

# Statecharts

- Development of the statechart
  - Problems with state-based approaches
- Modelling with statecharts
  - Simple examples
  - Concurrency
  - Other features

# About Statecharts

- The statechart notation was developed by David Harel
  - inadequacies of state transition diagrams (STDs).
- A statechart diagram shows a state machine.
  - Statechart diagrams are useful for modelling the lifetime of an object.
  - A statechart diagram shows flow of control from state to state.
- *Typically whole system not sum of object states*

# When to use Statecharts

- When you have a complex object
  - and are not sure how it is going to function.
- When you REALLY need to know the states of an object
  - because it is vital to the success of the application you are working on.
- *However, can use many other (non UML alternatives).*

# Behaviour a reactive objects

- Model the behaviour of *reactive objects*.
  - one whose behaviour is characterised by its response to events dispatched from outside its context.
  - A reactive object has a clear lifetime whose current behaviour is affected by its past.
- Statecharts are good for modelling the behaviour of **user interface objects** so that one can code from them.

# More about statechart usage

- Statecharts model the event order of dynamic aspects on the system.
- UML statecharts (an ‘extension’ of Harel’s original notation) show:
  1. States: simple and composite
  2. Transitions, including events and actions.

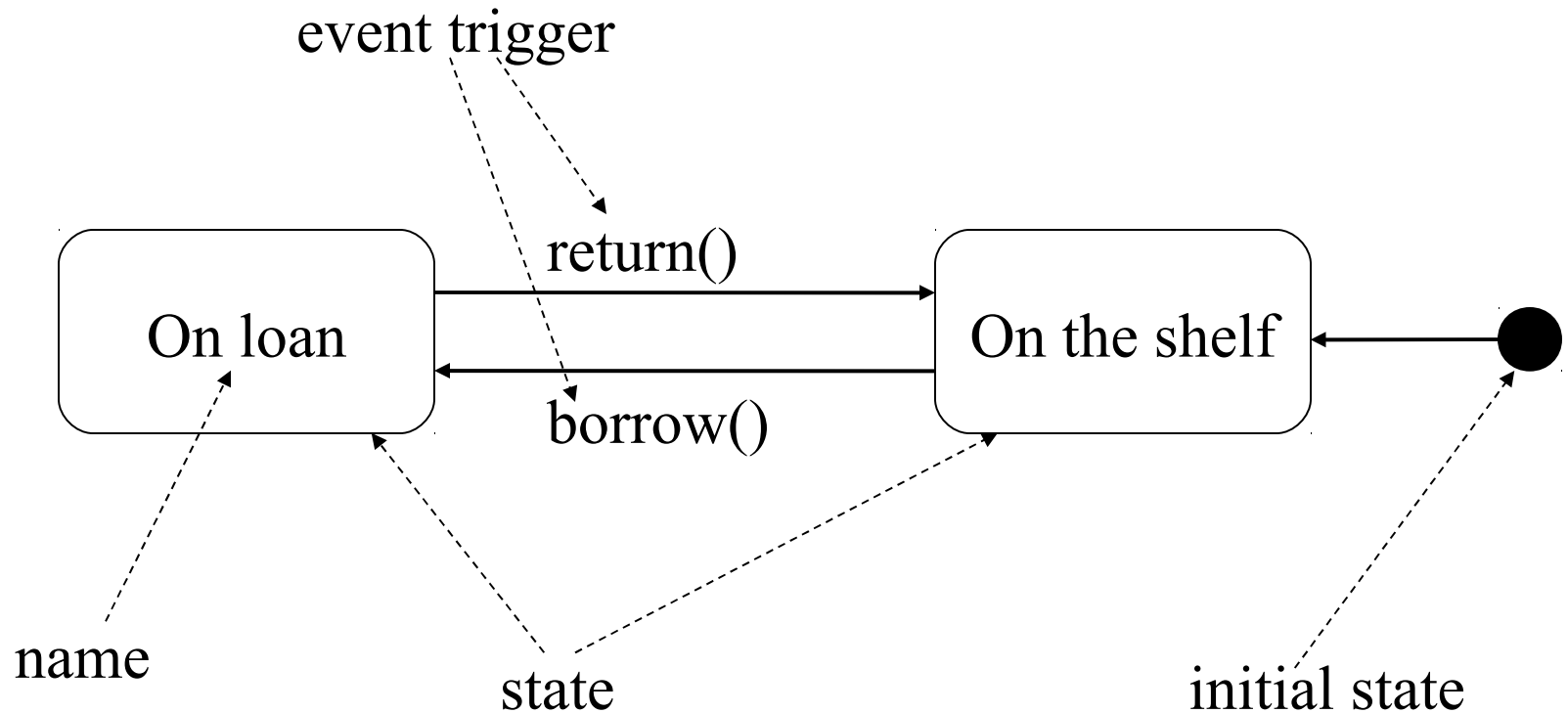
# Modelling with statecharts

- When modelling the behaviour of a reactive object with statecharts one is specifying:
  1. The stable state in which that object may live.
  2. The events that trigger a transition from state to state.
  3. The actions that occur on each state change.
- Can also model the creation and destruction of the object.

# States

- Stable state represents a condition in which an object may exist for some identifiable period of time.
  - When an event occurs, the object may move from state to state (a transition).
  - Events may also trigger self- and internal transitions, in which the source and the target of the transition are the same state.
  - In reaction to an event or a state change, the object may respond by dispatching an action.

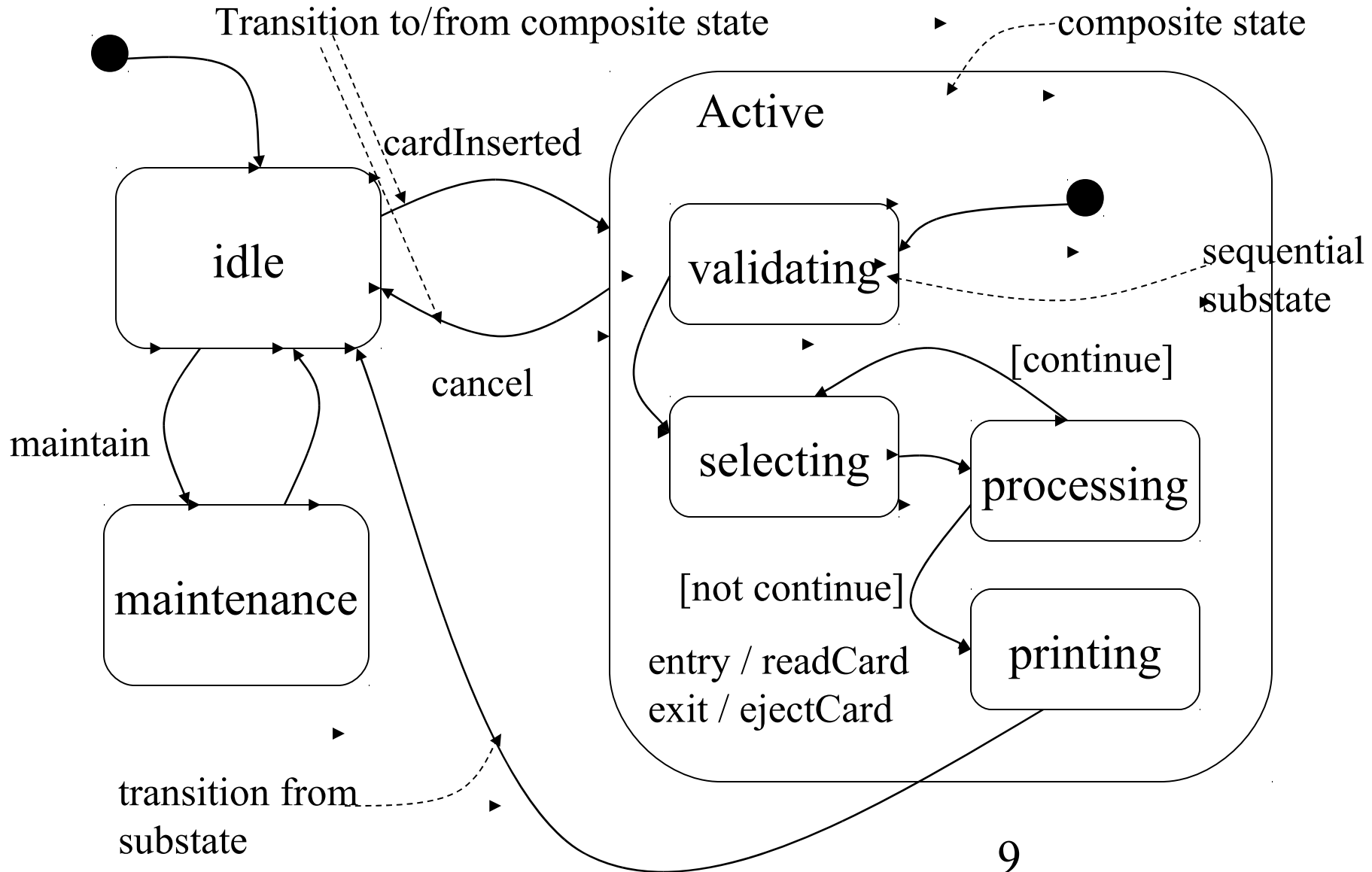
# Simple Statechart example



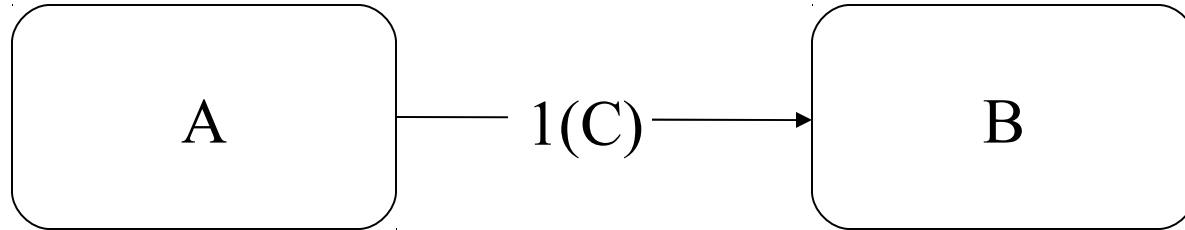
Describing a Copy of a Book object



# Composite State & Substate



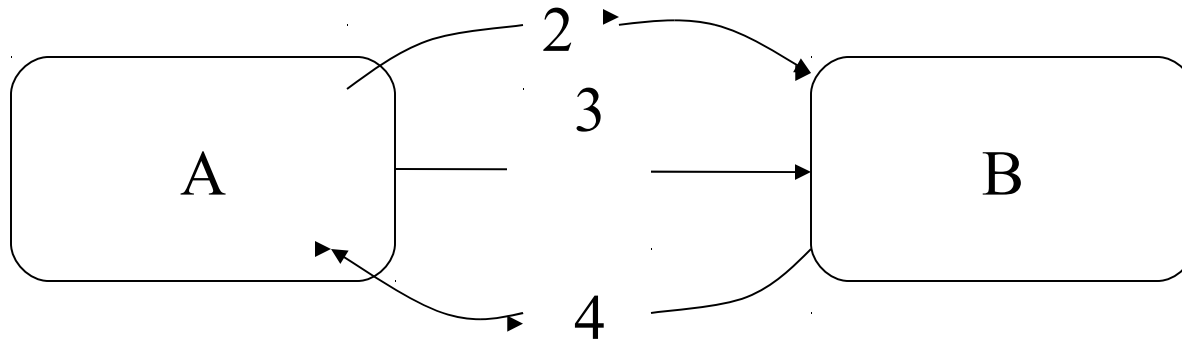
# Simple Transition



If event 1 occurs in state A and condition C is true at the time, then make the transition to state B

# States with Depth

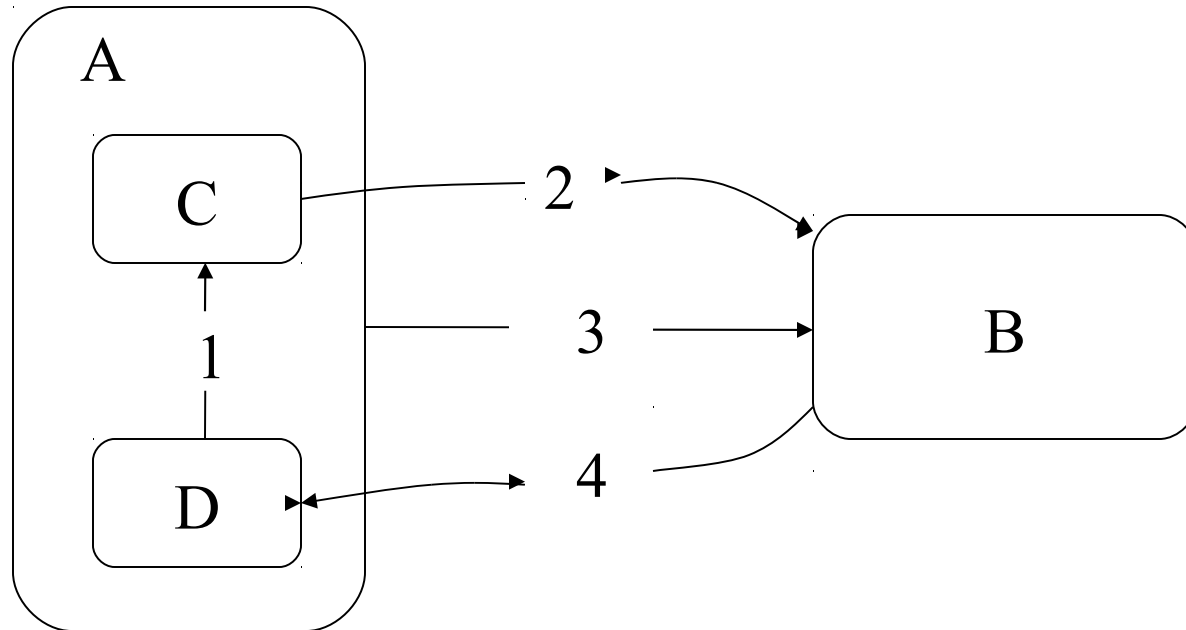
In a state transition diagram all the states appear at the same level in the diagram. In a statechart, however, the states are usually arranged in a hierarchy. In other words, states have depth. For example, state A in the picture has a lower level of detail. Although this is not shown, it is apparent that this exists because event 2 starts inside state A and event 4 terminates inside state A.



We can therefore model a high-level abstraction of a system and not bother with lower-level details just yet.

# Inside States

To understand what happens inside state A we can draw the lower level of detail for that state.



When the system is in state A, it is also in either state C or D at the next level in the hierarchy. There can be any number of levels and events can originate and terminate at different levels. E.g. event 2 starts at state C and terminates at state B. Event 4 starts at state B and terminates at state D.

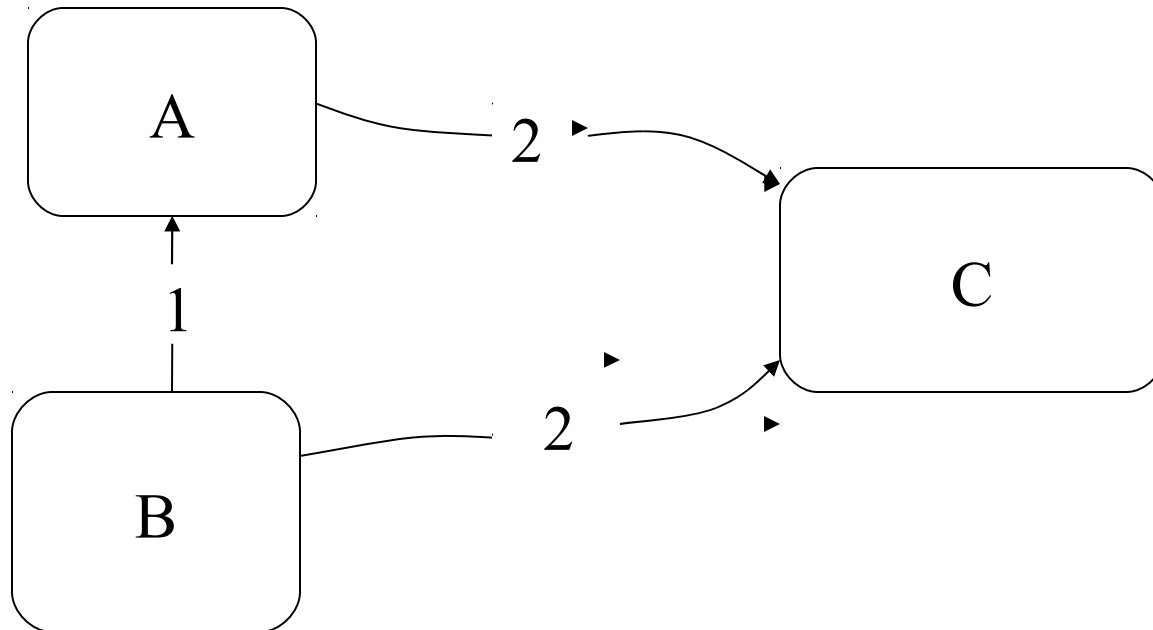
# Using Depth

Depth allows a state diagram to be viewed at different levels of abstraction. It provides an effective way of managing the design of a complex system. The high-level states can be identified first and the lower-level details deferred.

Systems designed as a hierarchy are generally easier to understand because the structure of the system is not obscured by irrelevant details.

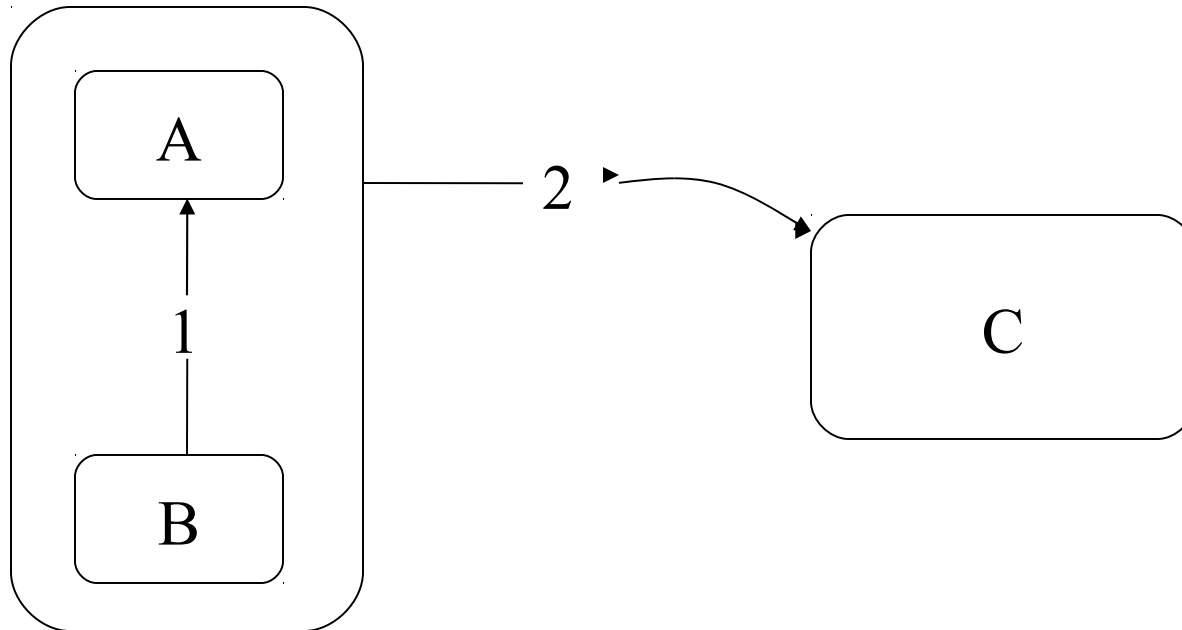
Depth in statecharts is not restricted to refining states into lower level states. States can also be used to cluster groups of states in order to reduce event arrows in a statechart.

# Depth & clustering



In the statechart, event 2 will cause a transition to state C when the system is in state A or B. The two states can be clustered into another state and the two event arrows can be replaced by just one that is attached to the clustering state.

# More on clustering



Although in this simple example the number of arrows has been reduced by just one, if a large group of states is clustered by a state, then a very significant reduction in the number of arrows can be achieved.

# Clustering and refinement

- Depth serves 2 purposes: *clustering* and *refinement*.
  - Clustering is concerned with grouping states together in order to reduce the number of event arrows - it is a bottom-up approach.
  - Refinement is concerned with identifying high-level states and, importantly, allowing the lower level details of those states to be delayed until a later time - a top-down approach.



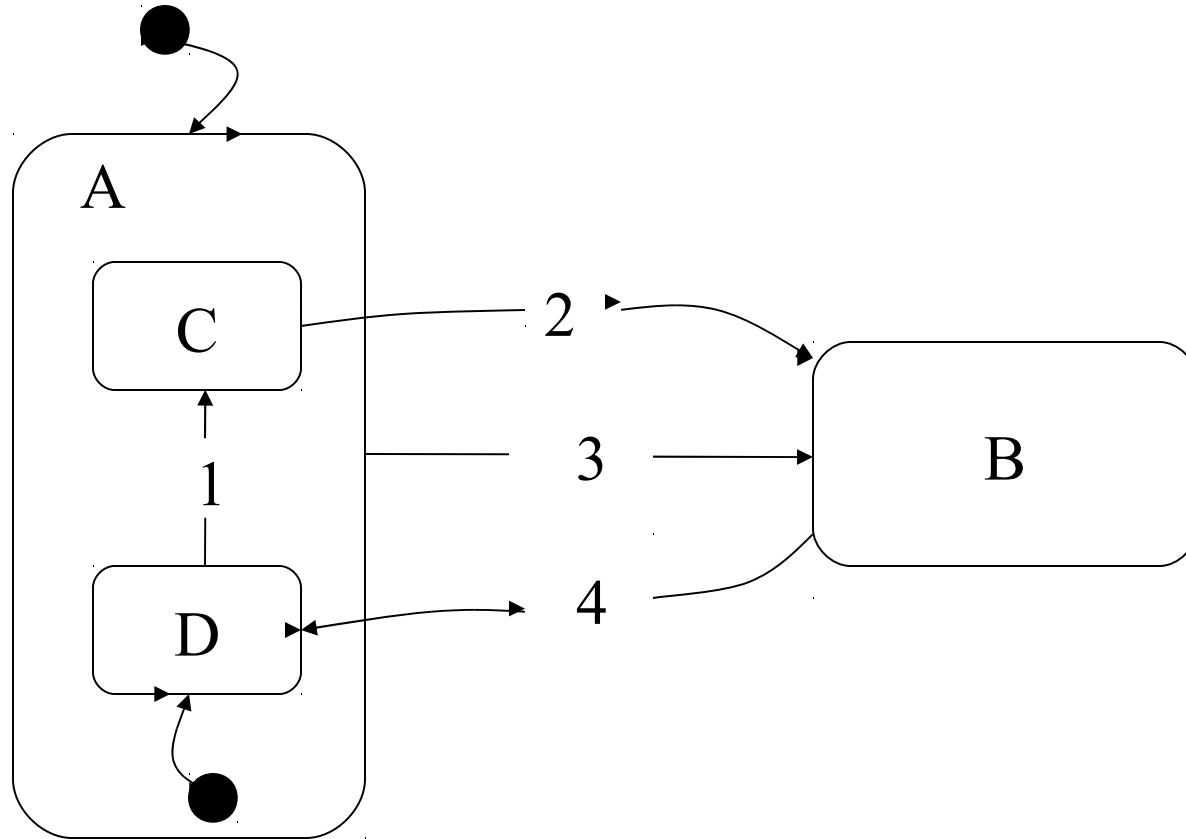
# Advantages of depth

- Low-level details of high-level states can be produced simultaneously by different people.
- The system is easy to understand
  - because it is viewed at different levels of abstraction and the details of one part can be understood without having to understand the entire system in detail.
- The design can be divided into small parts
  - that can fit onto pages of a design document.
- Number of event arrows in a can be reduced.

# Default start states (1)

- Can be several start states in a statechart
  - every level in the hierarchy requires a default start somewhere.
- A start state is identified by a short arrow terminated by a solid circle.
  - The next slide has the default start state at the highest level being state A (rather than state B).
    - Within state A the default starting state is D.
    - Thus, when the system is started the initial state will be state D within state A.

# Default start states (2)

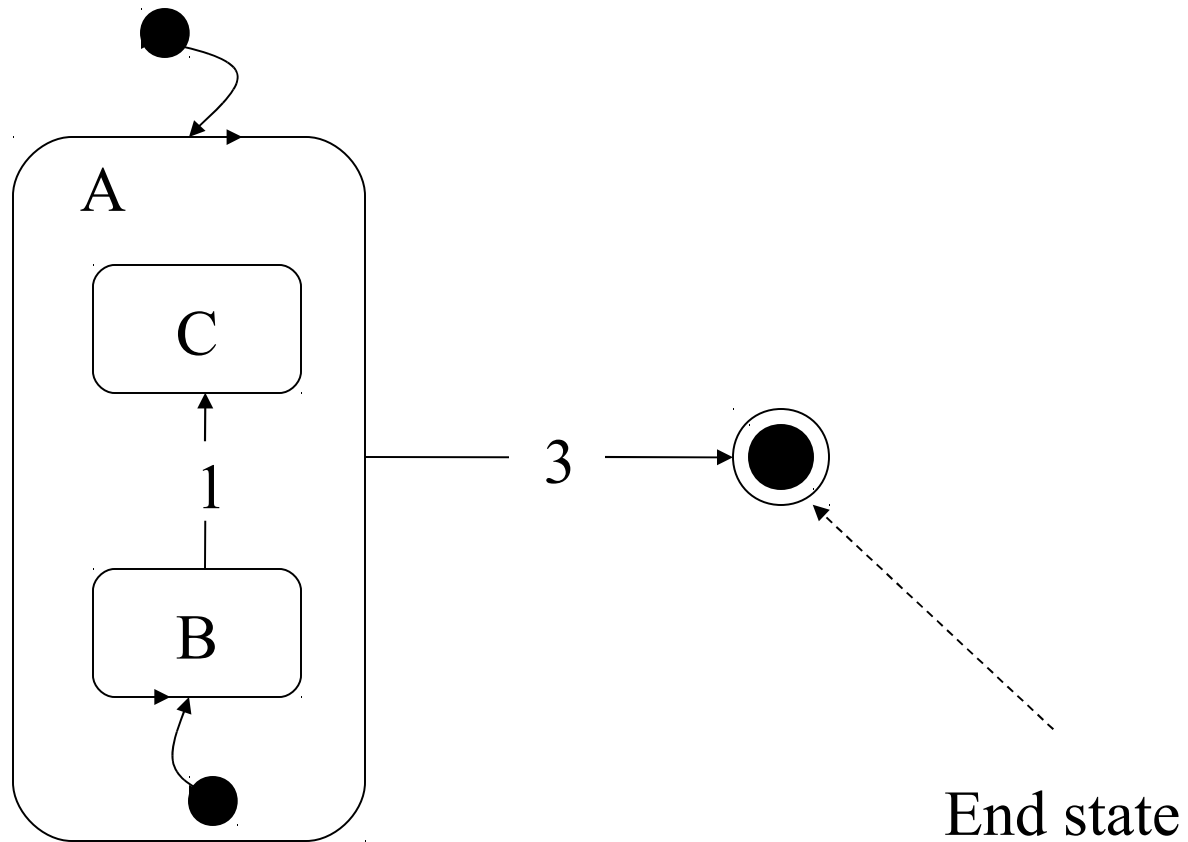


When the system is started the initial state will be state D within state A.

# End states (1)

- Harel (originally) did not define an end state to his notation though the UML does.
  - After all, most systems are designed to shut down at some point during their execution.
  - The UML symbol for the end state is an arrow terminating at a solid circle surrounded by an empty circle as seen in the next slide.
  - An action typically associated with an event that terminates at the end state is one that will cause the application to close down.

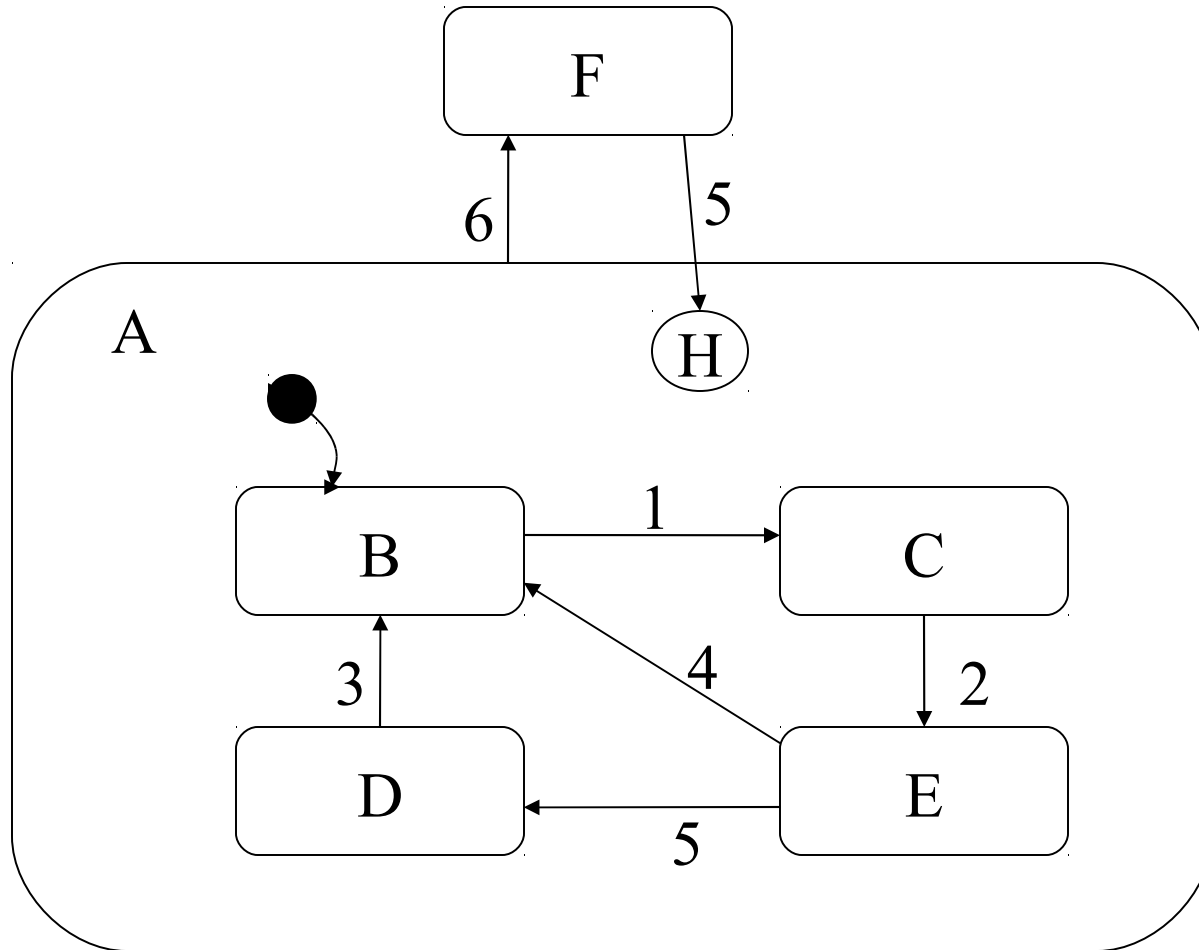
# End states (2)



# The history mechanism

- Provides a way of entering a group of states based on the system's history in that group.
  - That is, the state entered is the most recently visited state in that group.
  - In the next slide when event 5 occurs and state A is entered the history mechanism is used to determine the next state within A.
    - This is read as 'enter the most recently visited state in the group (B, C, D, E) or enter state B if this is the first visit to the state'.

# History mechanism in action

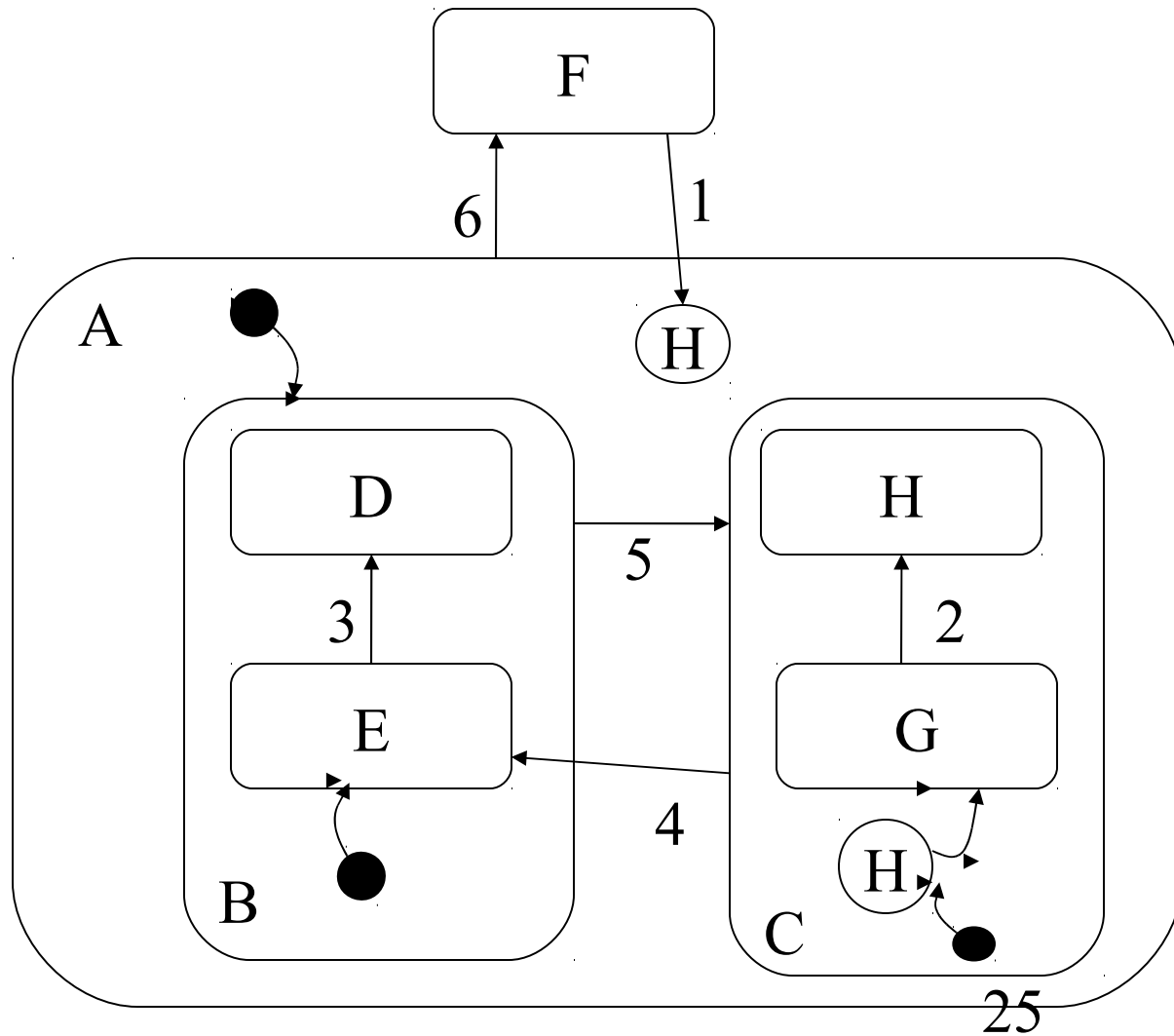


# History mechanism usage

- The history of a system overrides the default start state.
- A default start state must be specified for a group that uses the history mechanism for when the group is entered for the first time.
- The history of a system is only applied to the level in the hierarchy in which it appears.
- To apply the history mechanism at a lower level in the state hierarchy it is necessary to use a history symbol in the lower levels.

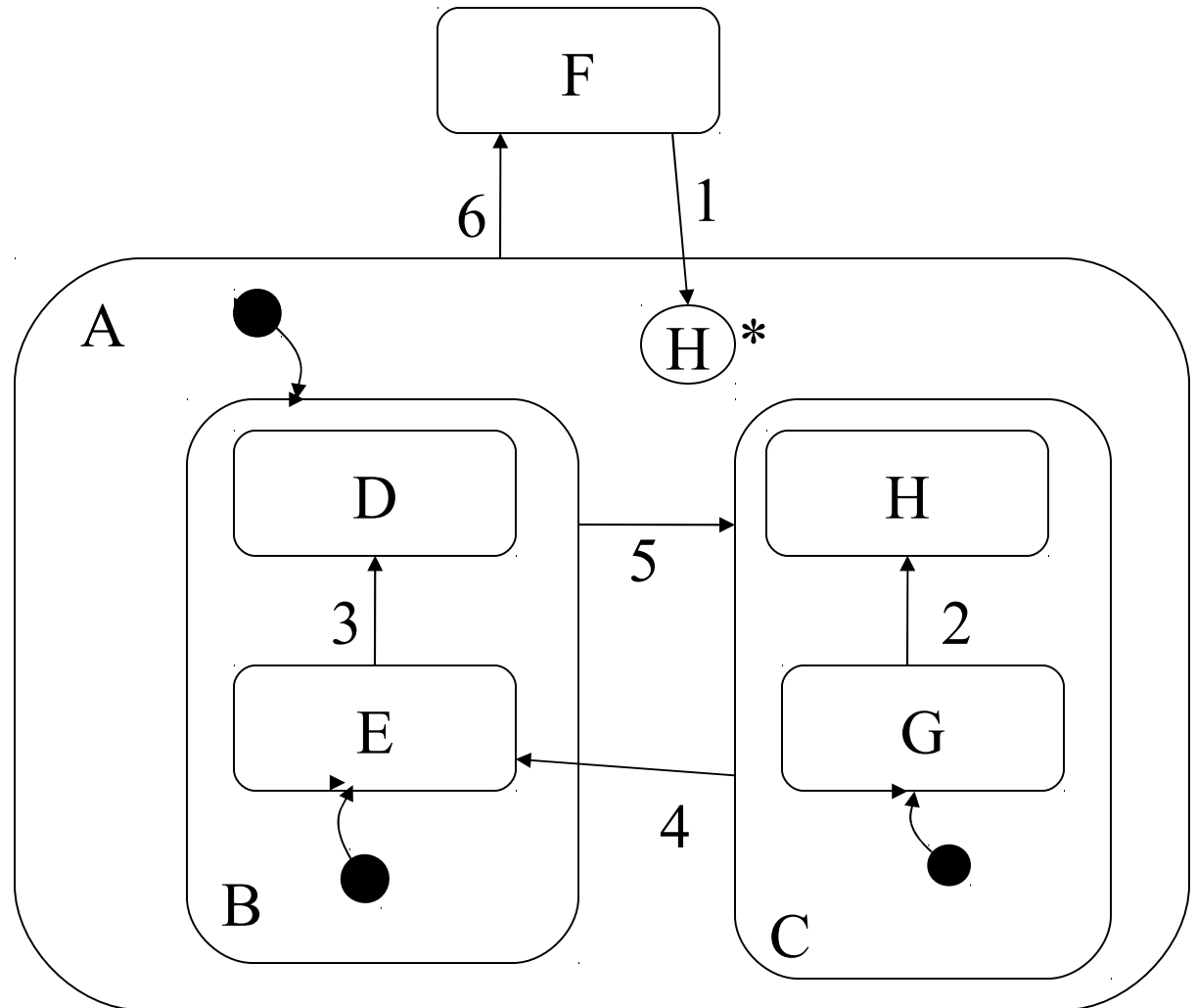


# The history mechanism again



# The history mechanism asterisk

An asterisk can be attached to the history symbol to indicate that the history of the system should be applied all the way down to the lowest level in the state hierarchy.



# Concurrency

- A major problem with STDs is the rapid growth of states to describe the simplest systems.
- The statechart approach overcomes this with the use of concurrency.
- As an example imagine the buttons you can use on the toolbar of Word - here let's model the **bold**, *italics* and underline in a statechart.

# Concurrency and state charts

- In terms of a state diagram for controlling these buttons, the format of the text selected in the application will determine the initial state of the system. For example, Bold off, Italics on, underlined off.
- The possible combination of these states are shown in the next slide.

# Concurrency Example

Bold off  
Italics off  
Underline off

Bold on  
Italics off  
Underline off

Bold off  
Italics on  
Underline off

Bold on  
Italics on  
Underline off

Bold off  
Italics off  
Underline on

Bold on  
Italics off  
Underline on

Bold off  
Italics on  
Underline on

Bold on  
Italics on  
Underline on

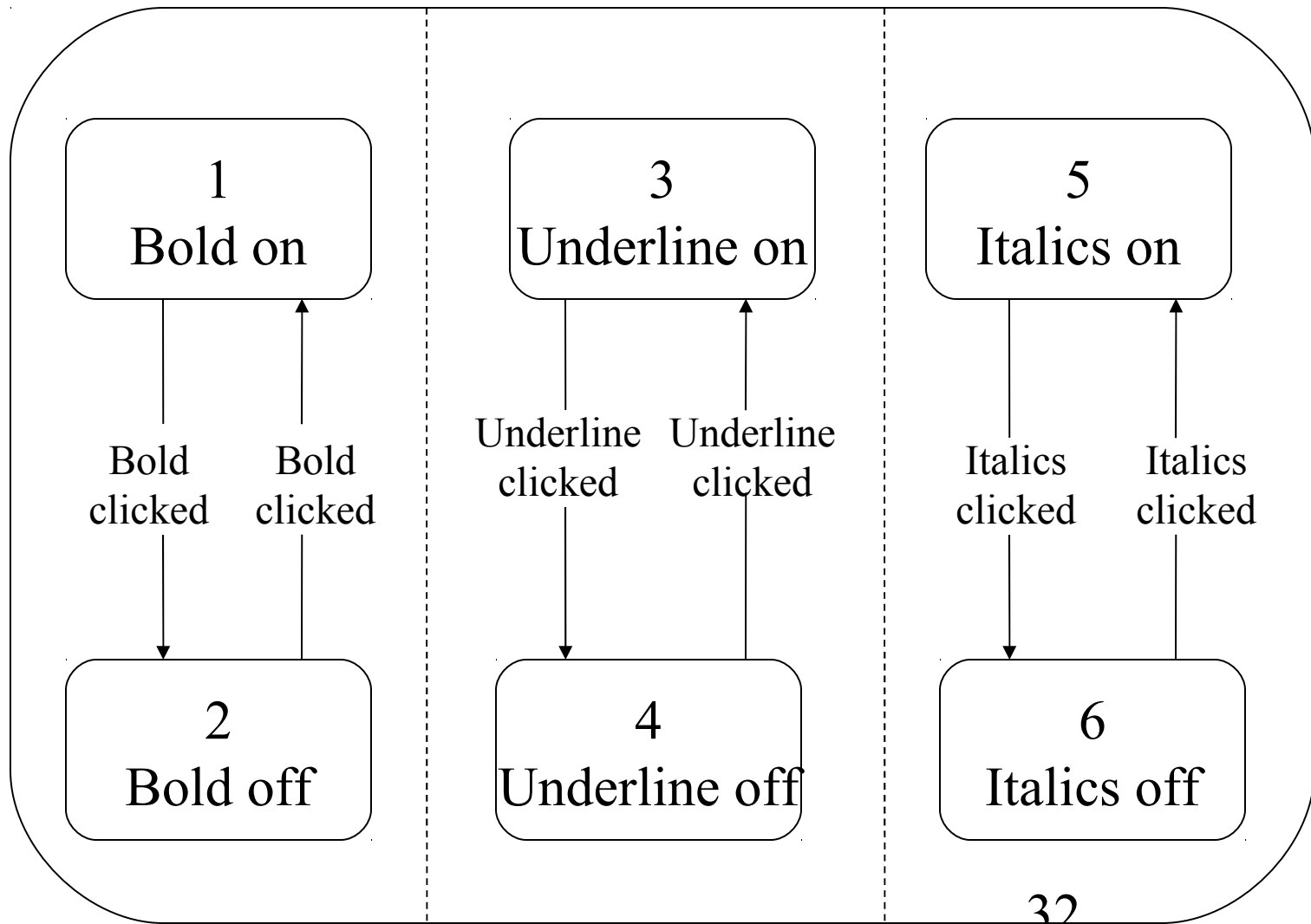
# Concurrency in use

- When a user clicks the **B**, *I* and U, the system will change states and cause the format of the highlighted text to change. A STD would be too chaotic to draw (each state has 3 possible exits because a user could click on any one of the buttons, and with 8 states, there are 24 transitions).
- Adding more functionality would make a STD unmanageable.

# Concurrency and independence

- If we examine the **B**, *I* and U buttons it is clear that they work independently of each other. The user can click any of these buttons and each will not affect the other. So there is no real reason to control them with the same STD.
- The next slide shows the statechart for these buttons.

# Simplicity of Concurrency

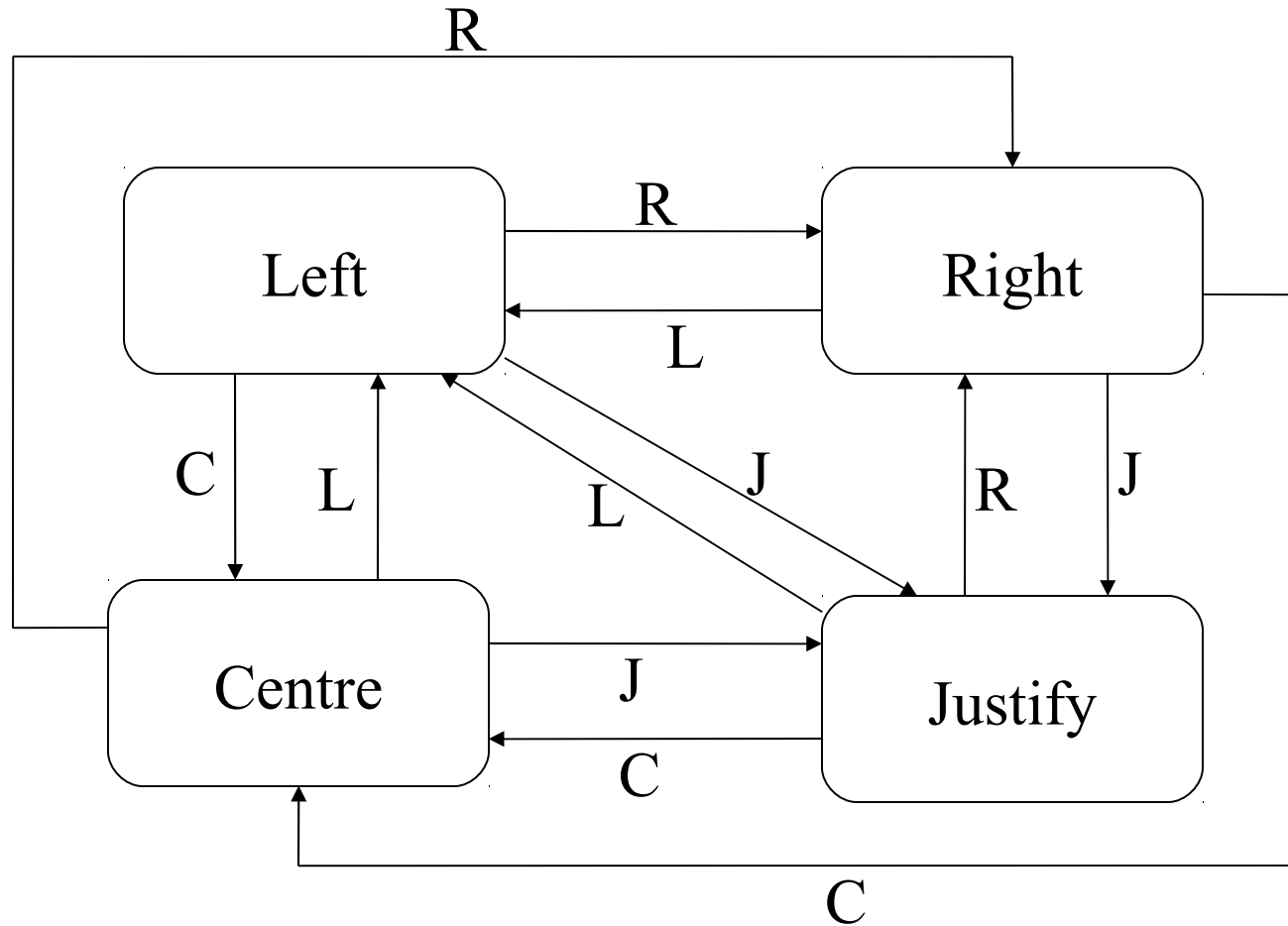




# A different example of Concurrency

- The left, right, centre and justify buttons do not operate independently of each other. They are in fact a group of radio buttons because at any one moment only one of them can be pushed down. If the user pushes another button, the previous button pops up.
- This is modelled on the next slide.

# Example modelled

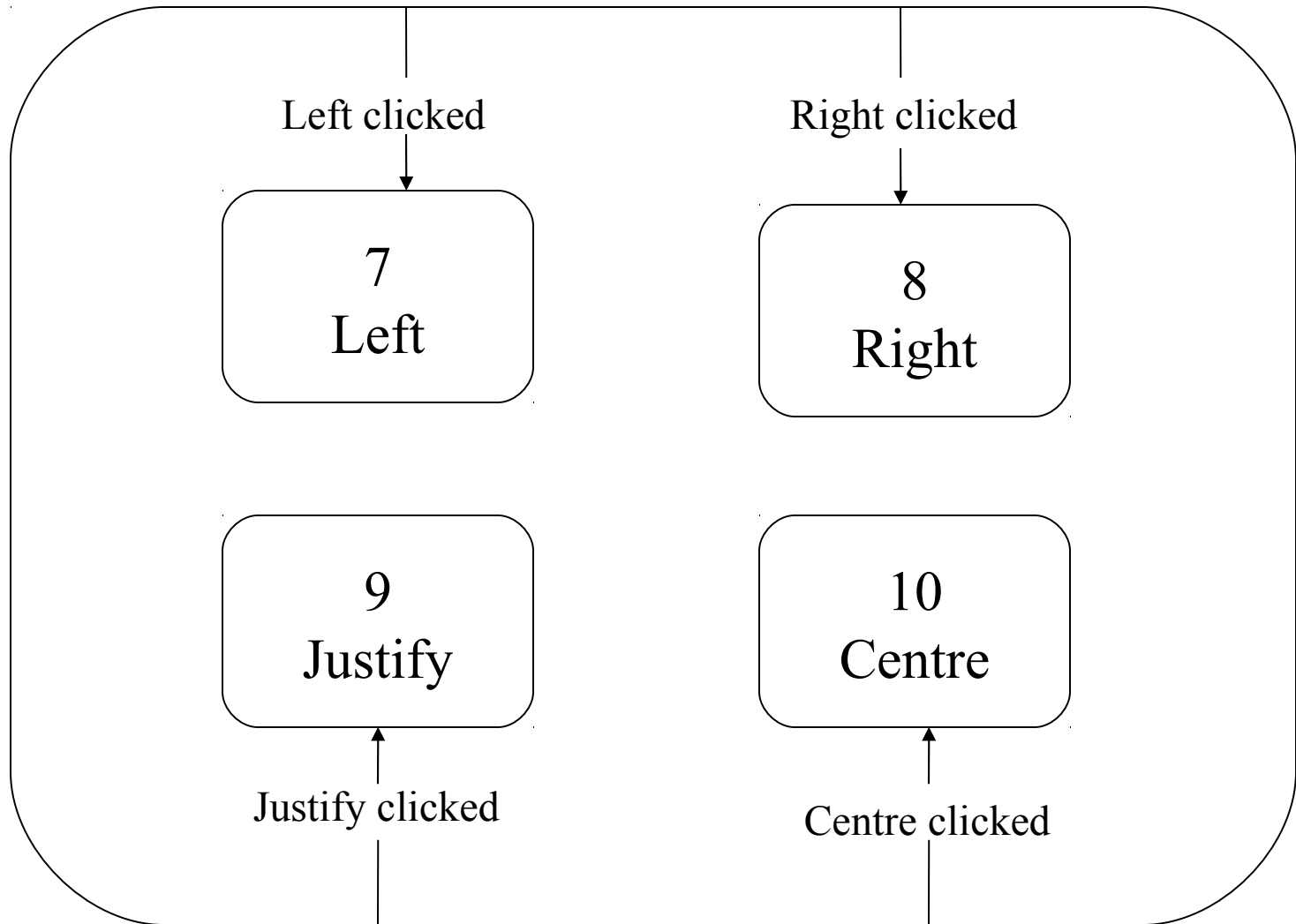


Modelling a radio button example -  
how to make a messy statechart

# Example and Concurrency

- The current state is defined by the last button the user clicked. There are a great deal of arrows but these can be reduced in a statechart because:
  - every state is connected to every other state
- So the modified statechart looks like the one in the next slide.

# Concurrency (9)

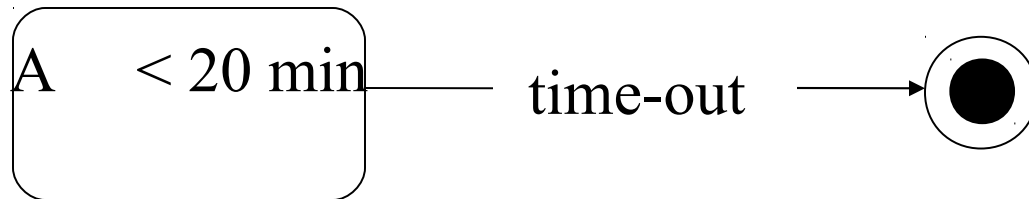


# Delays



A delay mechanism can be imposed on any state within a statechart. A delay on a state will prevent a user being able to perform events in that state for a specific period of time after entering that state. On entry to state A clicking OK will have no effect for 10 seconds. After the statechart has been in state A for 10 seconds, if the user clicks OK then this will cause a transition to state B.

# Time-outs

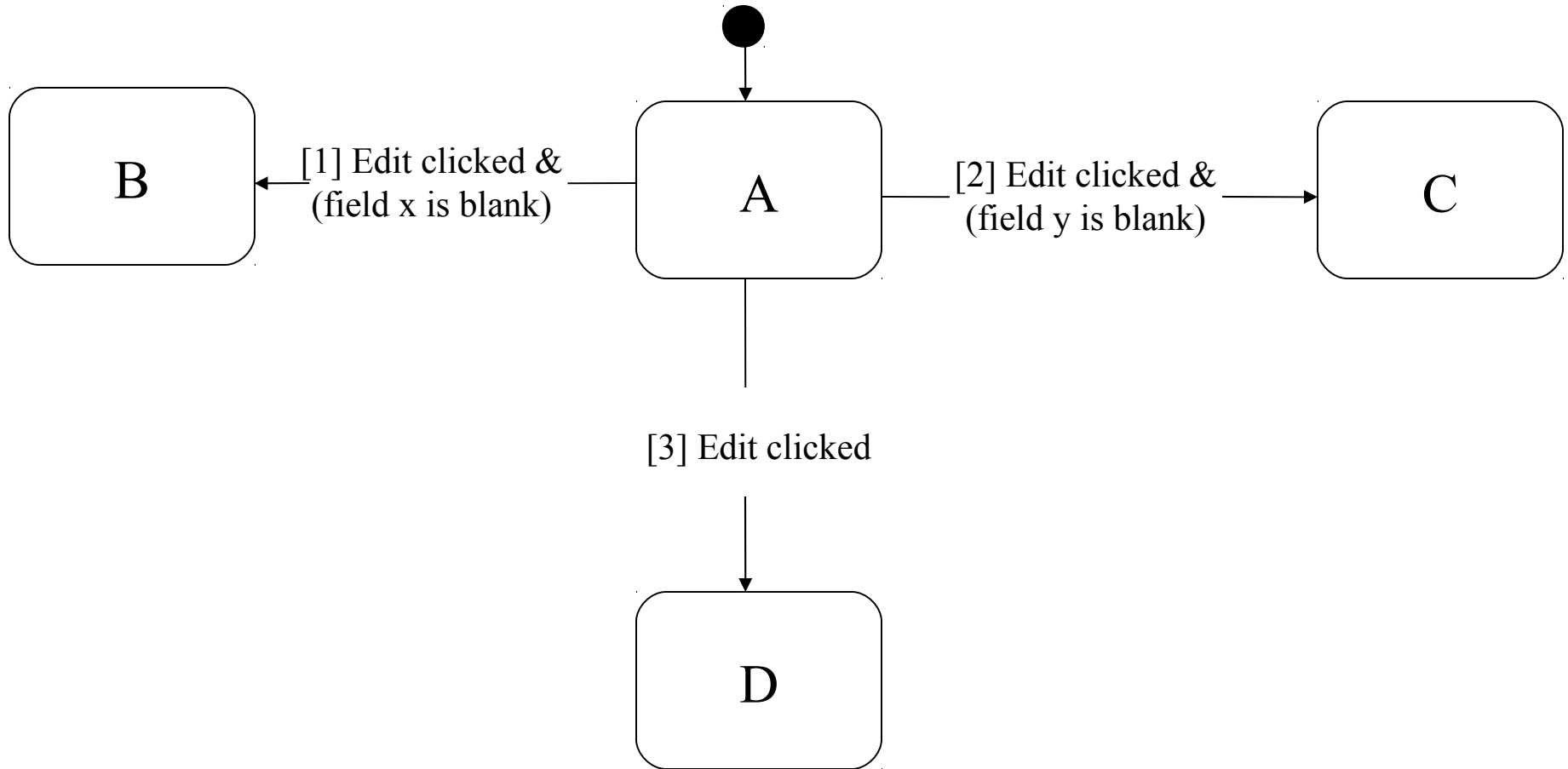


Time-outs can be very useful when specifying the behaviour of a user interface. For instance, suppose a user forgets to close down an application at the end of the day. To guard against this, the software can be designed to shut itself down automatically if it is not used for 20 minutes.

# Event priorities (1)

- When event arrows contain conditions, there is a possibility that more than one condition could evaluate to true. To avoid introducing non-deterministic behaviour, relative priorities can be assigned to events leaving the state (as seen in the next slide).
- In the example if fields x and y are both blank then Edit clicked could lead to states B, C or D. But because of the priorities, the transition to state B will occur.

# Event priorities (2)





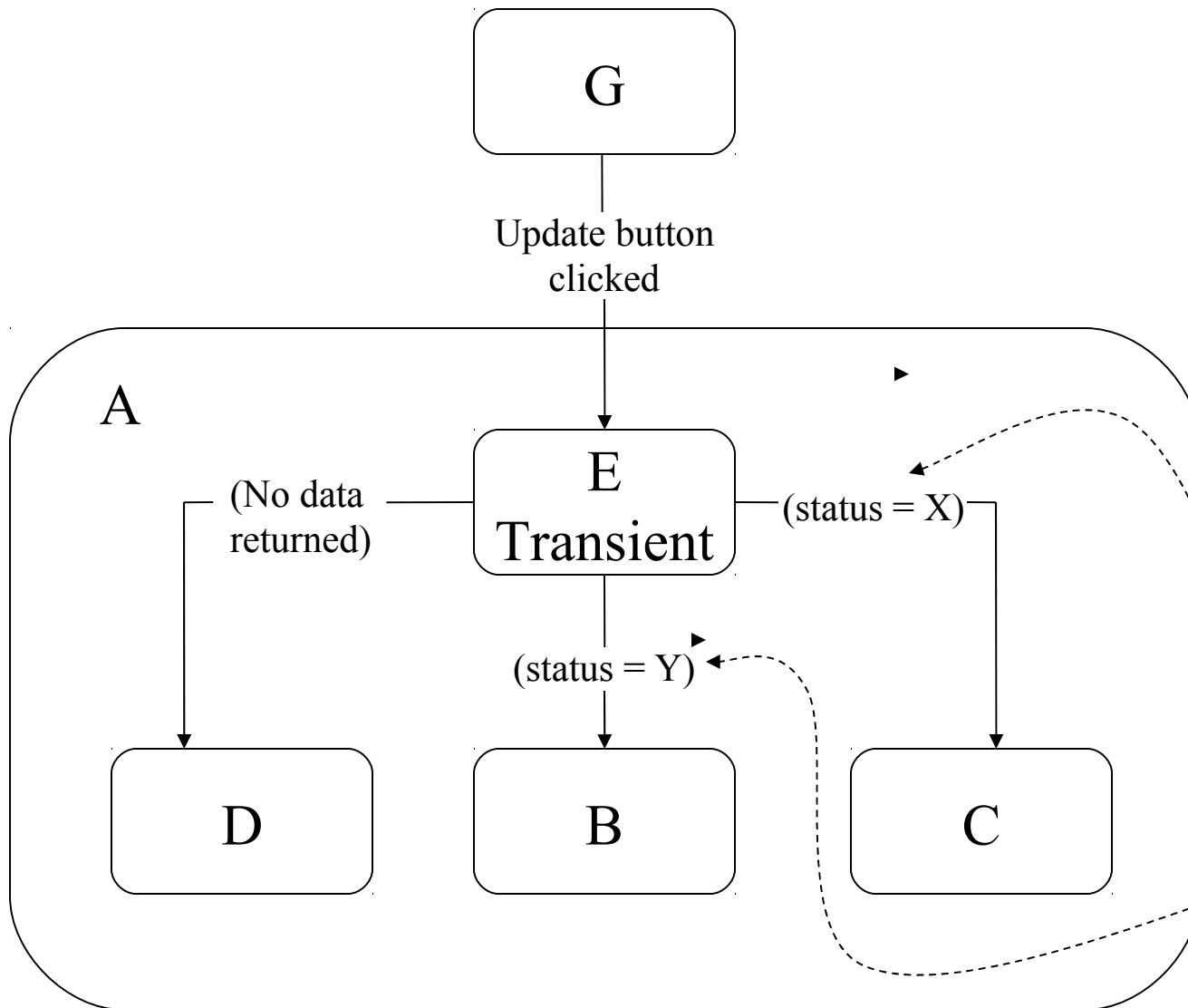
# Transient states (1)

- Transient states are useful for designing databases. In certain circumstances, a user event in itself may not determine what the next state is. For instance, suppose a user event triggers the following actions:
  - 1. Retrieve a set of data from a database
  - 2. Put the retrieved data into a scrolling list in a particular screen.
  - 3. Highlight the first row in the list.
- Suppose the data retrieved from the database contains some kind of status value. When a user selects a row in a scrolling list, the status value of that row will determine how the user will be able to interact with other screen items.

# Transient states (2)

- The difficulty with modelling following behaviour lies with the event that first retrieves the data from the database.
  - Under normal circumstances, the current state and the user event would determine the next state.
  - However, in this case, the next state will be determined by the status of the first row in the list item *after* the data is retrieved.
  - In other words, at the point when the user performs the event that causes the 3 actions, the next state cannot be determined until the first 2 actions have been performed.
  - To get round this we can use a transient state.

# Transient states (3)



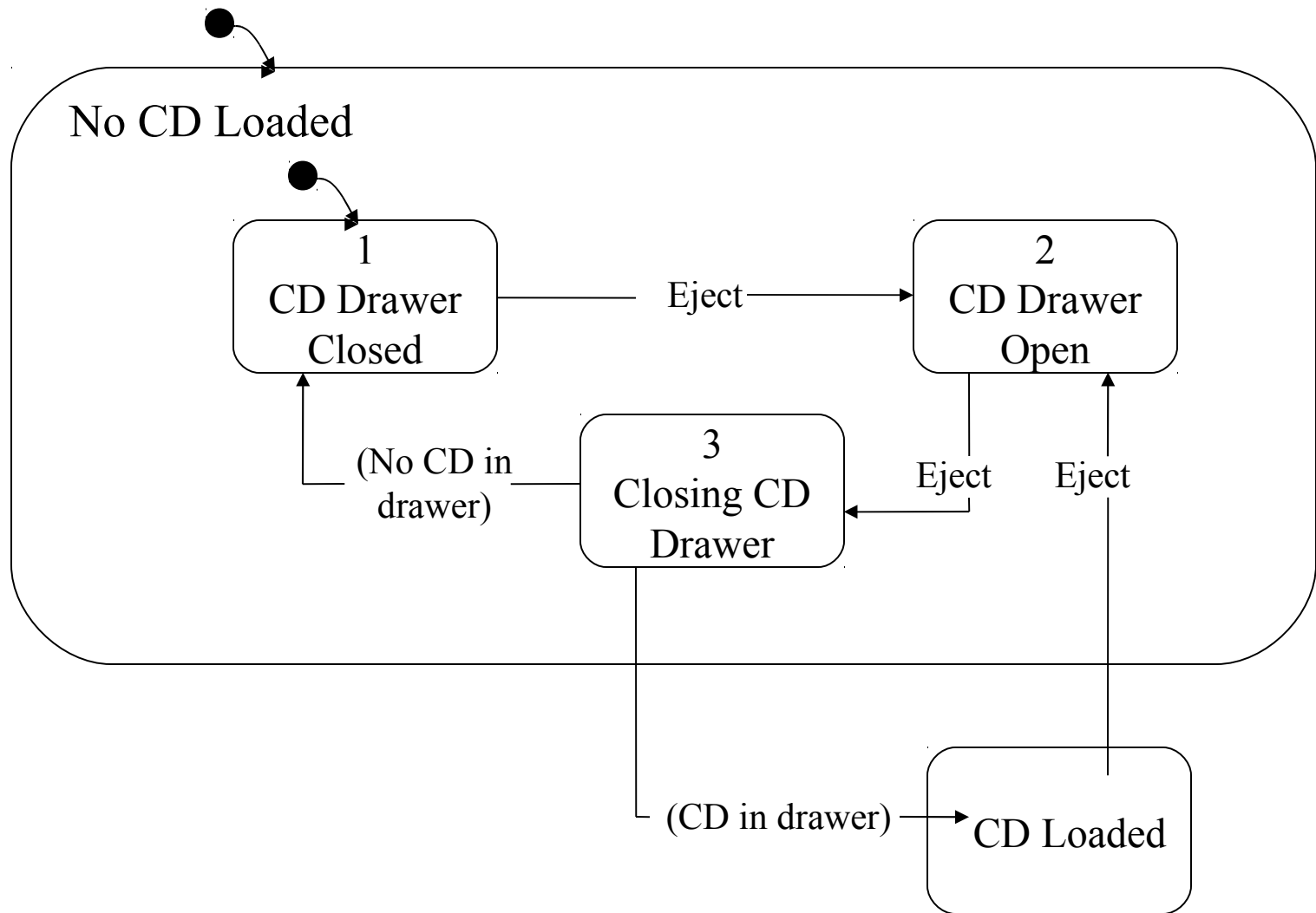
While the data is being retrieved from the database, the statechart will wait in state E. When the database query is complete and the status of the first row is known, the transition can be made to the appropriate state.

Conditions,  
not events

# Event-action tables (1)

- May describe all events and actions in a table.
  - This allows the developer to give more information than can be described in the statechart alone.
  - A system can be broken into smaller parts and precisely what is modelled in those parts can be shown in the event-action table.
  - When modelling different screens it is important to describe the transitions from screen to screen in the event-action tables.

# Event-action tables (2)



Example CD application statechart 45

# Event-action tables (3)

Current state	Event	Actions	Next state
start	Application started	cd_player.close_drawer;	1
1	Eject button clicked	cd_player.open_drawer;	2
2	Eject button clicked	cd_player.close_drawer;	3
3	(cd_player.cd_loaded = false)	none	1
3	(cd_player.cd_loaded = true)	none	CD loaded

Note that there are no actions for the last 2 entries. This is because state 3 is a transient state. It moves to the next state dependent upon the conditions stated.