It will be seen later in this chapter that behavioural modelling using state machine diagrams will lead directly into other types of behavioural modelling diagrams. For more in-depth discussion concerning state machine diagrams and behavioural modelling, see Reference 1.

3.5 Behavioural modelling using state machine diagrams

3.5.1 Introduction

State machine diagrams are used to model behaviour over the lifetime of a block, or, to put it another way, they describe the behaviour of objects. Both of these statements are true and are used interchangeably in many books. Remember that objects don’t actually exist in SysML, depending which part of the specification you read, but, in order for state machine diagrams to make any sense whatsoever, it is essential that they do exist! All objects are instances of blocks and thus use the block as a template for the creation of its instantiations, which includes any behaviour.

The most obvious question to put forward at this point is, ‘Does every block need to have an associated state machine diagram?’ The simple answer is no, since only blocks that exhibit some form of behaviour can possibly have their behaviour defined. Some blocks will not exhibit any sort of behaviour, such as data structures and database structures. The simple way to spot whether a block exhibits any behaviour is to see whether it has any operations: if a block has operations, it does something; if it does not have any operations, it does nothing. If a block does nothing, it is impossible to model the behaviour of it. Therefore, a simple rule of thumb is that any block that has one or more operations must have its behaviour defined using, for example, a state machine diagram.

3.5.2 Basic modelling

The basic modelling elements in a state machine diagram are states, transitions and events. States describe what is happening within a system at any given point in time; transitions show how to change between such states and events dictate which messages are passed on these transitions. Each of these elements will now be looked at in more detail, starting with the state, an example of which is shown in Figure 3.20.

Figure 3.20  A SysML representation of a state

Figure 3.20 shows a very simple state, which is shown in the SysML by a box with rounded corners. This particular state has the name ‘State 1’ and this diagram should be read as: ‘there is a single state, called “State 1”’. This shows what a state
looks like, but what exactly is a state? The following three points discuss the basics of a state.

- A state may describe situations in which the system satisfies a particular condition, in terms of its property values or events that have occurred. This may, for example, be ‘loaded’ or ‘saved’, so that it gives an indication as to something that has already happened. States that satisfy a particular condition tend to be used when an action-based approach is taken to creating state machine diagrams. This will be discussed in more detail in due course.

- A state may describe a situation in which the system performs a particular activity or set of actions, or, to put it another way, is actually doing something. States are assumed to take a finite amount of time, whereas transitions are assumed to take no time. There are two things that can be happening during such a state: one or more activities and/or one or more actions. Activities are non-atomic and, as such, can be interrupted, hence, they take up a certain amount of logical time. Actions, on the other hand, are atomic and, as such, cannot be interrupted, and, hence, take up zero logical time. Therefore, activities can appear only inside a state, whereas an action can exist either within a state or on a transition. Activities can be differentiated from actions inside states by the presence of the keyword ‘do’, whereas action will have other keywords, including: ‘entry’ and ‘exit’.

- A state may also describe a situation in which a system does nothing or is waiting for an event to occur. This is often the case with event-driven systems, such as Windows-style software, where, in fact, most of the time the system is sitting idle and is waiting for an event to occur.

There are different types of state in the SysML; however, the states used in state machine diagrams are known as normal states. This implies that complexity may exist within a state, such as: a number of things that may happen, certain actions that may take place on the entry or exit of the state, and so forth.

In order for the object to move from one state to another, a transition must be crossed. In order to cross a transition, some sort of event must occur. Figure 3.21 shows a simple example of how states and transitions are represented using the SysML.

From the model in Figure 3.21, it can be seen that two states exist: ‘State 1’ and ‘State 2’, represented by rounded boxes. There is a single transition that goes from

---

Figure 3.21  States and transitions
'State 1' to 'State 2', which is represented by a directed line that shows the direction of the transition. These transitions are unidirectional and, in the event of another transition being required going in the other direction, an entirely new transition is required – the original transition cannot be made bidirectional.

In order to cross a transition, which will make the object exit one state and enter another, an event must occur. This event may be something simple, such as the termination of an activity in a state (the state has finished what it is doing); then it leaves its present state, or may be more complex and involve receiving messages from another object in another part of the system. Event names are written on the transition lines inside special symbols, which will be introduced later in this section.

State machine diagrams will now be taken one step further by considering a simple example.

3.5.3 Behavioural modelling – a simple example

3.5.3.1 Introduction

In order to illustrate the use of state machine diagrams, a simple example will be used that will not only show all basic concepts associated with state machine diagrams, but will also provide a basis for further modelling and provide a consistent example that will be used throughout this section.

The simple example chosen is that of a game of chess, as it is a game with which most people are at least vaguely familiar, and thus most people will have some understanding of what the game is all about. In addition, chess is, on one level, very simple, which means that it can be modelled very simply, yet, on the other hand, it possesses a great deal of hidden complexity that should emerge as the example is taken further.

![Chess match diagram](image)

**Figure 3.22** A simple block definition diagram for a game of chess

Figure 3.22 shows a block definition diagram that represents, at a very simple level, a game of chess. From the model, a 'Chess match' is made up of two 'Player'. Each 'Player' has properties – 'result', 'status' and 'initiator' – and a single operation 'move'. The property 'result' reflects the result of the game and may have values: 'this player win', other player win', 'draw' or 'undecided'. The property 'status' reflects the current status of the game and may take the values 'checkmate', 'stalemate' and
‘game in progress’. Finally, the property ‘initiator’ represents the player who goes first.

3.5.3.2 Simple behaviour

It was stated previously that any block that exhibits behaviour must have an associated state machine diagram. Applying this rule reveals that, of the two blocks present in the block definition diagram, only the block ‘Player’ needs to have a state machine diagram. A very simple state machine diagram for a game of chess is shown in Figure 3.23, by defining the behaviour of the block ‘Player’.

![State machine diagram for a game of chess](image)

Figure 3.23 A simple state machine diagram for a game of chess

From the model, the object may be in one of two states: ‘waiting’ or ‘moving’. In order to cross from ‘waiting’ to ‘moving’, the event ‘player 2 moved’ must have occurred. In order to cross from ‘moving’ to ‘waiting’, the event ‘player 1 moved’ must have occurred.

The two pentagons, one convex and one concave, represent ‘send events’ and ‘receive events’ respectively. A send event is an event that is broadcast outside the boundary of the state machine diagram or, to put it another way, is broadcast to other objects. A receive event is a complement of the send event in that it is accepted from outside the boundary of the state machine diagram.

At a very simple level, this is ‘how’ a game of chess is played, by modelling the behaviour of each player. However, the model is by no means complete as the chess game described here has no beginning or end and will thus go on for ever. Despite the fact that chess games may seem to go on for ever, there is an actual distinct start and end point for the game. This is modelled in the SysML by introducing ‘start states’ and ‘end states’.

The next step, therefore, is to add more detail to this state machine diagram. It is interesting to note that this is actually how a state machine diagram (or any other SysML diagram, for that matter) is created. The diagram almost always starts off as a
simple collection of states and then evolves over time and, as more detail is added, so the model starts to get closer and closer to the reality that it is intended to represent.

3.5.3.3 Adding more detail
The next step is to add a beginning and an end for the state machine diagram, using start and end states. A start state describes what has happened before the object is created, and is shown visually by a filled-in circle. An end state, by comparison, shows the state of the object once the object has been destroyed, and is represented visually by a bull’s-eye symbol.

Start and end states are treated just like other states in that they require transitions to take the object into another state, which will require appropriate events. The events would typically be creation- and destruction-type events that are responsible for the birth and subsequent demise of the object.

![State Machine Diagram]

Figure 3.24 Expanded state machine diagram showing start and end states

Figure 3.24 shows the expanded state machine diagram that has start and end states along with appropriate events.

It can be seen from the model that there is a single start state and then a receive event is shown (the concave pentagon). A decision then occurs and, depending on which player goes first, the control of the state machine diagram will go to either the ‘waiting state’ or the ‘moving’ state. There are three different end states, depending on which player, if either, wins, or whether the game is a draw. The model is now becoming more realistic; its connection to reality is getting closer, but there is still room for ambiguity.

Notice also in this diagram that some conditions have been introduced that are defined on the outputs of the decision symbol (the diamond). These conditions should relate directly back to properties from its parent block, hence providing some basic consistency between a block and its associated state machine diagram.

We know from the block definition diagram that the block ‘Player’ does one thing, ‘move’, but we do not know in which state of the state machine diagram that this
operation is executed. As it happens, it is fairly obvious which state the operation, occurs in: ‘moving’. Operations on a state machine diagram may appear as either ‘activities’ or ‘actions’, which will be discussed in more detail in due course. This may now be shown by adding the activity to its appropriate state by writing ‘do’ in the state box and then adding the activity (operation from the block definition diagram) name, which is shown in Figure 3.25.

![Figure 3.25 Expanded state machine diagram showing activity](image)

The model is now getting even closer to reality; in fact, the model is evolving, which they always do! It is almost impossible to get a model right the first time, since they are living entities and will continue to evolve for as long as they exist. However, there is yet another problem with the state machine diagram, as, although it seems to work well for any situation in a chess game, it is impossible for the game to run! To illustrate this, consider what happens when we begin a game of chess.

### 3.5.4 Ensuring consistency

The first thing that will happen is that two instances of the block ‘Player’ need to be created so that we have the correct number of players. The behaviour of each player is described by the state machine diagram for the block ‘Player’. For argument’s sake, we shall name the two players ‘Player 1’ and ‘Player 2’ and see if the state machine diagram will hold up to the full game of chess. Figure 3.26 shows two identical state machine diagrams, one for each object, that are positioned side by side to make comparisons easier.

In order to begin a game of chess, an instance of the block ‘Chess match’ would be created, which would in turn create two instances of the block ‘Player’. In this example, the object names ‘Player 1’ and ‘Player 2’ have been chosen. Let us now imagine that a game of chess has been started and that ‘Player 1’ is to begin. The event that will occur is ‘player 1 start’, which is present on both state machine diagrams. However, this will put both players straight into the ‘moving’ state, which will make
the game of chess impossible to play, despite its being slightly more entertaining. This is because the events were named with reference to one player, rather than being generic so that they are applicable to any player. In order to make the game work, it is necessary to rename the events so that they are player-independent.

It is important to run through (by simulation or animation, if a suitable tool is available) a state machine diagram to check for consistency. In this case, the error was a simple, almost trivial, misnaming of an event. However, this trivial mistake will lead to the system actually failing!

This serves to illustrate a very important point that relates back to the section about modelling: the different levels of abstraction of the same model. The chess game was modelled only at the object level in terms of its behaviour, which, it is entirely possible, would have resulted in the block being implemented and even

**Figure 3.26  Side-by-side comparison of two state machine diagrams**
tested successfully if treated in isolation and under test conditions. It would only be at the final system test level that this error would have come to light. It would be useful, therefore, if it were possible to model behaviour at a higher level, where the interactions between objects could be shown, as in Figure 3.27.

![Diagram](image.png)

**Figure 3.27** Wouldn't-it-be-nice model

Figure 3.27 shows the two objects from the original block definition diagram, but this time the events from the state machine diagrams have been shown. It is clear from this diagram that there would be a problem with the event names at a higher level of abstraction, which could lead to the problem being sorted out far earlier than the previous case. It would have been nice if we had drawn this diagram as part of our behavioural modelling of the game of chess. Luckily, such a diagram does exist in the UML and is known as a communication diagram – unfortunately, this diagram has been omitted from the SysML, as it is ‘not needed’ (although, ironically, the SysML specification does contain some communications diagrams).

3.5.5 Solving the inconsistency

There are many ways to solve the inconsistency problems that were highlighted in the previous section, two of which are presented here. The first solution is to make the generic state machine diagram correct, while the second is to change the block definition diagram to make the state machine diagrams correct.

3.5.5.1 Changing the state machine diagram

The first solution to the inconsistency problem that is presented here is to make the state machine diagram more generic, so that the events now match up. The state machine diagram for this solution is shown in Figure 3.28.

Figure 3.28 represents a correct solution to the chess model. Note that in this model the event names have been changed to make them more generic. The first events that have been changed are the events on the transitions from the two start states. Note that this time the names have been changed so that they are no longer specific to either player one or player two. The names are now relative to each instance, rather than specific, so the event names have changed from ‘Player 1 begins’ to ‘this’ (player begins) and from ‘player 2 begins’ to ‘other’ (player begins).

The other event names that have been changed are on the transitions between the two normal states. Previously, the names were object-specific, being, as they were,
‘Player 1 moved’ and ‘Player 2 moved’. These names have now changed to a more generic ‘moved’ event, which will apply equally to both objects.

This is by no means the only solution to the problem and another possible solution is presented in the next section, where the block definition diagram is changed to make the state machine diagram correct, rather than changing the state machine diagram.

### 3.5.5.2 Changing the block definition diagram

The second solution to the consistency problem is to change the block definition diagram rather than the state machine diagram, as shown in Figure 3.29.

Figure 3.29 shows a modified block definition diagram in which two new sub-blocks of ‘Player’ have been added. This would mean that, rather than the block ‘Player’ being instantiated, one instance of each block ‘Player 1’ and ‘Player 2’ would be created. This has implications on the state machine diagrams, as the block definition diagram shown here would require a state machine diagram for both ‘Player 1’ and ‘Player 2’, rather than a single state machine diagram for ‘Player’. This would also mean that the initial state machine diagram shown in Figure 3.25 would now be
correct for ‘Player 1’, but that a new state machine diagram would have to be created for the block ‘Player 2’.

Taking this idea a step further, it is also possible to make the two sub-blocks more specific as, in the game of chess, one player always controls white pieces and the other player controls only black pieces. This would have an impact on the block definition diagram again, as each subclass could now be named according to its colour. This is shown in Figure 3.30.

![Chess match diagram]

**Figure 3.30 Further modification of the chess block definition diagram**

Figure 3.30 shows a block definition diagram where the blocks have been named according to colour, rather than simply ‘Player 1’ and ‘Player 2’. This has even more effect on the state machine diagram, as, in the standard rules of chess, the white player always moves first.

### 3.5.6 Alternative state machine modelling

#### 3.5.6.1 Actions and activities

The state machine diagrams presented so far have had an emphasis on the activities in the system where the main state in the state machine diagram had an explicit activity associated with it. This is slightly different from another approach that is often used in many texts, which is referred to here as an action-based approach. In order to appreciate the difference between the action-based and activity-based approaches, it is important to distinguish between actions and activities.

An activity describes an ongoing, non-atomic execution within a system. In simple terms, this means that an activity takes time and can be interrupted. An activity is also directly related to the operations on a block.

An action is an atomic execution within the system that results in either a change in state or the return of a value. In simple terms, this means that an action takes no time and cannot be interrupted.

Activities may be differentiated from actions as they will always appear inside a state (as what is known as an ‘internal transition’) and will use the keyword ‘do/’ as a prefix. Any other keyword used as a prefix (for example, ‘entry/’, ‘exit’) is assumed to be an action.
These two types of execution may change the way that a state machine diagram is created and the type of names that are chosen to define its state machine diagram.

### 3.5.6.2 Activity-based state machine diagrams

Activity-based state machine diagrams are created with an emphasis on the activities within states. Activities must be directly related back to the block that the state machine diagram is describing and provide the basis for a very simple, yet effective, consistency check between the block definition diagram and state machine diagram. As they take time to execute, activities may be present only on a state machine diagram within a state, as states are the only element on the state machine diagram that takes time – transitions are always assumed to take zero time.

The state names, when using the activity-based approach, tend to be verbs that describe what is going on during the state and thus tend to end with ‘ing’, such as the two states seen so far in this chapter: ‘moving’ and ‘waiting’.

### 3.5.6.3 Action-based state machine diagrams

Action-based state machine diagrams are created with an emphasis on actions rather than activities. Actions are often not related directly back to blocks, although they should be, in order to ensure that consistency checks are performed between the two types of diagram. Actions are assumed to take zero time to execute and thus may be present either inside states or on transitions – transitions also take zero time. With an action-based approach, the actions tend to be written on transitions rather than inside states, which leads to a number of empty states. These states have names that reflect the values of the system at that point in time.

The simplest way to demonstrate the difference is to create an action-based state machine diagram for the chess example and then to compare the two different styles of state machine diagram.

Figure 3.31 shows an action-based state machine diagram where the emphasis is on the action ‘/activate’ and ‘/deactivate’ rather than on activities. Notice that the state names here reflect what has happened in the system, implying that the system is doing nothing while waiting for events.

### 3.5.6.4 Comparing the approaches

Both of the approaches shown here are valid and useful; however, there are a number of observations that can be made concerning the use of both approaches.

- The activity-based approach may lead to a more rigorous approach to modelling as the consistency checks with its associated block are far more obvious.
- The activity-based approach makes use of executions that take time and may be interrupted for whatever reason.
- The action-based approach is more applicable to event-based systems such as Windows-based software, where for much of the time the system is idle while waiting for things to happen.
In terms of real-time systems, the action-based approach is less suitable than the activity-based approach, as the basic assumption of executions taking zero time goes against much real-time thinking.

Action-based diagrams tend to be less complex than activity-based, as they tend to have fewer states.

So which approach is the better of the two? The answer is whichever is more appropriate for the application at hand.

3.5.7 Other behavioural diagrams

This chapter has concentrated mainly on state machine diagrams, but the principles that apply to state machine diagrams apply to the other types of behavioural model.

If you can have a good grasp of the models shown here and, just as importantly, understand how they must be checked to ensure consistency between different diagrams (and models), the other diagrams are relatively straightforward.

State machine diagrams are used to describe the behaviour of a block and, hence, its associated objects. They represent a medium level of abstraction.

Activity diagrams emphasize the actions within a system and are generally used at a very low level, or algorithmic level, of abstraction. Activity diagrams may also be used at a medium level of abstraction, but with an emphasis on actions within a system, rather than states. The main difference between an activity diagram and a state machine diagram is that a state machine diagram contains normal states (that may be complex), whereas an activity diagram will only contain activation instances. This is elaborated upon in Chapter 4.
Sequence diagrams model interactions at a high level of abstraction and can be used, among other purposes, to ensure that events passed between state machine diagrams are consistent. It should be pointed out that UML has three other diagrams that operate at the same level of abstraction as the sequence diagram, collectively known as ‘interaction diagrams’. These diagrams are: the communication diagram, the interaction overview diagram and the timing diagram. As has been stated previously, these diagrams have been omitted from the SysML.

Use case diagrams represent the highest level of abstraction and show the interactions between the system and its external actors. These are strongly related to sequence diagrams, which may be used to model scenarios as instances of use cases.

All of the points raised above are examined in Chapter 4, where each diagram is looked at in turn; however, one point that should emerge here is that there is a relationship between types of behavioural diagram and they each possess a number of common features.

3.6 Identifying complexity through levels of abstraction

3.6.1 Introduction

One issue that has come up on a number of occasions in this book is that of complexity. Complexity is guilty of being one of the three evils of life and is one of the fundamental reasons why we need to model in the first place. In this section, two systems will be compared in terms of their complexity, which will be identified through looking at the system from different levels of abstraction. A number of modelling issues will arise from this example.

3.6.2 The systems

Imagine the situation where you are put in charge of two projects, neither of which you have any domain knowledge for, and you have to rely on descriptions of each project. This seems to be a prime application for modelling – there is a lack of understanding about the two systems, there is no indication of how complex each is and this information will need to be communicated to the project teams.

The two examples that have been chosen are chess, as seen in the current example, and also Monopoly. Before the modelling begins, ask yourself a simple question – which is the more complex, chess or Monopoly? Take a moment to consider this before reading on.

3.6.3 Structural view

An initial modelling exercise was carried out and two block definition diagrams were produced – one for chess and one for Monopoly.

The diagram in Figure 3.32 shows a simple block-definition-diagram representation of each of the two systems. It is now time to do a simple complexity comparison of the two diagrams – which of the two is more complex? There is really not much to choose between the two, but, if forced to choose one, then Monopoly could be considered slightly more complex than chess, since it has a higher multiplicity on the