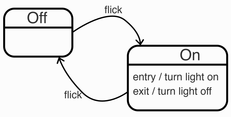
Welcome to the world of Statecharts

What is a statechart?

A statechart can be explained in many ways, and we'll get to those explanations, but essentially, a statechart is a drawing. Here's a simple statechart:



However, this drawing isn't very useful for software engineers who want to reap the benefits outlined elsewhere on this site, so let's dive into some other ways of describing what a statechart is. The original paper that defines statecharts bills them as "A visual formalism for complex systems" (Harel, 1987). With that out of the way, let's try to explain statecharts.

Introduction to statecharts

Put simply, a statechart is a beefed up state machine. The beefing up solves a lot of the problems that state machines have, especially state explosion that happens as state machines grow. One of the goals of this site is to help explain what statecharts are and how they are useful.

- What is a state machine?
- What is a statechart?

Why should you use statecharts?

Statecharts offer a surprising array of benefits

- It's easier to understand a statechart than many other forms of code.
- The behaviour is decoupled from the component in question.
  - This makes it easier to make changes to the behaviour.
  - It also makes it easier to reason about the code.
  - And the behaviour can be tested independently of the component.
- The process of building a statechart causes all the states to be explored.
- Studies have shown that statechart based code has lower bug counts than traditional code.
- Statecharts lends itself to dealing with exceptional situations that might otherwise be overlooked.
- As complexity grows, statecharts scale well.
- A statechart is a great communicator: Non-developers can understand the statecharts, while QA can use a statecharts as an exploratory tool.

It's worth noting that you're already coding state machines, except that they're hidden in the code.

Why should you not use statecharts?

There are a few downsides to using statecharts that you should be aware of.

- Programmers typically need to learn something new, although the underpinnings (state machines) would be something that most programmers are familiar with.
- It's usually a very foreign way of coding, so teams might experience pushback based on how very different it is.
- There is an overhead to extracting the behaviour in that the number of lines of code might increase with smaller statecharts.

Why are they not used?

- People don't know about them, and YAGNI.

What are the main arguments against statecharts?

There are a few common arguments against statecharts in addition to the ones listed above:

- It's simply not needed.
- It goes against the grain of *[insert name of technology]*.
- It increases the number of libraries, for web applications this means increased load time.

The benefits outlined above should make it clear that the introduction of statecharts is generally a *net positive*.

How do you use statecharts?

First of all, know that a W3C committee spent 10+ years (2005 to 2015) standardizing something called *SCXML* (yes, Statechart XML), and that it defines a lot of the semantics and specifies how to deal with certain edge cases. There are tools to read, author and even execute statecharts written in SCXML, in various languages. There are also some derivatives that support the same *model* as SCXML, but using a different syntax.

Additionally, there are statechart libaries for a variety of platforms, that in varying degrees support the semantics described by SCXML. You should consider using these libraries just to get those edge cases taken care of. The libraries generally perform entry and exit actions *in the right order* and so on.

With that out of the way, read on!

Executable statecharts

In addition to just using statecharts to model the behaviour in documents separate from the actual running code, it's possible to use one of various machine formats, both to design the behaviour, and at run-time to actually *be* the behaviour. The idea is to have a single source of truth that describes the behaviour of a component, and that this single source drives both the actual run-time code, but that it can also be used to generate a precise diagram that visualises the statechart.

This carries along some different pros and cons:

Why should you use executable statecharts?

- No need to translate diagrams into code
- No bugs introduced by hand translation of diagrams
- The diagrams are always in sync
- The diagrams are more precise

Why should you not use executable statecharts?

- The diagrams may become quite complex
- The format and tools for executable statecharts is limited
- Type safety between statechart and the component is hard to enforce

How do you use executable statecharts?

In essence, if you have any definition of a statechart in your code, all you need to do is to take that representation and automate the generation of the visual statechart. This is of course simpler when the definition is in a separate file, e.g. in a JSON or XML file.

This is all explained on the page on how to use statecharts!

Anything else?

If you feel like chatting to someone about statecharts, you should go to spectrum.chat/statecharts, where you'll find a community of like minded developers that can help you understand and reap the benefits of using Statecharts.

Quite a few people have written books or held presentations that deal with statecharts in various ways, and they're included in our resources page.

There are some pages that haven't found any place in the web of documents, so they're honourably mentioned here:

- Use case: Statecharts in User Interfaces
- Concepts — The most important concepts in a statechart and what they look like in a diagram.
- Glossary — A list of terms that get thrown around when talking about statecharts, with their definitions.
- FizzBuzz — FizzBuzz is a well known problem, and it's been used as a backdrop to explain various statechart concepts.

Acknowledgements