CA314 – Object Oriented Analysis & Design - 7

File name: CA314_Section_07_Ver01

Author: L Tuohey

No. of pages: 16

Table of Contents

7. UML State & Activity Diagrams (see ref 1, Chap. 11, 12) 3
7.1 Introduction 3
7.2 State diagram basics 3
7.2.1 Recap from library case study example 3
7.2.2 Level of abstraction 3
7.2.3 UML syntax and notation for states, transitions, and events 4
7.2.4 Kinds of events 4
7.3 Actions 4
7.3.1 Case where an action is in response to a specific event 5
7.3.2 Case where same action applies for all transitions into a state 5
7.3.3 Case where same action applies for all transitions out of a state ... 6
7.3.4 Example of “combined” entry and exit events 6
7.3.5 Sequences of actions - example 6
7.4 Guards 7
7.5 State diagram examples to illustrate "substates" 8
7.5.1 On-line shopper [Ref. 3] 8
7.5.2 Customer in a Queue [Ref. 1] 9
7.6 Some other points on state diagrams 9
7.7 Activity Diagrams 10
7.8 Concurrency inside states 12
7.9 A note on Harel state charts 12
7.9.1 Introduction 12
7.9.2 Depiction of initial states in a StateChart 13
7.9.3 Concurrent states in a StateChart 14
7.9.4 Windshield Wiper Controller 14

7. UML State & Activity Diagrams (see ref 1, Chap. 11, 12)

7.1 Introduction

Here, we are concerned with showing how an object "decides" to react to a message. To begin with, our focus is on the commonest use of state (or "statechart") diagrams namely to "show how an object reacts to receiving a message by sending messages". Activity diagrams are (in a sense) a variant of state diagrams that *can* aid understanding of complex activities. Sometimes it may be better to use interaction (previous section) rather than activity diagrams.

7.2 State diagram basics

7.2.1 Recap from library case study example

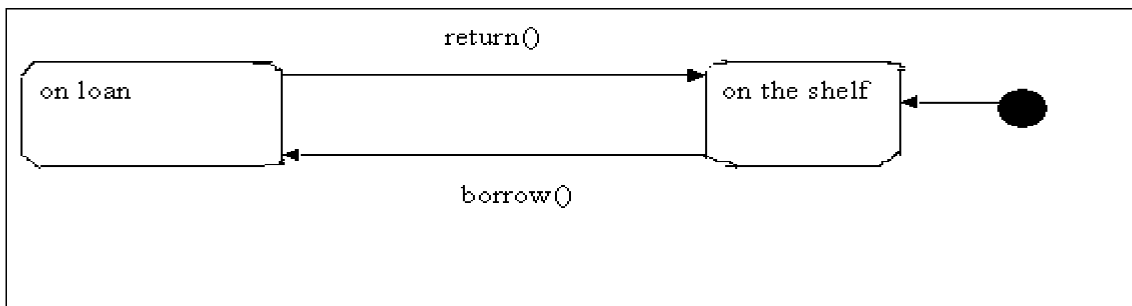


Figure 7.2-1: State diagram of class "Copy" (from Ref. 1)

-- It may be decided that the class `Copy` should have an attribute `onShelf` (true/false if copy is/is not in library)
-- Messages `borrow()` and `return()` are *events* that cause state *transitions*
-- Message `borrow()`informs `Copy` object it has just been borrowed. It *SHOULD* only arrive if `onShelf` = true but what happens otherwise?
-- [*Practical suggestion-departure from UML which "ignores"*] A common solution to the situation that an event arrives when an object is in the "wrong" state is "to have a single, globally accessible object of a class `Error`, whose sole responsibility is to report errors. An object that receives a message it was not expecting sends a message to the error object describing what happened. This error handling is not shown on the state diagram to avoid clutter. [***Implications for coupling are ???***]

Black "blob" is an optional *start marker*. Useful if objects of a class always start in the same state

7.2.2 Level of abstraction

In general, the values of *all* the attributes of an object define its state. In fact, to be complete, one should say that the object's state also depends on the state of objects it

is linked to etc. However, in a state diagram, we are often concerned with a particular aspect of an object which is defined by just a few of its attributes (e.g. onShelf) - for this purpose we are indifferent to the values of its other attributes (e.g. library number of copy, book it is a copy of) and do not depict them.

7.2.3 UML syntax and notation for states, transitions, and events

states - shown as rounded boxes

transitions between states - shown as arrows

events that cause transitions - most common kind is receipt of a message, which is shown by writing the message on the transition arrow (for other event kinds, see section 7.2.4).

start marker - shown as black blob with unlabeled arrow to initial state

stop marker - shown as ringed black blob with unlabeled arrow to it.

There may be several stop markers, or none, present. Meaning is that the object has reached the end of its life & will be destroyed

7.2.4 Kinds of events

Event Type	Description
Call event	Most usual and already discussed An object receives a call for one of its operations Annotated by 'signature' of event (See below)
Signal event	An object receives a signal (or asynchronous communication) Annotated by 'signature' of event (See below)
Change event	Occurs when a condition becomes true Annotated by **when** keyword with a condition (usually Boolean).
Elapsed-time event	Caused by the passage of a designated period of time after a specified event (often the entry to current state) Annotated by **after**(time expression).

Note: The basic syntax for a 'call' or 'signal' event is

```
event-name parameter-list
```

where `parameter-list` is optional.

7.3 Actions

We have

An event is something done to an object, such as *it being sent* a message.

An action is something that an object does, such as *it sending* a message.

7.3.1 Case where an action is in response to a specific event

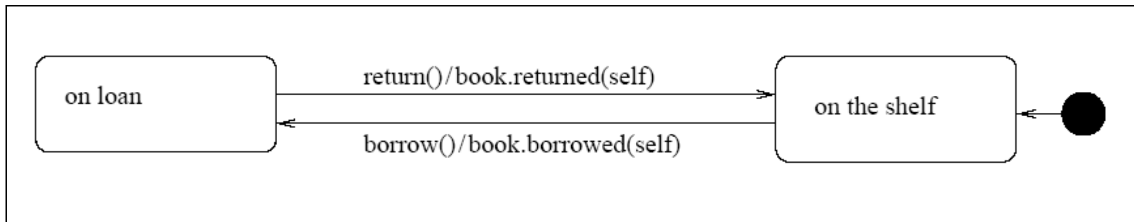


Figure 7.3-1: State diagram of class "Copy", with actions (from Ref. 1)

Figure 7.3-1 illustrates that an action is shown after the event on transition arrow, separated by "/".

In the figure, "book." identifies the object to which the message is being sent. Also, "returned(self)" is a message with a parameter. Here, the parameter is an object of class Copy, namely (a reference to) the object itself. The same applies for "/book.borrowed(self)".

In summary, the meaning of Figure 7.3-1 is that the Copy object sends messages borrowed(self) and returned(self) as part of its reactions to receiving the messages borrow() and return(), respectively.

7.3.2 Case where same action applies for all transitions into a state

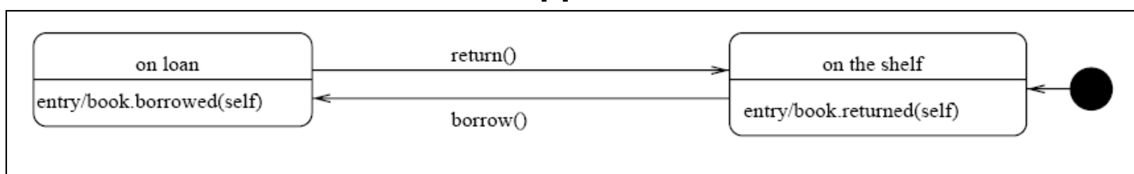


Figure 7.3-2: State diagram of class "Copy", with entry actions (from Ref. 1)

Every time an object enters a state there is an entry event. Normally, this is not shown explicitly unless, as here, there are actions are associated with it.

In each state of Figure 7.3-2, an action is shown inside the state, as a reaction to the special event entry.

The Copy object sends messages borrowed(self) and returned(self) in reactions to entering the "on loan" and "on the shelf states", respectively.

7.3.3 Case where same action applies for all transitions out of a state

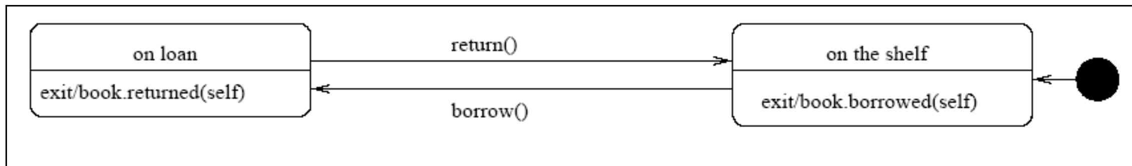


Figure 7.3-3: State diagram of class "Copy", with exit actions (from Ref. 1)

Every time an object leaves a state there is an exit event. Normally, this is not shown explicitly unless, as here, there are actions are associated with it.

In each state of Figure 7.3-3, an action is shown inside the state, as a reaction to the special event exit.

The Copy object sends borrowed(self) and returned(self) in reactions to exiting "on the shelf" and "on loan" states, respectively.

7.3.4 Example of "combined" entry and exit events

Various alternatives to Figures 7.3-2 or Figure 7.3-3 are possible, though sometimes the implications may not be quite equivalent. Figure 7.3-4 depicts one such alternative:

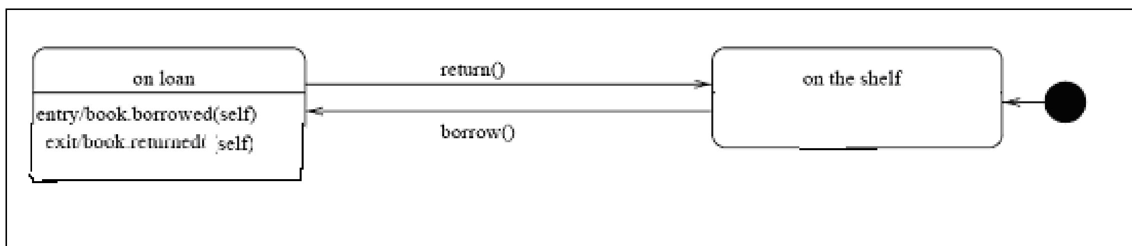


Figure 7.3-4: State diagram of class "Copy", with entry & exit actions on 1 state

7.3.5 Sequences of actions - example

Consider the following figure (Figure 7.3-5), in which the "Average" object waits for "update" messages from other objects. Such messages contain a parameter "val" to be added to the current total.

"update" is put within the icon so that "entry" and "exit" events are **not** triggered on its receipt.

There are two (external) events that can cause a self-transition ("report", "reset").

Finally, action sequences contain "/" to separate actions.

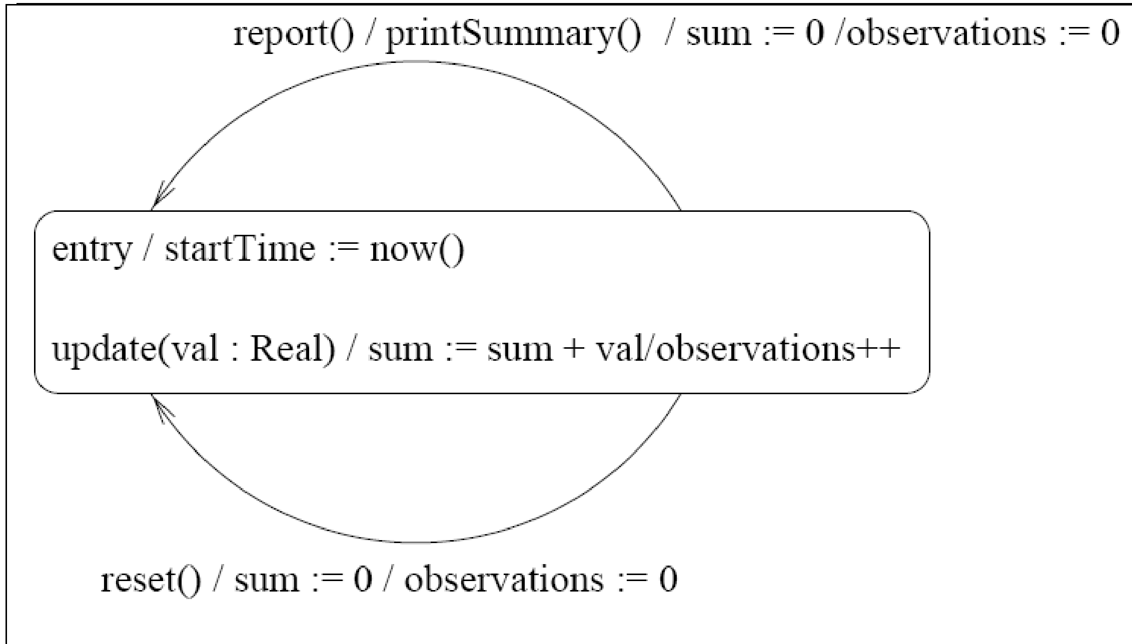


Figure 7.3-5: State diagram of class "Average" (not a recommended style)

7.4 Guards

We had before

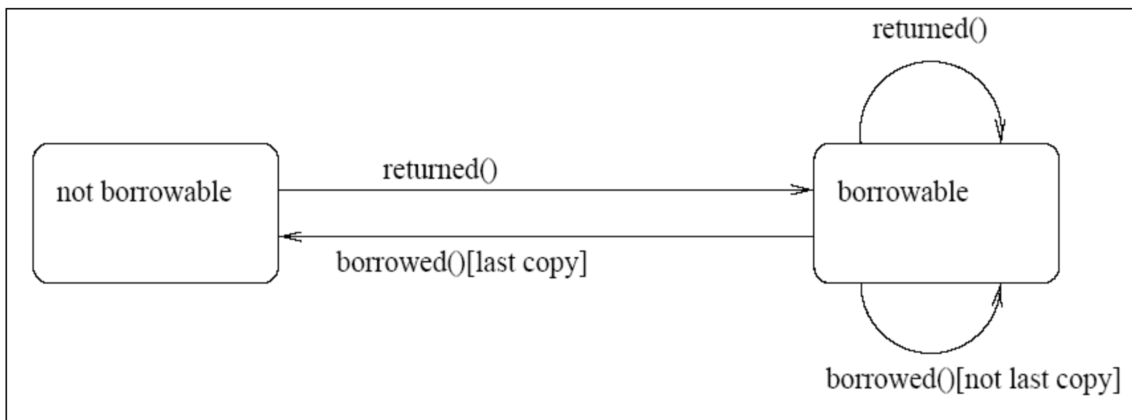


Figure 7.4-1: State diagram of class "Book" (from Ref. 1)

-- This illustrates the situation where *sometimes an event will cause a state transition and sometimes not*. Basically, it means we have to look in more detail at the attribute values. In the above example, message borrowed() causes a transition to Not Borrowable only when the copy borrowed is the last copy on the shelf.

This is shown in the diagram by including two *conditions* , [last copy] and [not last copy]. The conditions guard the transition. The transition caused by receiving returned() is unguarded.

Figure 7.4-16 also illustrates the case of self-transitions.

7.5 State diagram examples to illustrate "substates"

7.5.1 On-line shopper [Ref. 3]

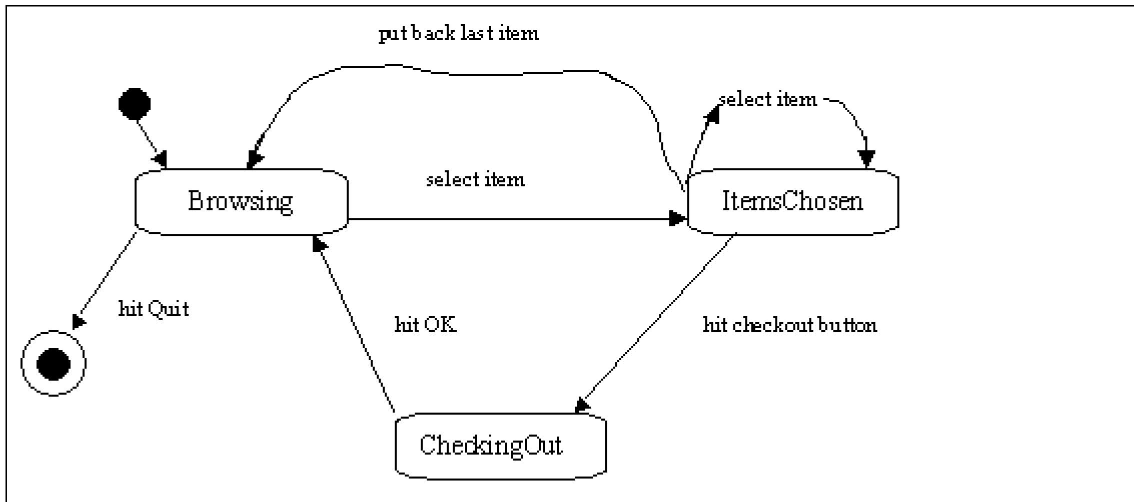


Figure 7.5-1: State Diagram for OnLineShopper Class, *without* substate detail

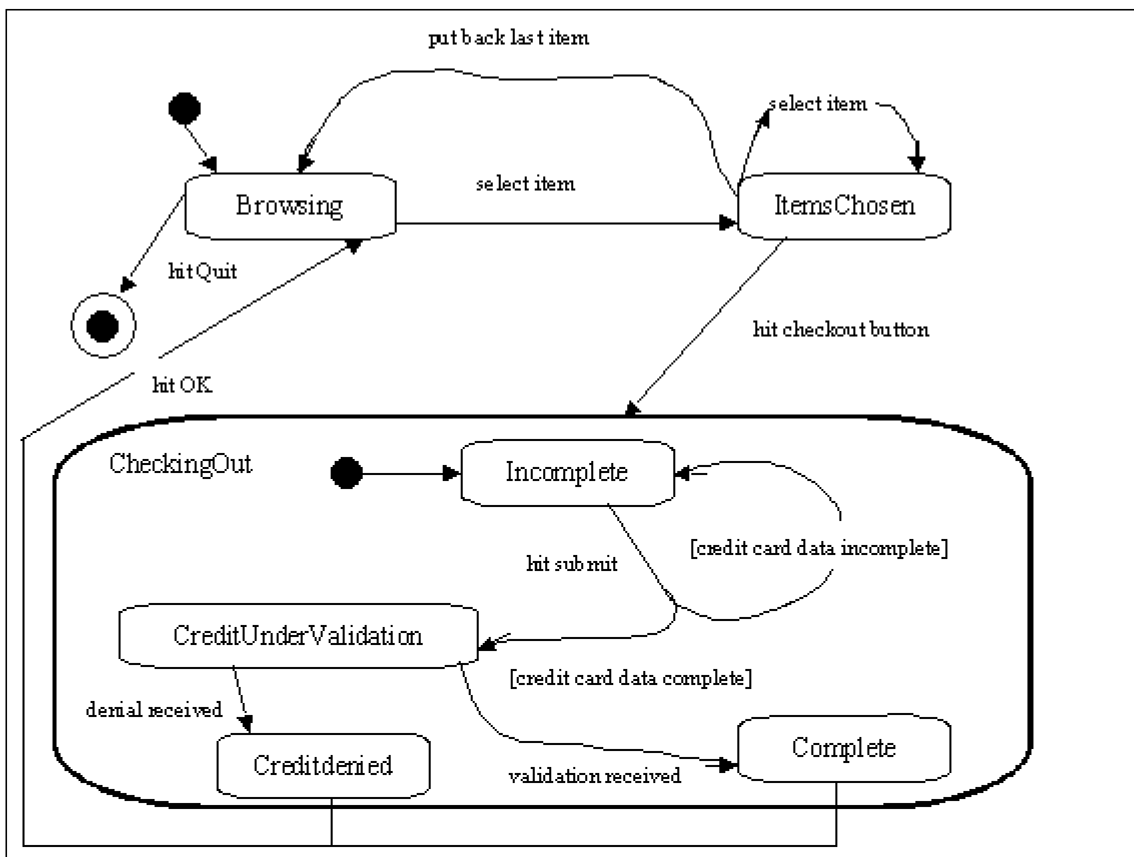


Figure 7.5-2: State Diagram for OnLineShopper Class, *with* substate detail

Note that when Shopper object enters the CheckingOut state, it automatically enters the substate Incomplete (i.e. CheckingOut.Incomplete).

7.5.2 Customer in a Queue [Ref. 1]

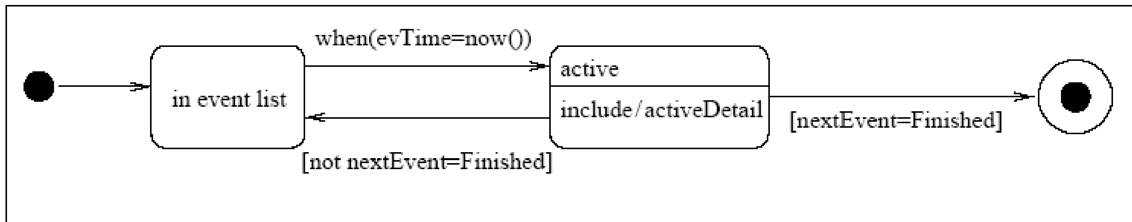


Figure 7.5-3: State Diagram for Customer Class

Notice that Figure 7.5-3 contains an example of an "elapse time" (or simply "time") event.

In the “active” state, "include/activeDetail" means that “active” is a compound state and that there is a detailed state diagram called “activeDetail” nested within it (Figure 7.5-4). There is an implicit "completion" event corresponding to when this internal state diagram reaches its end state.

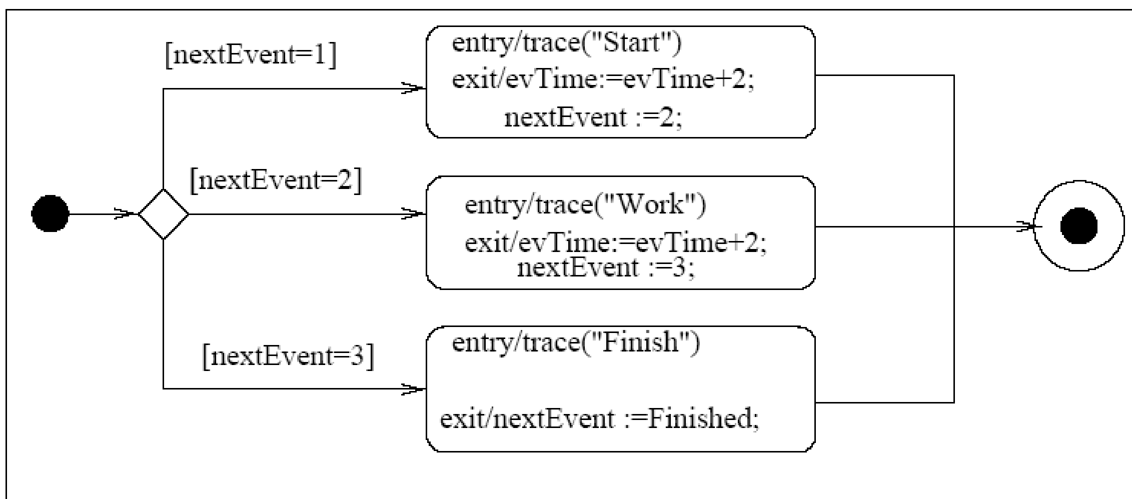


Figure 7.5-4: Nested state diag. “activeDetail” for Class Customer’s *active* state

This example is drawn from the case study in Chapter 17 of Ref. 1. In that case study the first, second and third sub-states represent "waiting to join a queue", "in queue, waiting for service" and "being served, waiting for service to be completed", respectively.

7.6 Some other points on state diagrams

- Braude [ref. 3] writes about `state diagrams as follows:

"*Some applications or parts thereof are conveniently thought of as being in one of several possible states. UML state diagrams help us to visualize these and the events that cause transitions among them*".

- State diagrams should be as simple as possible. Otherwise, with complex state diagrams, one gets such problems as:
  - -- Harder to understand
  - -- Harder to write code correctly for - end up with many conditional sections
  - -- Harder to test
  - -- Harder for external code to use the class correctly

7.7 Activity Diagrams

Activity diagrams describe how activities are coordinated. In general, they are useful to describe situations where there are dependencies between activities, such as

- to show how an operation is implemented (in a method, say)
- to describe carrying out of a "use Case" and how it may depend on other "Use Cases"

Activity diagrams incorporate much of the traditional "flow charts".

The following diagram (from ref. 3, Braude) depicts the various activity chart features:

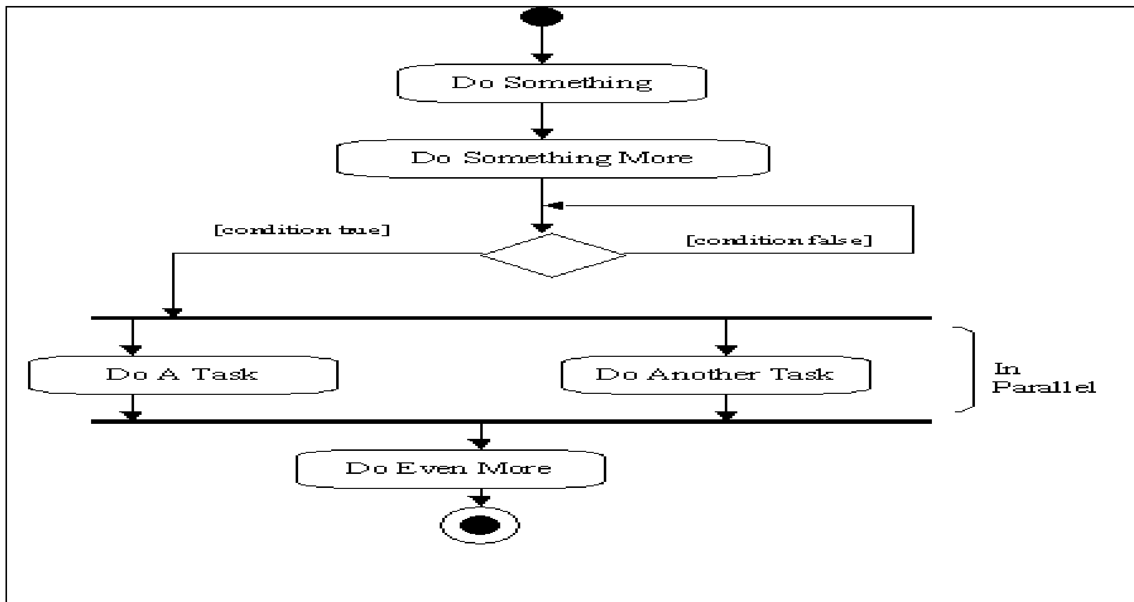


Figure 7.7-1: Summary of Activity Chart Notation

As a specific example, consider Figure 7.7-2 which is related to our initial case study. Notice, in this figure, that "Swim lanes" can be useful in arranging activity diagrams to be clearer for human readers; here, they are used to indicate which activities are performed by whom.

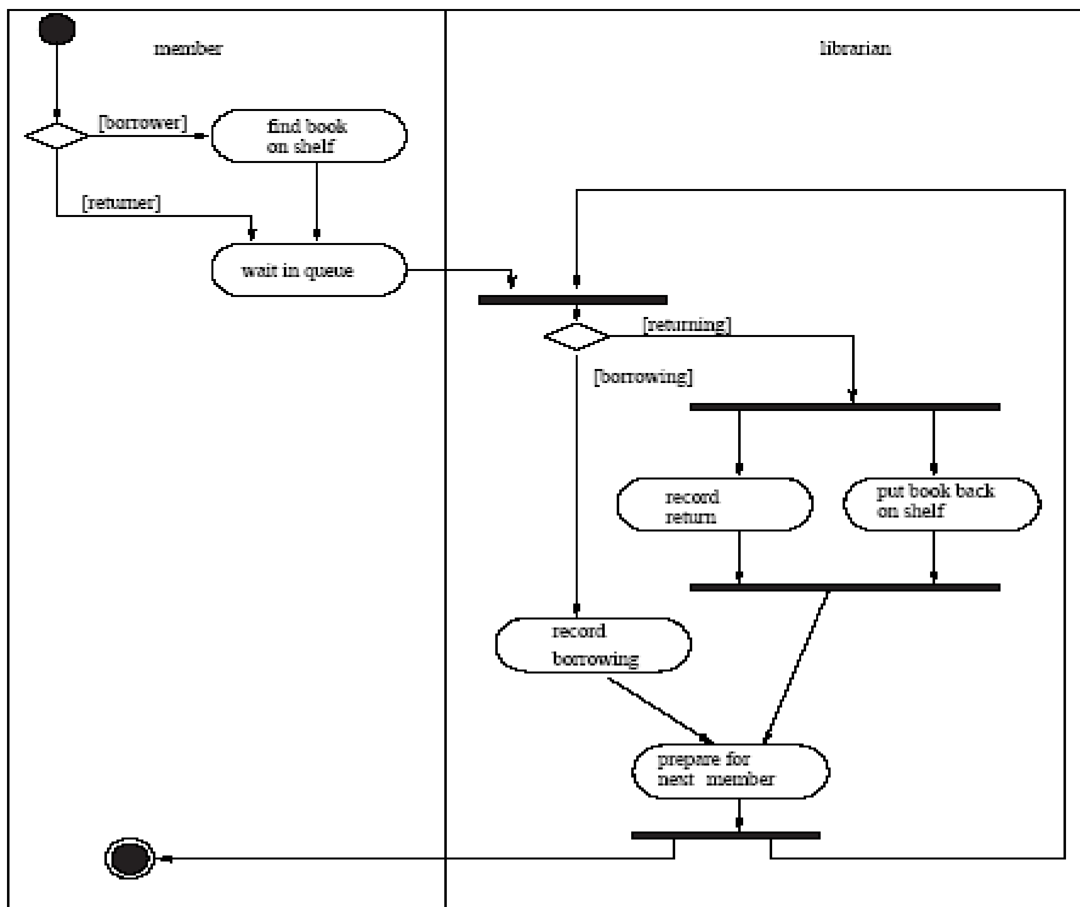


Figure 7.7-2: Business level activity diagram of the library

This figure describes the human interactions into which the new system must fit.

7.8 Concurrency inside states

Consider Figure 7.8-1, which depicts two equivalent UML ways of expressing the same thing.

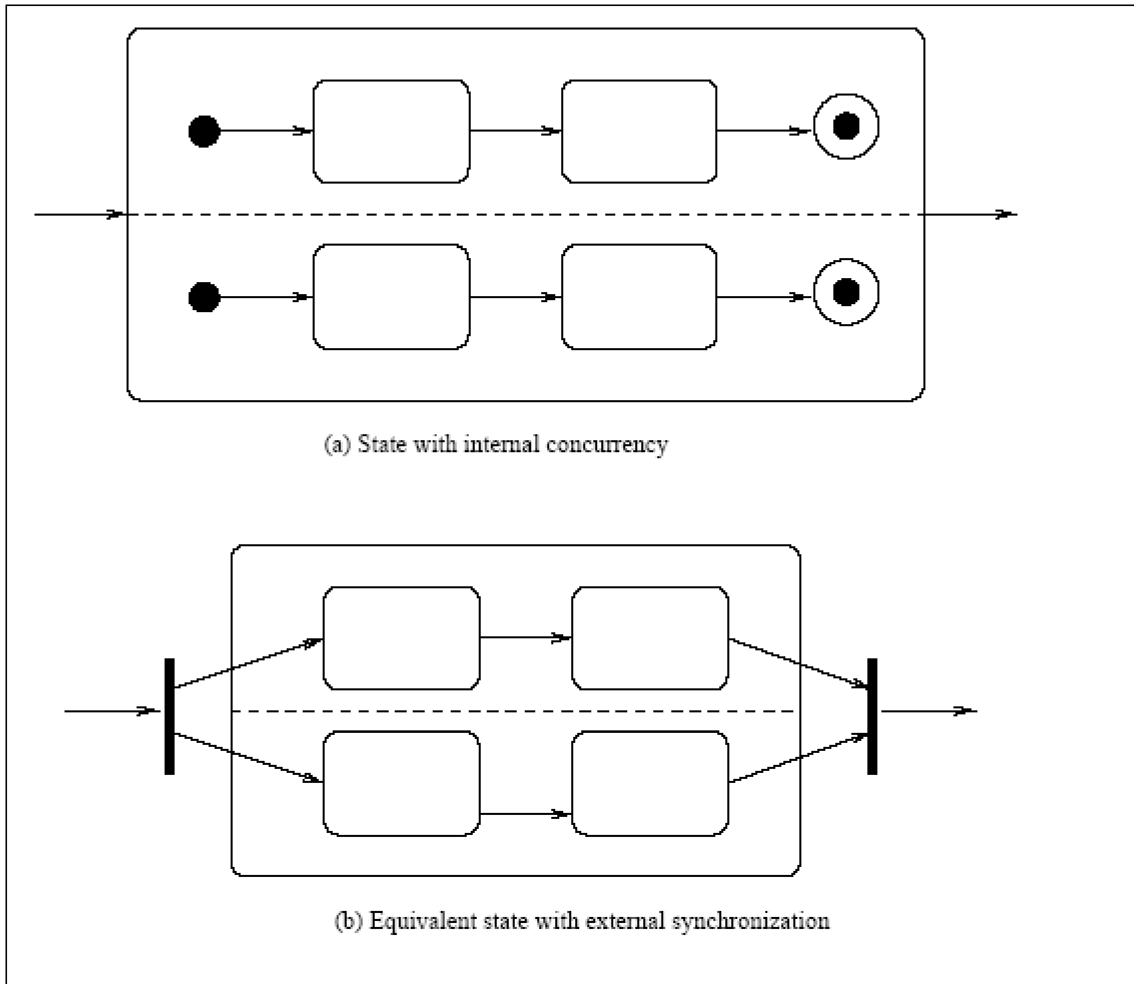


Figure 7.8-1: State diagrams with concurrency

In (a): - At entry, start states of all (both) regions are reached at the same time.
- Leaving transitions leaving "fire" when all regions have reached the end state

In (b), use of "synchronization bars" in a state diagram is depicted.

7.9 A note on Harel state charts

7.9.1 Introduction

1) David Harel[1] developed the StateChart notation in order to have a visual notation that combined

[1] Harel, David, "On visual formalisms", Communications of the ACM, Vol. 31, No. 5, pp.514-530, May 1988.

- the ability of Venn diagrams to express hierarchy
- the ability of directed graphs to express connectedness

His idea was to beat the **state explosion** problem of ordinary state machines.

2) Essentially, StateCharts are used as state diagrams in UML with some minor notational differences (see examples below).

3) In Harel's language, a **blob** is the basic building block. We have that

- a blob can contain other blobs (**hierarchy**)
- Blobs can be **connected** by **edges**

Blobs are interpreted as states and edges as transitions.

4) We do not present the full StateChart system but instead just the main features via examples. Essentially, these examples can be viewed as examples of UML state diagrams also.

7.9.2 Depiction of initial states in a StateChart

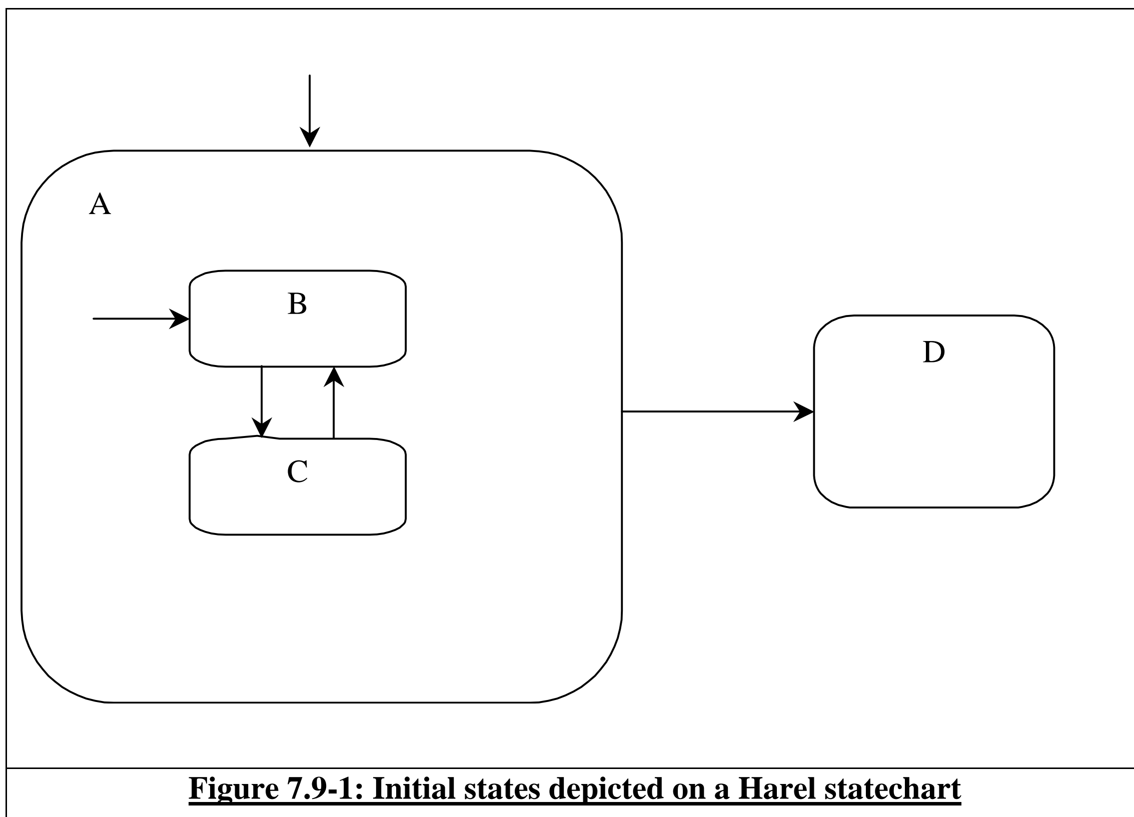


Figure 7.9-1: Initial states depicted on a Harel statechart

This should be interpreted exactly as before for UML state diagrams except that UML uses a black circle (•) to identify an initial state whereas in the above diagram this is indicated by an arrow (or edge) that has no source state. We have,

- State A is entered implies that at a lower level state B is also entered

- When a state is entered, we can think of it as being active
- The convention is that an edge must start and end on a state outline. If a state contains sub-states the edge refers to all the sub-states. So, above, the edge from A to D means that the transition can occur from either B or C.

7.9.3 Concurrent states in a StateChart

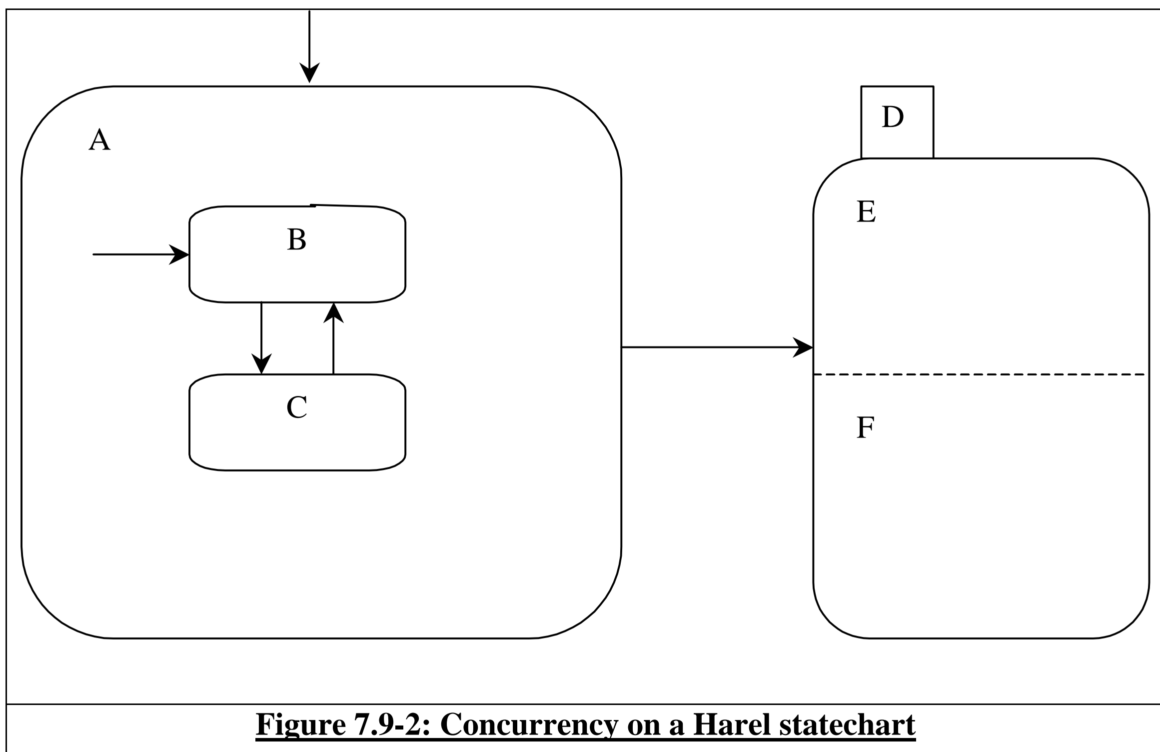


Figure 7.9-2: Concurrency on a Harel statechart

As before for UML, the dotted line indicates that state D really refers to 2 concurrent states E and F. Harel's convention is to move D's state label to a rectangular tab (as shown)

When the transition from A to D occurs both E and F are active

7.9.4 Windshield Wiper Controller

Consider a system where a windshield wiper is controlled by a **lever** with a **dial**.

The lever has 4 positions: OFF, INT (i.e. intermittent), LOW, and HIGH.

The dial positions indicate 3 intermittent speeds and the dial position is relevant only when the lever is in the INT position.

The "decision table"[2] below shows the wiper speeds in wipes per minute for the lever and dial positions:

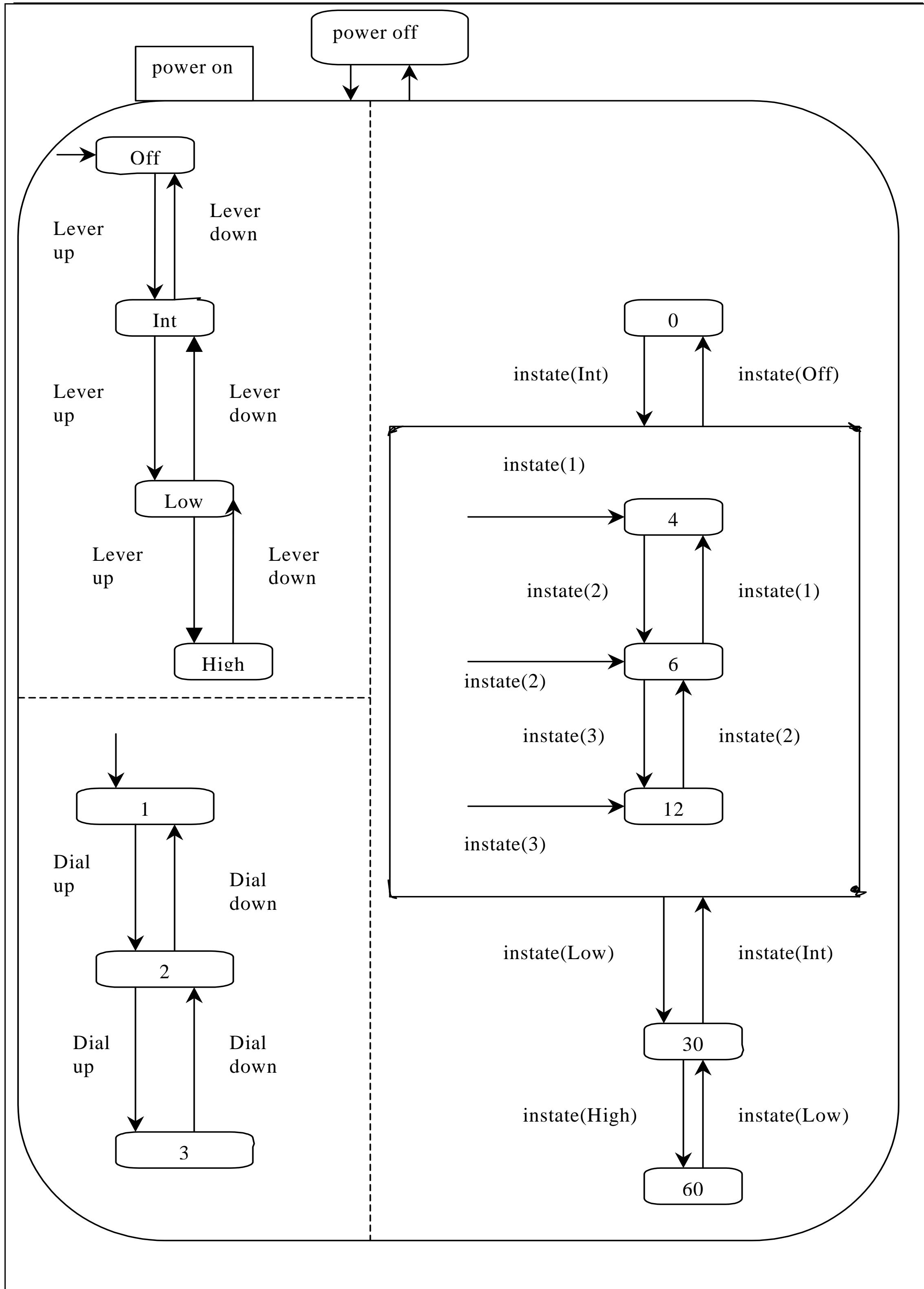
[2] We will come back to "decision tables" in a later section when discussing testing.

c1. Lever	OFF	INT	INT	INT	LOW	HIGH
c2. Dial	n/a	1	2	3	n/a	n/a
a1. Wiper	0	4	6	12	30	60

Table 7.9-1: Decision table for wiper system example

The StateChart below (Figure 7.9-3) describes the system's behaviour.

Note that transitions in the **lever** and **dial** "components" are caused by events whereas those in the **wiper** "component" are all caused by propositions that refer to what state is active in the dial or lever component. (*These "propositions" are part of the StateChart syntax as permitted by the StateMate product*). In UML we would use "guard conditions" to do the job of the "propositions.



<u>Figure 7.9-3: Windscreen wiper system</u>