ports and sockets) between TLM modules can be abstracted by using module parameters in SMV. Third, like SystemC, SMV provides a rich set of programming language constructs such as *if-then-else*, *case-switch* and *for loop* statements. Fourth, SMV main module connects, similar to SystemC, each component of the system. Finally, SMV supports various kinds of data types and data operations. Especially users can define their own data type. All of these SMV features facilitate the translation from TLMs into SMV specification.

As an intermediate form for TLM to SMV translation, the formal model defined in Definition 2.3 provides both structure and behavior information. Such information needs to be collected for translation into a SMV representation. The structure information includes the data-type definition and connectivity between modules. It

<pre> class packet{ public: sc_uint<2> to_chan; sc_uint<6> payload_sz; sc_uint<8> payload[4]; sc_uint<8> parity; }; </pre>	<pre> typedef packet struct{ to_chan : 0..3; payload_sz : 0..63; payload : array 3..0 of 0..255; parity : 0..255; }; </pre>
a) packet in SystemC TLM	b) packet in SMV

Fig. 2.3 An example of data type transformation

corresponds to the description of transaction data token as well as interconnection of transitions and places in the graph model. The behavior information contains token processing and token routing. In the formal model, it represents the internal processing of a transition. This section discusses how to extract both structural and behavioral information (i.e., the graph model and FSM models) and transform it into an SMV specification. We use the example shown in Fig. 2.8 to illustrate how to extract the formal model from a router example.

2.3.2.1 Structure Extraction

In TLM, the content of a transaction data token indicates the transaction flow and the output of each component. Generally, a transaction token consists of several attributes of different types. Because data type determines the size of the specified variable which in turn affects the model checking performance, it is necessary to figure out the data type of a token. Besides all native C++ types, SystemC defines a set of data type classes within the namespace *sc_dt* to represent values with application-specific word lengths applicable to digital hardware. SMV also supports various data types such as array, Boolean, integer, struct, and so on. Such data type definitions facilitate the mapping of data types between SystemC TLM and SMV specification. During the transformation, the word lengths of user-defined type need to be considered. Figure 2.3 shows an example of the router *packet* in the form of SystemC TLM and SMV respectively. For example, *sc_uint* < 2 > has 2 bits and will be transformed into a range 0–3 in SMV.

Derived from the base class *sc_module*, TLM modules are the main processing units for the transaction data. Generally each *sc_module* contains the definitions of processes whose types are *SC_METHOD* or *SC_THREAD*. Modules communicate with each other by sending and receiving transaction data tokens via output and input ports. SystemC provides a communication wrapper for the system components (modules). Various binding mechanisms exist in SystemC (e.g., port to export binding, export to export binding, and port to channel binding) to establish interconnection between modules. Usually each binding corresponds to a channel (e.g., a FIFO channel) to temporarily hold transaction tokens.

```

class router : public sc_module{
public:
    sc_export<tlm_put_if<packet> > packet_in;
    sc_export<tlm_fifo_get_if<packet> > packet_out0;
    sc_export<tlm_fifo_get_if<packet> > packet_out1;
    sc_export<tlm_fifo_get_if<packet> > packet_out2;

    router(sc_module_name module_name);
    void route();
private:
    tlm_fifo<packet> chan0, chan1, chan2, input_;
    packet tmp_packet;
};

```

Fig. 2.4 An example of SystemC TLM module

Figure 2.4 shows the TLM module structure of a router. The class *sc_export* can be used as a port to communicate with other modules. Because the interface type of port *packet_in* is *tlm_put_if<packet>*, it is an input port. In contrast, *packet_out_x* ($x = 0, 1, 2$) have the interface *tlm_fifo_get_if<packet>*, so they are output ports. During the router communication, each connection between a port and an export uses a FIFO channel to temporarily hold a packet.

Structurally similar to SystemC TLMs, SMV specification is also modularized and hierarchically organized. So the extraction of structure information needs to map the TLM constructs in the right place of the SMV specification. Figure 2.5 shows the SMV module skeleton corresponding to example in Fig. 2.4 after the structure extraction. In SMV, a module uses the parameters as the input and output ports to both communicate with other modules and configure the system status defined in the *main* module. In the example of Fig. 2.5, the SMV module has one input port and three output ports. The type of the input and output ports is *packet*. All the declarations of member variables except for the FIFO channels are declared in the SMV specification. Because a FIFO channel together with its port pairs are abstracted as an SMV parameter, it is not necessary to create a variable in SMV explicitly. Based on context during the elaboration, some of the declared variables will be initialized. In SMV specification, each output ports and local variables need to be initialized. For example, *packet_out0* is a parameter which refers to an output port, so it will be initialized with a value “0”. During the translation, it is required that all such module connections should be defined in the module *sc_top*.

2.3.2.2 Behavior Extraction

TLM behavior describes the run-time information of TLM including transaction creation, transaction manipulation, and module communication. Transaction creation initializes a transaction by creating a data token (i.e., a C++ object) with proper


```

module router(packet_in, packet_out0, packet_out1, packet_out2){
    input  packet_in : packet;
    output packet_out0, packet_out1, packet_out2: packet;
    tmp_packet : packet;

    init(packet_out0):=0;
    init(packet_out1):=0;
    init(packet_out2):=0;
    init(tmp_packet):=0;
    .....
}

```

Fig. 2.5 An example of SMV module

```

router::router( sc_module_name mname ): sc_module(mname){
    packet_in(input_);    packet_out0(chan0);
    packet_out1(chan1);   packet_out2(chan2);
    SC_METHOD(route);
    sensitive << input_.ok_to_get();
    dont_initialize();
}

void router::route() {
    input_.nb_get(tmp_packet);
    if(tmp_packet.to_chan == (sc_uint<2>)0)
        chan0.nb_put(tmp_packet);
    else if(tmp_packet.to_chan == (sc_uint<2>)1)
        chan1.nb_put(tmp_packet);
    else chan2.nb_put(tmp_packet);
}

```

Fig. 2.6 An example of TLM process

values. Transaction execution describes the transaction flow among the modules. A module is a container which has a cluster of relevant processes. Such processes will handle the incoming transaction tokens and decide where to send them according to the specified conditions. Thus, a different value of a token will lead to different transaction flows. The following two types of process communication are widely supported in transaction flows: (1) direct procedure call from one process to another process, and (2) channel-based events triggered by the procedure call. For example, in the blocking mode, a process can fetch a transaction data token from the specified input port only when the corresponding channel is not empty. Otherwise, the operation “get” will be blocked until there is an event triggered by the “put” operation by other processes.

Figure 2.6 shows the module process *route* of the router example. The process receives a packet from the driver via channel *input_*, and then it decides where to send data based on the packet header information *to_chan*.

TLM modeling provides some synchronization mechanism for the communications between modules. As shown in Fig. 2.6, the router can fetch the data from the

Fig. 2.7 An example of SMV process

```

module router(packet_in, packet_out0,
              packet_out1, packet_out2){
    .....
    next(tmp_packet) := packet_in;
    if(tmp_packet.to_chan = 0){
        next(packet_out0) := tmp_packet;
        next(packet_out1) := 0;
        next(packet_out2) := 0;
    }else if(tmp_packet.to_chan = 1){
        next(packet_out0) := 0;
        next(packet_out1) := tmp_packet;
        next(packet_out2) := 0;
    }else{
        next(packet_out0) := 0;
        next(packet_out1) := 0;
        next(packet_out2) := tmp_packet;
    }
}

```

FIFO queue *input_* only when the driver puts a package and the FIFO channel event *ok_to_get* is triggered. Thus the synchronization between two modules is implicitly achieved.

SMV supports many constructs similar to the common programming language such as *if-then-else*, *switch-case* and *for loop*. So these constructs facilitate the behavior modeling of processes from TLMs to SMV specifications. Figure 2.7 shows the translated SMV specification of the TLM example presented in Fig. 2.6. During the translation from TLM into SMV, a channel is abstracted as an implicit buffer between two ports. So an SMV module will get the input data from its input ports. There is no mapping of the channel in the transformed SMV specification. For example, the *tmp_packet* is assigned the value of the *packet_in* instead of the value of *input_* shown in the TLM example in Fig. 2.6.

2.3.3 Case Study: A Router Example

A prototype tool called *TLM2SMV* was developed to enable the transformation from SystemC TLM specifications to corresponding SMV models for automated directed test generation. The details of the implementation are described in Sect. 12.3.3.

Figure 2.8 shows the TLM structure of a router design. The router consists of five modules: one master, one router, and three slaves. The SystemC program consists of four classes (one class for packet definition, one class for the driver, one class for the router, and one class for the slave), eight functions, and 143 lines of code. The main function of the router is to analyze and distribute the packets received from the master to target slaves.

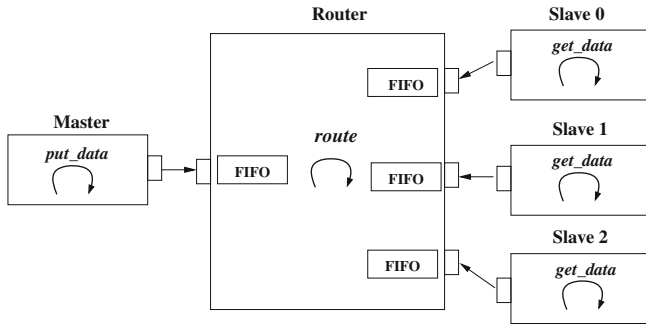


Fig. 2.8 The TLM structure of the router

At the beginning of a transaction, the master module creates a packet. Then, the driver sends the packet to the router for package distribution. The router has one input port and three output ports. Each port is connected to a FIFO buffer (channel) which temporarily stores packets. The router has one process *route* which is implemented as an *SC_METHOD*. The *route* first collects a packet from the channel connected to the driver, decodes the header of the packet to get the target address of a slave, and then sends the packet to the channel connected to the target slave. Finally, the slave modules read the packets when data are available in the respective FIFOs. The transaction data (i.e. packet) flows from the master to its target slave via the router. The flow is controlled by the address *to_chan* in the packet header. By using the proposed approach in Sect. 2.3.2, the automatically generated SMV model contains four modules and 145 lines of code.

2.4 Specification Using UML Activity Diagrams

Formal verification can be used to verify the correctness of specifications, so it can be used to guarantee the quality of UML models [19]. UML activity diagram adopts Petri-net semantics which is promising to describe the concurrent behavior [20, 21]. There are several approaches that use model checking techniques to verify UML activity diagrams. Eshuis [22] presented a translation procedure from UML activity diagrams to the input language of NuSMV [23]. However, the translation is used to verify the consistency between UML activity diagrams and class diagrams. It focuses on checking the consistency between two different models. Guelfi and Mammar [24] provided a formal definition for timed activity diagrams. They outlined the translation from the semantic specifications into PROMELA—an input language of the SPIN model checker. Das et al. [25] proposed a method to deal with timing verification of UML activity diagrams. All these verification work primarily focus on checking the consistency or correctness of the model itself instead of generating directed test cases.

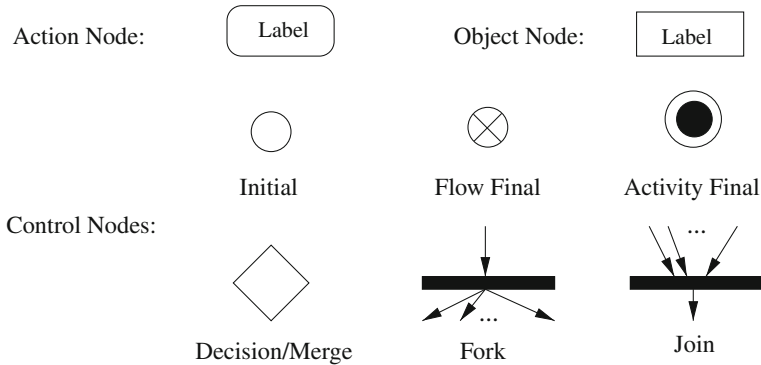


Fig. 2.9 UML activity nodes

In this section, UML 2.1.2 [26] is used as the SoC specification. To reduce the complexity of the testing work, we restrict the testing target and investigate a subset of activity diagrams. The subset mainly contains action nodes, control nodes, object nodes, and control and data flow. Especially for the object node, we assume that it can hold at most one object at a time and it does not support *competition* and *data store*.

2.4.1 Graphic Notations

UML activity diagram is used to coordinate the execution of actions. An action takes a set of inputs and converts them into corresponding outputs. An activity (behavior) consists of a set of actions and flow edges. The actions are connected by object flow edges to show how object tokens flow through and connected by control flow edges to indicate the execution order.

UML activity diagrams adopt the semantics like Petri-net [27]. It is a type of directed graphical representation. Tokens which indicate control or data values flow along the edges from the source node to the sink nodes driven by the actions and conditions. An activity diagram has two kinds of modeling elements: activity nodes and activity edges. More specially, there are three kinds of nodes in activity diagrams:

- **Action Node.** Action nodes consume all input data/control tokens when they are ready; generate new tokens; and send them to output activity edges.
- **Object Node.** Object nodes provide and accept data tokens, and may act as buffers, collecting data tokens as they wait to move downstream.
- **Control Node.** Control nodes route tokens through the graph. The control nodes include constructs to choose between alternative flows (decision/ merge), to split or merge the flow for concurrent processing (fork/ join).

Figure 2.9 shows the basic constructs of activity nodes. An action node is denoted by a round cornered box. It represents an execution of operations on input tokens, and the generated new tokens will be delivered to outgoing edges. An object node denoted by a rectangle box is used to temporarily hold the data tokens waiting to be processed or delivered. For simplicity, it is assumed that object nodes do not support *competition* and *data store* for test case generation. A flow in an activity starts from the initial node. When a token arrives at a flow final node, it will be destroyed. The flow final node has no outgoing edges, so there is no downstream effect. When no tokens exist in an activity diagram, the activity will be terminated. The activity final nodes are similar to flow final nodes, except that when a token reaches one activity final node, the entire flow will be terminated. Decision nodes and merge nodes use the same shape of diamond. Decision nodes choose one of the outgoing flows according to the value of Boolean expressions labeled on the outgoing edge. Merge nodes select only one of the incoming flows to deliver to the next activity node. Forks or joins are shown by multiple arrows leaving or entering the synchronization bar, respectively, to describe the concurrent behavior of a system. When a token arrives at a fork node, it will be duplicated across the outgoing edges. Join nodes synchronize multiple flows. The tokens must be available on every incoming edge in order to be passed to outgoing edges.

Activity nodes are connected by activity edges along which tokens may flow under some condition. Activity edges include control and data flow edges as follows:

- *Control Flow Edge*. Control flow edges indicate the execution sequence of actions.
- *Object Flow Edge*. Object flow edges indicate the relation of data token transmissions. It provides the inputs to actions.

To simplify the discussion, we combine the control and data token together as a new kind of token which contains both control and data information. Such token can flow through activity edges. In other words, we do not distinguish control flow edges and object flow edges.

Figure 2.10 shows an example which uses most of the elements shown in Fig. 2.9. It describes the functionality of withdrawing money from an automated teller machine (ATM) [28]. A user needs to enter the access code first. In case of failure, the user can input the access code again. The operation will abort if access code is wrong in both cases. If the input access code is right, the user can enter the amount of money he wants to withdraw. At the same time, the printer will be ready to print a receipt. Once the ATM decides whether there is enough money the user can withdraw, it provides the cash and generates the information for this transaction. Finally, the printer prints the receipt and the transaction is complete.

The token for this example contains the ATM transaction information such as the input access code and input cash amount, the context information such as the available cash amount and correct access code. In general, a token reflects all the data information required for this activity. Table 2.1 shows the composition of a token of the ATM activity diagram. It consists of five variables which will be used to make the decisions illustrated in Table 2.2.

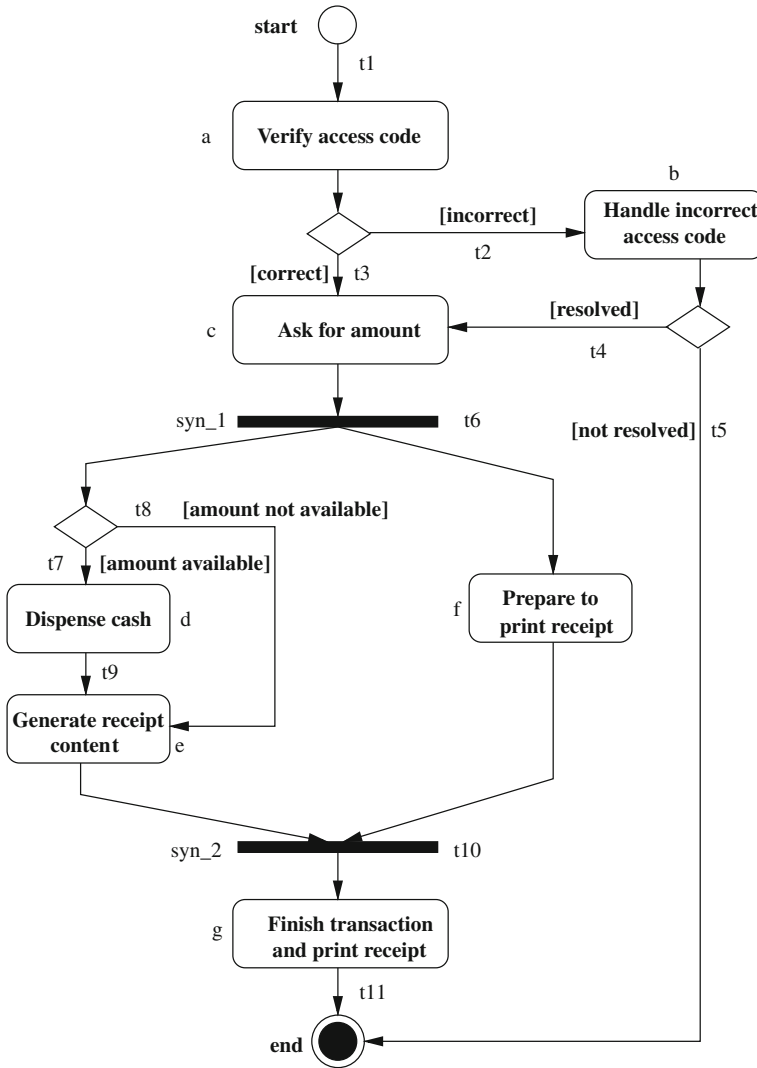


Fig. 2.10 The UML activity diagram of an ATM

2.4.2 Formal Modeling of UML Activity Diagrams

Without formalism, it is hard to describe and model the activity diagrams accurately. UML activity diagram itself is a semi-formal specification that cannot be directly mapped to a model checker input (e.g., SMV models). In practice, Petri-net is used as an intermediate formal model between activity diagrams and SMV model, because

Table 2.1 Breakdown of a token in Fig. 2.10

Variable	Type	Description
<i>access_code</i>	String	User's access code
<i>access_code_input</i>	String	User access code input
<i>access_code_resolve</i>	String	User access code input correction
<i>amount_input</i>	Integer	User cash amount input
<i>amount_available</i>	Integer	Cash amount available

Table 2.2 Condition on the flow edges in Fig. 2.10

Flow edge	Condition	Description
t2	Incorrect	$access_code \neq access_code_input$
t3	Correct	$access_code = access_code_input$
t4	Resolved	$access_code = access_code_resolve$
t5	Not resolved	$access_code \neq access_code_resolve$
t7	Amount available	$amount_input \leq amount_available$
t8	Amount not available	$amount_input > amount_available$

the Petri-net formalism can capture the major functional scenarios as well as guide the translation.

Definition 2.4 describes the relation between the activity nodes and flow edges with a Petri-net semantics. Although it does not model the full features of activity diagrams, it formally depicts the static abstracted structure of activity diagrams which can be used to describe the scenarios that need to be tested.

Definition 2.4 An activity diagram is a directed graph described using an eight-tuple $(A, T, F, C, V, A, a_I, a_F)$ where

- $A = \{a_1, a_2, \dots, a_m\}$ is a set of action nodes.
- $T = \{t_1, t_2, \dots, t_n\}$ is a set of completion transitions.
- $F \subseteq \{A \times T\} \cup \{T \times A\}$ is a set of flow edges between activity nodes and completion transitions.
- $C = \{c_1, c_2, \dots, c_n\}$ is a finite set of guard conditions. Here, c_i ($1 \leq i \leq n$) is a predicate (expression) based on the input variables. There is a mapping from $f_i \in F$ to c_i , referred as $Cond(f_i) = c_i$.
- Let V be the set of all possible assignments for input variables V_1, V_2, \dots, V_k where k is a positive integer.
- $M : A \times V \rightarrow V$ is a mapping that describes the value change of the input variables inside an activity node.
- $a_I \in A$ is the *initial node*, and $a_F \in A$ is the *final node*. There is only one completion transition $t \in T$ and $c \in C$ such that $(a_I, t) \in F$, and for any $t' \in T$, $(t', a_I) \notin F$ and $(a_F, t') \notin F$. ■

A node can be an action node, an initial node or a final node. The *completion transition* and *flow edge* are used to model the behavior of the control nodes. In the graph, the nodes are connected by flow edges associated with a completion transition. Because activity diagrams allow tokens to exist in the flows concurrently, the completion transition can be used to synchronize the token flows. If a completion transition has multiple incoming flow edges, it will do the join operation. If there are multiple outgoing flow edges, then it will do the fork operation. For each flow, e.g., there may be a condition which can guide the token traverse. The graph has one initial node that indicates the start of control and data flows. Activity diagrams have two kinds of final nodes: flow final nodes and activity final nodes. We can combine them together and use a join operation to get a new activity final node. So in the definition there is only one final node.

When analyzing dynamic behaviors of an activity diagram, we need to use the *states* (a set of actions executing concurrently) to model the status of a system. Current state (denoted by CS) of an activity diagram indicates the actions which are being activated.

Definition 2.5 Let D be an activity diagram. The *current state* CS of D is a subset of A . For any transition $t \in T$,

- $\bullet t$ denotes the preset of t , then $\bullet t = \{a \mid (a, t) \in F\}$.
- t^\bullet denotes the postset of t , then $t^\bullet = \{a \mid (t, a) \in F\}$.
- $enabled(CS)$ denotes the set of completion transitions that are associated with the outgoing flow edges of CS , then $enabled(CS) = \{t \mid \bullet t \subseteq CS\}$.
- $firable(CS)$ denotes the set of transitions that can be fired from CS , then $firable(CS) = \{t \mid t \in enabled(CS) \wedge \bullet t \text{ are all completed} \wedge \exists n \in A. Cond((t, n)) \text{ is satisfied} \wedge (CS - \bullet t) \cap t^\bullet = \emptyset\}$. After some t is fired, the new current state $CS' = fire(CS, t) = (CS - \bullet t) \cup t^\bullet$. ■

The current state of an activity diagram indicates which activity nodes are holding the tokens. For example, when $\{d, f\}$ is the current state of the activity diagram in Fig. 2.10, two tokens are in the activity nodes d and f individually. At this time, only the transition associated with t_9 is firable. If it is fired, then the next state is $\{e, f\}$.

Because of the inherent concurrency, several transitions can be fired at the same time. For an activity diagram, all the firable transitions in a state form a *concurrent transition*.

Definition 2.6 Let D be an activity diagram. For a state CS of D , a concurrent transition τ is a set of completion transitions $t_1, t_2, \dots, t_n \in firable(CS)$ where

1. $\forall i, j (1 \leq i < j \leq n), \bullet t_i \cap \bullet t_j = \emptyset$;
2. $\forall t \in (enabled(CS) - \{t_1, t_2, \dots, t_n\})$, there exists an $i (1 \leq i \leq n)$ such that $\bullet t \cap \bullet t_i \neq \emptyset$.

After firing τ from state CS , the current state $CS' = fire(CS, \tau) = \bigcup_{i=1}^n (fire(CS, t_i)) = \bigcup_{i=1}^n ((CS - \bullet t_i) \cup t_i^\bullet)$. ■

An instance of dynamic behavior of an activity diagram can be represented by a sequence of states and concurrent transitions. We call it a *path* of the activity diagram. Because a path may have cycles, during the model checking, it is hard to determine the cycle numbers, so we neglect the cycles on a path. We call such a path as *key path*.

Definition 2.7 A path ρ of the activity diagram D is a sequence of states and concurrent transitions, let

$$\rho = s_0 \xrightarrow{\tau_0} s_1 \xrightarrow{\tau_1} \dots \xrightarrow{\tau_{n-1}} s_n$$

where $s_0 = \{a_I\}$, $s_n = \{a_F\}$, and $s_{i+1} = \text{fire}(s_i, \tau_i)$ for any i ($0 \leq i < n$). ρ is a *key path* if there is no state repetition in ρ , i.e. $\forall i, j$ ($0 < i < j \leq n$), $s_i \cap s_j = \emptyset$.

■

There are five key paths in Fig. 2.10:

- $\rho_1 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t2\}} \{b\} \xrightarrow{\{t5\}} \{end\}$
- $\rho_2 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t3\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t7\}} \{d, f\} \xrightarrow{\{t9\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\},$
- $\rho_3 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t3\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t8\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\},$
- $\rho_4 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t2\}} \{b\} \xrightarrow{\{t4\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t7\}} \{d, f\} \xrightarrow{\{t9\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\},$
- $\rho_5 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t2\}} \{b\} \xrightarrow{\{t4\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t8\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}.$

The *dummy* node is inserted here because it assumes that outgoing edges of the fork node must connect to an activity rather than a selection node. For a key path, when firing transitions, we need to consider guard conditions. For clarity, in Fig. 2.10, we did not label the condition guards for each transition.

Definition 2.8 Let D be an activity diagram. An interaction of the activity diagram is a set of activity nodes (actions) that can be activated simultaneously. A “k-interaction” is a set that contains k activity nodes. ■

In order to detect whether a concurrent state of an activity diagram is reachable or can be activated, we use the term *interaction*¹ to describe the scenario that a set of actions can be activated simultaneously. For example, in Fig. 2.10, $\{d, f\}$ is an example of “2-interaction” in the ATM.

¹ Unlike the interaction in UML Interaction overview diagram, the interaction here means that several actions are activated at the same time.

2.4.3 Transformation from UML Activity Diagrams to SMV

By parsing a UML activity diagram, both the control and data flows can be extracted. The translation consists of two parts: static information extraction and dynamic information extraction. Static information extraction analyzes the structure of an activity diagram and then generates a skeleton of the SMV input. The dynamic information extraction analyzes the dynamic behavior of the system by focusing on control and data flow analysis (i.e., the state change of activities, data manipulation in activities and the condition of the transitions).

2.4.3.1 Static Information Extraction

This step collects both the input data manipulated by the activities and the predicates used as guard conditions of the transitions. For example in Fig. 2.10, there are five input data variables that determine the data and control flows: *access_code*, *access_code_input*, *access_code_resolve*, *amount_input*, and *amount_available*. Since there may be a large number of possible values for a variable, during model checking it will cause the state space explosion. SMV does not support complex data types (e.g., float, double, etc.). For each variable, it is required that the value range should be specified explicitly. To avoid state space explosion, the following methods can be used to reduce the complexity of data types:

- *Scaling*. Scaling is to proportionally reduce the value range of a variable.
- *Reduction*. Reduction is to reduce the cardinality of possible values for a variable.

Since it is hard to implement the above techniques automatically, before the SMV translation, the variable type information is tuned manually for activity diagrams.

During translation, each activity is assigned with a *state variable* which has three possible state values: *unvisited* (0), *unvisited* (1) and *visited* (2). *visited* indicates that no token has passed through this activity node. *Visiting* indicates currently the activity is holding one or more tokens. *visited* indicates that some token has passed through this activity node and currently there is no token in this activity node. The extraction procedure instantiates the activity state variables and assigns suitable values to them. During initialization, the initial activity node is assigned *unvisited* that means there is a token ready at the initial state. Other nodes are initialized to *unvisited*. Also, each flow edge is assigned with a state variable which has two possible values: *fired* (1) and *unfired* (0). *Fired* means some tokens have flowed from the incoming activity nodes to its outgoing activity nodes. *Unfired* means no token has passed through this activity edge. Initially, they are set with value 0.

Figure 2.11 shows the generated skeleton of Fig. 2.10 in SMV format [18, 23]. There are three modules in this skeleton. The module *state* defines the token information (described in Table 2.1) as well as the state variable for activity nodes and flow edges. For example, *verify_access_code* is a state variable for an action with three states. Initially it is assigned the state *unvisited* (0). Module *ATM* shows a

Fig. 2.11 The generated skeleton after structure extraction

```

MODULE state
VAR
  access_code: { A1, B1, C1 };
  access_code_input: { A1, B1, C1, D1 };
  start: 0..2;
  syn_1: 0..2;
  verify_access_code: 0..2;
  t2_cond: 0..1;
  t3_cond: 0..1;
  .....
ASSIGN
  init(start):=1;
  init(syn_1):=0;
  init(verify_access_code):=0;
  .....
MODULE ATM(st)
ASSIGN
  next(st.start):=
  next(st.t2_cond):=
  .....
  next(st.prepare_print_receipt):=
  .....
  next(st.dispense_cash):=
  next(st.t7_cond):=
  .....
MODULE main() {
  st: state; atm: ATM(st);
  p_print: prepare_print(st);
  check: check_amount(st);
}

```

static skeleton without dynamic behavior information. In this phase, variables are collected without any processing. The missing state transition details will be described in Sect. 2.4.3.2. The module *main* creates the module instances and elaborates them together. For example, *st* is an instance of state module and *atm* is an instance of ATM module. Both the *st* and *atm* are bound together, because *atm* will handle the state changes of variables in *st*.

2.4.3.2 Dynamic Information Extraction

After static information extraction, it needs to extract both data manipulations and transitions of state variables, because they will determine the data and control flows. Figure 2.12 defines a set of rules that specify the state transition for each activity node and the value changes of each data. In these rules, the preset and postset notations are used. In these rules, the assignment and constraint to a set means the assignment and constraint to each element in the set. For example, if $\bullet t = \{a_1, a_2, \dots, a_k\}$,

<p>Rule 1: If n is an initial node</p> <pre> init(n) := 1; next(n) := 2; </pre>
<p>Rule 2: If n is a final node, and there are k incoming transitions $t_1, t_2 \dots t_k$.</p> <pre> init(n) := 0; next(n) := case (($\bullet t_1 = 1 \ \& \ cond(t_1)$) ($\bullet t_2 = 1 \ \& \ cond(t_2)$) ... ($\bullet t_k = 1 \ \& \ cond(t_k)$)) : 2; 1 : n; esac; </pre>
<p>Rule 3: If n is an activity node (not join or fork), and there are k incoming transitions $t_1, t_2 \dots t_k$.</p> <pre> init(n) := 0; next(n) := case $n = 1$: 2; ($\bullet t_1 = 1 \ \& \ cond(t_1)$) ($\bullet t_2 = 1 \ \& \ cond(t_2)$) ... ($\bullet t_k = 1 \ \& \ cond(t_k)$)) : 1; 1 : n; esac; </pre>
<p>Rule 4: If n is a fork node, and the corresponding transition is t.</p> <pre> init(n) := 0; next(n) := case $n = 1 \ \& \ \bullet t > 0$: 2; $\bullet t = 1$: 1; 1 : n; esac; </pre>
<p>Rule 5: If n is a join node of transition t, and $a_1, a_2 \dots a_k$ are k elements of $\bullet t$.</p> <pre> init(n) := 0; next(n) := case $n = 1$: 2; $n = 0 \ \& \ (a_1 + a_2 + \dots + a_k = 2 * k)$: 1; $n = 2 \ \& \ (a_1 + a_2 + \dots + a_k < 2 * k)$: 0; 1 : n; esac; </pre>
<p>Rule 6: If t is a transition which corresponds to the flow edges.</p> <pre> init(t) := 0; next(t) := case ! $cond(t) \ \& \ \bullet t = 1$: 0; $cond(t) \ \& \ \bullet t = 1$: 1; 1 : t; esac; </pre>
<p>Rule 7: If v is a variable whose new value is changed by exp_i in activity act_i ($1 \leq i \leq n$).</p> <pre> next(v) := case $act_1 = 1$: exp_1; $act_2 = 1$: exp_2; $act_n = 1$: exp_n; 1 : v; esac; </pre>

Fig. 2.12 Translation rules for state and data transitions

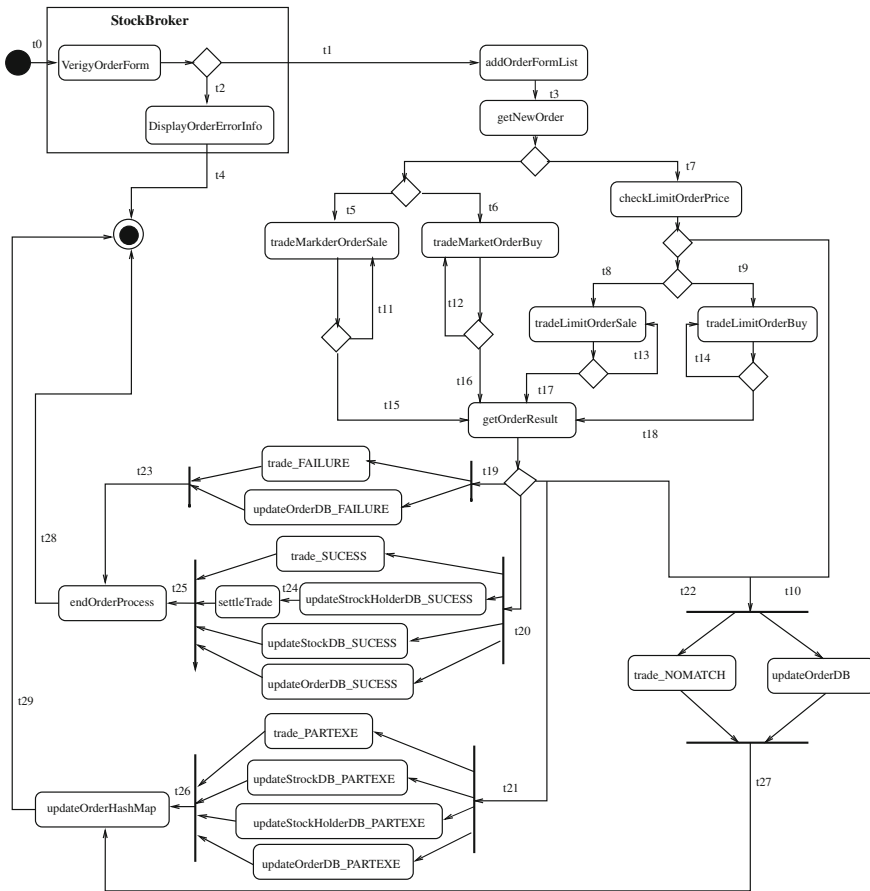


Fig. 2.13 The activity diagram for a stock exchange system

then $\bullet t = 1$ means $a_1 = 1$ & $a_2 = 1$ & ... & $a_k = 1$ and $cond(t)$ means $cond((a_1, t)) \& cond((a_2, t)) \& \dots \& cond((a_k, t))$.

Rule 1 specifies the translation rule for the initial node. The token will be first put at the initial state and the node is marked as *visited* in the next step. Rule 2 specifies the translation rule for the final node. At first, the state is *unvisited*, when one of the incoming edges is activated, its state will become *visited*. Rule 3 defines the state changes of an activity. Initially, the state of an activity is *unvisited*. If the incoming edge is activated, the state will become *unvisited* in the next step. If the current state is *unvisited*, the state will change to be *visited* in the next step. Rule 4 presents the state transition of the fork nodes. When the incoming edge is activated, the fork node will maintain the *unvisited* status until all the outgoing edges are visiting or visited. Rule 5 provides the state transition of join nodes. The join node is used to synchronize the token flows. When all the incoming flows are ready, the transition corresponding

to the join node can be fired. In this rule, if we want to fire the transition, it needs to wait until all the activity nodes in the preset of the transition are visited. Rule 6 shows how to manipulate the state change of the transition when it is fired. Rule 7 presents the translation for value change of the variables. If an activity performs some operation on the variable, the value of the variable can be modified only when the activity state is *unvisited*.

2.4.4 Case Study: A Stock Exchange System

Based on the framework proposed in Sect. 2.4.3, a prototype tool is developed to automate the process of test generation. The tool takes three inputs: type definition of the data which is used in the activity diagram, the context information which sets the parameters for the execution of an activity diagram (e.g. when to trigger the initial node and so on), and UML activity diagrams. The UML activity diagrams are stored in the format of XML Metadata Interchange (XMI) files. The tool can parse the XMI files to get the static and dynamic information for formal model translation. Combined with the context information and data type information, a formal model can be generated using the proposed mapping rules.

The purpose of the Online Stock Exchange System (OSSES) is to process three scenarios: accept, check, and execute the customers' orders (market orders and limit orders). The system uses the UML activity diagram as its behavior specification. Fig. 2.13 shows the specification of the stock system. It has 27 activities, 29 transitions, and 18 key paths. The generated SMV model has 756 lines of code.

2.5 Chapter Summary

This chapter presented the basic concepts of graph/FSM-based modeling of systems. It also introduced two system-level design specifications for SoC designs: SystemC TLM and UML activity diagrams. Based on the structural and behavior models extracted from these two specifications, this chapter presented mechanisms to convert them into formal SMV models to enable automatic analysis and directed test generation, which will be discussed in the following chapters.

References

1. Rose A, Swan S, Pierce J, Fernandez J (2005) Transaction level modeling in SystemC. Open SystemC Initiative. <http://www.systemc.org>
2. Object Management Group (2006) UML profile for system on a chip (SoC), v 1.0.1. http://www.omg.org/technology/documents/formal/profile_soc.htm

3. Riccobene E, Scandurra P, Rosti A, Bocchio S (2005) A UML 2.0 profile for systemc: toward high-level soc design. In: Proceedings of ACM international conference on embedded software, pp 38–141
4. Chureau A, Savaria Y, Aboulhamid EM (2005) The role of model-level transactors and UML in functional prototyping of systems-on-chip: a software-radio application. In: Proceedings of design automation and test in Europe (DATE), pp 698–703
5. Mueller W, Rosti A, Bocchio S, Riccobene E, Scandurra P, Dehaene W, Vanderperren Y (2006) UML for ESL design: basic principles, tools, and applications. In: Proceedings of international conference on computer-aided design (ICCAD), pp 73–80
6. Ammann P, Black P, Majurski W (1998) Using model checking to generate tests from specifications. In: Proceedings of international conference on formal engineering methods (ICFEM), pp 46–54
7. Mishra P, Dutt N (2008) Processor description languages. Morgan Kaufmann Publishers, San Francisco
8. Hennessy J, Patterson D (2003) Computer architecture: a quantitative approach. Morgan Kaufmann Publishers, San Francisco
9. Hopcroft JE, Motwani R, Ullman FD (2006) Introduction to automata theory, language, and computation 3rd edn. Addison-Wesley
10. Open SystemC Initiative (OSCI) (2006) Systemc. <http://www.systemc.org>
11. Cai L, Gajski D (2003) Transaction level modeling: an overview. In: Proceedings of international conference on hardware/software codesign and system, synthesis (CODES+ISSS), pp 19–24
12. Ghenassia F (2005) Transaction level modeling with systemC. Springer, Dordrecht
13. Abdi S, Gajski D (2005) A formalism for functionality preserving system level transformations. In: Proceedings of Asia and South Pacific design automation conference (ASPDAC), pp 139–144
14. Kroening D, Sharygina N (2005) Formal verification of systemc by automatic hardware/software partitioning. In: Proceedings of international conference on formal methods and models for co-design (MEMOCODE), pp 101–110
15. Moy M, Maraninchi F, Maillet-Contoz L (2005) Lussy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In: Proceedings of the international conference on application of concurrency to system design, pp 26–35
16. Karlsson D, Eles P, Peng Z (2006) Formal verification of systemc designs using a petri-net based representation. In: Proceedings of design, automation, and test in Europe (DATE), pp 1228–1233
17. Habibi A, Tahar S (2006) Design and verification of systemC transaction-level models. IEEE Trans Very Large Scale Integr Syst (TVLSI) 14(1):57–68
18. McMillan KL (OSCI) (2006) SMV model checker. <http://www.kenmcml.com/>
19. Unhelkar B (2005) Verification and Validation for Quality of UML 2.0 Models. Wiley, New York
20. Chen M, Qiu X, Li X (2006) Automatic test case generation for uml activity diagrams. In: Proceedings of international workshop on automation on software test, pp 2–8
21. Chen M, Qiu X, Xu W, Wang L, Zhao J, Li X (2009) UML activity diagram based automatic test case generation for java programs. Comput J 52(5):545–556
22. Eshuis R (2006) Symbolic model checking of UML activity diagrams. ACM Trans on Softw Eng Methodol 15(1):1–38
23. Cimatti A, Clarke EM, Giunchiglia F, Roveri M (1999) NUSMV: A new symbolic model verifier. In: Proceedings of international conference on computer aided verification (CAV), pp 495–499
24. Guelfi N, Mammar A (2005) NUSMV: A formal semantics of timed activity diagrams and its Promela translation. In: Proceedings of Asia-Pacific software engineering conference (APSEC), pp 283–290
25. Das D, Kumar R, Chakrabarti PP (2006) Timing verification of UML activity diagram based code block level models for real time multiprocessor system-on-chip applications. In: Proceedings of Asia-Pacific software engineering conference (APSEC), pp 199–208

26. Object Management Group (2007) UML superstructure V2.1.2. <http://www.omg.org/docs/formal/07-11-02.pdf>
27. Peterson J (1981) Petri nets theory and the modeling of systems. Prentice-Hall, Englewood Cliffs
28. Ericsson M (2004) Activity diagrams: what they are and how to use them. The Rational Edge. <http://www.ibm.com/developerworks/rational/library/2802.html>

System-Level Validation
High-Level Modeling and Directed Test Generation
Techniques

Chen, M.; Qin, X.; Koo, H.-M.; Mishra, P.

2013, XXII, 250 p., Hardcover

ISBN: 978-1-4614-1358-5