Reactive Systems

A finite state machine is similar to a finite state recognizer, but may include output as well as input.

Finite State Machine

Consider a machine $A = (S, O, Q, q_{\text{start}}, T)$ where

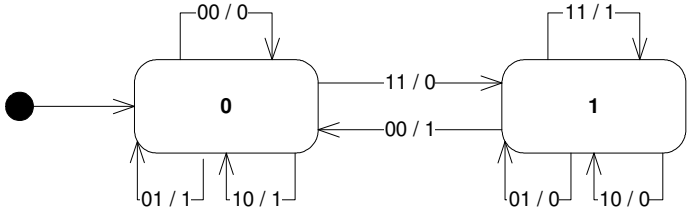
- S is an input alphabet
- O is an output alphabet
- Q is a finite set of states
- q_{start} is the initial state
- T is a set of transitions from $Q \times \big(S^1 \cup \{\epsilon\}\big) \times \big(O^1 \cup \{\epsilon\}\big) \times Q$

We assume $S \cap O = \emptyset$.
Such a machine defines a set of strings $L(A) \subseteq (S \cup O)^*$:
If there is a path from q_{start} to some state in Q and

- s is the catenation of input and output labels along that path,
- then $s \in L(A)$.

Application to digital circuits

If the machine is free of ϵs on both inputs or outputs, then each input is followed by a corresponding output. This gives a good model for synchronous digital systems.
Example: Let $S = \{00, 01, 10, 11\}$ and $O = \{0, 1\}$



Now we have a serial adder. If we feed in two numbers, least-significant bit first, the machine produces the sum. E.g. we have

$$\lfloor 10\,1\;00\,0\;11\,0\;01\,0\;10\,0\;00\,1\;11\,0\;01\,0\;10\,0\;00\,1$$

corresponding to the sum.

$$\begin{array}{r} 0101010101 \\ 0011001100 \\ \hline 1000100001 \end{array}$$

System modelling and StateCharts

Reactive systems

Reactive systems are systems that must react to events.

- A Calculator must react to the keypresses
- A Microwave oven must react to keypresses and also to the passage of time.
- An internet Router must react to the arrival of packets
- A synchronous hardware circuit must react to the clock ticks.
- Most systems can be viewed as reactive.

We can specify and/or model a reactive system using state machines.

StateCharts

StateCharts is a diagrammatic language for modelling finite state systems.

There are various flavours of StateCharts.

I'll be using UML *StateCharts.*

Transitions

Our terminal alphabet now consist of a set of *events*.

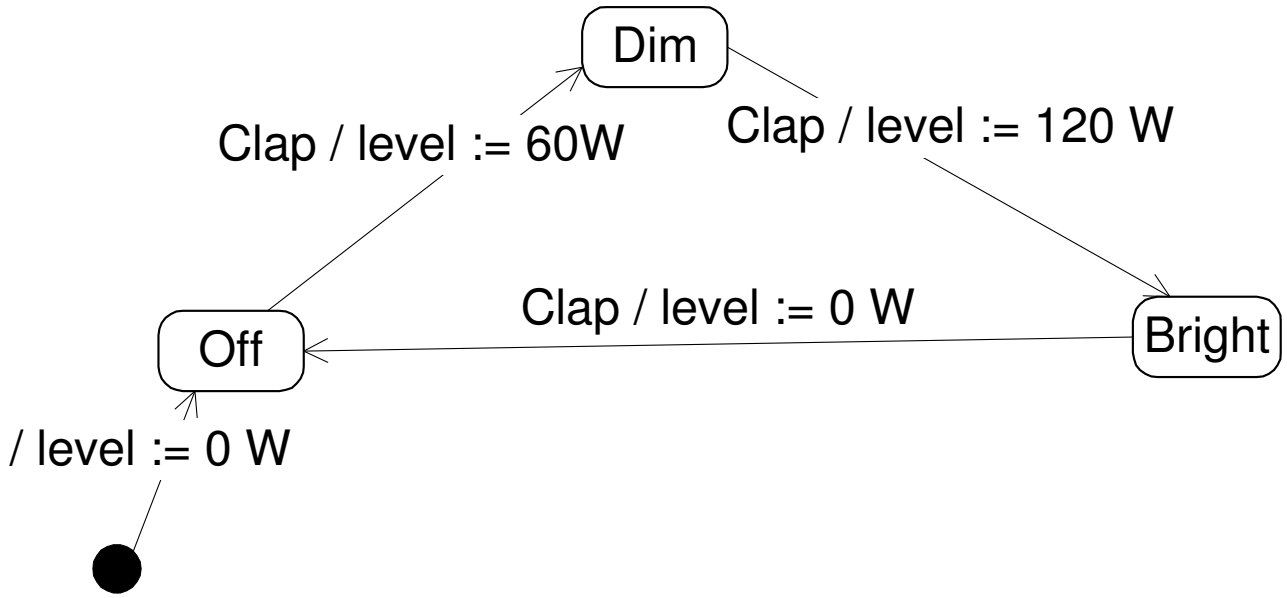
Each transition from state to state is labelled

trigger [condition] / reaction

where

- the *trigger* is an event that triggers the transition
- the *condition* is a boolean expression
- the *reaction* is either
  - ∗ an event that is generated,
  - ∗ or a change to the system's state variables

Example: A Clap-light.



Data Dictionary:

Entity	**Kind**	**Description**
Clap	Event	Occurs when a single clapping sound is detected.
level	System variable	The amount of power sent to the lights.

Here there are 3 states. One event: **Clap** and reactions that change the system's state variable level.

If the system is in state **Off** and a **Clap** event occurs, then

- *immediately* and *simultaneously*
- level is set to 60W and the system state changes of **Dim**.

Transitions in detail

Transitions are labeled as:

$$(trigger)^{?}\ ([condition])^{?}(\ /\ reaction\ )^{?}$$

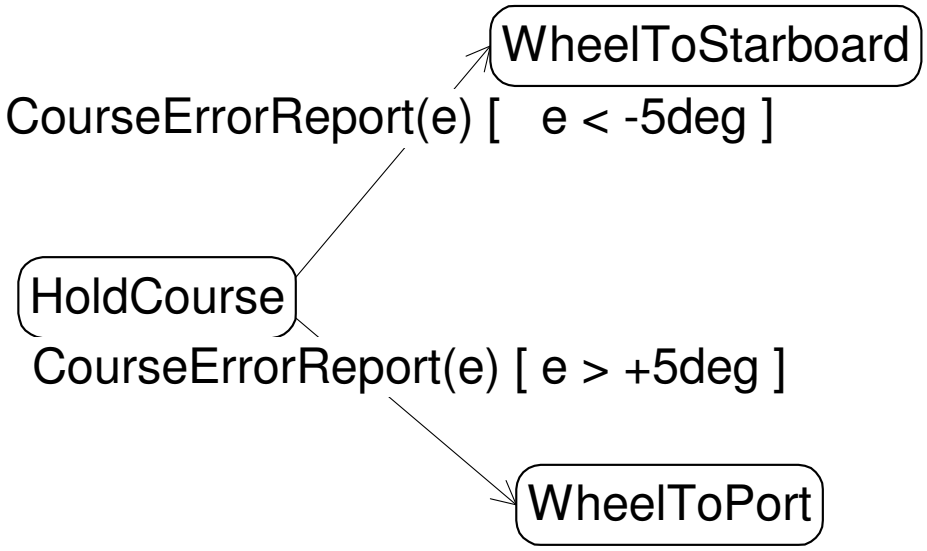
- If the *trigger* event is omitted,
  - the transition is taken as soon as the condition is true.
- If the *condition* part is omitted,
  - the condition is *true*
- If the *reaction* is omitted
  - there is no reaction, only a state change.

Conditions

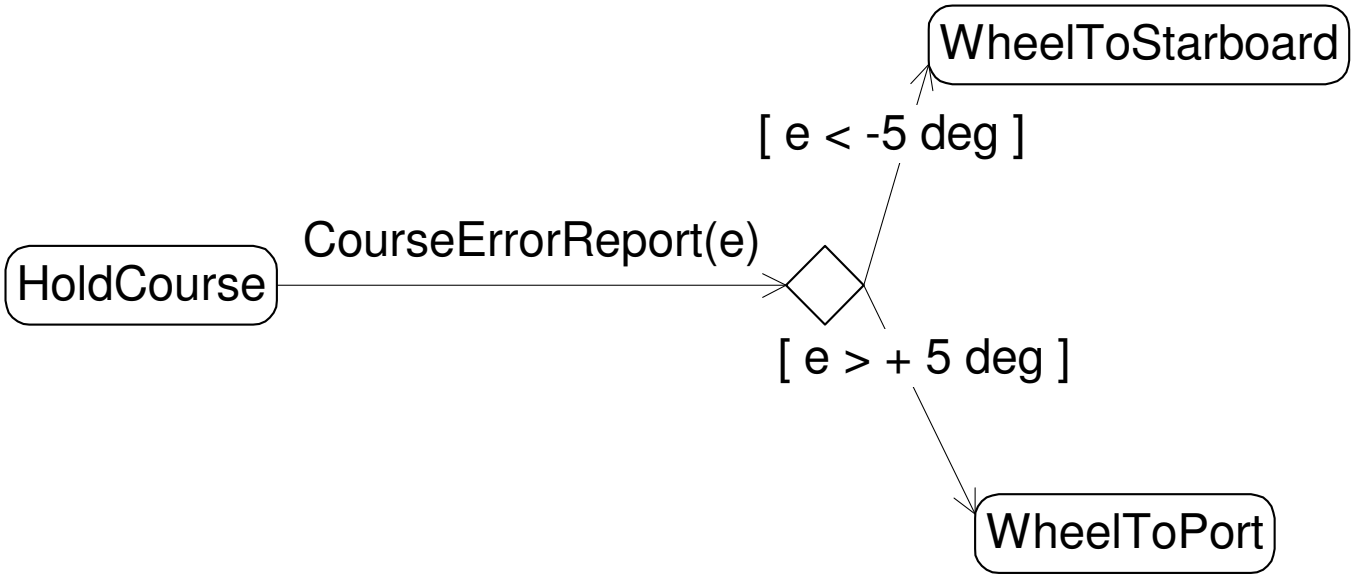
Conditions can be used to inhibit transitions:

Notice the event is parameterized.

Conditions can also be used to choose between transitions:



We could also illustrate this as: (The diamond is not a state.)



Time

It is important to realize:

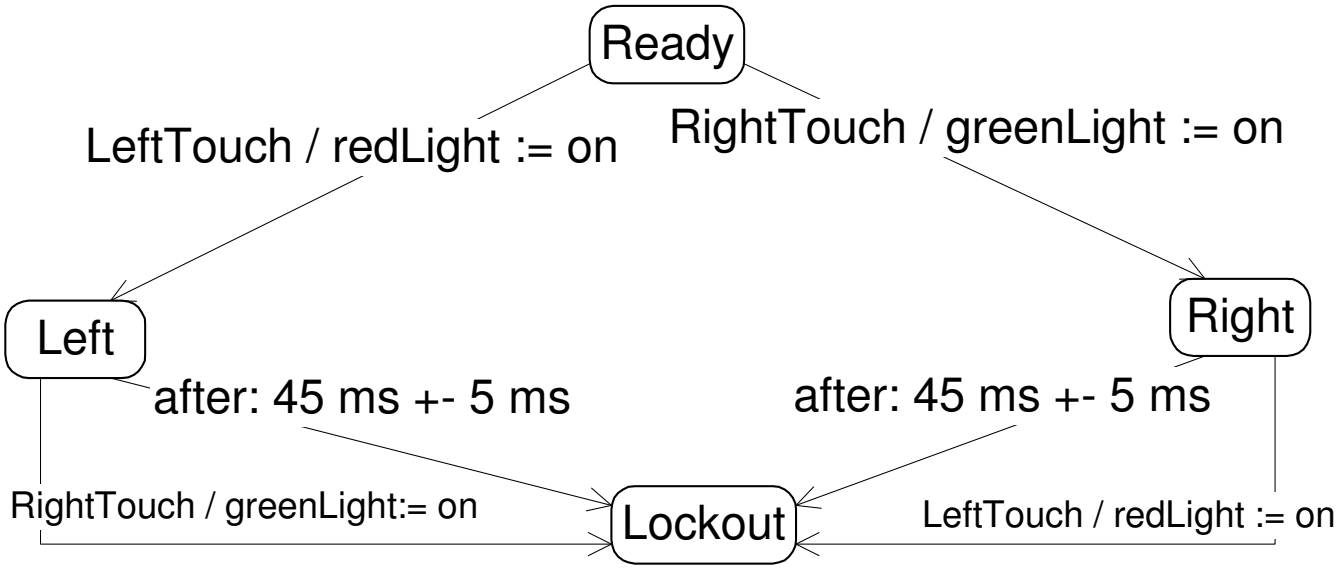
Time passes in the states.

Thus transitions are essentially instantaneous.

What if you want a delay?

For real-time systems, the passage of time is an important kind of event.

- after: 1 second is an event that happens 1 second after the source state was entered.
- when(11:59 AM) is an event that takes place at 11:59 AM.

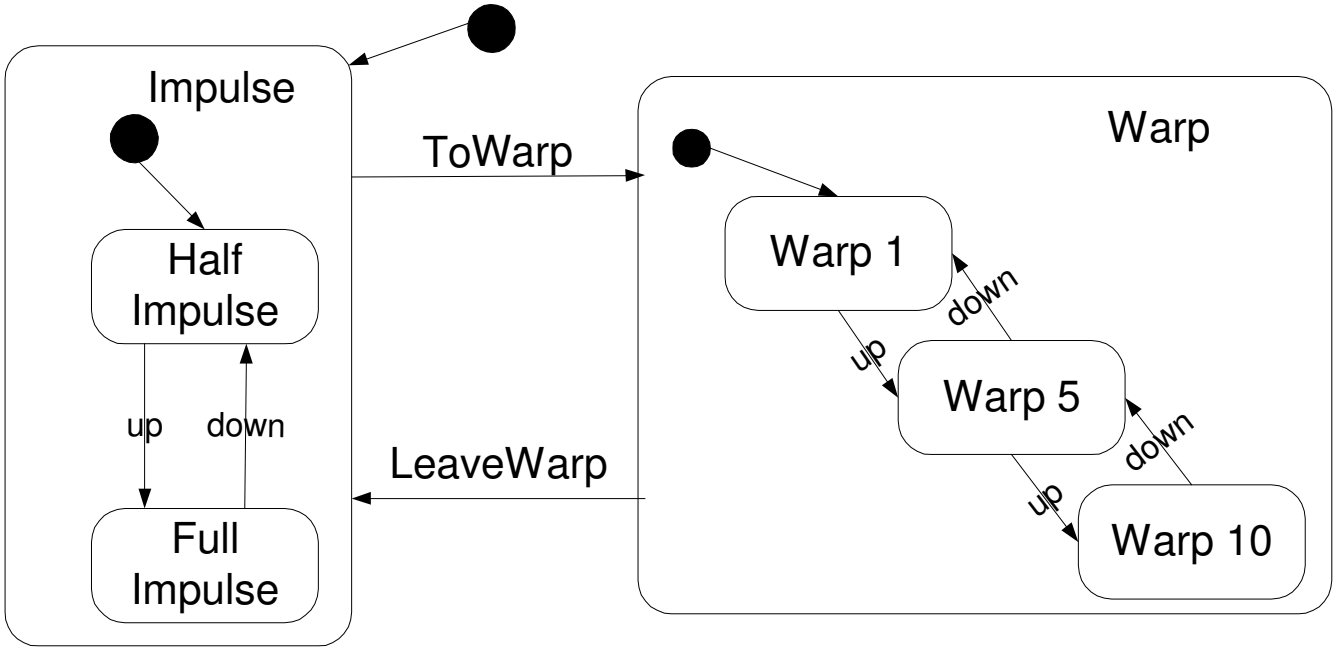


Hierarchy

Time passes within states.

During that time the system need not be idle.

We can divide states into substates



When this system is in state Impulse it will be in exactly one of the substates Half Impulse or Full Impulse.

When it is in state Warp it will be in exactly one of its 3 substates.

At system start the system is in both states Impulse and Half Impulse.

If the system is in state Impulse and a ToWarp event happens, the system will transition to both states Warp and Warp 1.

States like Impulse and Warp are called “or” states, since when the system is in an “or” state, it is also in *exactly one* of the “or” state’s substates.

States with no substates are called “basic” states.

The term “state” is being abused here since a system should really be in exactly one state at a time. The set of StateChart states the system is in constitutes its “true” state.

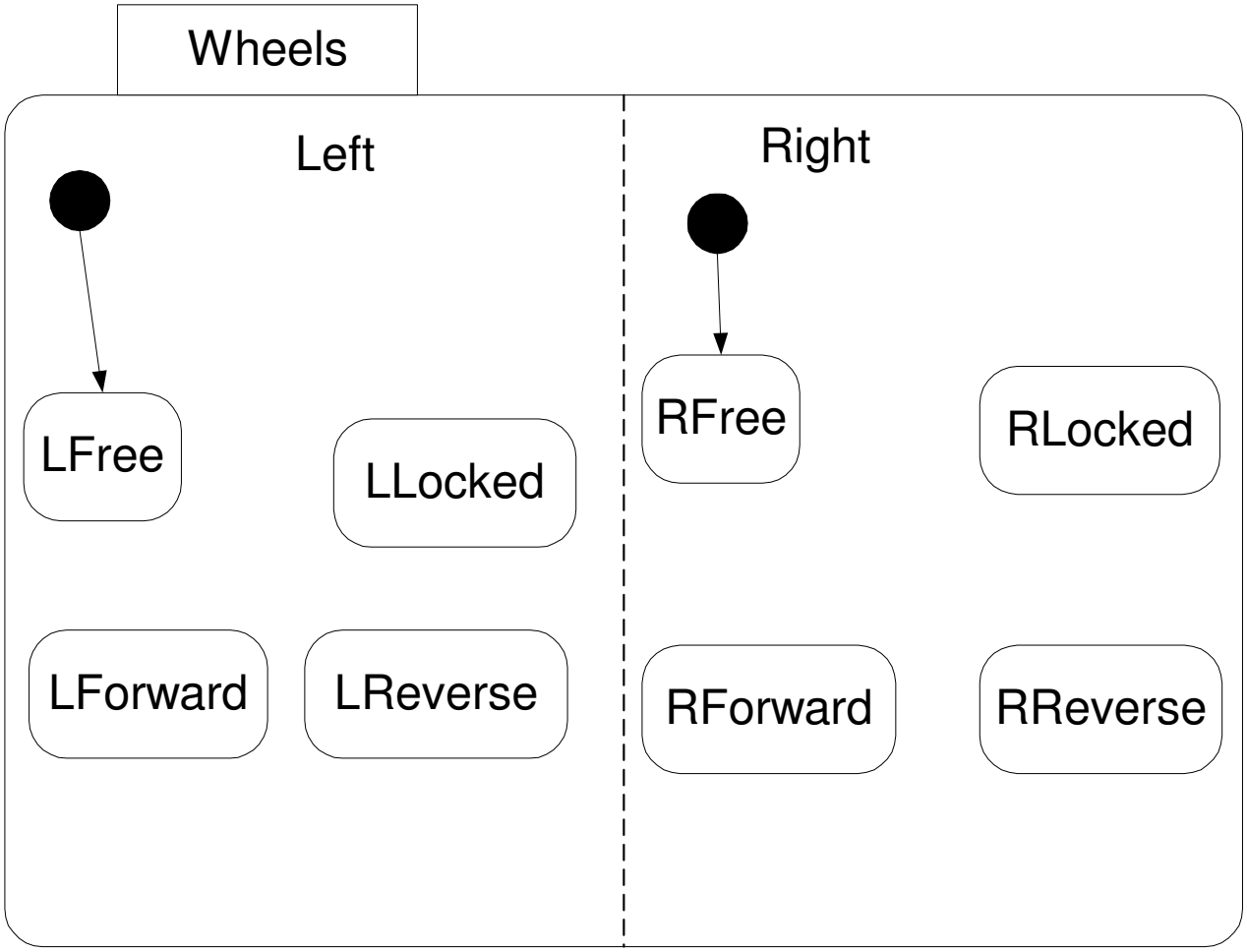
Some people prefer to call "or states" "modes".

Concurrency.

When the system is in an "and" state, it is also in all of the "and" state's substates.

The substates of an "and" state are always "or" states.

An example (transitions omitted)



Wheels is an "and" state.

Its substates are Left and Right which are both "or" states.

The system could be in states Wheels, Left, Right, LReverse, and RLocked all at the same time.

The substates of an "and" state act as concurrent state machines.

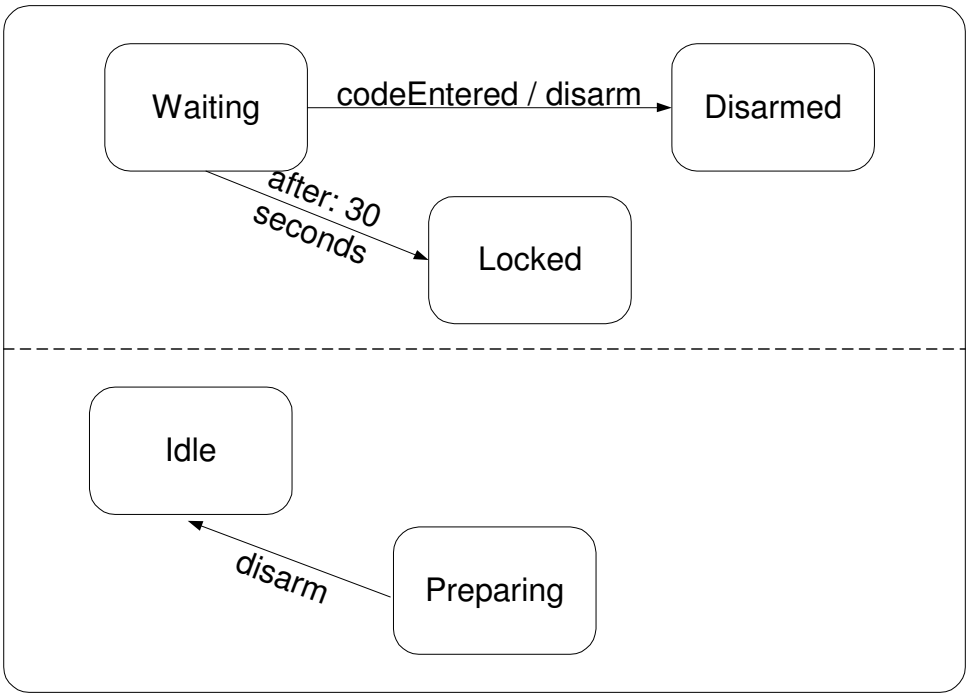
Communication

You can use events to coordinate actions of concurrent state machines.

If a transition has an event as its reaction,

- that event can trigger a transition out of any state the system is in.

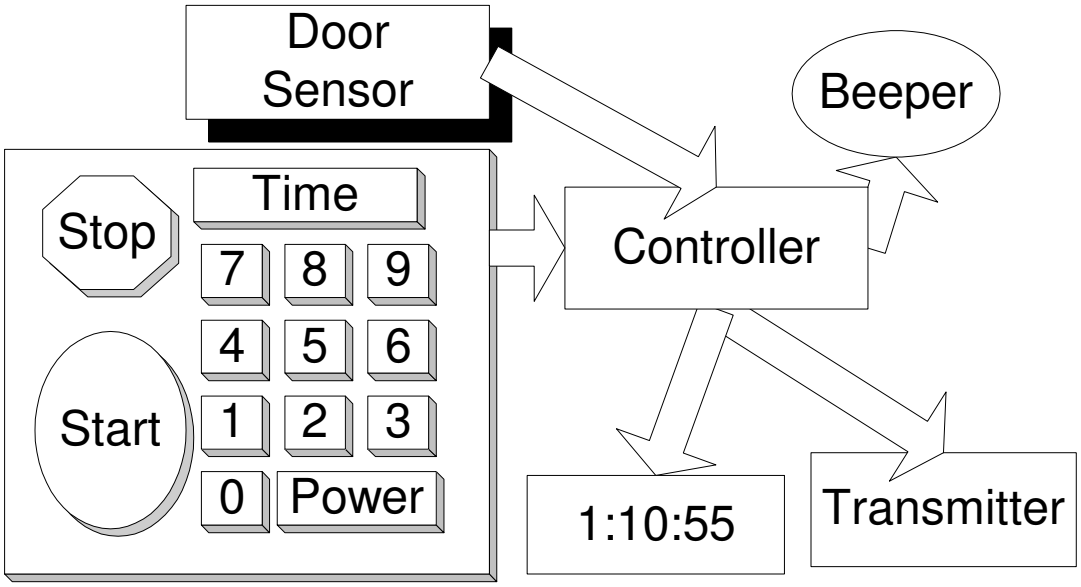
Example:



If the system is in both Waiting and Preparing then a codeEntered event causes a disarm event and the new states will include Disarmed and Idle.

System modelling and StateCharts

A *Microwave oven* Example



Inputs

Entity	Type	Description
timeButton	Event	The time button is pressed
powerButton	Event	The power button is pressed
startButton	Event	The start button is pressed
stopButton	Event	The stop button is pressed
digitButton(d:0..9)	Event	A button 0 to 9 is pressed
door	Variable	Can be in states open or closed.

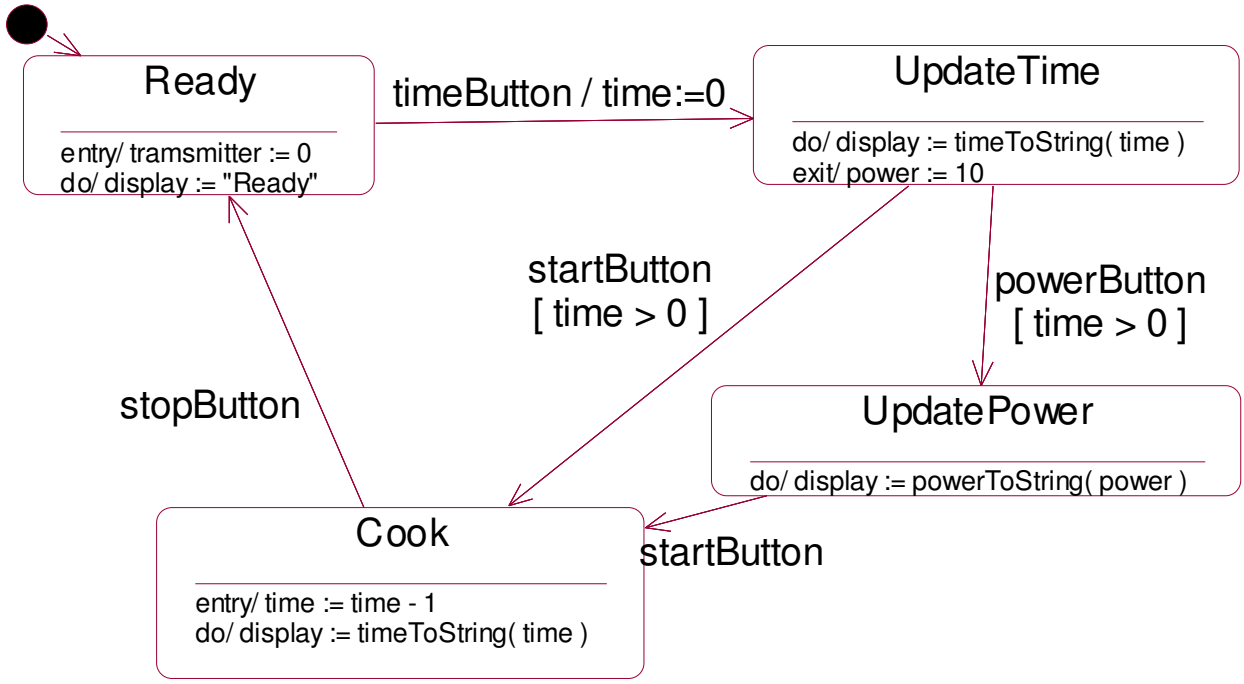
Outputs

Entity	Type	Description
transmitter	Variable	The power level of the transmitter {0,1,..,10}
display	Variable	An alpha-numeric display with 6 characters.
beep	Event	Initiate a beeping sound

Local entities

Entity	Type	Description
time	Variable	A natual number. In units of seconds
power	Variable	A natural number {1,..,10}

Overall behavior

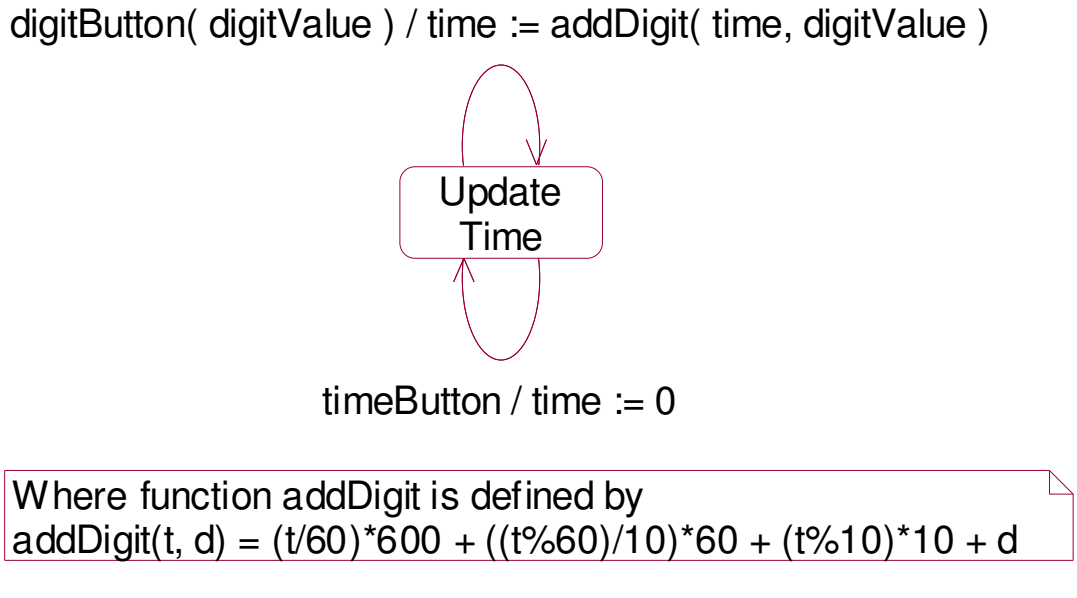


Note

- "entry" actions are performed whenever the state is entered.
- "do" actions are performed continuously when the state is active
- "exit" actions are performed whenever the state is executed.

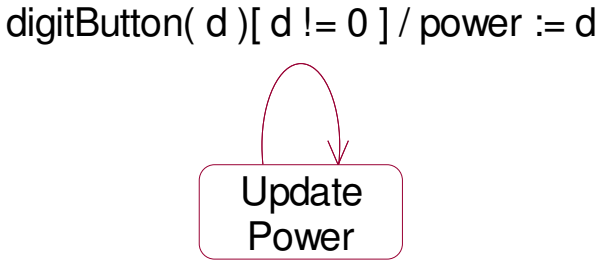
A closer look at the UpdateTime state

We add more detail by adding a transition for digitButton events in the UpdateTime state.

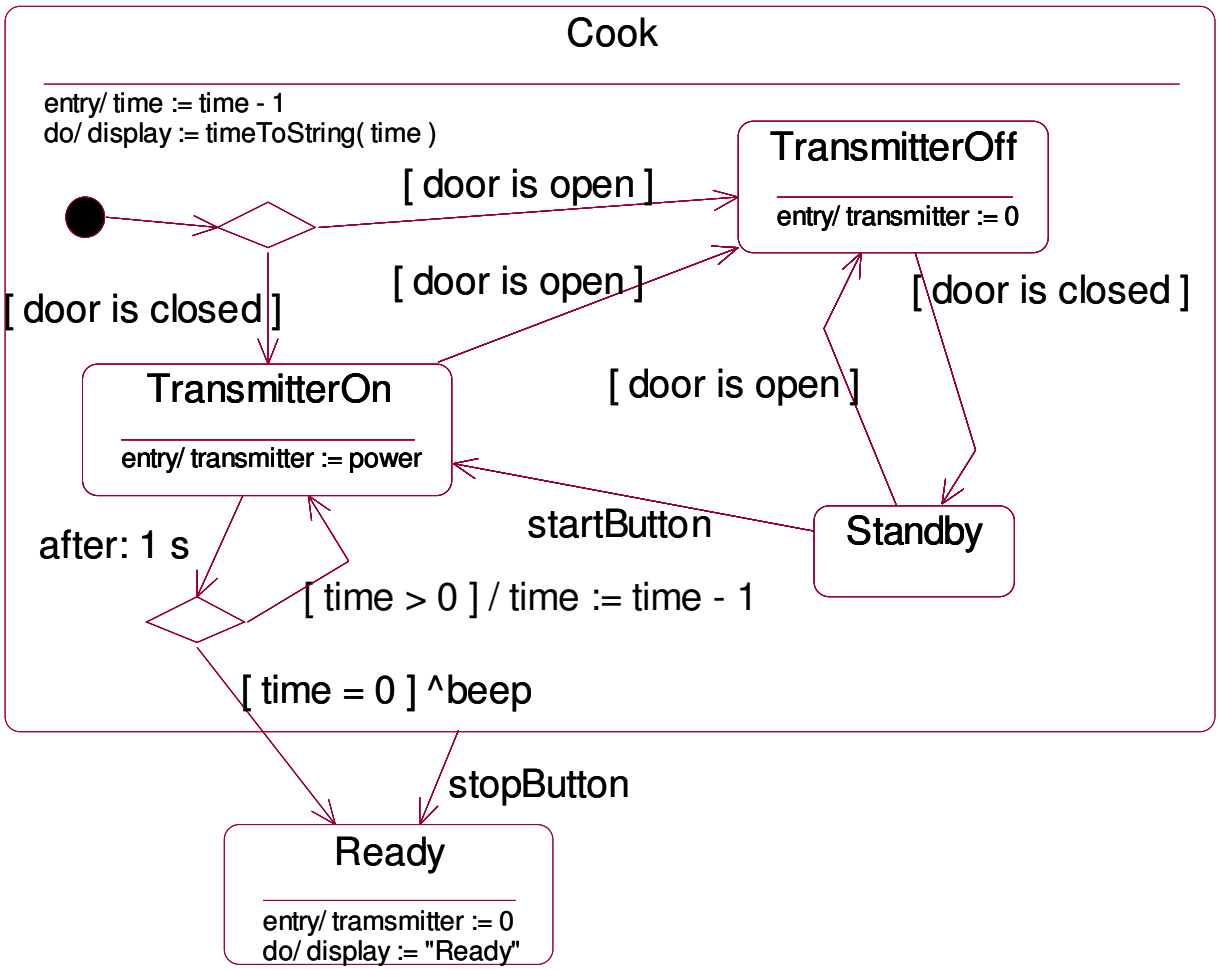


The UpdatePower

DigitButton events in the UpdatePower state.



The Cook state



Note that ^beep means a beep event occurs as a reaction.

I claim that “the door is open implies transmitter = 0” is a global invariant.

Using StateCharts to model software classes

Reactive Systems and objects have a lot in common.

- Reactive systems react to events
- Classes react to method invocations.
- Reactive systems have states on which behaviour depends
- Classes have states on which behaviour depends.

Input Events:

- Method calls to this object

Input variables:

- Variables and objects known to the class

Output events

- Calls to objects known to the class

Output variables

- Variables and objects changeable by this object.

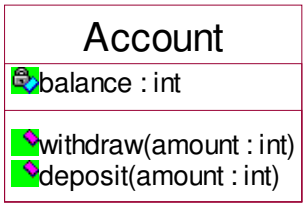
Relationship to other UML diagrams:

- Class diagrams describe the “static” relationships between classes.
- Sequence and collaboration diagrams show examples of behaviour.
- StateCharts describe behaviour.

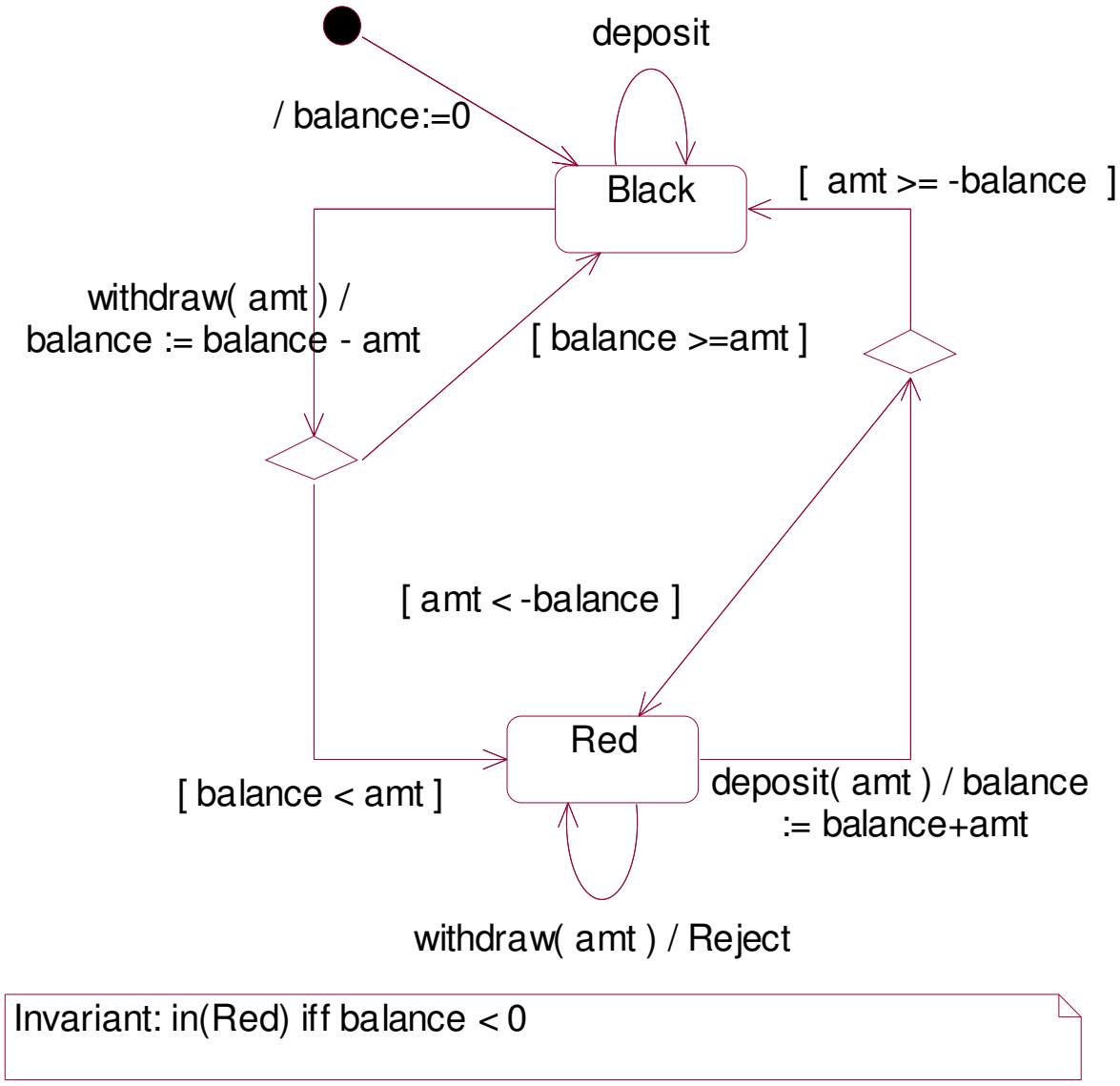
An Example

A savings account.

Account can not be debited once overdrawn:



The behaviour as a state chart:



In C++

```
class Account {
        public: Account() { balance = 0 ; }
        public: withdraw( int amt ) {
                if( balance >= 0 ) balance -= amt ;
                else throw Reject ; }
        public: deposit( int amt ) {
                balance += amt ; }
} ;
```

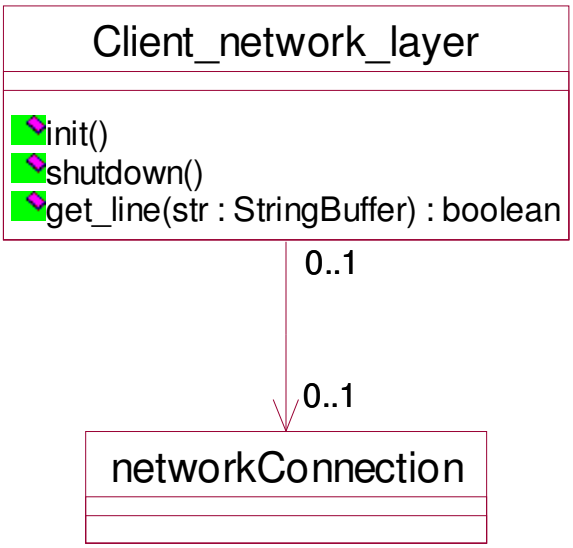
In this case the state diagram is not simpler than the implementation itself.

This is because the class is very simple.

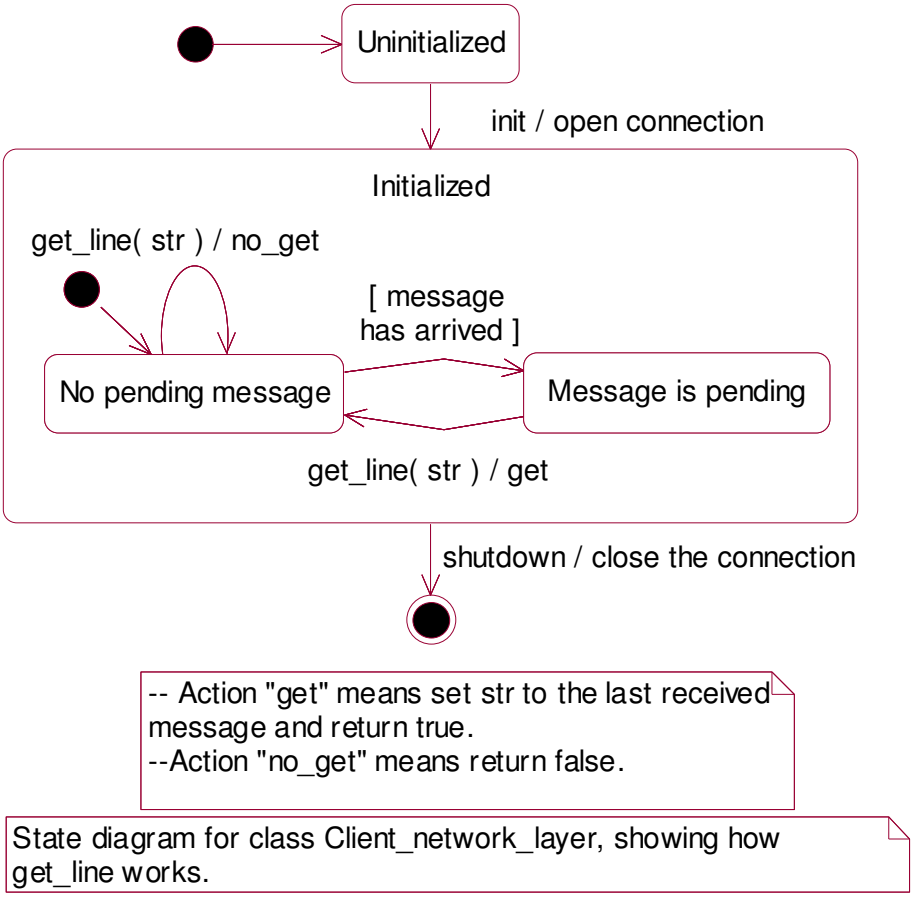
Another example:

This class represents a source of network messages.

- It must be initialized before being used and should be shutdown after use.
- It observes a network channel. Thus the network channel is an input variable.
- The client of this class may use it to get lines.



State diagram for Client_network_layer:



Notice that the state can change in response to changes in the observed variable.

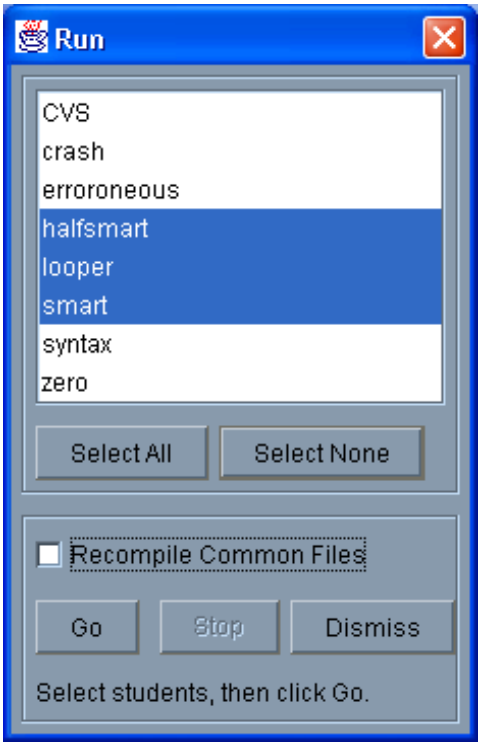
The actual implementation of this class is quite a bit more complex than the state machine, since the implementation must attempt to determine which state the object is in by querying the network connection.

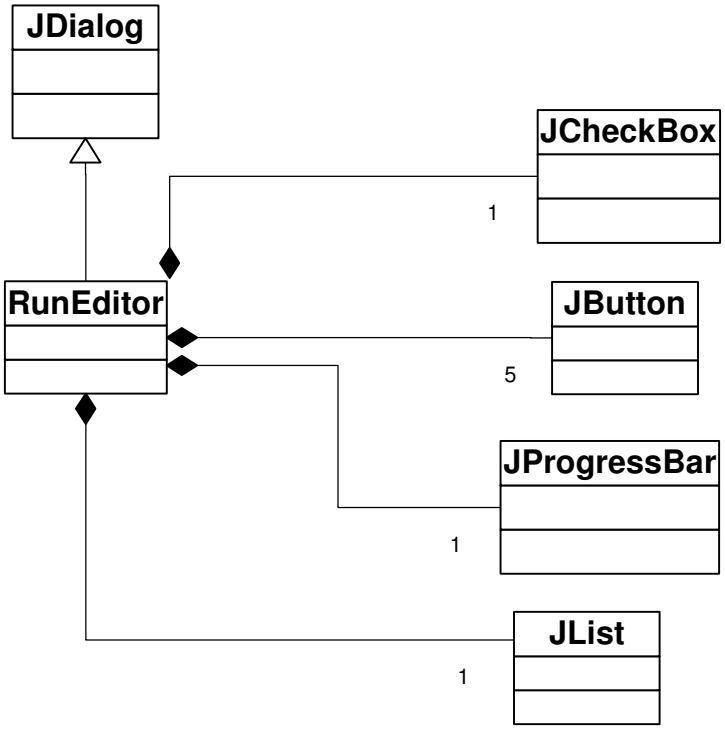
A Case Study — The RunEditor Dialog

The RunEditor is a GUI class that controls the running of a series of tests on student assignments. The user can pick one or more students and select go. The actual running of the tests is done by a separate worker thread, which takes some time to stop.

The dialog includes 5 buttons, a progress bar, and a list of students.

Buttons are: Go, Dismiss, Stop, Select-All, Select-None





StateChart

The class has 3 major states

- Waiting. The user: can select students from list, initiate run (if students are selected), dismiss the dialog.
- Running. The user: can initiate a stop.
- Stop. The user must wait until the stop is complete.

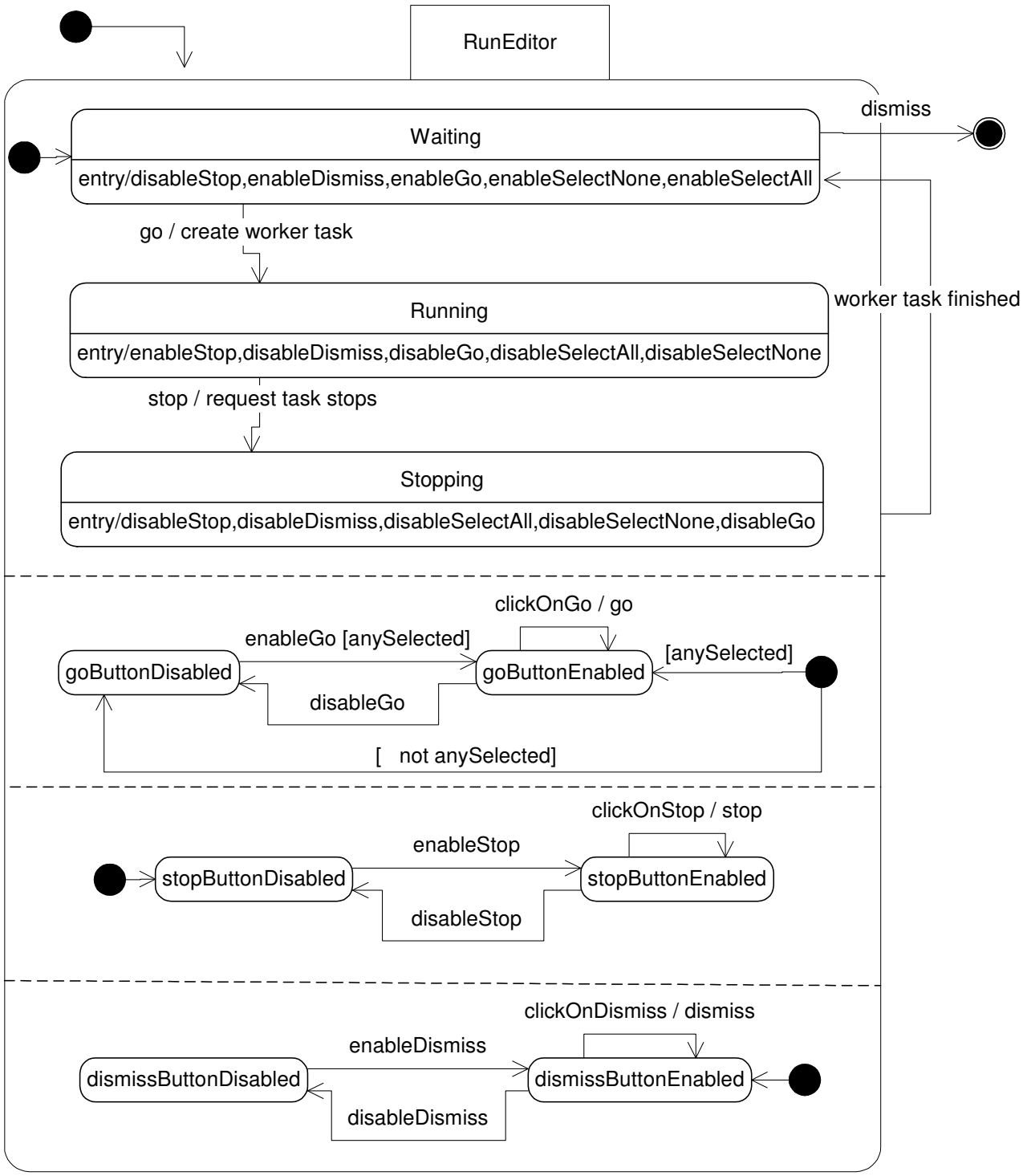
Note the StateChart on the next slide is a simplification as it omits the state of the selection buttons, the list, and the progress bar, as well as state inherited from JDialog.

Note that the state of the buttons is properly a part of the state of the RunEditor. This is because the relationship between these classes is one of Aggregation.

Data Dictionary:

External Entity	Kind	In/Out	Description
clickOnGo	event	in	User clicks on go button
clickOnStop	event	in	User clicks on stop button
clickOnDismiss	event	in	User clicks on dismiss button
create worker task	event	out	A worker thread is created.
request task stops	event	out	Request thread to stop.
worker task finished	event	in	A worker thread completes.

All other signals on the diagram are internal.


RunEditor
Waiting
entry/disableStop,enableDismiss,enableGo,enableSelectNone,enableSelectAll
dismiss
go / create worker task
Running
entry/enableStop,disableDismiss,disableGo,disableSelectAll,disableSelectNone
worker task finished
stop / request task stops
Stopping
entry/disableStop,disableDismiss,disableSelectAll,disableSelectNone,disableGo
clickOnGo / go
enableGo [anySelected]
goButtonDisabled
goButtonEnabled
[anySelected]
disableGo
[not anySelected]
clickOnStop / stop
enableStop
stopButtonDisabled
stopButtonEnabled
disableStop
clickOnDismiss / dismiss
enableDismiss
dismissButtonDisabled
dismissButtonEnabled
disableDismiss

Implementation

JButtons 'know' if the are enabled. We represent the state with fields

---

```
private final int WAITING = 0, RUNNING = 1, STOPPING
= 2 , INIT = 3 ;
private int state = INIT ;
JButton goButton = new JButton();
JButton dismissButton = new JButton();
JButton stopButton = new JButton();
... // and more
```

(The INIT 'state' is used only during construction)

The abstraction relation relates the states in the chart (left) to those in the code (right)

	(in Waiting	iff	state=WAITING)
and	(in Running	iff	state=RUNNING)
and	(in Stopping	iff	state=STOPPING)
and	(in goButtonEnabled	iff	goButton.isEnabled())
and	(in goButtonDisabled	iff	not goButton.isEnabled())
and	(in stopButtonEnabled	iff	stopButton.isEnabled())
and	(in stopButtonDisabled	iff	not stopButtonIsEnabled())
and	(in dismissButtonEnabled	iff	dismissButton.isEnabled())
and	(in dismissButtonDisabled	iff	not dismissButton.isEnabled())
	... and more		

Implementation continued

It is useful to centralize the state switching code in one subroutine

```
private void changeState( int newState ) {
        if( newState == WAITING ) {
            if( studentList.getModel().getSize() != 0 )
                listLabel.setText("Select students...");
            else listLabel.setText("No students ...") ;
            removeProgressBar() ;
            goButton.getModel().setEnabled(anySlctd() );
            dismissButton.getModel().setEnabled( true );
            stopButton.getModel().setEnabled( false );
            allButton.getModel().setEnabled( true );
            noneButton.getModel().setEnabled( true ); }
        else if( newState == RUNNING ) {
            ... code to enter Running state... }
        else if( newState == STOPPING ) {
            ...code to enter Stopping State... }
        state = newState ;
}
```