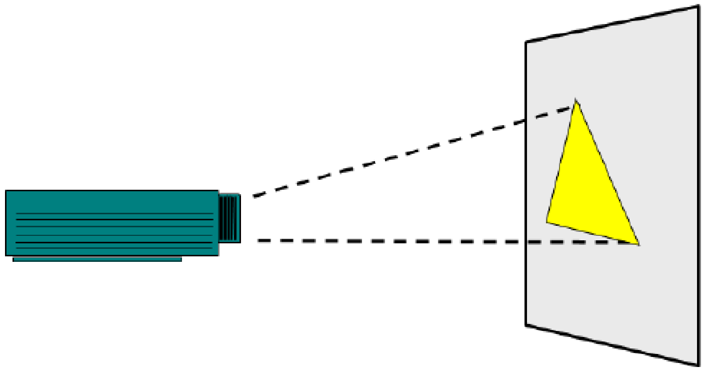
TVVS - Test, Verification and Validation of Software

Model Based Testing

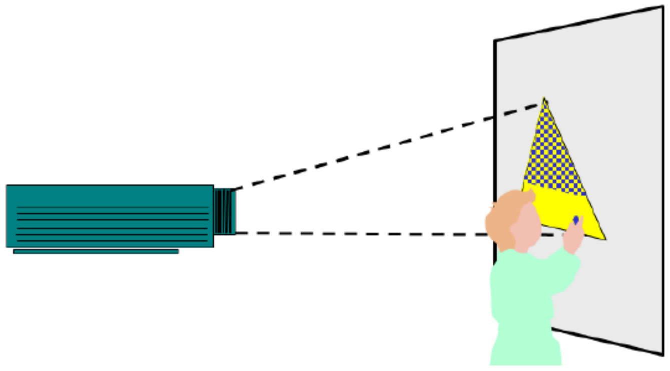
Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

“Traditional” Automated Testing

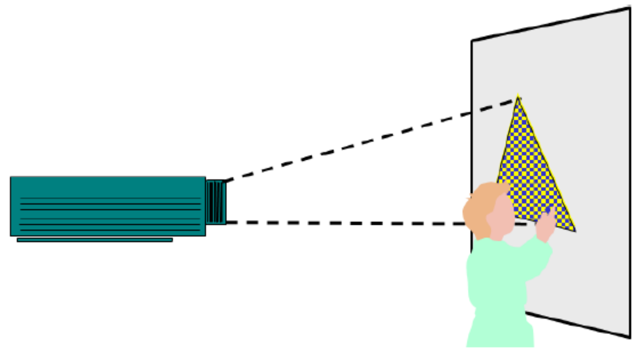


Imagine that this projector is the software you are testing.

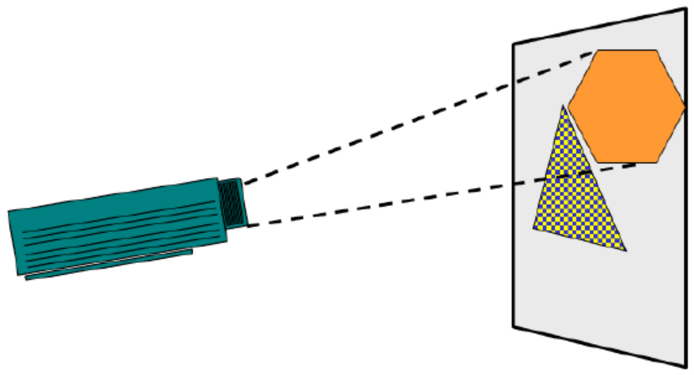


Typically, testers automate by creating static scripts.

“Traditional” Automated Testing

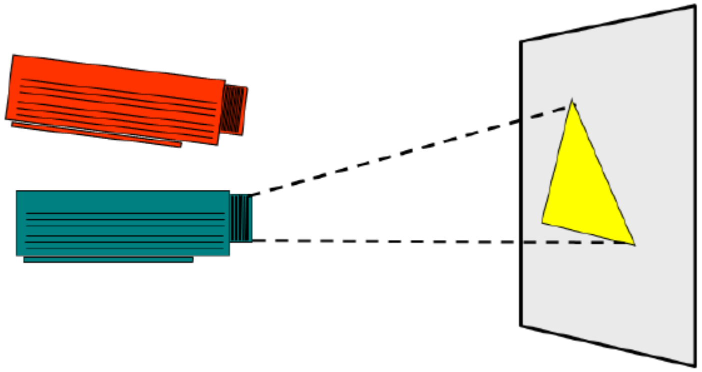


Given enough time, these scripts will cover the behavior.

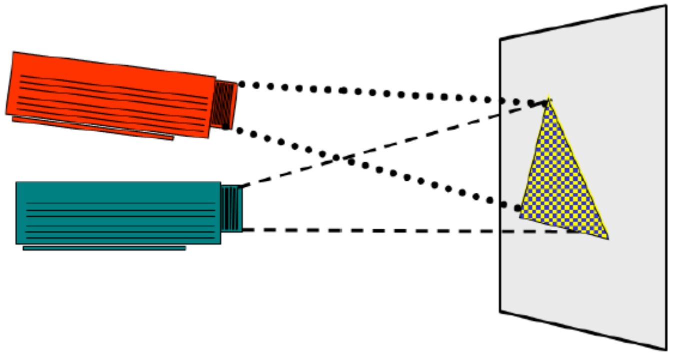


But what happens when the software’s behavior changes?

Model Based Testing

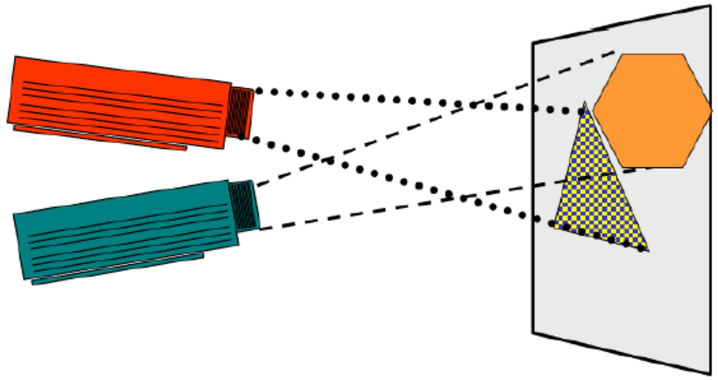


Now, imagine that the top projector is your model.

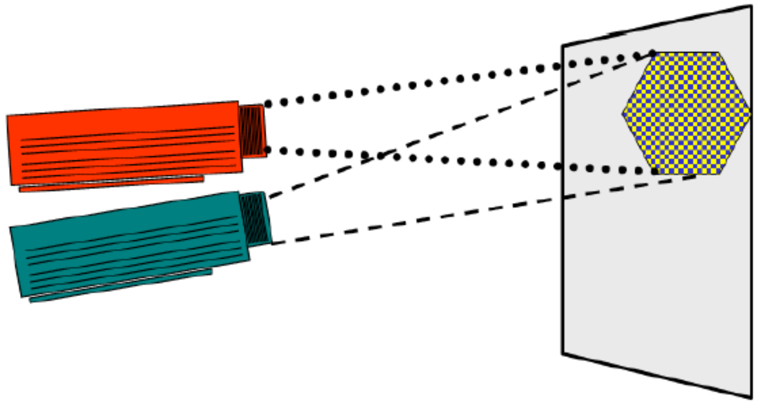


The model generates tests to cover the behavior.

Model Based Testing



… and when the behavior changes…



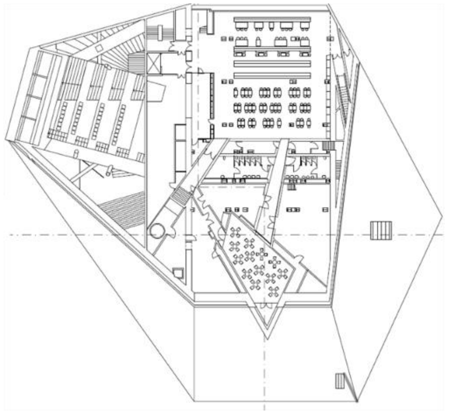
… so do the tests.

What is a model?

- In model-based testing, we use a software system model to help us **systematically** derive tests for that system.
- A Model is an **abstract representation** of the system (the model preserves, or approximates, key attributes of the artefact it shapes).
- Models **are simpler** than the systems they describe (a model is simpler than the original artefact, e.g., the entire car).
- Models help us **understand and predict** the system's behaviour
    - the model can subsequently be used to analyze properties that can be translated back to the original artefact, in this case, the aerodynamics.). The

Model Based Testing

- Models are widely used in engineering in general.
  - As an example, the car industry uses physical models of cars to test a car's aerodynamic properties. The car tested is a simplification of a real car. It may not even have an engine, but it preserves the properties needed to analyze its aerodynamics.
  - Another example is architecture, where models describe what to build.
  - In Software Engineering we use models, too.



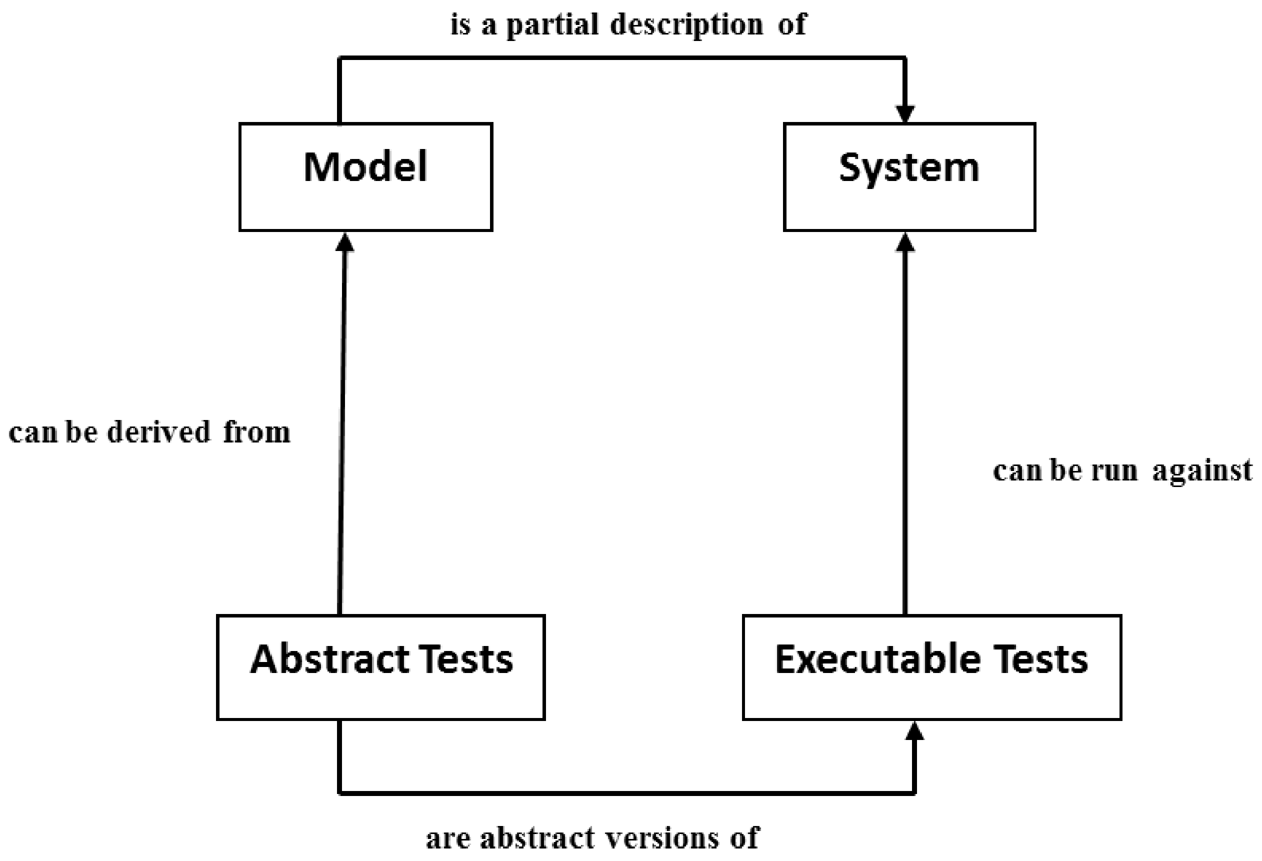
Model

Real

Model Based Testing

- Model-based testing uses a model program to generate test cases or act as an oracle.
- An oracle is the authority which provides the correct result used to make a judgment about the outcome of a test case - whether the test passed or failed

Model Based Testing



Model Based Testing

- Advantages
  - Improved quality of the product
  - Better quality of the product
  - Higher degree of automation (test case generation)
  - Allows more exhaustive testing
  - Good for correctness/functional testing
  - Model can be easily adapted to changes
  - Early exposure of ambiguities in specification and design
  - Generate variety of test suites from the same model by different test selection criteria

Model Based Testing

- Disadvantages
  - Requires a formal specification/model
  - Test case explosion problem
  - Test case generation has to be controlled appropriately to generate a test case of manageable size
  - Small changes to the model can result in a totally different test suite
  - Time to analyse failed tests (model, SUT, adaptor code)

Model Based Testing

Some Tools

- ModelJUnit
- SpecTest
- Mulsaw
- Nmodel and SpecExplorer
- PBGT
- GUITAR
- ...

Model Based Testing

- In software we use models too. You may have, for example, used the UML, the Unified Modeling Language. It allows you to create models of your software, for example by means of class or package diagrams, which model the static structure of the software system.
- For testing purposes, we are mostly interested in **models of the behaviour of software systems**, that is, of the dynamic characteristics.
- In the UML, examples of such behavioural models include **state machine diagrams** and **activity diagrams**.

Model Based Testing

- Models obtained from requirements, e.g., user stories
  - Meaningful to domain experts. The developers can then use them to identify test cases for behaviour that “must be there”, specified in the requirements.
- Models “reverse engineered” from the code
  - In this case, the model reflects what is in the current code base. Such models are typically most meaningful to developers for their testing purposes. In this case, developers will use the model to derive tests that systematically exercise certain aspects of the code base, reflected in the model.

Model Based Testing - languages

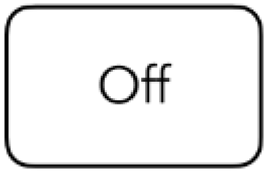
- Pre/Post (or Model-based). Ex.: VDM, Z, Spec#.
- Transition-based. Ex.: FSM, Petri nets.
- Behavior-based (or history-based). Ex.: CSP.
- Property-based (or functional-based). Ex.: OBJ.
- Hybrid approaches. Ex.: RAISE.
- UML
- GUI modeling: Abstract models (e.g., PiE e RED-PiE), grammars, FSM, Property based, Behavioral based, hybrid, EFG, Spec#, etc.

State Machines

- State machines are used to model the behaviour of software systems.
- State machines model, as the name suggests, the state of a software system, and what actions can lead to a change in that state.
- Most systems around us have some notion of “state” inside them. For example, your phone can be switched off, in standby mode, or being used.

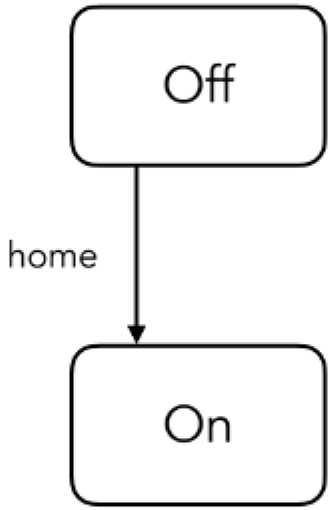
State Machines, on/off example

- Here is a state, in the box, labelled "Off". It represents a phone switched off.



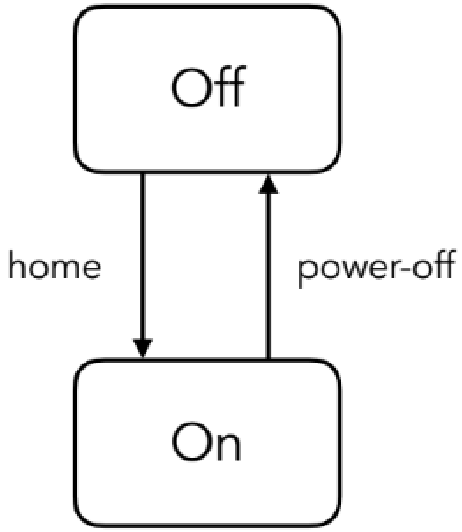
State Machines, on/off example

- If we switch it on using the “home” button, we enter a state labelled “On”. We see a new state, as well as a transition from the “Off” to the “On” state, shown as an arrow. An event triggers this transition: in this case, pressing the “home” button. The diagram represents this as the “home” label on the transition.



State Machines, on/off example

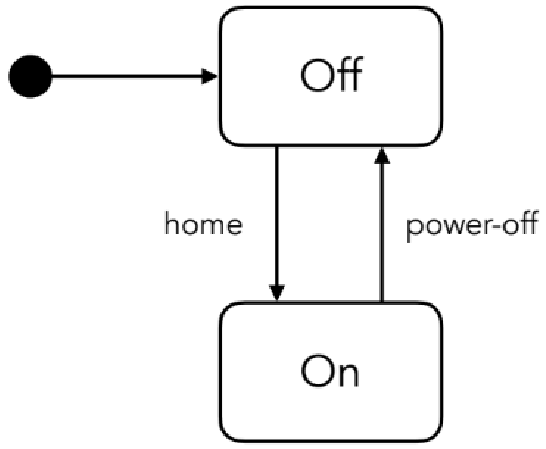
- Once the phone is on, we can also switch it off. This leads to another transition, this time triggered by pressing the "power-off" button for a few seconds.



State Machines, on/off example

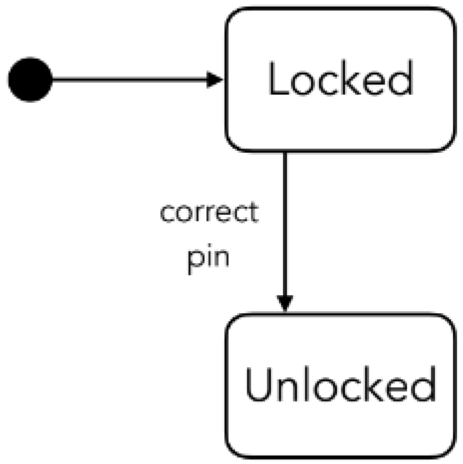
- With two states, we should decide which is the “**initial**” state. For the phone, we will assume that the phone initially is switched off. In the diagram, we indicate this with the little arrow into the “Off” state shown at the top left.

  - This very simple model represents one aspect of the phone's behavior: namely how it can be switched on and off using the home and power-off buttons.



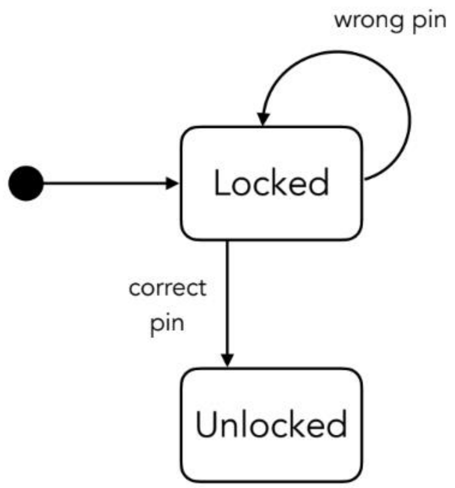
State Machines, unlocking example

- Here we see a “Locked” state, which we also mark as initial since by default a phone is locked. We can unlock the phone by entering the correct PIN, as shown in the diagram.



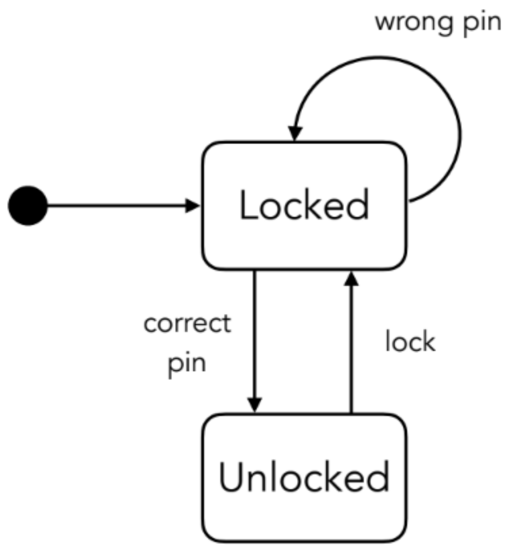
State Machines, unlocking example

- However, the entered PIN code may also be wrong. In that case, the phone stays in the "Locked" state. This is shown using a "self-transition", which goes from the "Locked" state back to the "Locked" state itself.



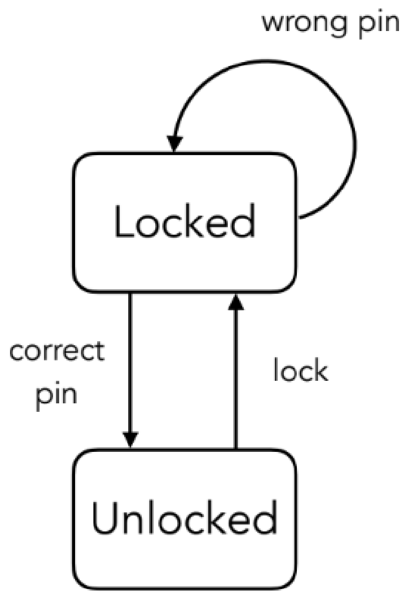
State Machines, unlocking example

- Once we are unlocked, we can press the “lock” button to lock the phone again to protect it against unintended use by others, modelled here as a transition from “Unlocked” to “Locked”.



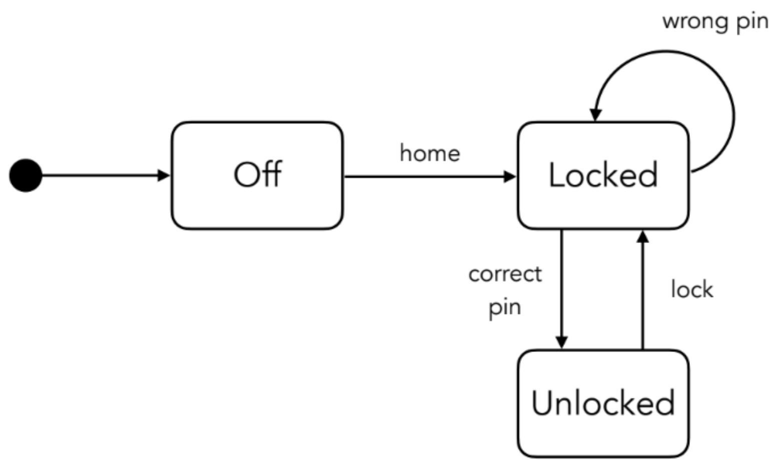
State Machines, on/off + unlocking

- Now we have modelled two aspects of the behaviour of using your phone: switching it on and unlocking it. We can combine these into a larger diagram.
- Here we start with the unlocking state diagram.



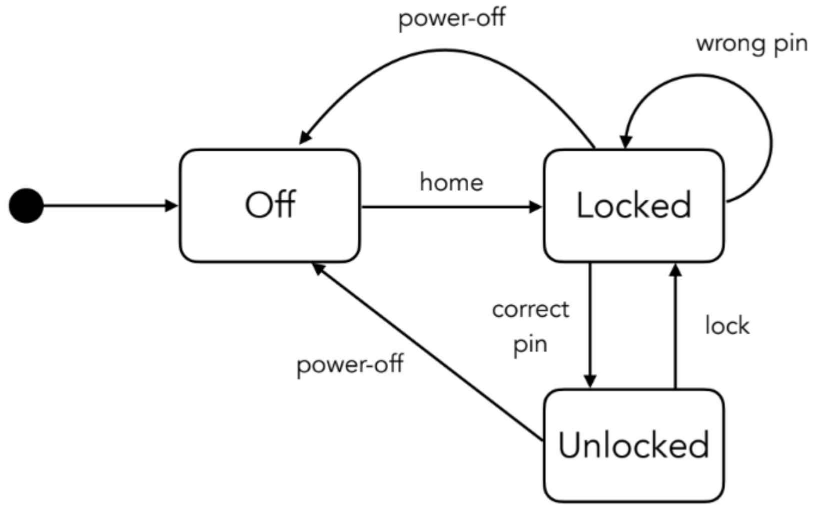
State Machines, on/off + unlocking

- We can add the “Off” state to this, together with the transition from “Off” to the “Locked” state, which is the initial state of the unlocking diagram.



State Machines, on/off + unlocking

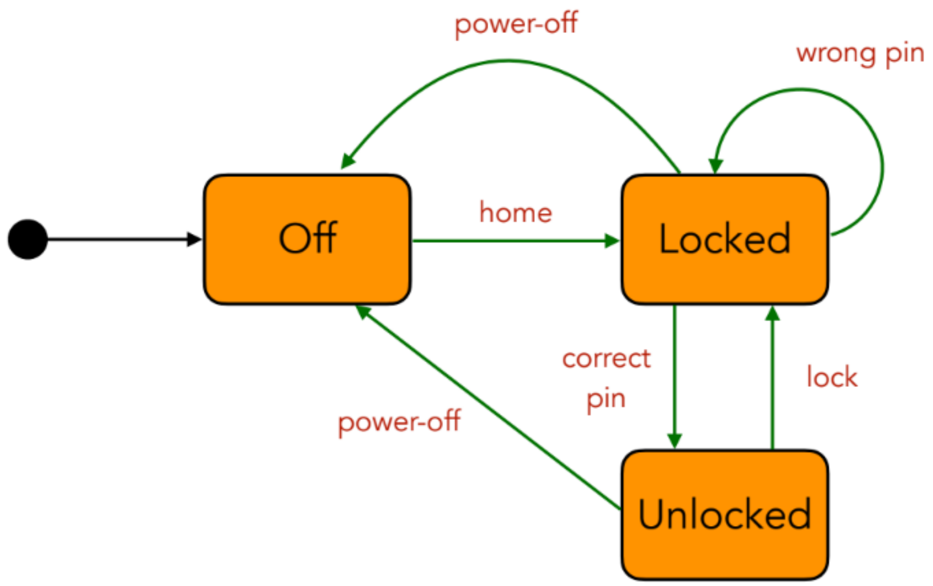
- Furthermore, we need to indicate how to switch off the phone. Pressing the power-off button in any state switches off the phone. Therefore, we add two transitions that go back to the “Off” state: one starting in the “Locked” state at the top, and one in the “Unlocked” state at the bottom.



State Machines, on/off + unlocking

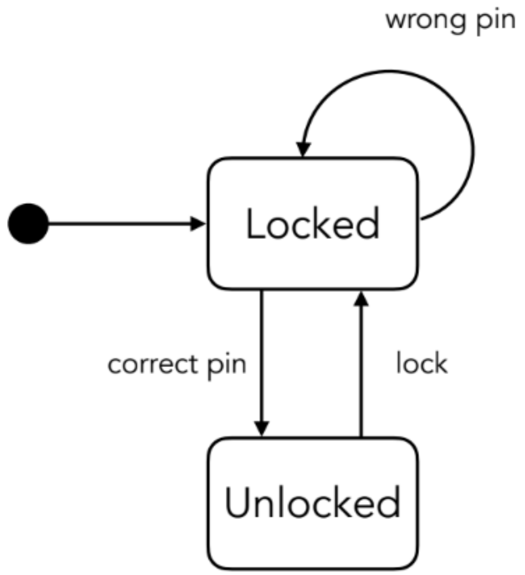
The diagram has now become a little more complex, with

- three states
- six transitions
- five event types



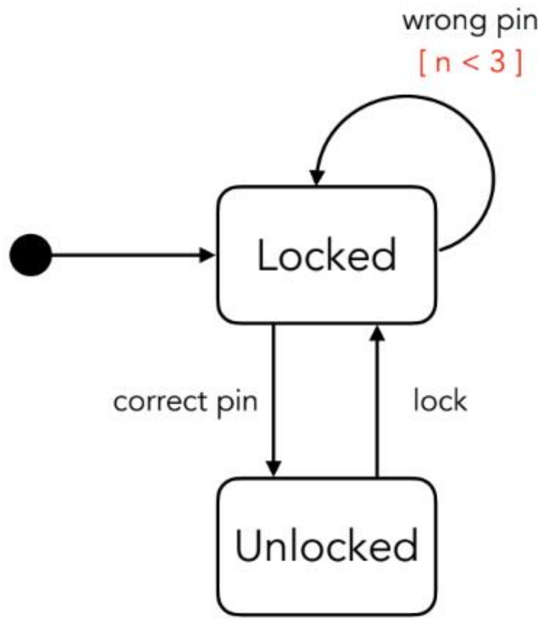
State Machines, conditional transitions

- State diagrams also support **conditional transitions.**



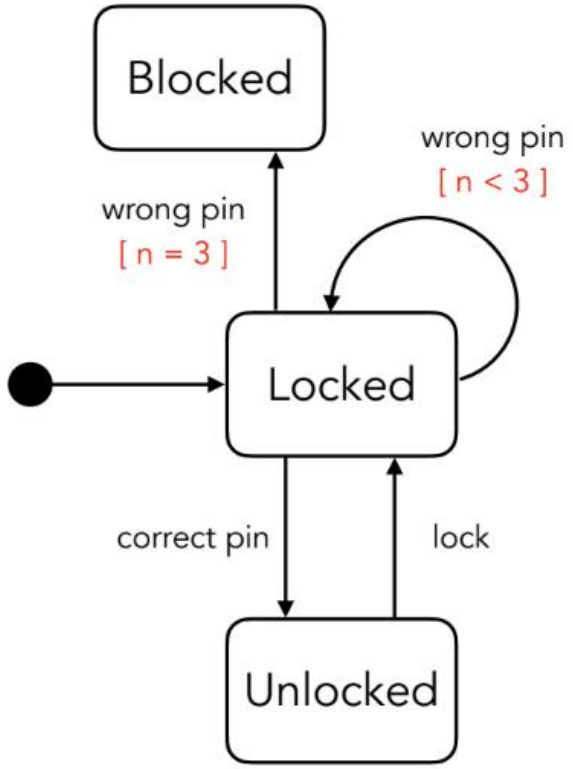
State Machines, conditional transitions

- An example is shown here for the self-transition with the wrong PIN code. The **condition, written with square brackets**, states that the number of attempts, denoted by n, should be less than 3.



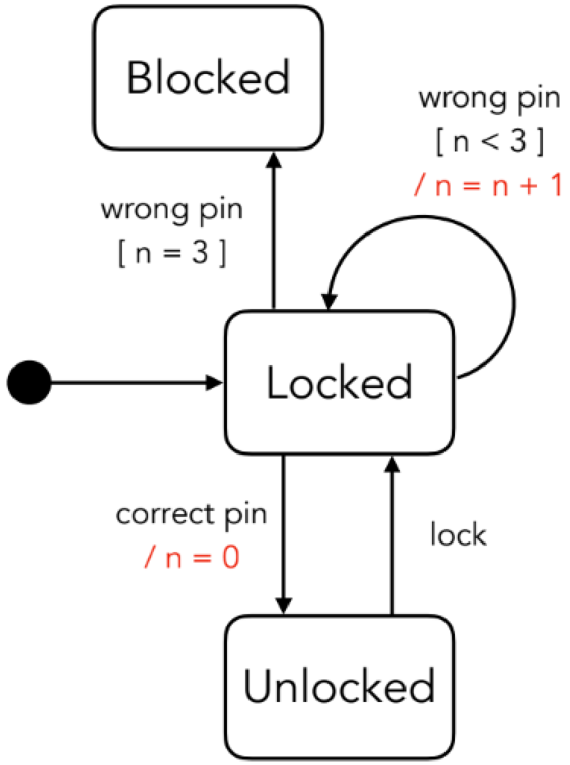
State Machines, conditional transitions

- After entering a wrong PIN three times, the phone gets blocked, as indicated here. Thus, the “Locked” state has two outgoing edges both triggered by the “wrong pin” event, and the condition determines the resulting state.



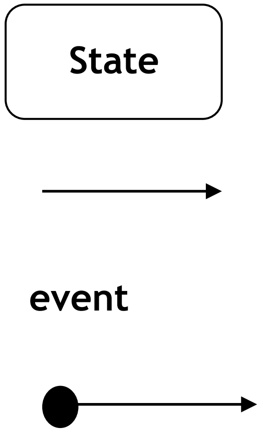
State Machines, actions

- Yet another convention of the UML is that transitions can have "actions". These are denoted by a forward slash "/", followed by the action.
- In our example, we have added two actions, both manipulating the attempt counter "n":
  - If we enter a wrong PIN, we increment the counter.
  - If we enter a correct PIN, we reset it to zero.



State Machines, recap

- States
- Transitions
- Events
- Initial State
- [Conditions]
- / Actions



State Machine Test Adequacy

- To test a state machine, various forms of state machine-specific test adequacy criteria can be used.
  - **State coverage**. The simplest, just ensures that every state is reached once.
  - **Transition coverage**. It is a bit stronger than state coverage, which insists that every transition is covered. Note that all states are also automatically covered if this is the case.
  - **Path coverage**. Exercise sequences of transitions, which are paths through the state machine.
    - Note that state machines very often have loops with an infinite number of possible paths. As it is impossible to take the loop as often as we want, leading to more and more and longer paths, we cannot have full path coverage. We can, however, insist that each loop is exercised at least once.

Testing one transition, recipe

To test one transition between states S1 to S2, we do the following:

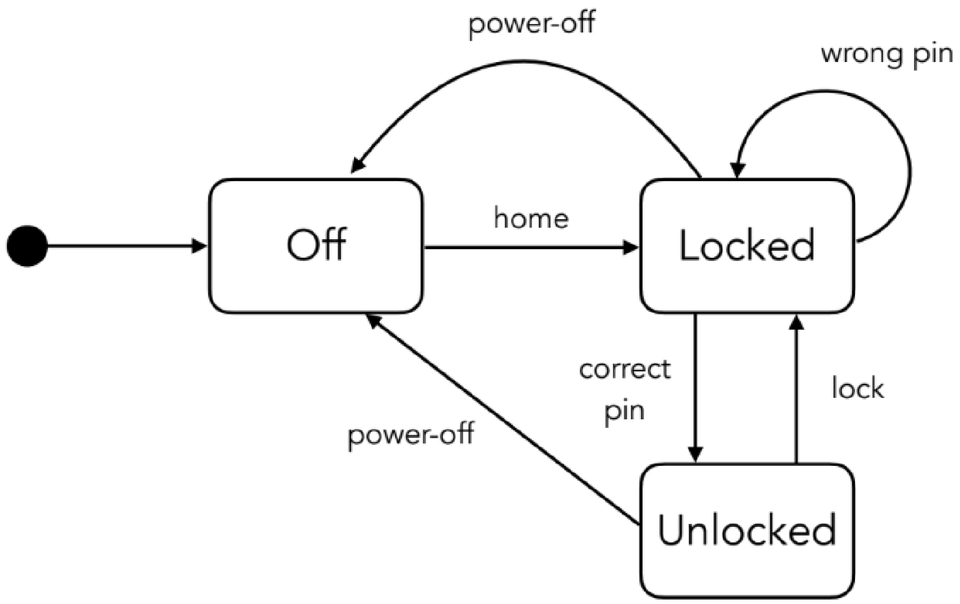
- 1. Bring the system into state S1 and ensure/verify that it is indeed in state S1.
- 2. Then trigger the event, e.g., event1, that should lead to state S2.
- 3. Assess that any action that should come with the event takes place.
- 4. Assess that the system has indeed reached state S2.

Testing a sequence of transitions

- A full test case will typically cover a sequence of transitions, which will exercise a path through the state machine.
- As there are potentially infinitely many paths, we will need some approach to decide which ones to exercise.
- Our way to do that is by creating a transition tree spanning the diagram.

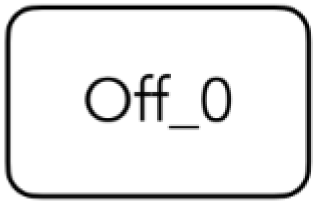
Transition tree

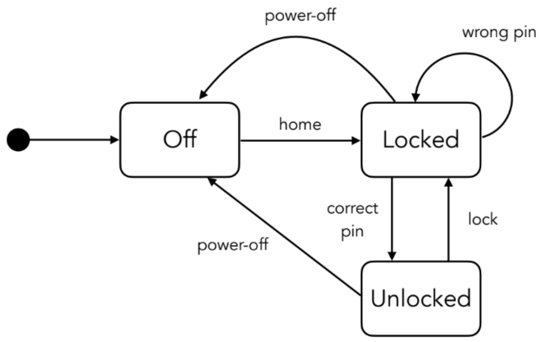
- Let us do this for our previous example, on/off + unlocking.



Transition tree

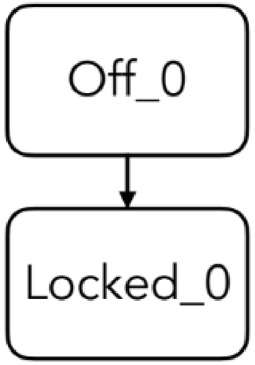
- We start with the initial state, named “Off”. Since our tree will duplicate some states, we add a number to the state to give it a unique name.

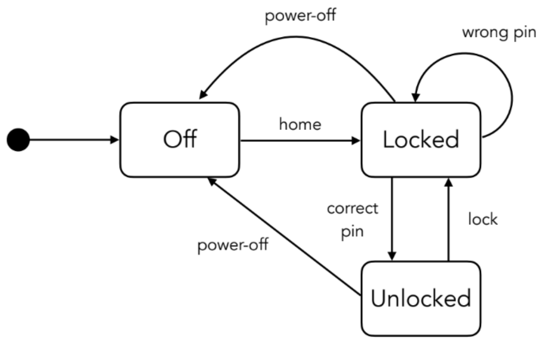




Transition tree

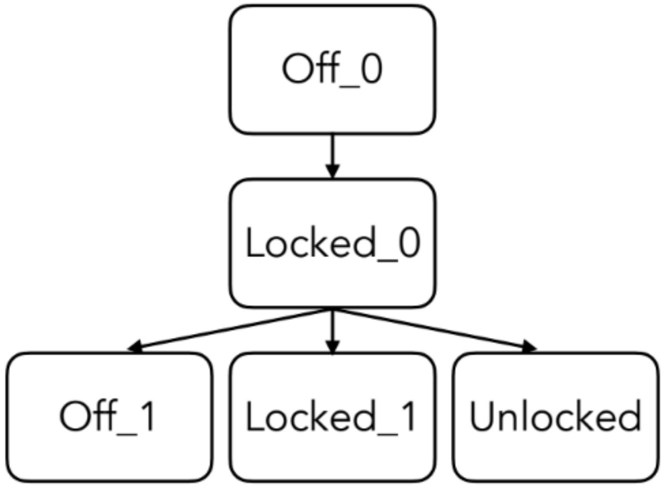
- From “Off”, we have one outgoing transition: we can go to the “Locked” state.

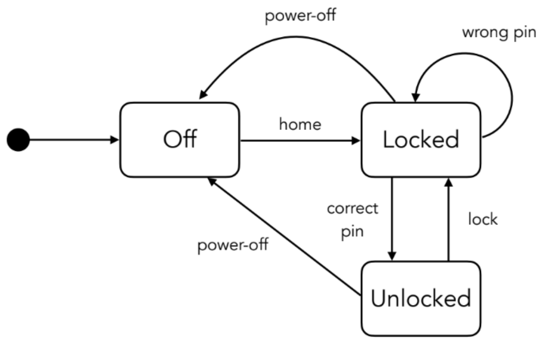




Transition tree

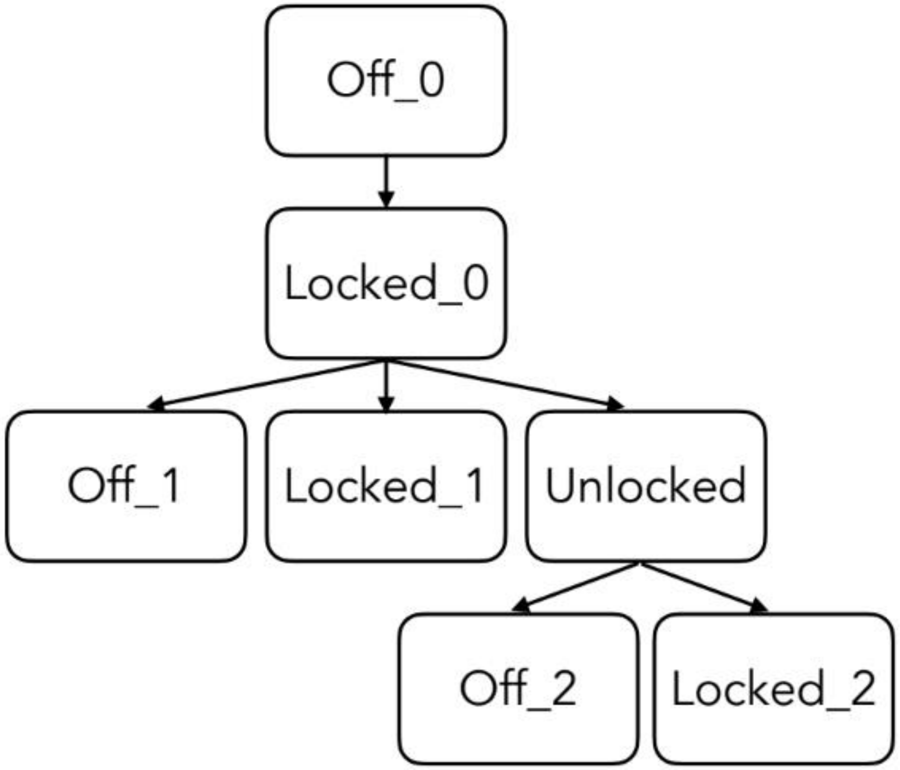
- From the “Locked” state, we have three outgoing edges: “Off”, “Locked”, and “Unlocked”.
- Note that the “Off” and “Locked” states are somewhat special, as we have been there before, i.e., the off-zero and locked-zero nodes in the tree.
- As the zero nodes in the tree already describe their behavior, we do not repeat it for the one-nodes.

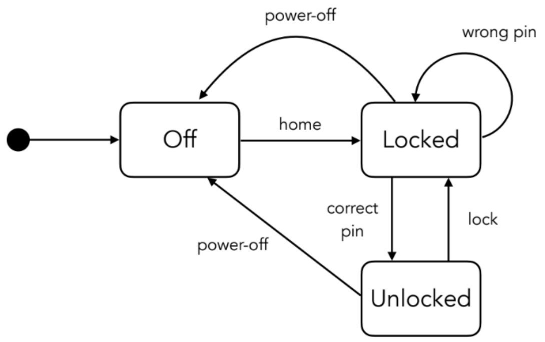




Transition tree

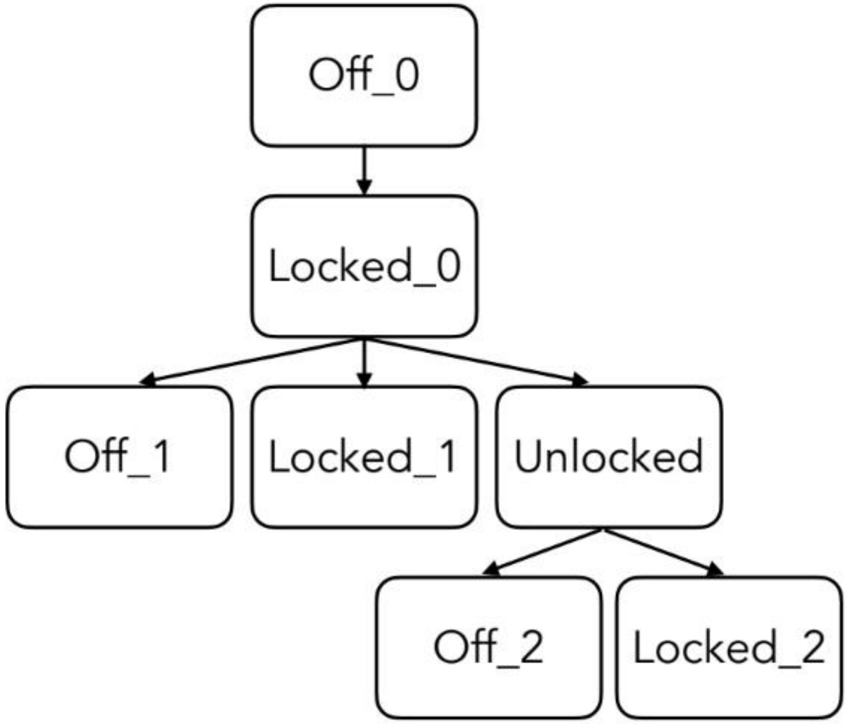
- From "Unlocked", we can reach two states, off and locked.
- Again, the two-level states were described at the zero level, so we do not repeat their behaviour.

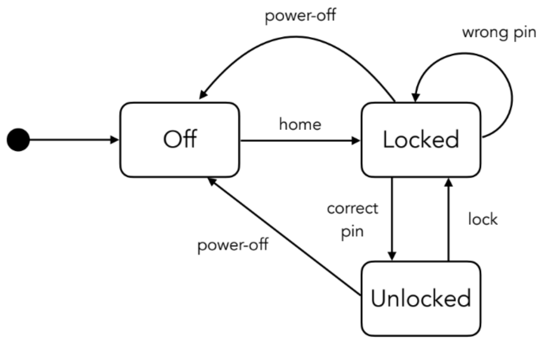




Transition tree

- This then gives a tree that matches the original state diagram.
  - Each loop is unfolded once.
  - Furthermore, it can help us to find the shortest path in the graph to any state, simply by following the tree.

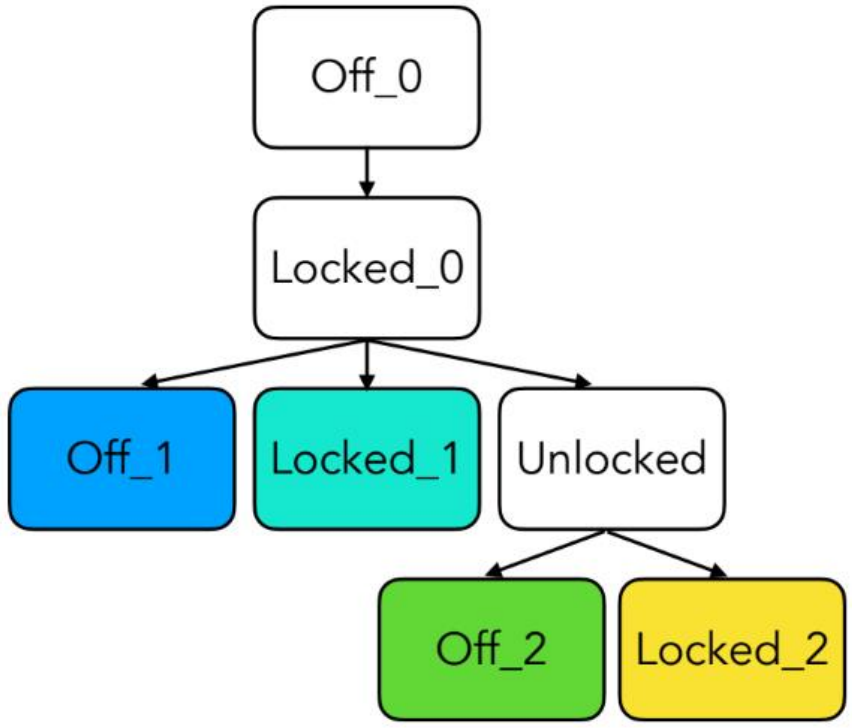




Transition tree

Once we have the tree, we can use it to derive test cases.

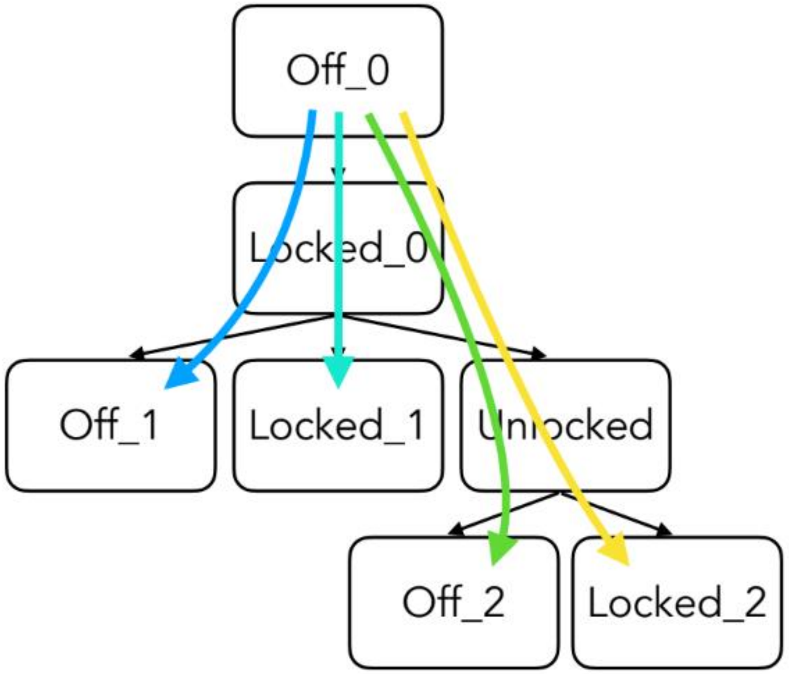
- **Each test case is a path from the root of the tree to one of the leaves.** With four leaves for this tree, we then have four test cases.



Transition tree

Here we show the four test cases, with different colors.

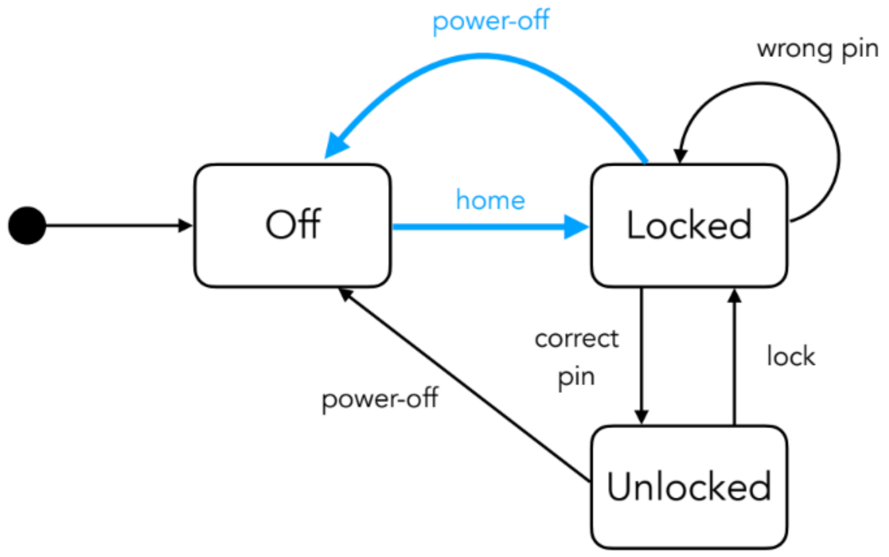
- For example, the **blue** path gives the simplest test case, corresponding to switching the phone on and off again, without unlocking it.
- The test path that also does the unlocking is the one drawn in **green**.



Testes from the transition tree

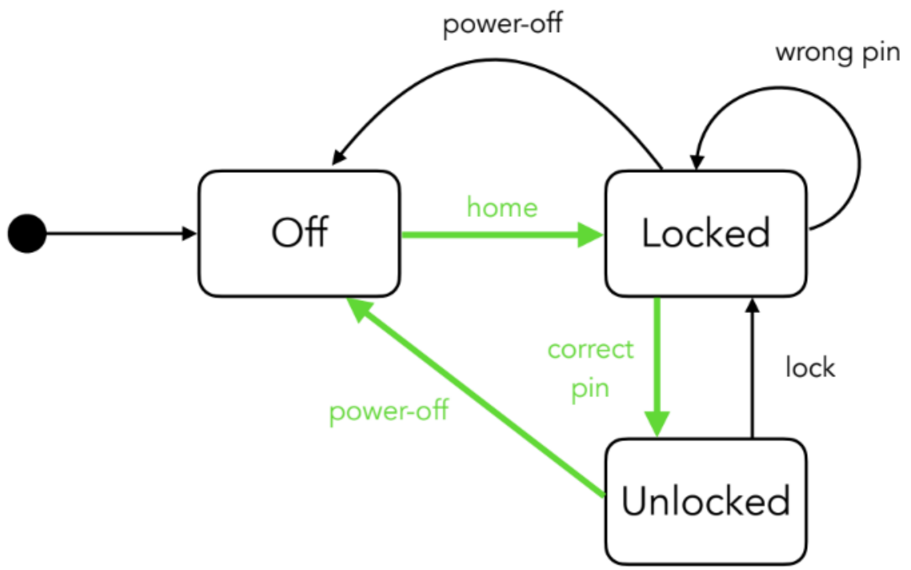
These paths can also be displayed in the state machine.

- For example, the blue path is from “Off” to “Locked” and back.



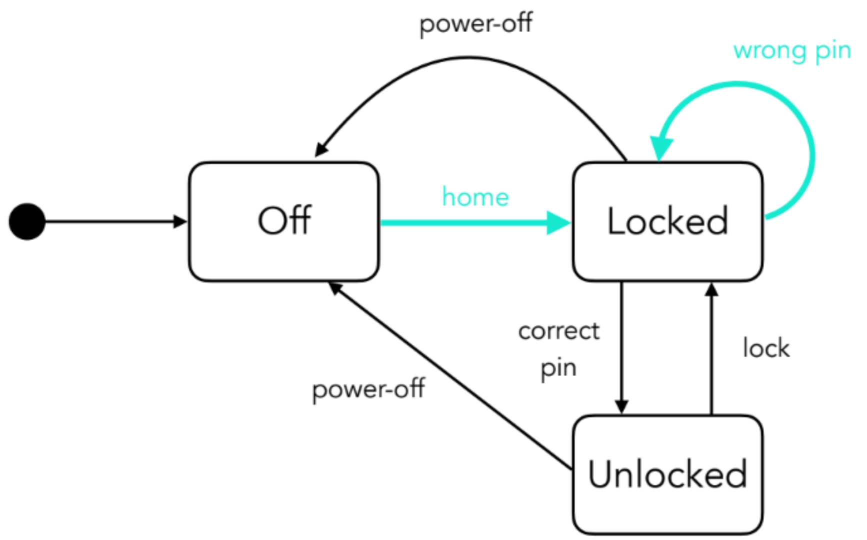
Testes from the transition tree

Another example, the green path goes from "Off", via "Locked" and "Unlocked" back to "Off"



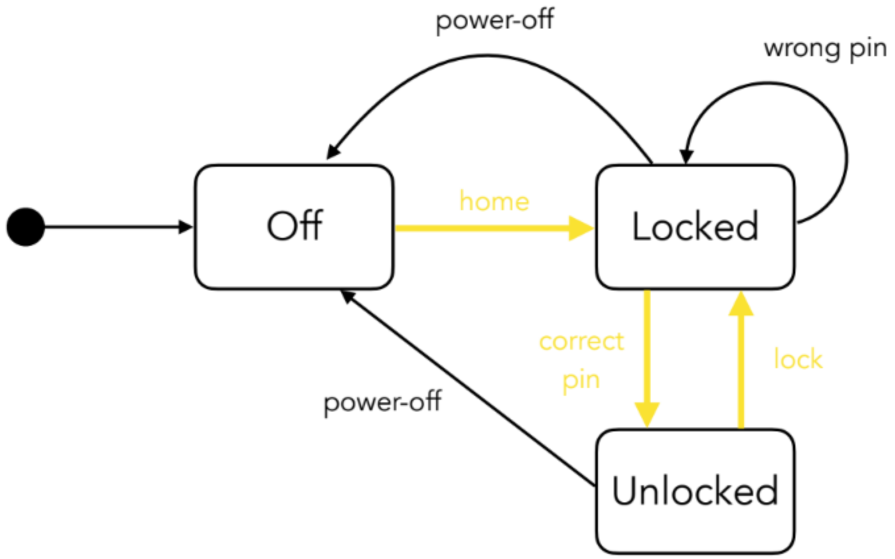
Testes from the transition tree

Another example, is the light blue path that goes from “Off” to “Locked” and remains there



Testes from the transition tree

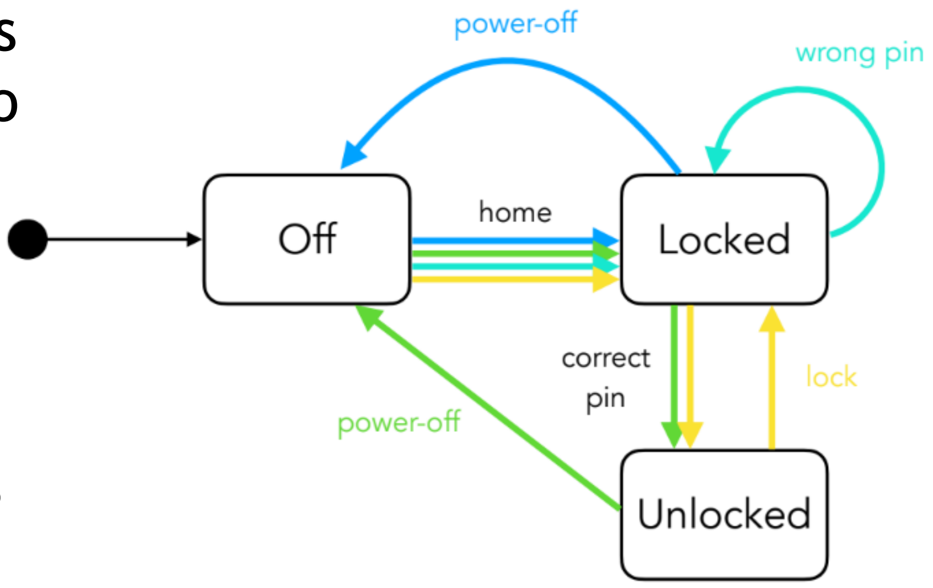
Another example, is the yellow path that goes from "Off" to "Unlocked" through the "Locked" state



Tests from the transition tree

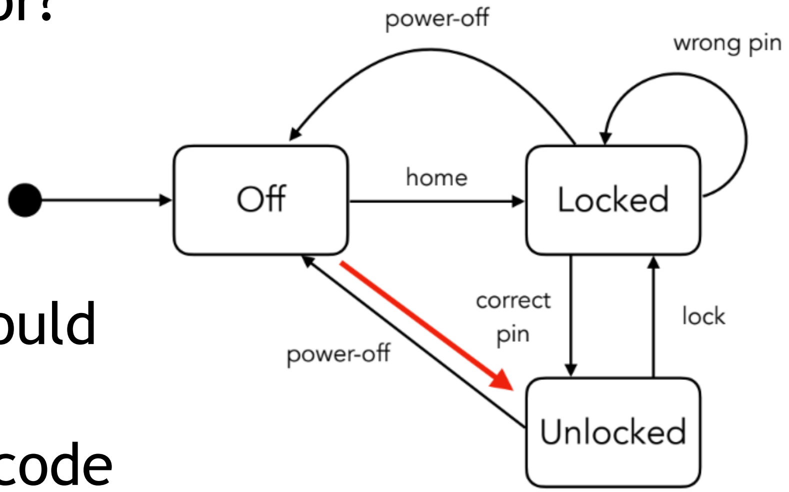
- The transition tree gives us guidance on which paths to take to ensure that each specified **transition behaves as intended**, and which loops to execute.

- With this, we test that the system under test implements all specified behaviour. In other words, we test that the implementation conforms to the model.



Sneak path testing

- But what about unspecified behavior? What if the system accidentally (or secretly) implements additional transitions?
- For example, what if the system would allow going from “Off” directly to “Unlocked”, circumventing the pincode mechanism? That would be a security problem, which we, of course, want to avoid.
- To check for such “sneak paths”, we create a tabular representation of the state machine.



Sneak path testing: transition table

States / Events	home	Power-off	lock	Wrong pin	Correct pin
Off					
Locked					
Unlocked					

- In the rows we have states, and in the columns, we have events

Sneak path testing: transition table

States / Events	home	Power-off	lock	Wrong pin	Correct pin
Off	Locked				
Locked					
Unlocked					

- If a state has a transition with the given event, we indicate the corresponding cell with the outgoing state.
- For example, in our state machine, we can move from “Off” to “Locked” by clicking the home button. No other events are possible from the “Off” state, which is why the remaining cells in the “Off” row are empty

Sneak path testing: transition table

States / Events	home	Power-off	lock	Wrong pin	Correct pin
Off	Locked				
Locked		Off		Locked	Unlocked
Unlocked		Off	Locked		

- … and we follow the same process for the other states.
- A table like this contains all the information from the diagram. Yet at the same time, the empty cells are directly visible. These empty cells correspond to “sneaky” transitions, which we will test them now.

“Sneaky” transitions

States / Events	home	Power-off	lock	Wrong pin	Correct pin
Off	Locked	🟥	🟥	🟥	🟥
Locked	🟥	Off	🟥	Locked	Unlocked
Unlocked	🟥	Off	Locked	🟥	🟥

- Determine the intended behaviour for empty cells. A common option is to ignore the event. An alternative possibility is to raise an exception. Ultimately, these choices should be made by the domain experts, but in some cases, developers can make a guess

“Sneaky” transitions

States / Events	home	Power-off	lock	Wrong pin	Correct pin
Off	Locked				
Locked		Off		Locked	Unlocked
Unlocked		Off	Locked		

- We start by bringing the system into a particular state, for example, the “Locked” state. We can again use the transition tree to bring the system to a particular state. That tree tells us exactly which path to take from the top to reach a given state. From there we trigger the two unspecified events (corresponding to home and lock in the empty cells).
- We then assess that these events have no observable effect and that the system stays in the “Locked” state.
- We repeat this for all empty cells. For our example, this then results in 9 additional sneak path tests.

Sneak path testing

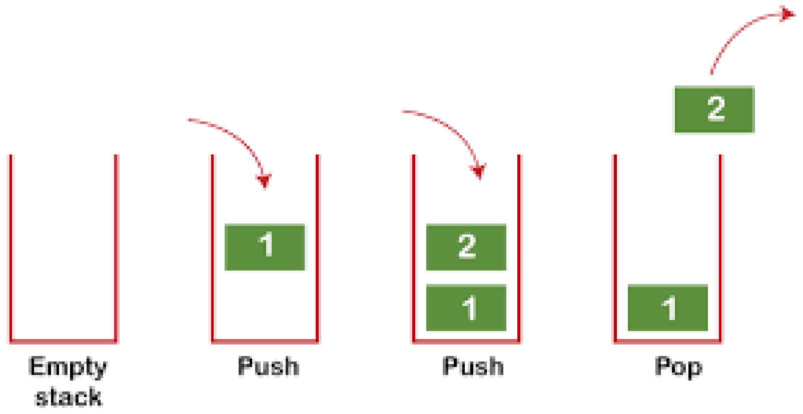
- The resulting “sneak path test suite” helps us to verify that illegal transactions cannot occur.

- As always in software testing, whether it is necessary to apply sneak path testing to your system is a tradeoff between risk and cost.

- Sneak paths are particularly relevant when the system has high demands concerning security (when we want no backdoors) or safety (when we want no accidents).

Test case explosion problem stack example

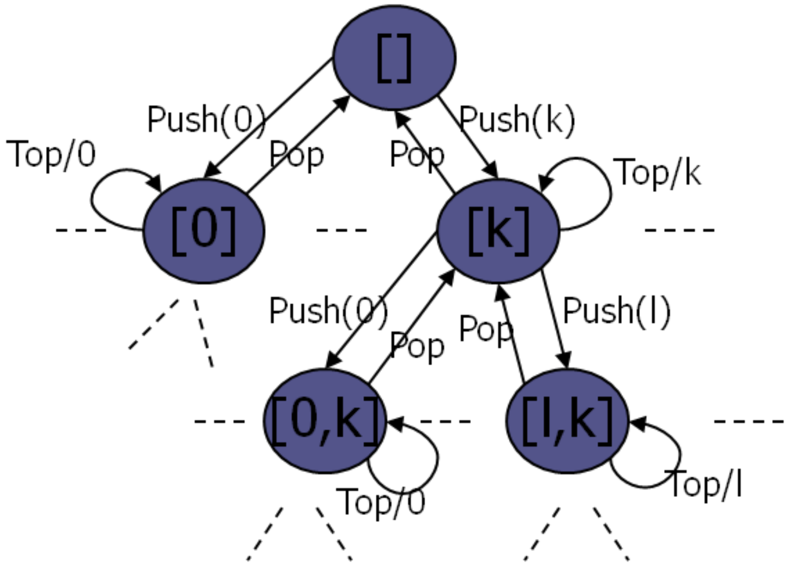
- Imagine a state machine for a stack where states describe the content of that stack
- How many states/transitions do we have? How many test cases do we need to cover all states/transitions?



Test case explosion problem
stack example

Stack Spec# model

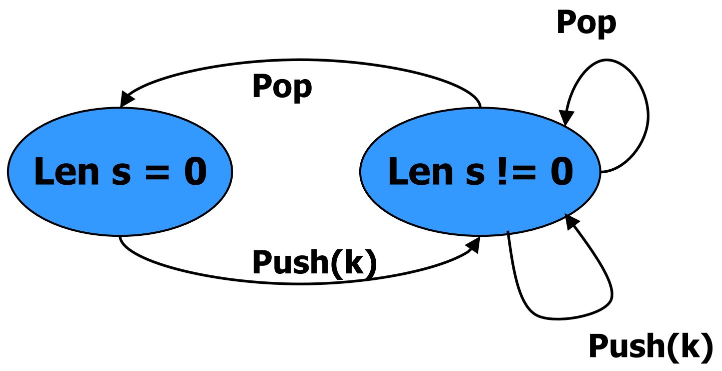
```
var Seq<int> content = Seq{};

[Action]
public void Push(int x) {
  content = Seq{x} + content;
}

[Action]
public void Pop() {
  requires !content.IsEmpty;
  content = content.Tail;
}

[Action]
public int Top() {
  requires !content.IsEmpty;
  return content.Head;
}
```



Test case explosion problem
stack example

- View all states in a G-group as being the same
- A way to define “what is an interesting state” from some testing point of view

- Example: **content.Size** in the stack model



Test case explosion problem
stack example

One or more **state groupings** may be defined

- A grouping G is a sequence of state based on expressions $g_1,\ldots,g_k$
- Two states *s* and *t* are in the same group G or G-equivalent if
  - $g_i^s = g_i^t$ for $1 \leq i \leq k$
- A G-group is the set of all G-equivalent states

Also used in viewing

Test case explosion problem
stack example

- The following techniques are used in Spec Explorer
  - State filters
  - Stopping conditions
  - State groupings
  - Search order
  - Enabling conditions on actions

- Usually a combination of all (or some) of the techniques is needed to achieve desired effect