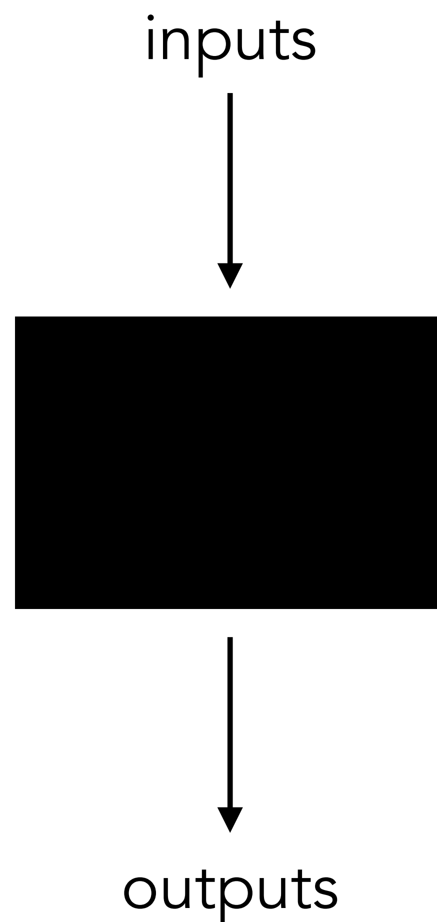
Software Testing, Verification and Validation

October 19, 2022
Week #5 — Lecture #4

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Last week, we revisited equivalence class partition / category-partition and introduced boundary values analysis as part of our set of black-box techniques. We also introduced random testing.

Black-box testing



Techniques

✅ Equivalence class partitioning
✅ Category partition
✅ Boundary value analysis
✅ Random testing
👉 Model-Based Testing
Cause effect graphing
Error guessing
…

Model-based Software Testing

Model-based Software Testing

In model-based testing, we use a model of a software system, to help us systematically derive tests for that system.

Models are widely used in engineering in general. As an example, the car industry uses physical models of cars to test a car's areo-dynamic 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 areo-dynamics. These characteristics hold for software models too.

Model

- a model is simpler than the original artifact, e.g., the entire car.
- the model preserves, or approximates, key attributes of the artifact, its shape.
- the model can subsequently be used to analyze properties that can be translated back to the original artifact, in this case the areo-dynamics.

Models

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 behavior of software systems**, that is, of the dynamic characteristics.

In the UML, examples of such behavioral models include **state machine diagrams** and **activity diagrams**.

Models

- **State machine diagrams** can be used whenever a system maintains state, which, in fact, most systems do. A typical example is a web or mobile application, which can be in various states. The transitions between those states and the events, clicks, swipes, inputs, triggering those transitions, are easily modeled using a state machine.
- **Decision tables** can be used whenever combinations of multiple inputs determine a system's behavior. A typical example here are payment plans for mobile phones, in which a number of user choices determine the monthly subscription fee and the features the user can enjoy.

Both types of models can typically be obtained from the system requirements or user stories.

Models for testing

- **Models obtained from requirements, e.g., user stories**
  - Meaninfull to domain experts. The developers can then use them to identify test cases for behavior 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, reflected in the model, of the code base.

State Machines

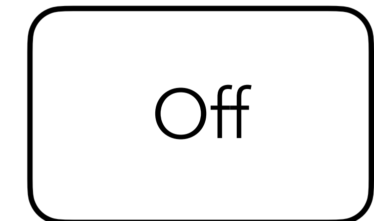
State machines are used to model behavior of software systems. In this course, we will learn how to design state machines, and how to use them to derive test cases in a systematic manner.

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 stand-by mode, or in use.
Let’s model that behavior using a state machine.

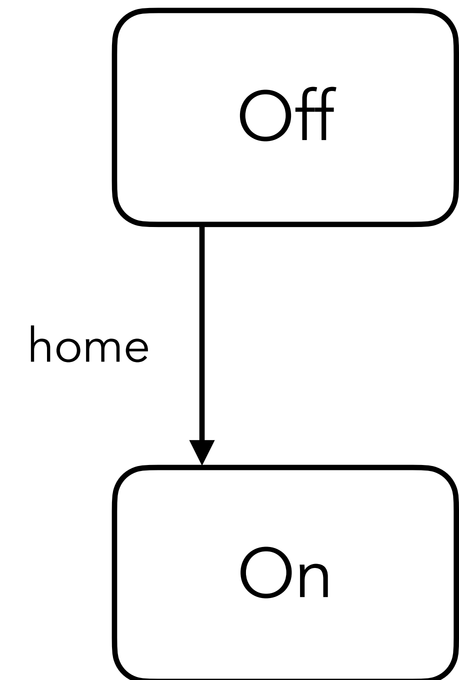
State Machines, on/off example

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



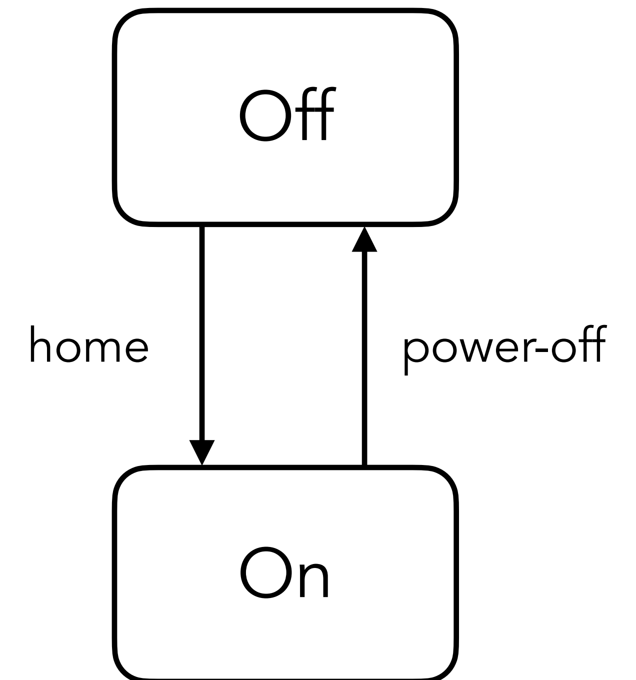
State Machines, on/off example

- If we switch it on using the “home” button, we enter a state labeled “On”. We see a new state, as well as a **transition** from the “Off” to the “On” state, shown as an arrow. This transition is triggered by an event: in this case pressing the “home” button. In the diagram, this is represented as the “home” **label** on the transition.



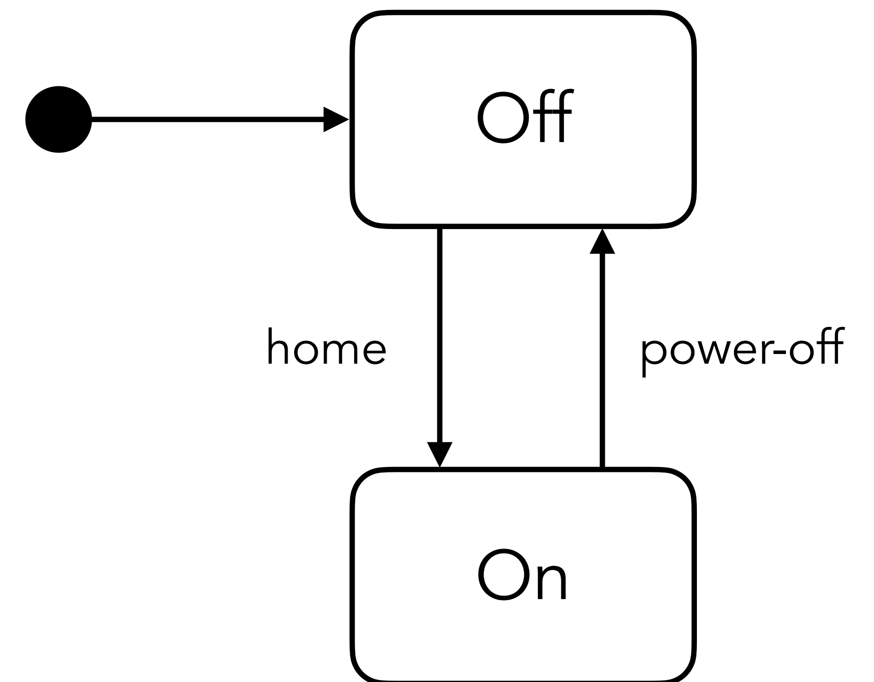
State Machines, on/off example

- Once the phone is on, we can also switch it off. This leads to another transition, this time triggered 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.



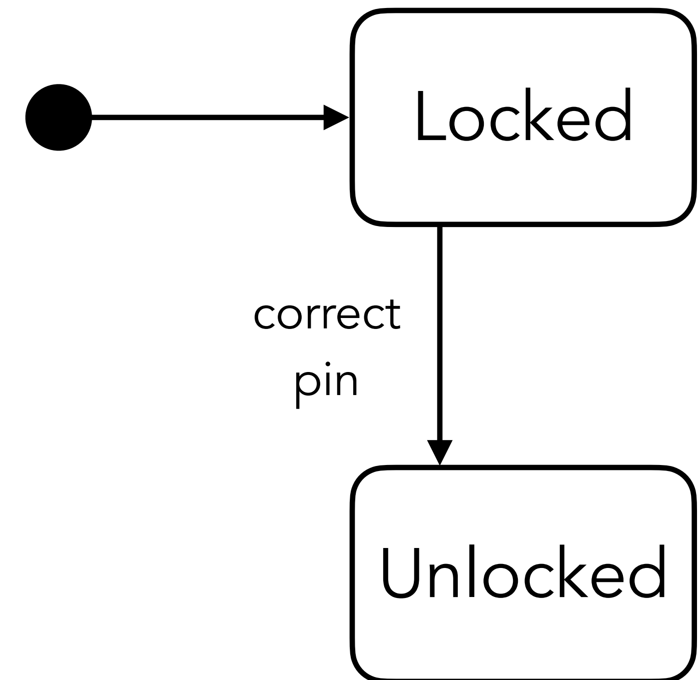
This very simple model represents one aspect of the phone's behavior: namely how it can be switched on and off using the home and power-off buttons.

State Machines, unlocking example

We can also model different behaviors, for example unlocking the phone.

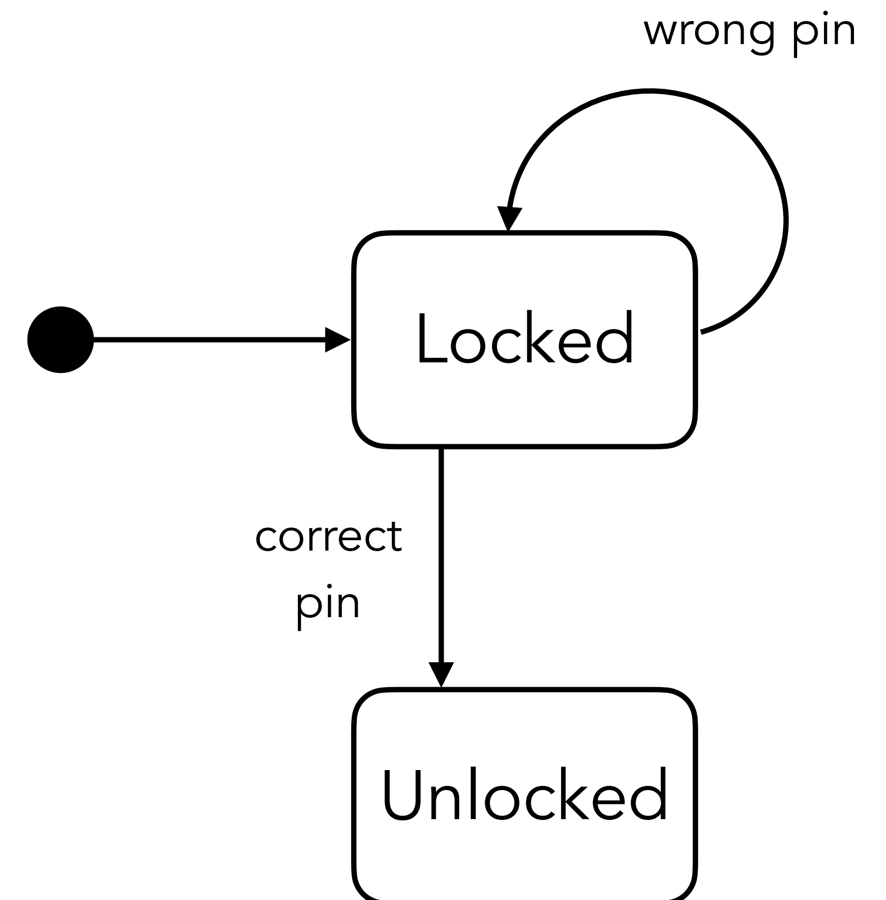
State Machines, unlocking example

- 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 a correct pin, as shown in the diagram.



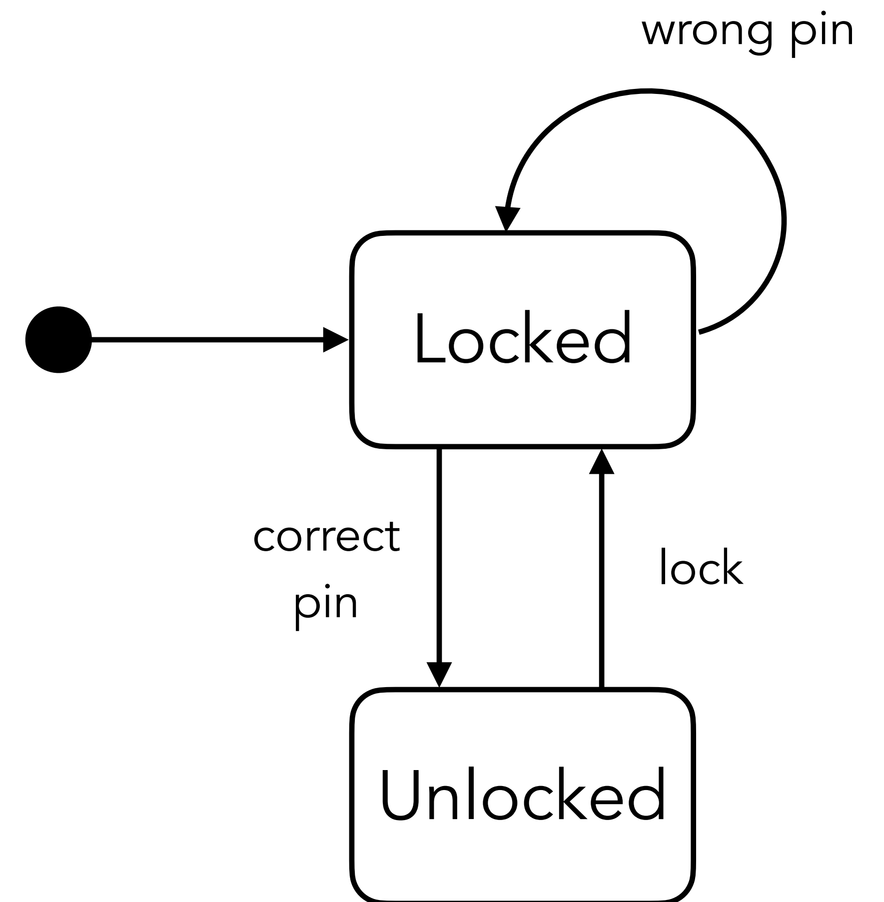
State Machines, unlocking example

- 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

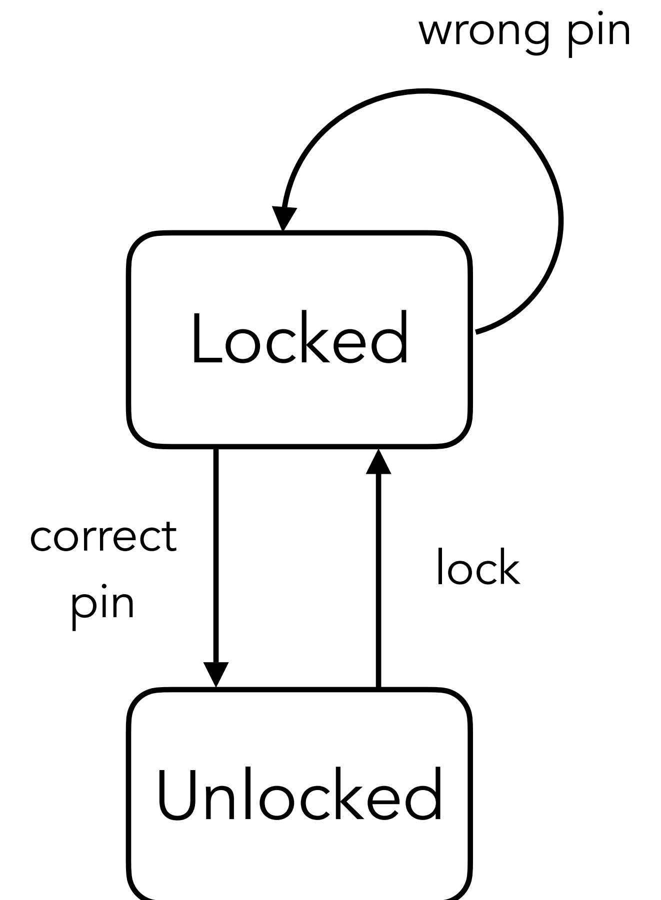
- Once we are unlocked, we can press the “lock” button to lock the phone again to protect it against unintended use by others, modeled here as a transition from “Unlocked” to “Locked”.



State Machines, on/off + unlocking

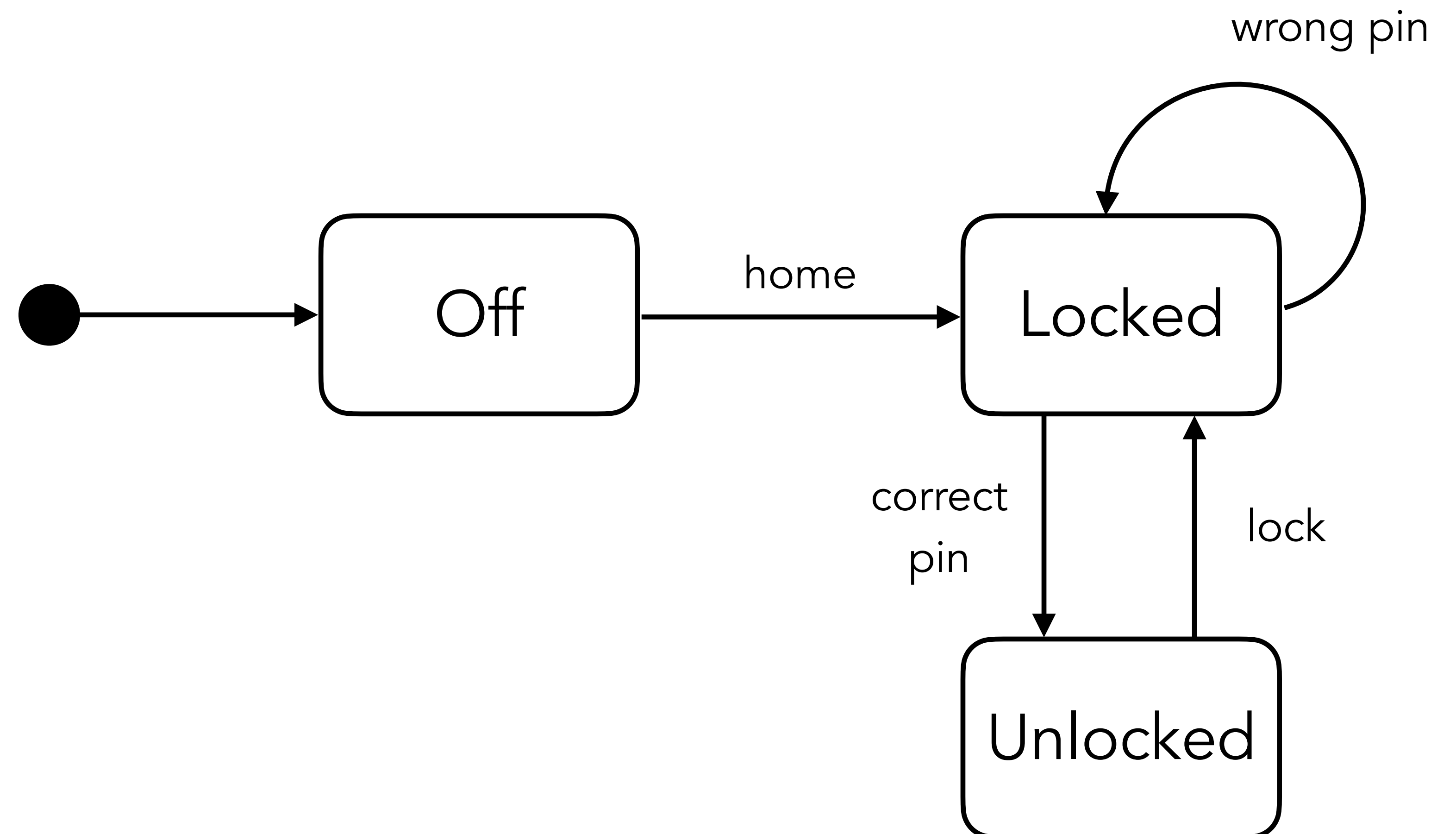
Now we have modeled two aspects of the behavior of using your phone: switching it on and unlocking it. We can combine these into a larger diagram.

Here we start with the unlocking state diagram.



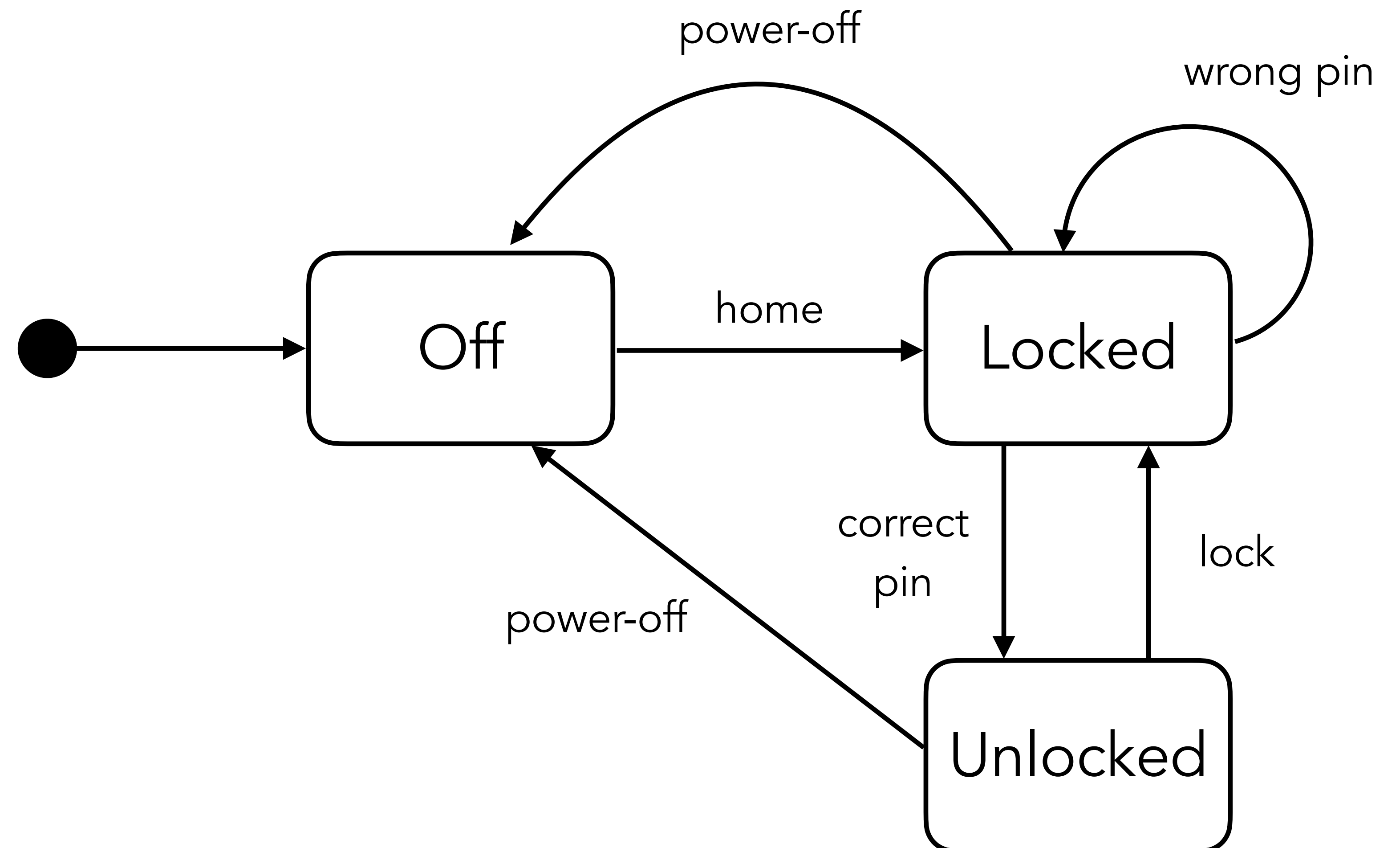
State Machines, on/off + unlocking

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



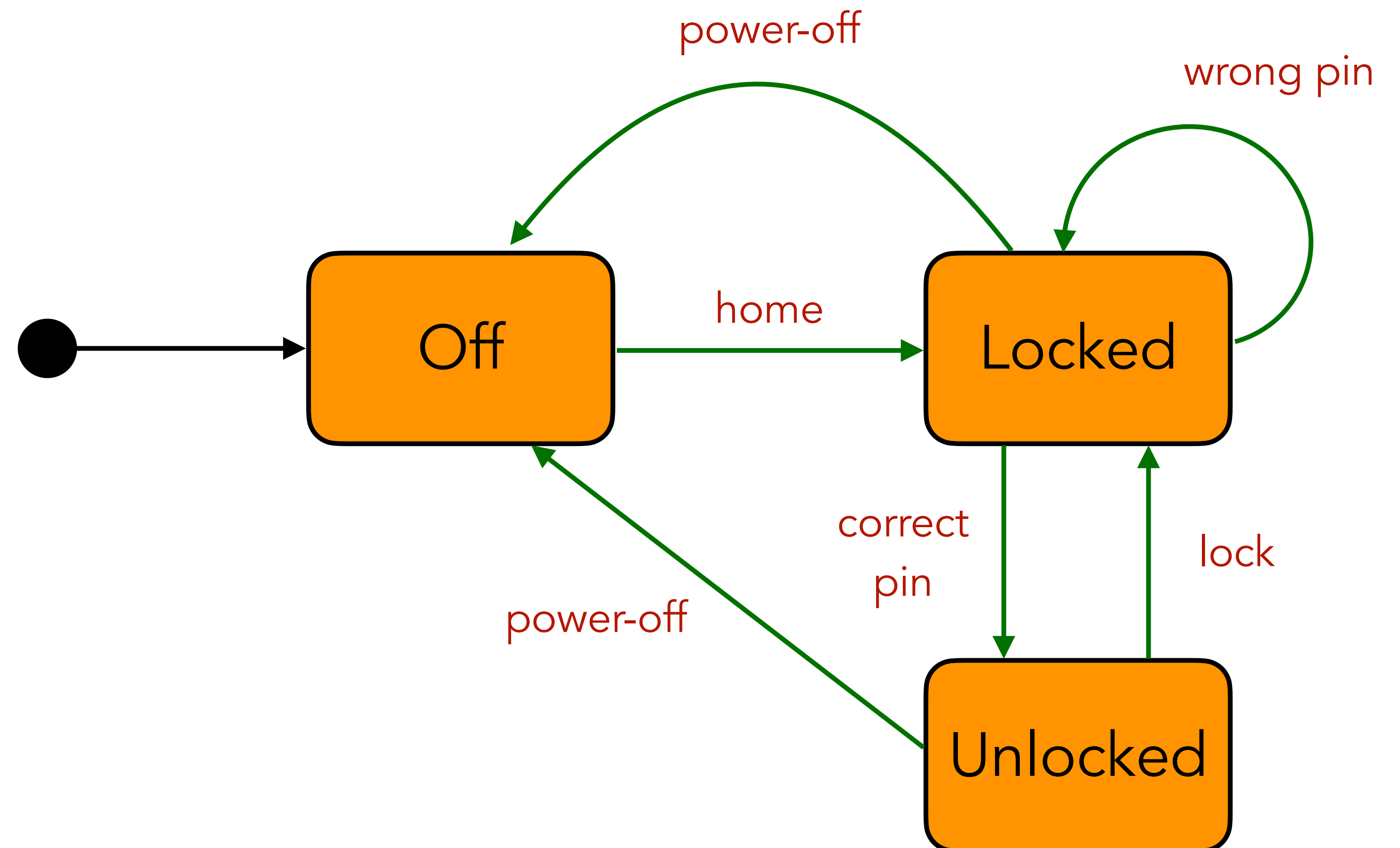
State Machines, on/off + unlocking

- 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.



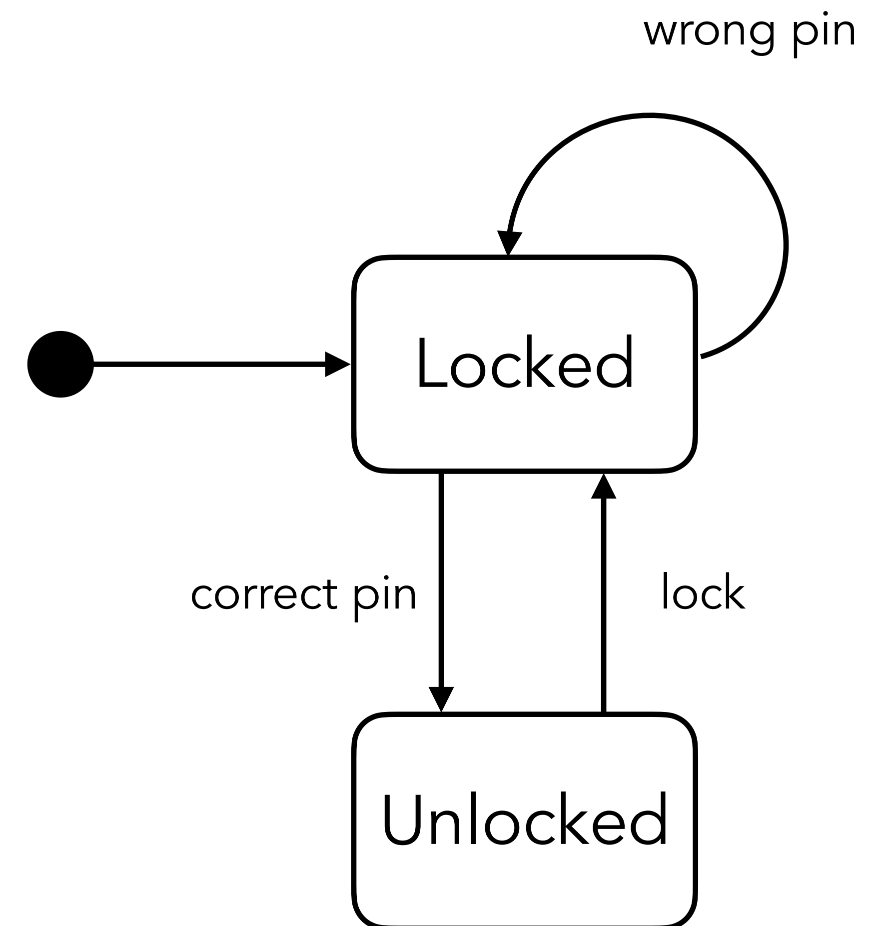
State Machines, on/off + unlocking

The diagram has now become a little more complex, with
- three states
- six transitions
- five event types



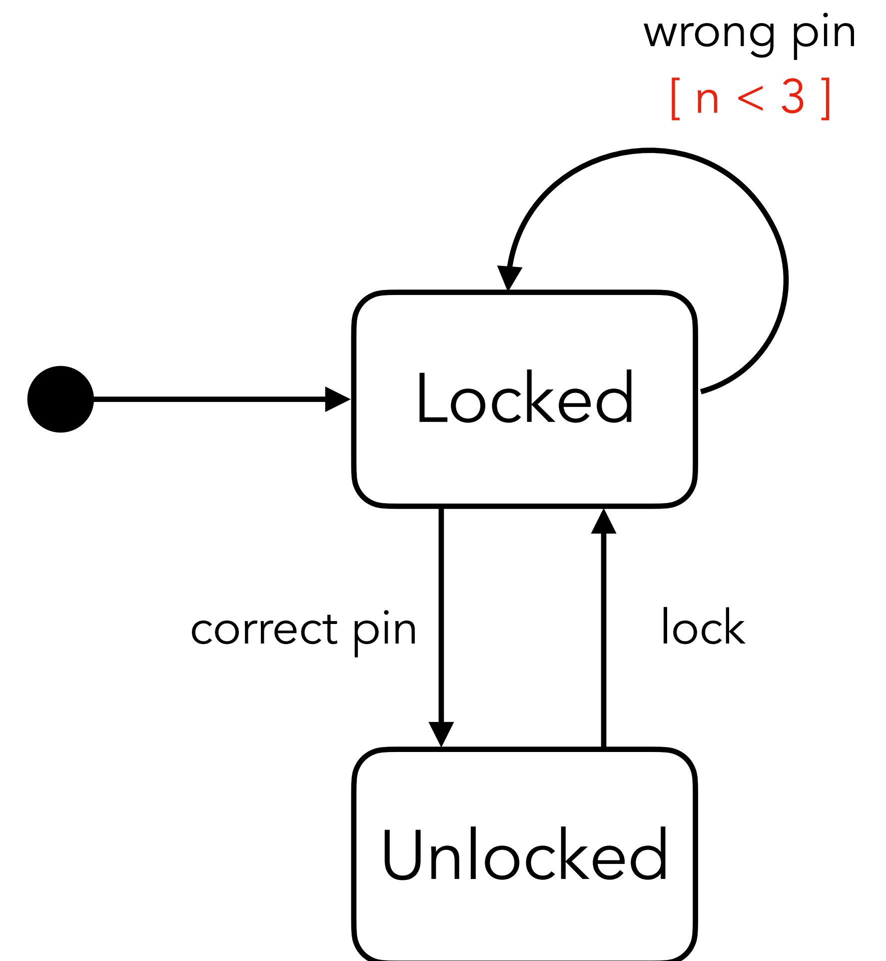
State Machines, conditional transitions

State diagrams also support **conditional transitions**.



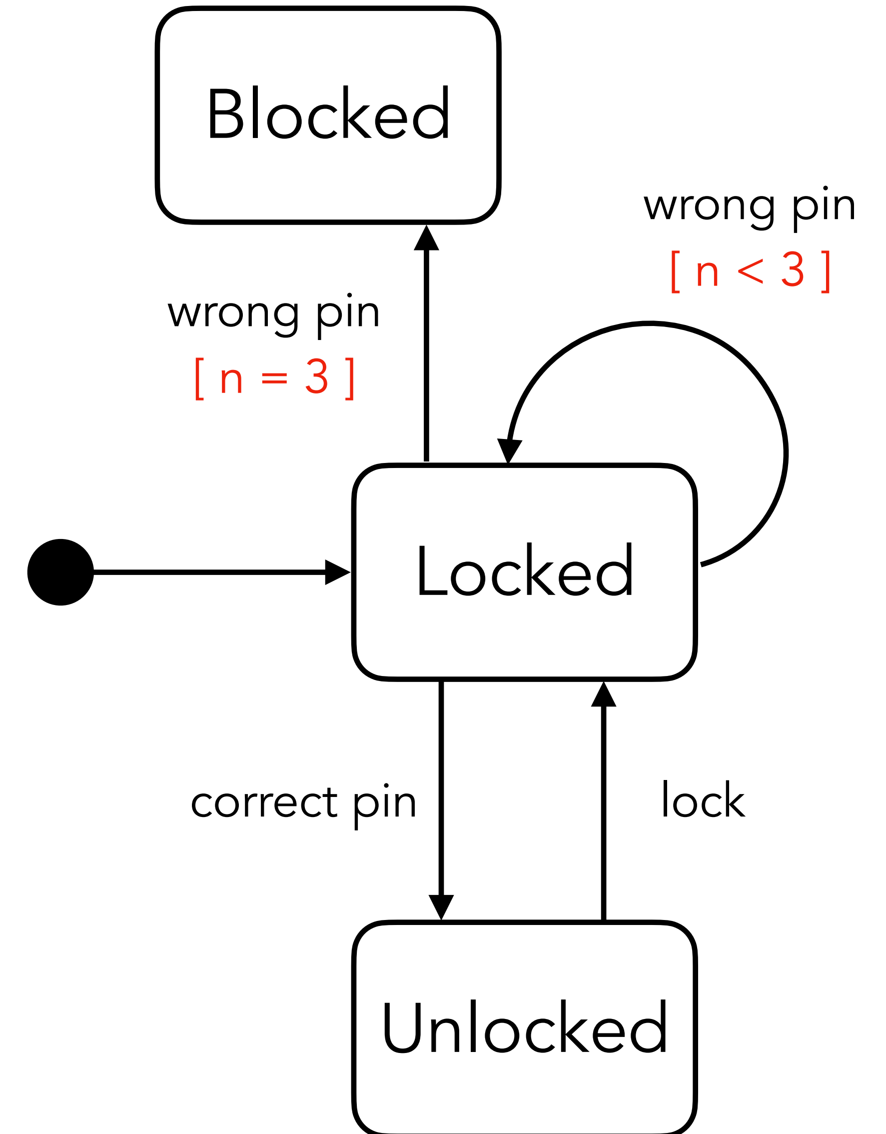
State Machines, conditional transitions

- 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

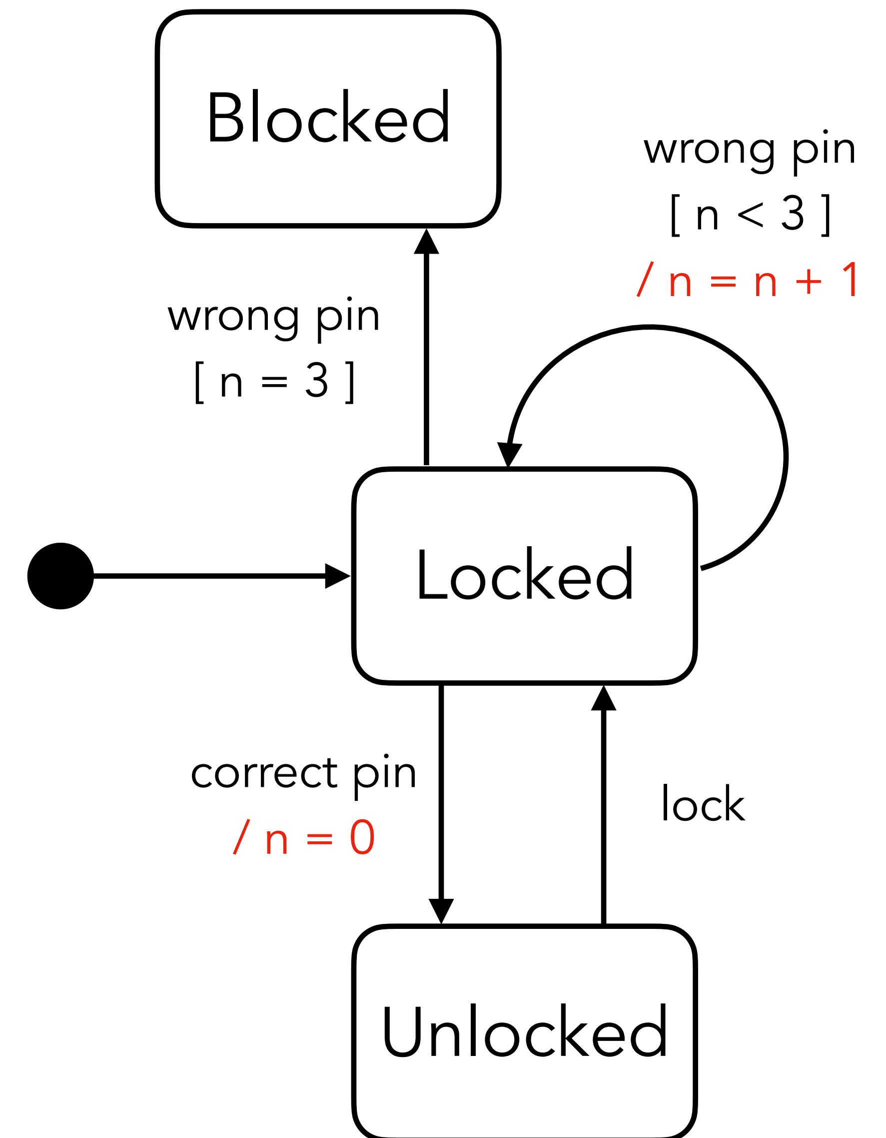
- After entering a wrong pin three times, the phone gets blocked, as indicated here. Thus, the “Locked” state has two outgoing edges both triggered by the “wrong pin” event, and the condition determines what the resulting state is.



State Machines, actions

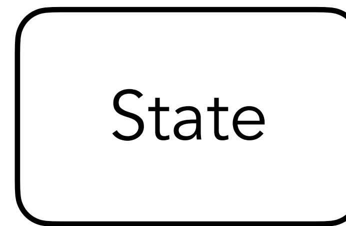
Yet another convention of the UML is that transitions can have "actions". These are denoted by a forward slash "/", followed by the action.

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.



UML State Diagram, recap

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



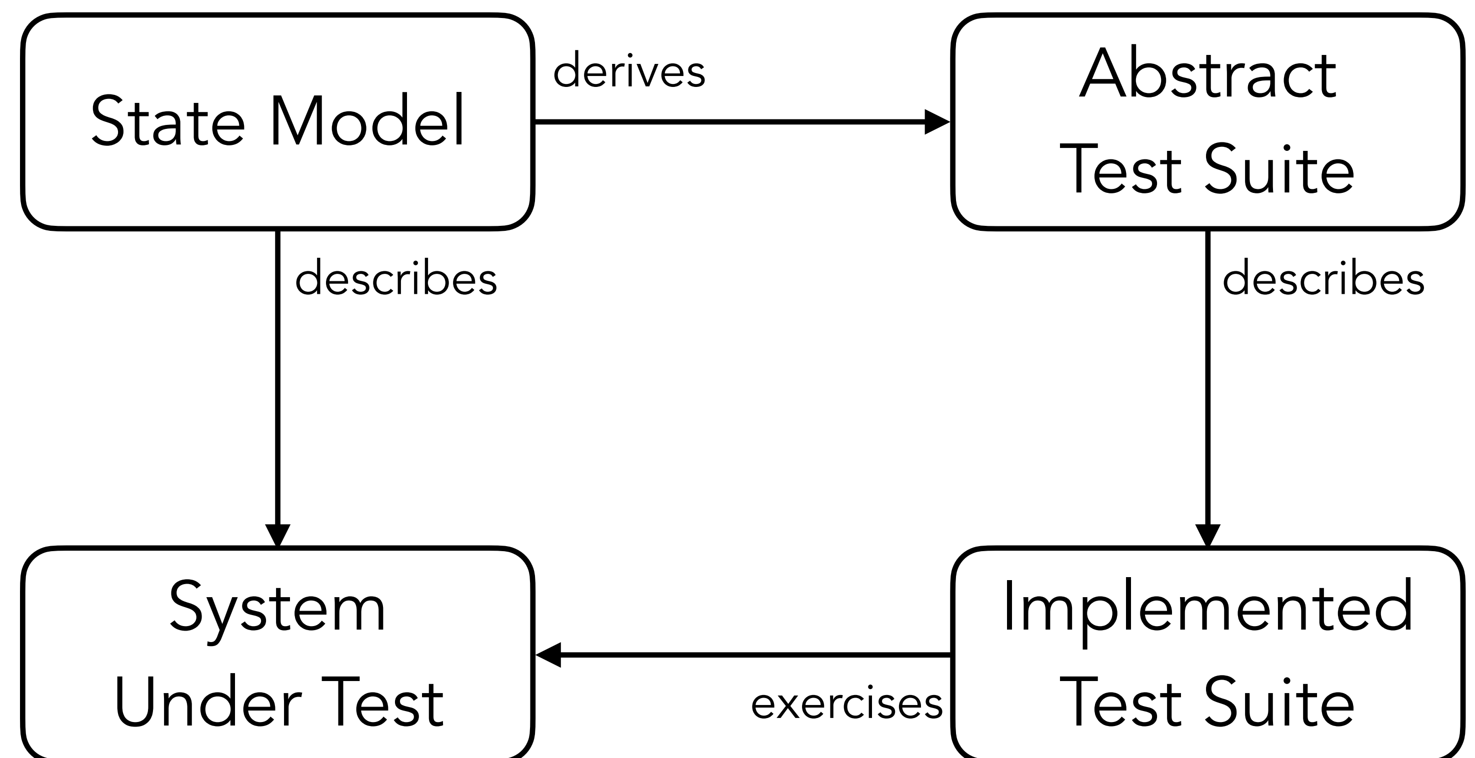
event

State-based Testing

The model describes the system under test.

From that model we can derive an abstract test suite, expressed in terms of paths through the diagram.

Once we have abstract test cases, we can implement them and actually exercise the system under test.



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, which just ensures that every state is reached once.

- **Transition coverage.** A bit stronger than state coverage, which insists that every transition is covered. Note that if this is the case, then automatically all states are covered as well.

- **Path coverage**, i.e., exercise sequences of transitions, which are paths through the state machine. Note that state machines very often have loops with an infinit number of possible paths. As it is impossible to take the loop as often as we want leading to more and more, and longer 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 the 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 actually takes place.

4. Assess that the system has indeed reached state S2.

Testing a sequence of transitions

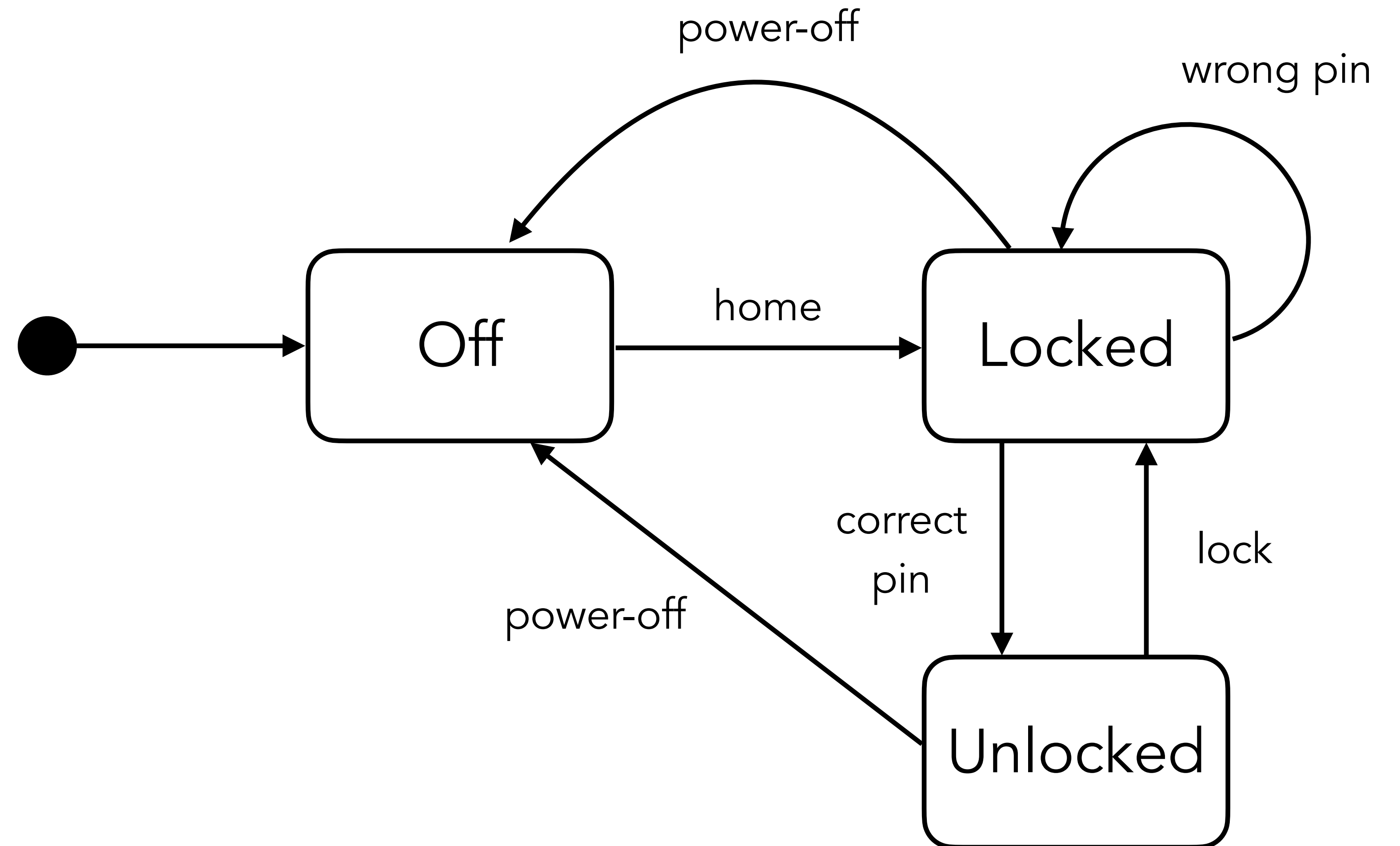
A full test case will typically cover a sequence of transitions, that is, it will exercise a **path** through the state machine.

As there are potentially infinitely many paths we will need some approach to decide which paths to exercise.

Our way to do that is by creating a transition tree, which spans the diagram.

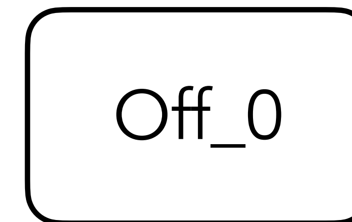
Transition tree

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



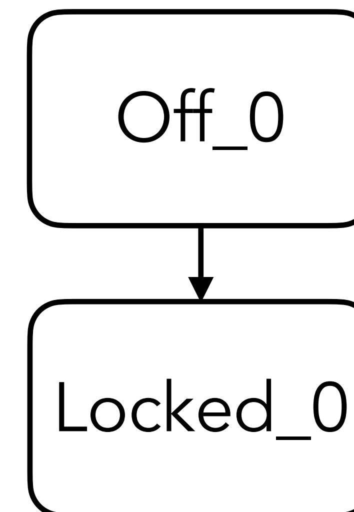
Transition tree

- 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.



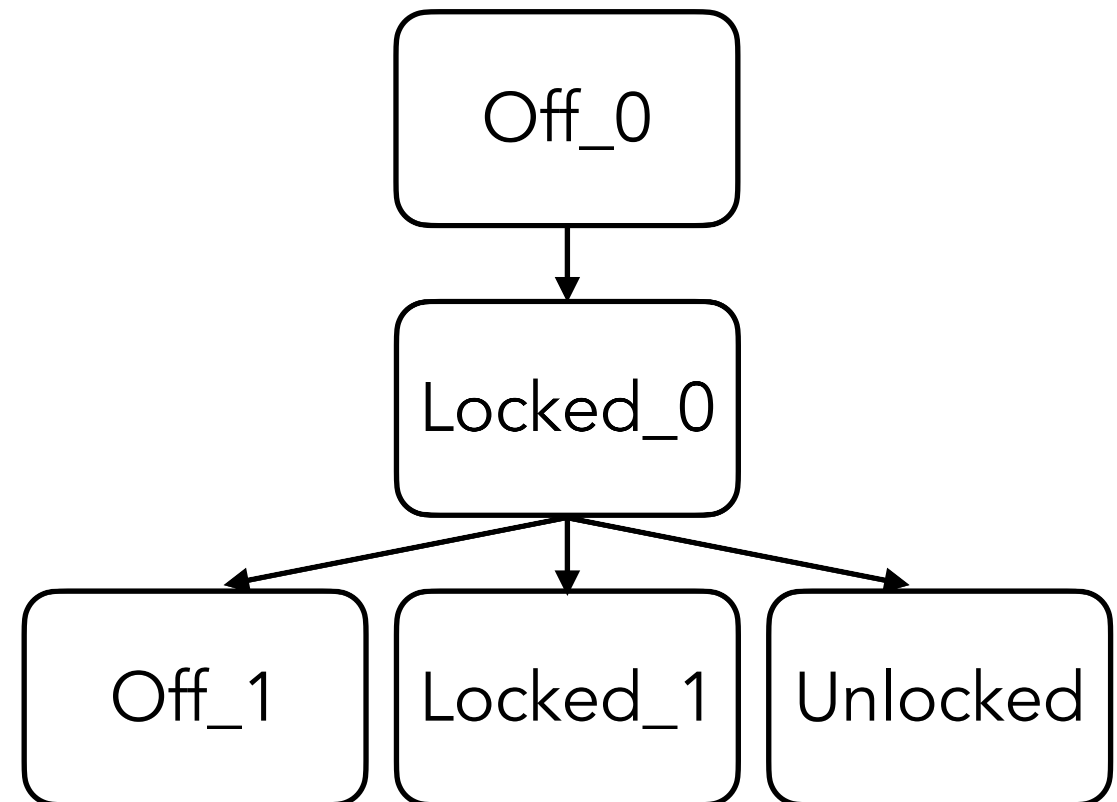
Transition tree

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



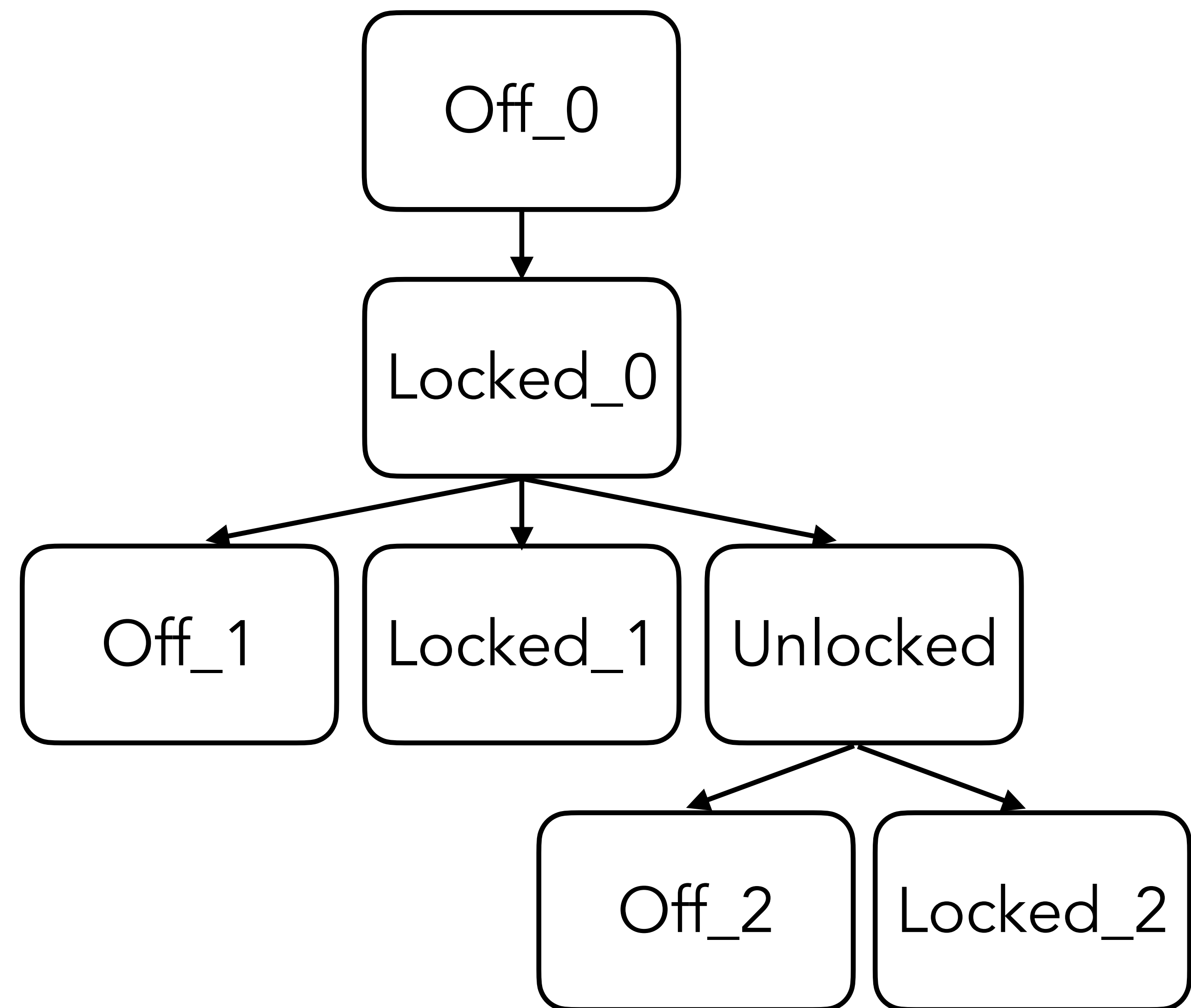
Transition tree

- From the “Locked” state, we have three outgoing edges, to “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.



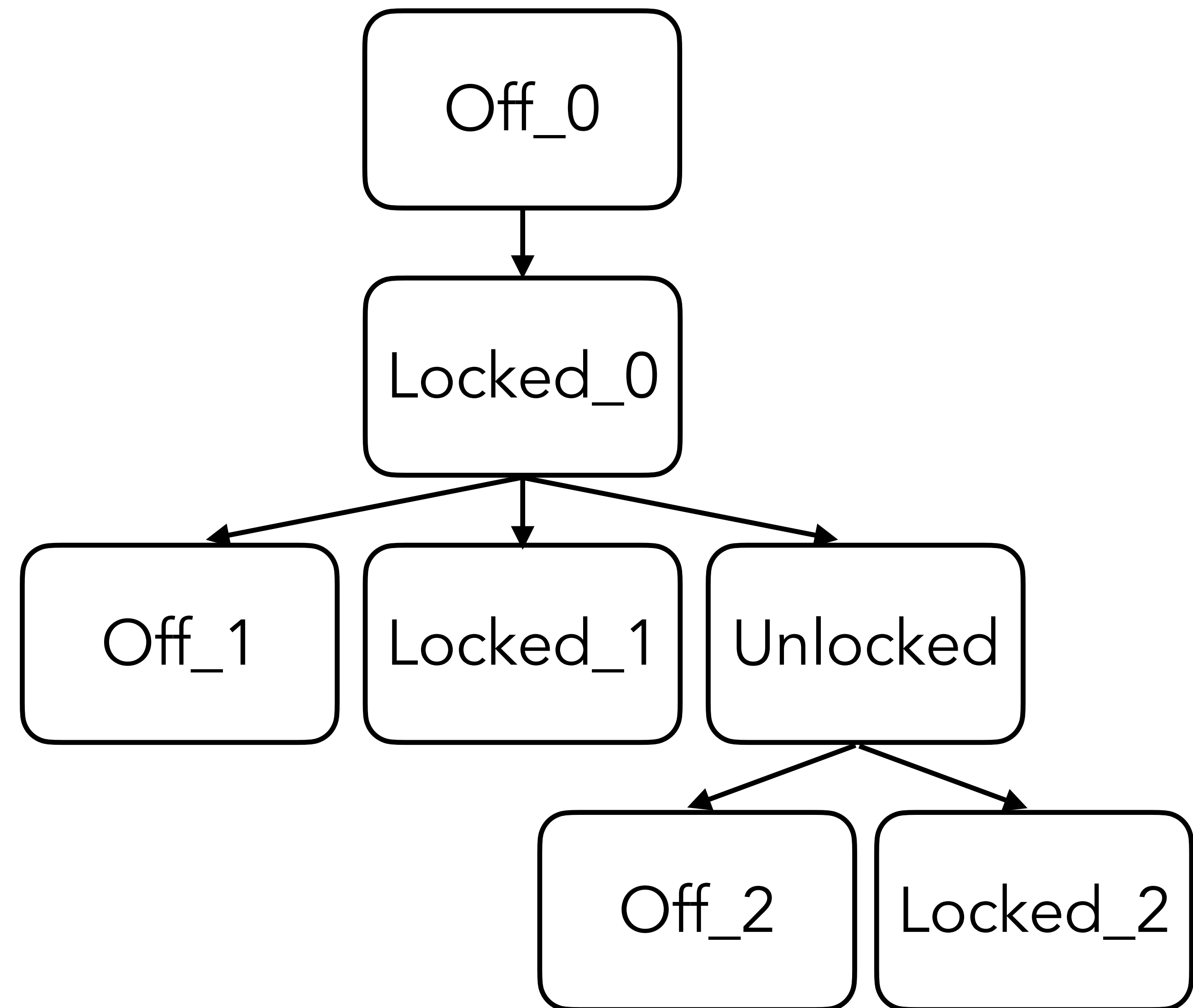
Transition tree

- From “Unlocked”, lastly, we can reach two different states, off and locked. Here again, the two-level states were described at the zero level, so we do not repeat their behavior.



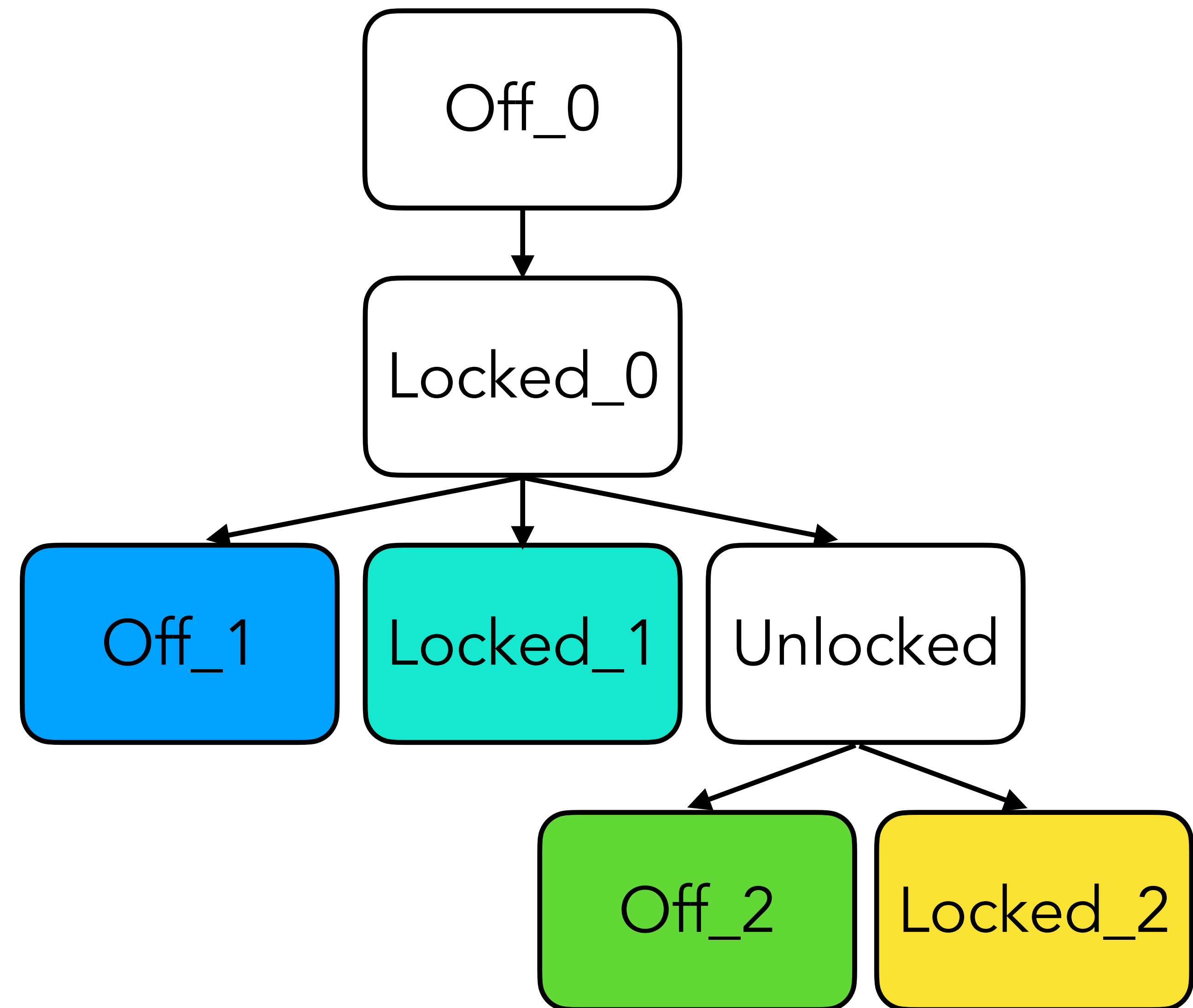
Transition tree

- 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.



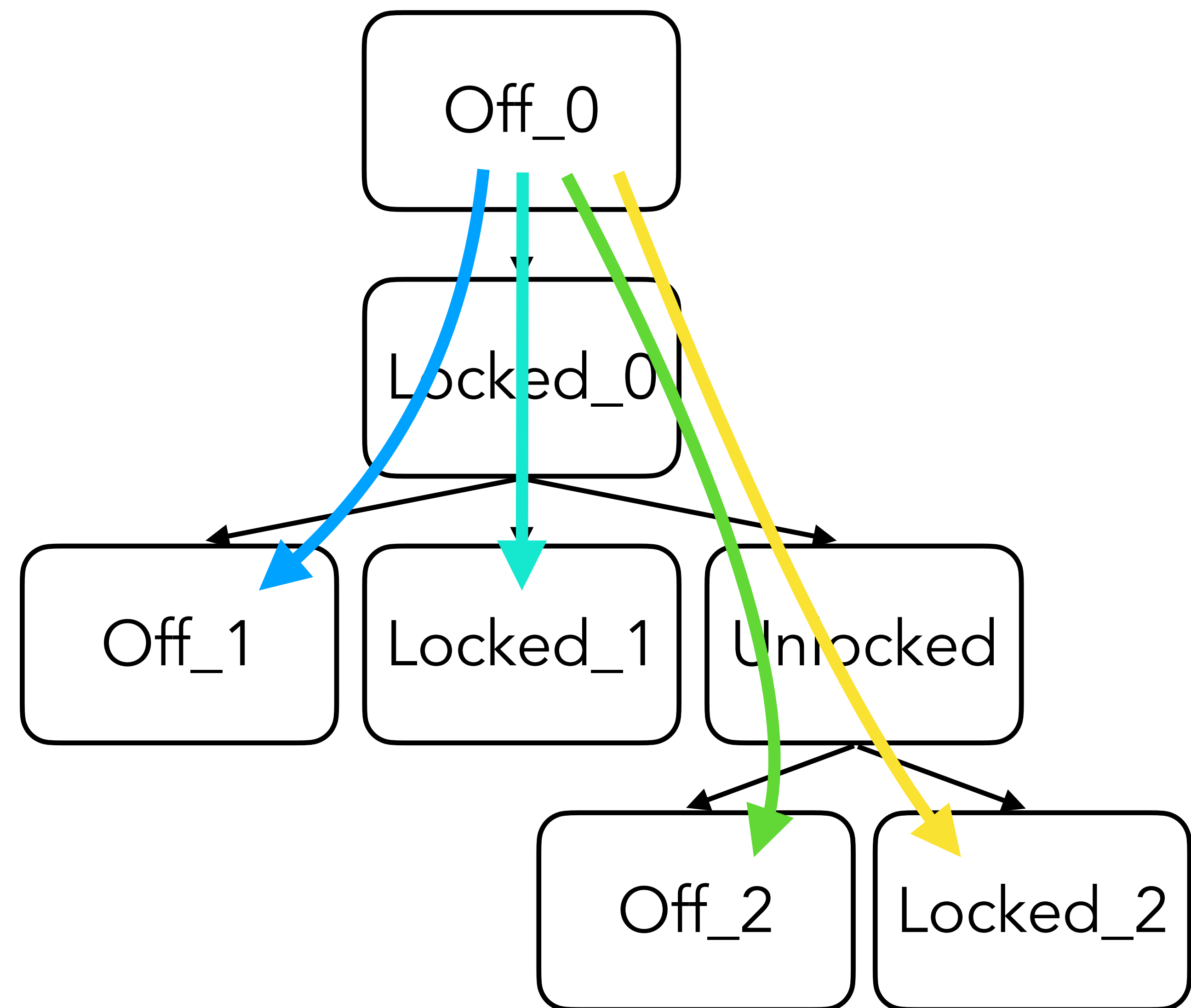
Tests from the transition tree

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.



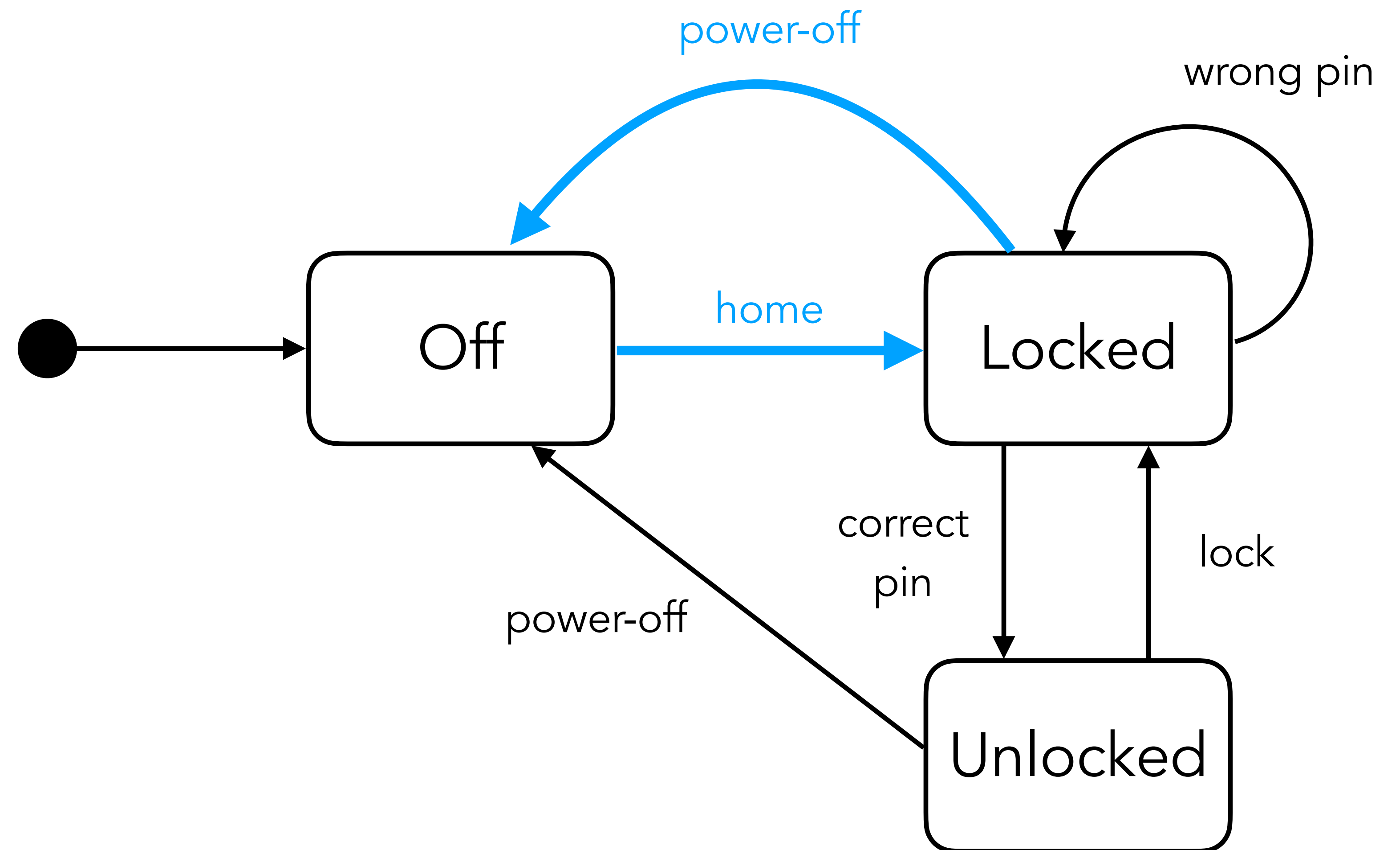
Tests from the transition tree

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 drawn in green.



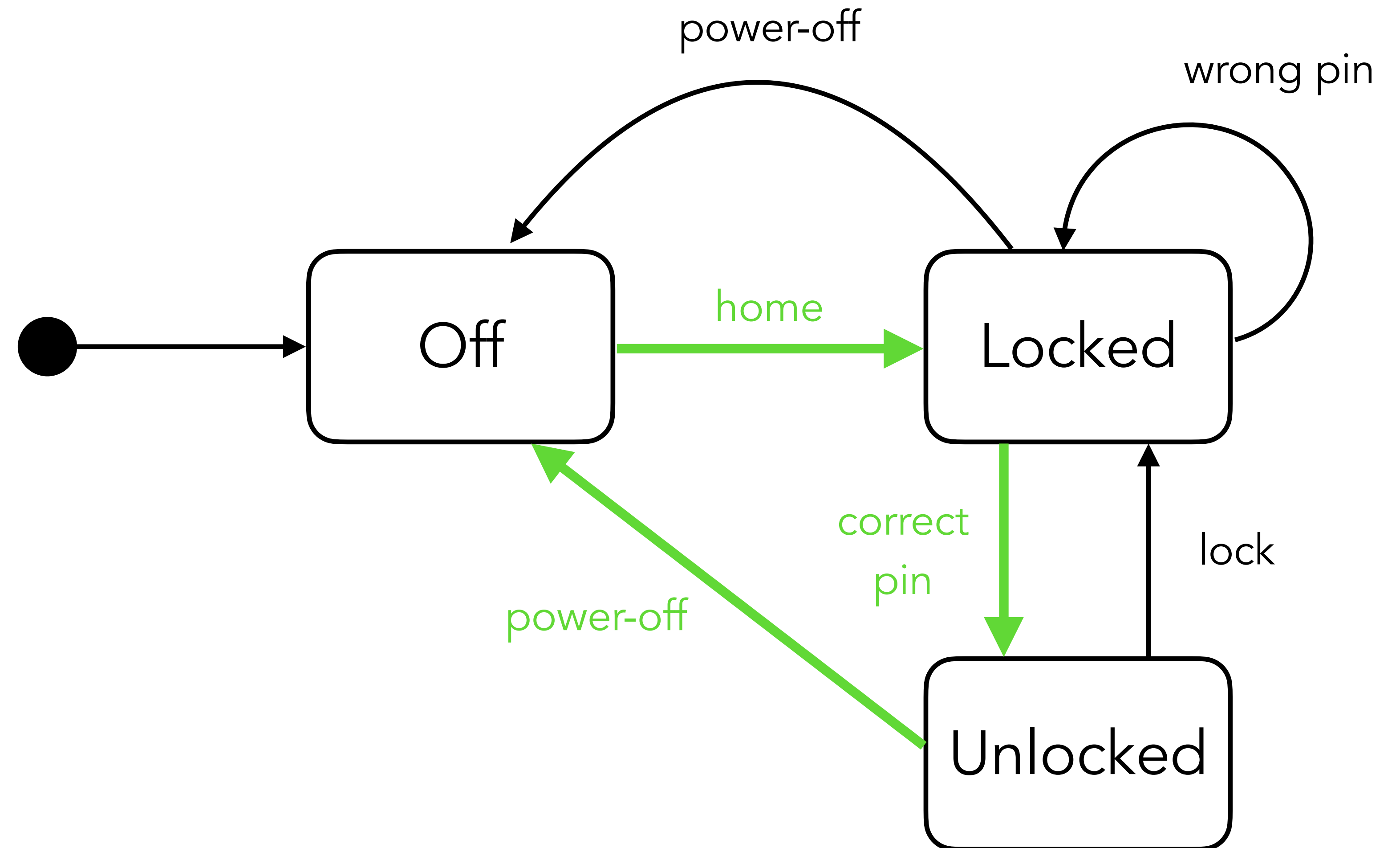
Tests from the transition tree

These paths can also be displayed in the state machine.
- For example, here is the **blue** path from "Off" to "Locked" and back.



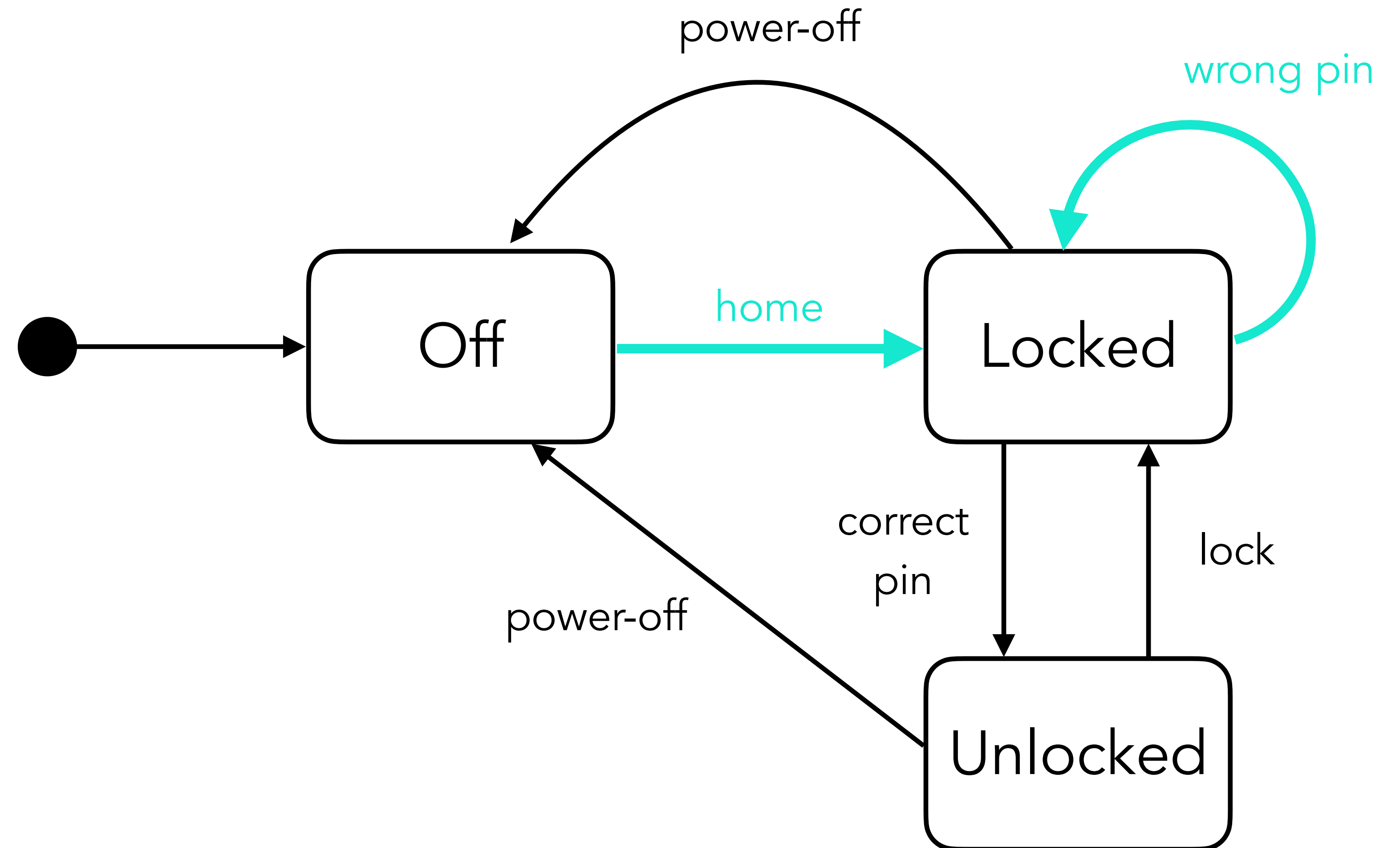
Tests from the transition tree

Another example, the **green** path goes from “Off”, via “Locked” and “Unlocked” back to “Off”.



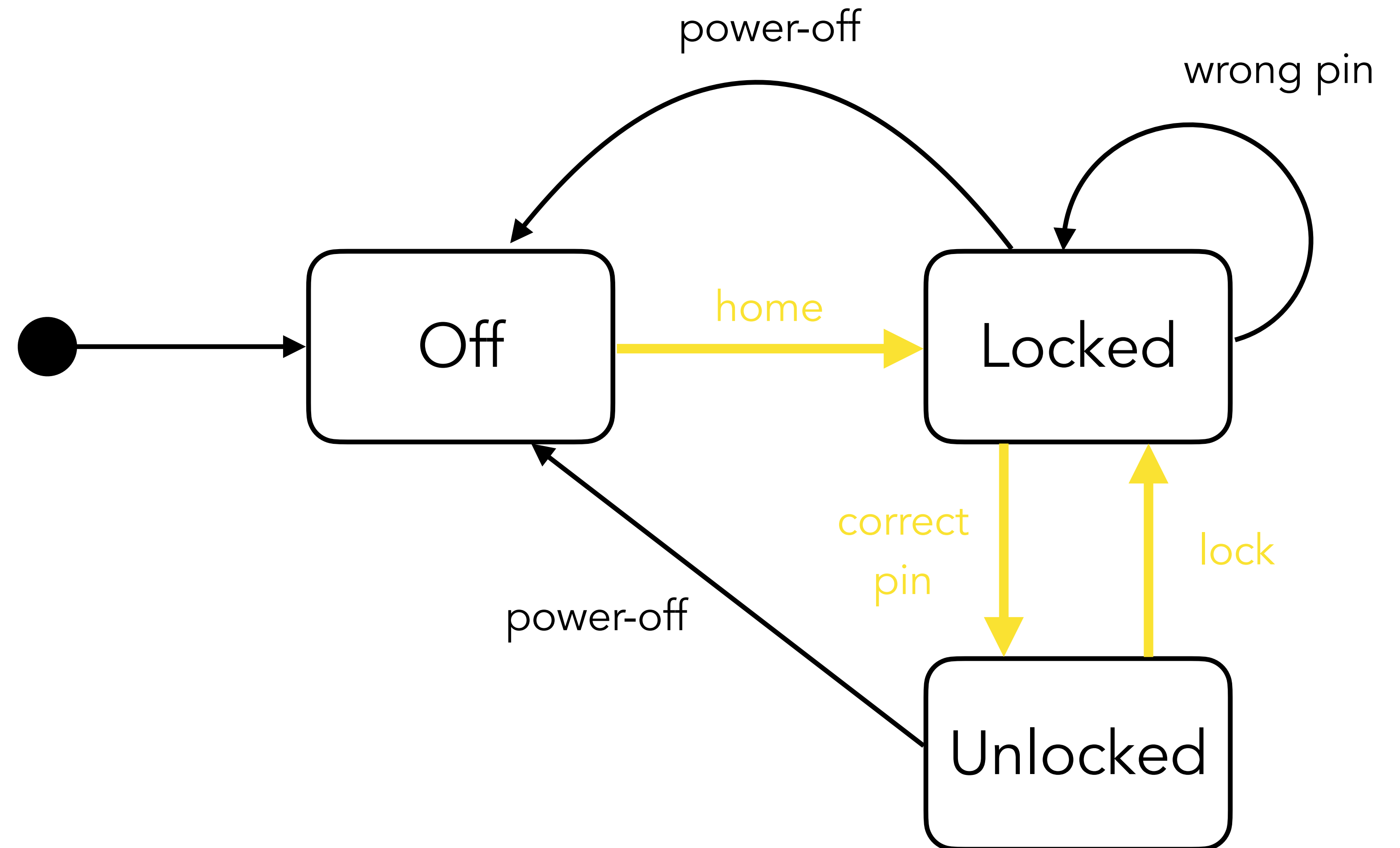
Tests from the transition tree

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



Tests from the transition tree

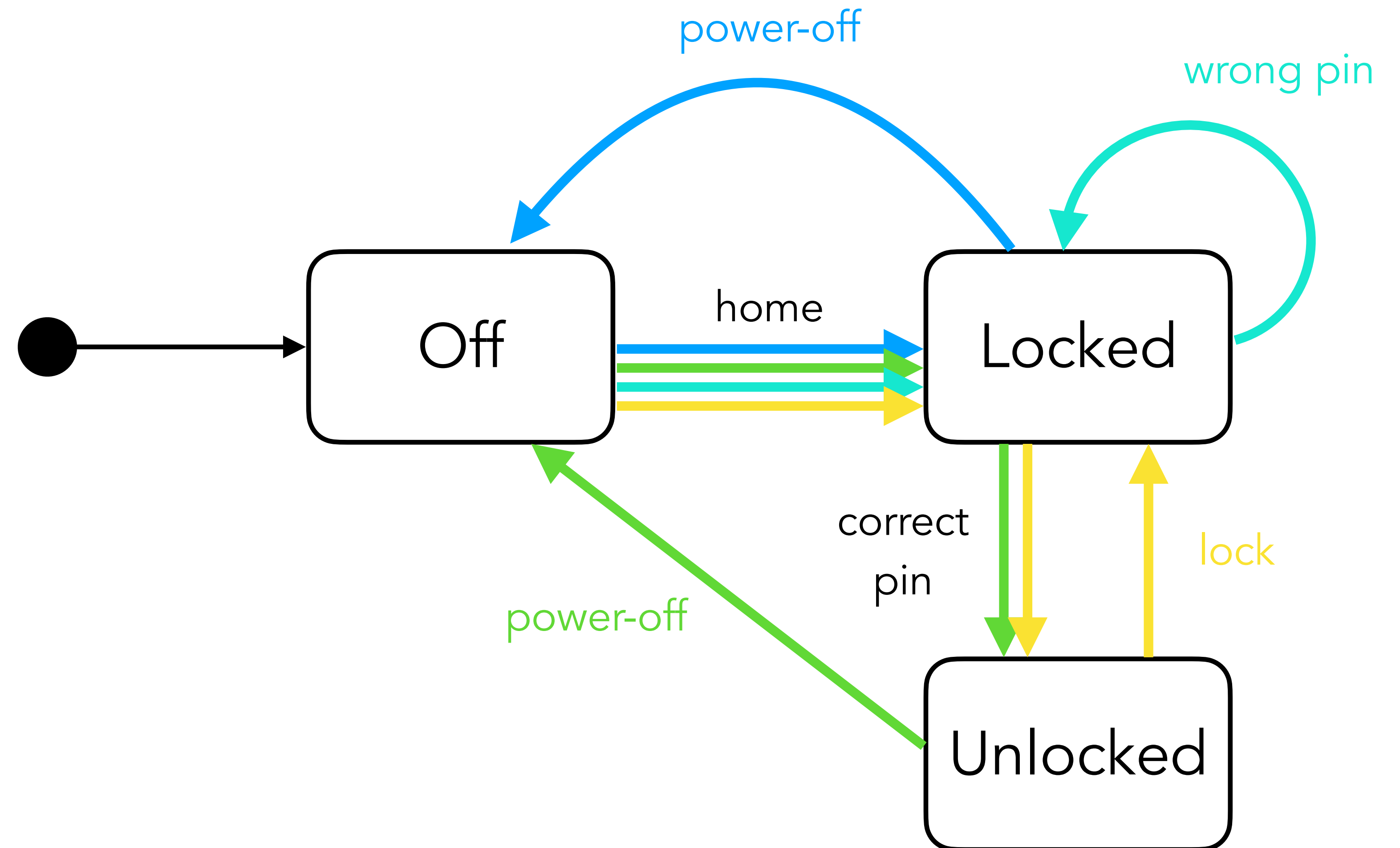
Another example, the yellow path goes from “Off” to “Unlocked” troughth the “Locked” state.



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.

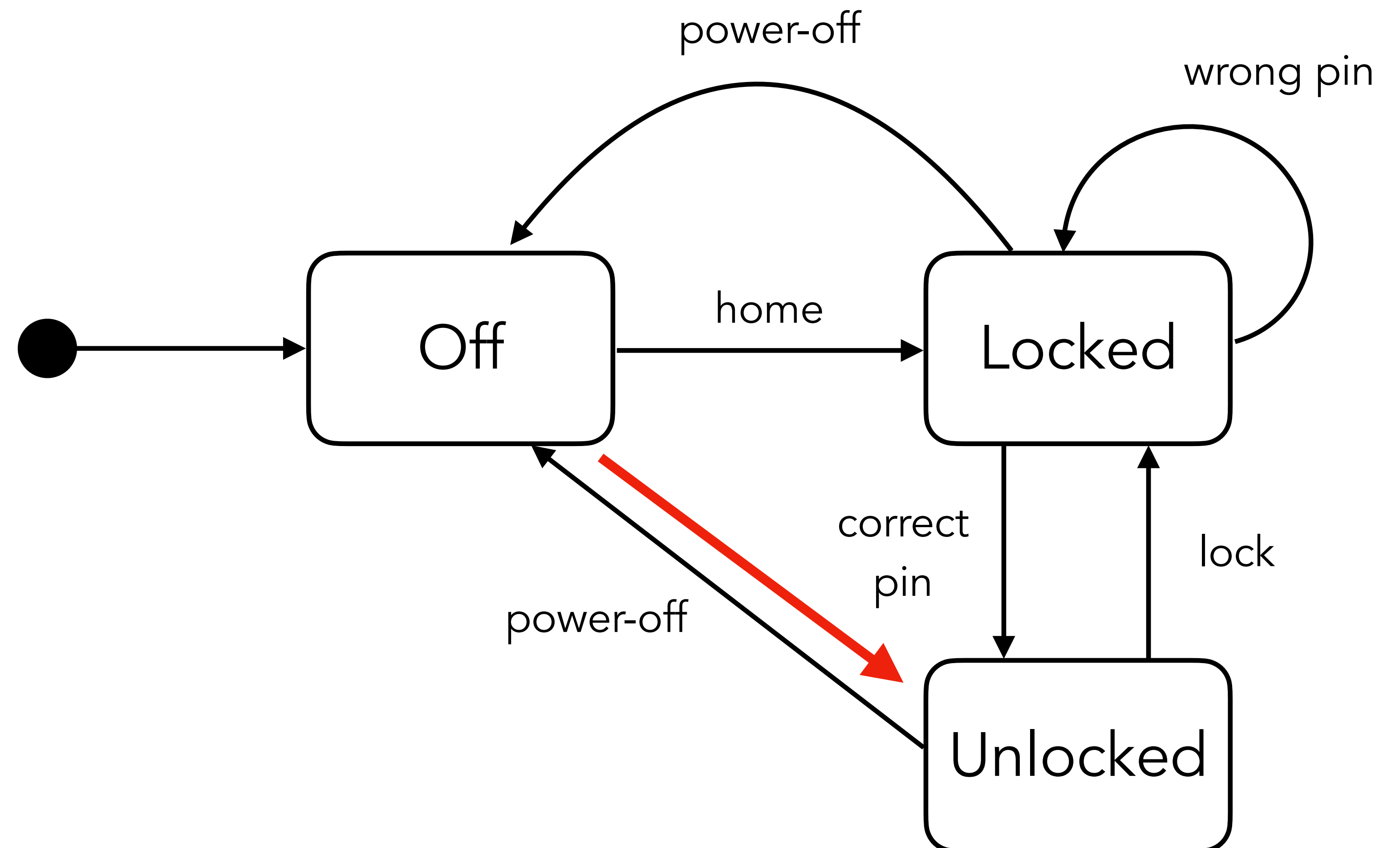
With this, we test that the system under test implements all specified behavior. In other words, we test that the implementation conforms to the model.



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		

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		

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 behavior for empty cells. A common option is to simply 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 an educated 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 in a particular state, for example the “Locked” state. To bring the system in a particular state we can again make use of the transition tree. 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.

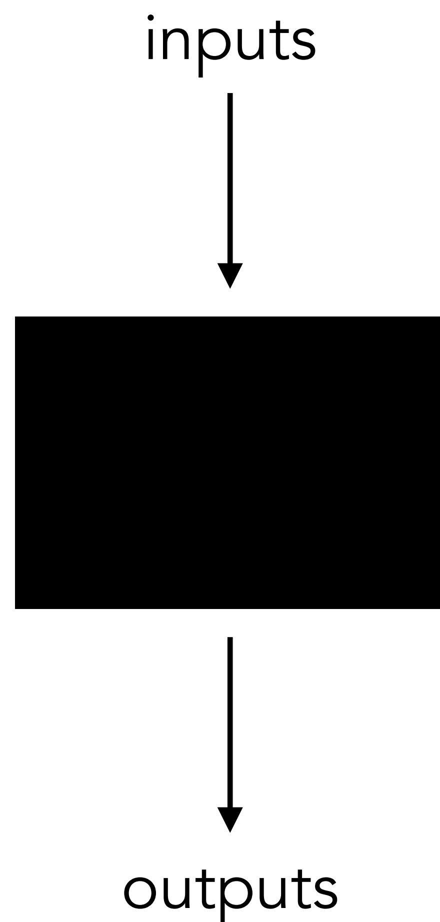
Sneak path testing

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).

Black-box testing: which one?



Techniques

Equivalence class partitioning
Category partition
Boundary value analysis
Random testing
Model-Based Testing
Cause effect graphing
Error guessing
…

Black-box testing: which one?

Always use a combination of techniques.

- Identify valid and invalid input equivalence classes.
- Identify output equivalence classes.
- Apply boundary value analysis.
- When a formal specification is available try to use it to further improve your tests.
- Apply model-based testing.
- Guess about possible errors.
- …

References

- Gordon Fraser and José Miguel Rojas; Software Testing, 2019. ISBN 978-3-030-00262-6
- Chapter 4, Foundations of Software Testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.
- Chapter 14, Software Testing and Analysis Process, Principles and Techniques, 2008. Mauro Pezze, Michal Young.
- Chapter 12, Software Testing A Craftsman's Approach, 2014. Jorgensen, Paul C.
- Practical Model-Based Testing A Tools Approach, 2007. Mark Utting, Bruno Legeard.