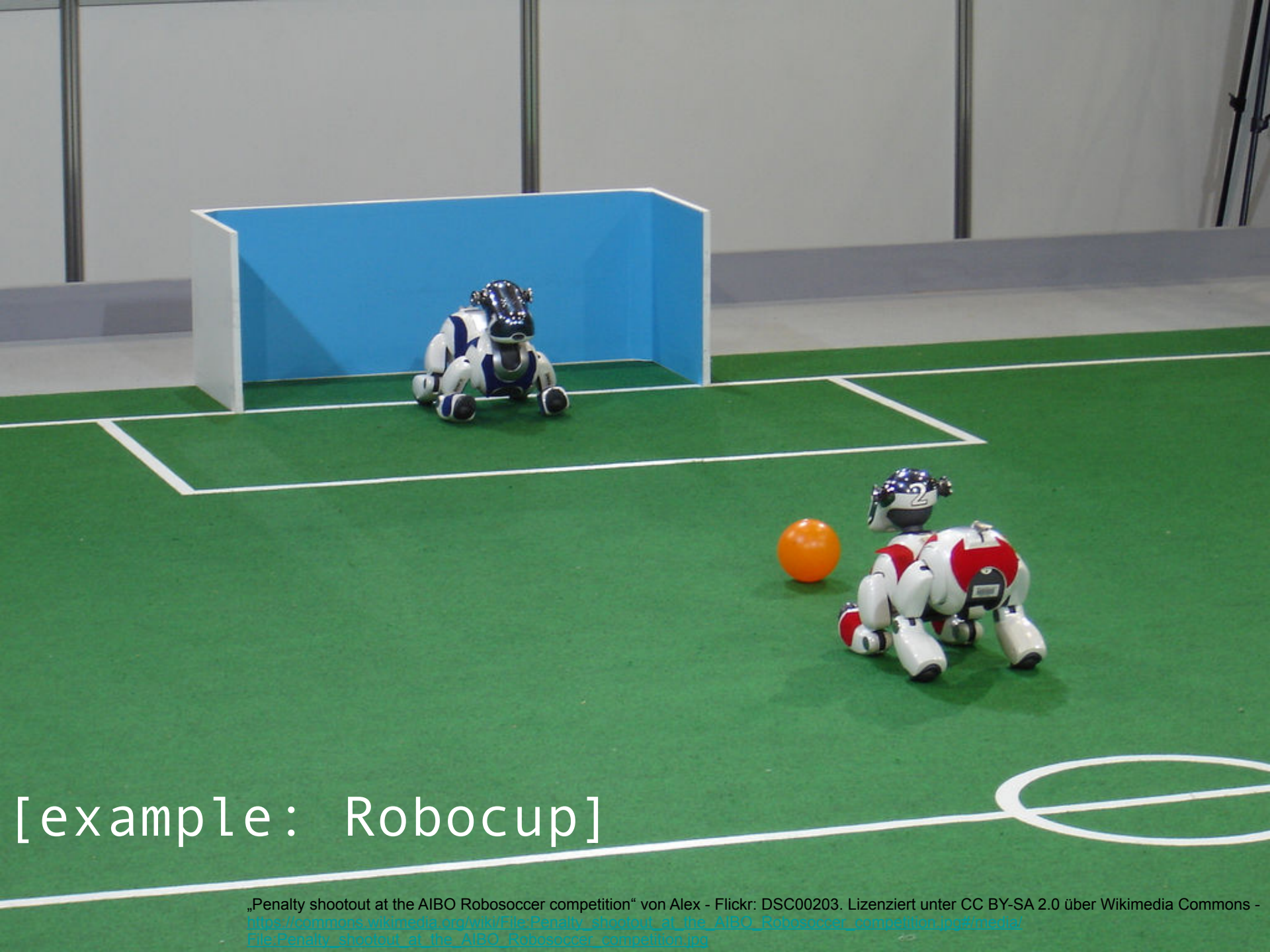
Anthony Anjorin

MODELLING REACTIVE / EVENT-DRIVEN SYSTEMS WITH STATE MACHINES



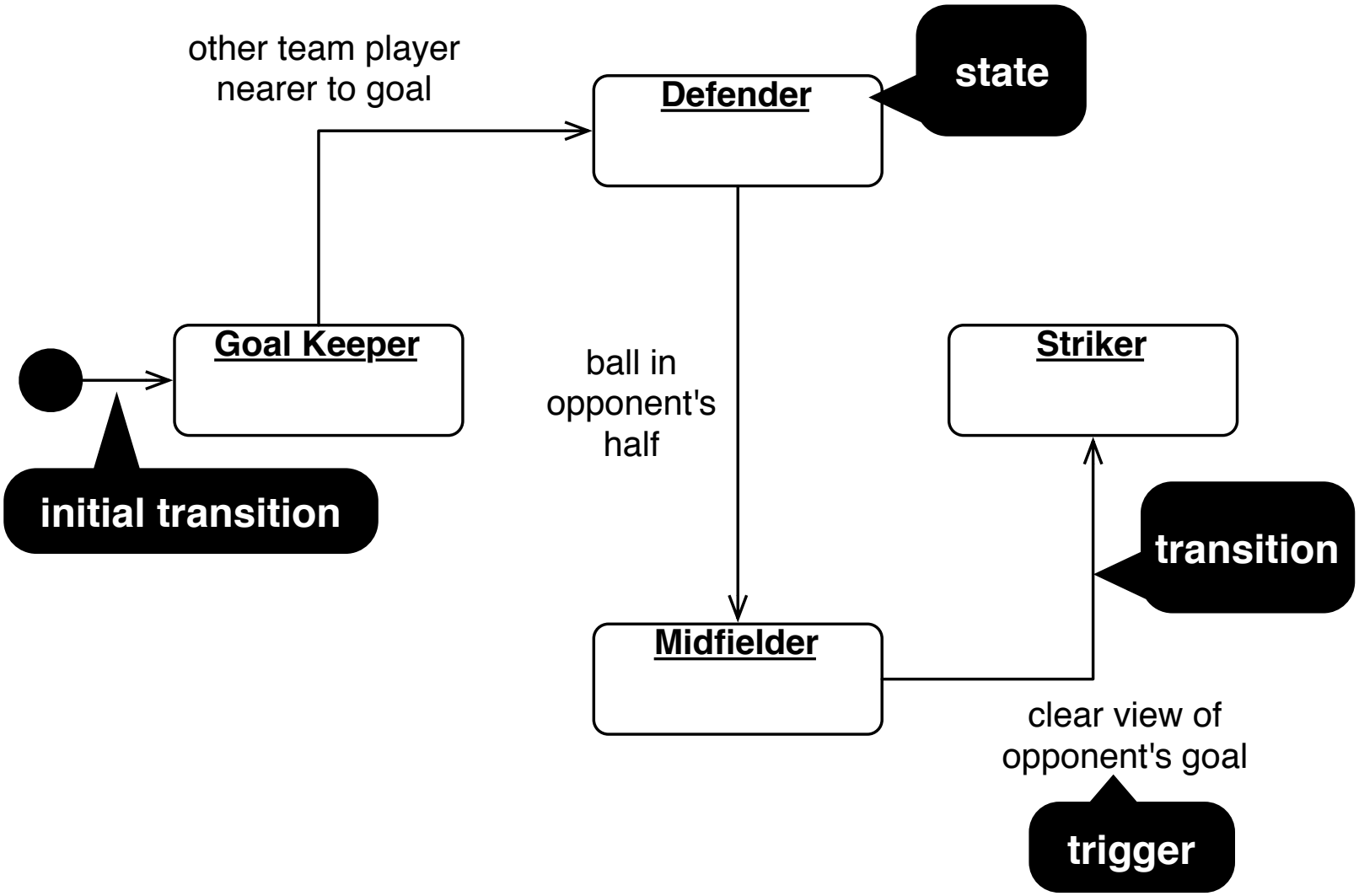
Reactive / Event-Driven System

current role (or “state”)

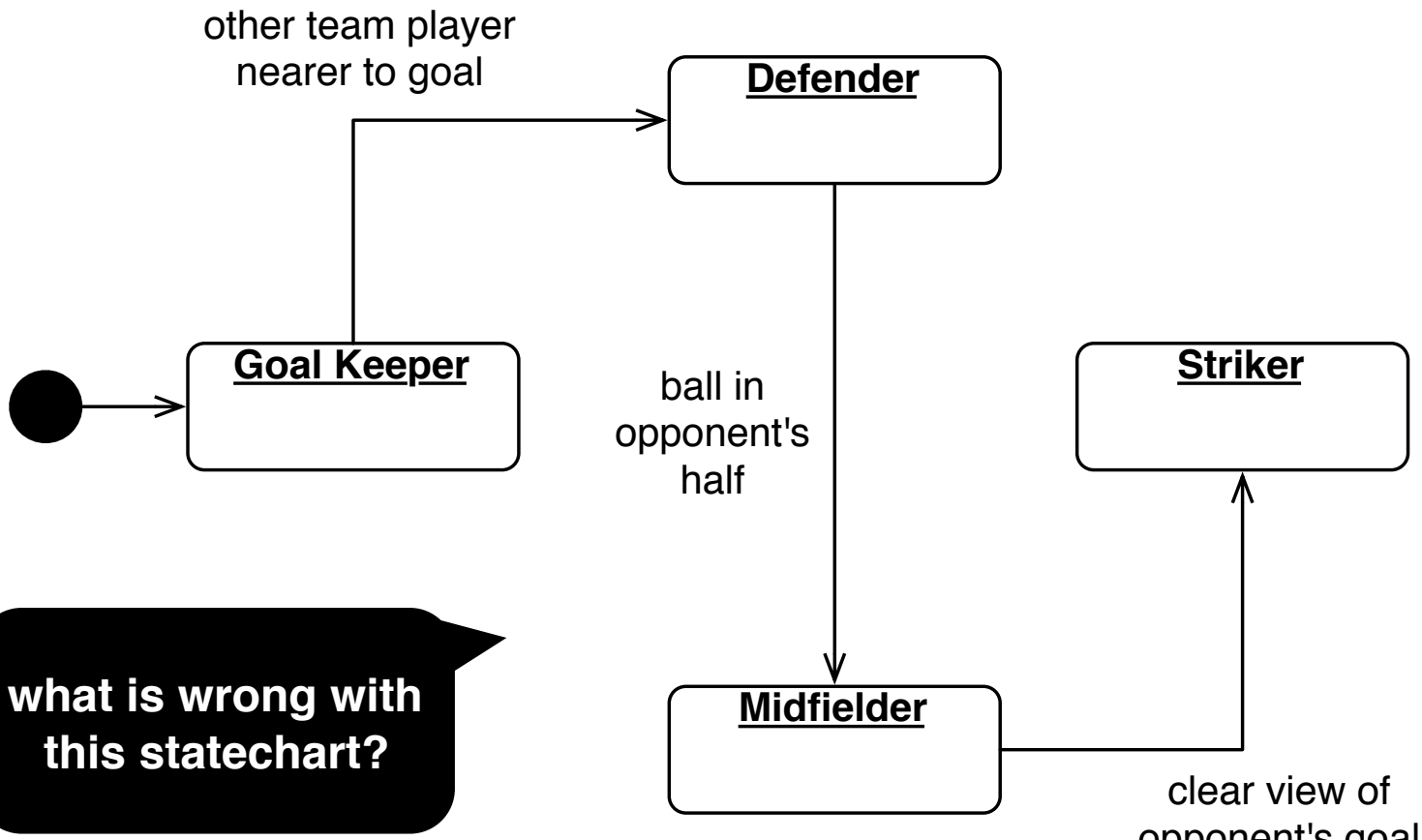
event	Goal Keeper	Defender	Striker	…
got the ball	clear with a long shot	pass	dribble	
lost the ball	intercept shot	chase	wait for pass	
clear view of opponents goal	…	…	score!	
ball in opponents half	…	…	chase	
…	…		…	

how can this be implemented?

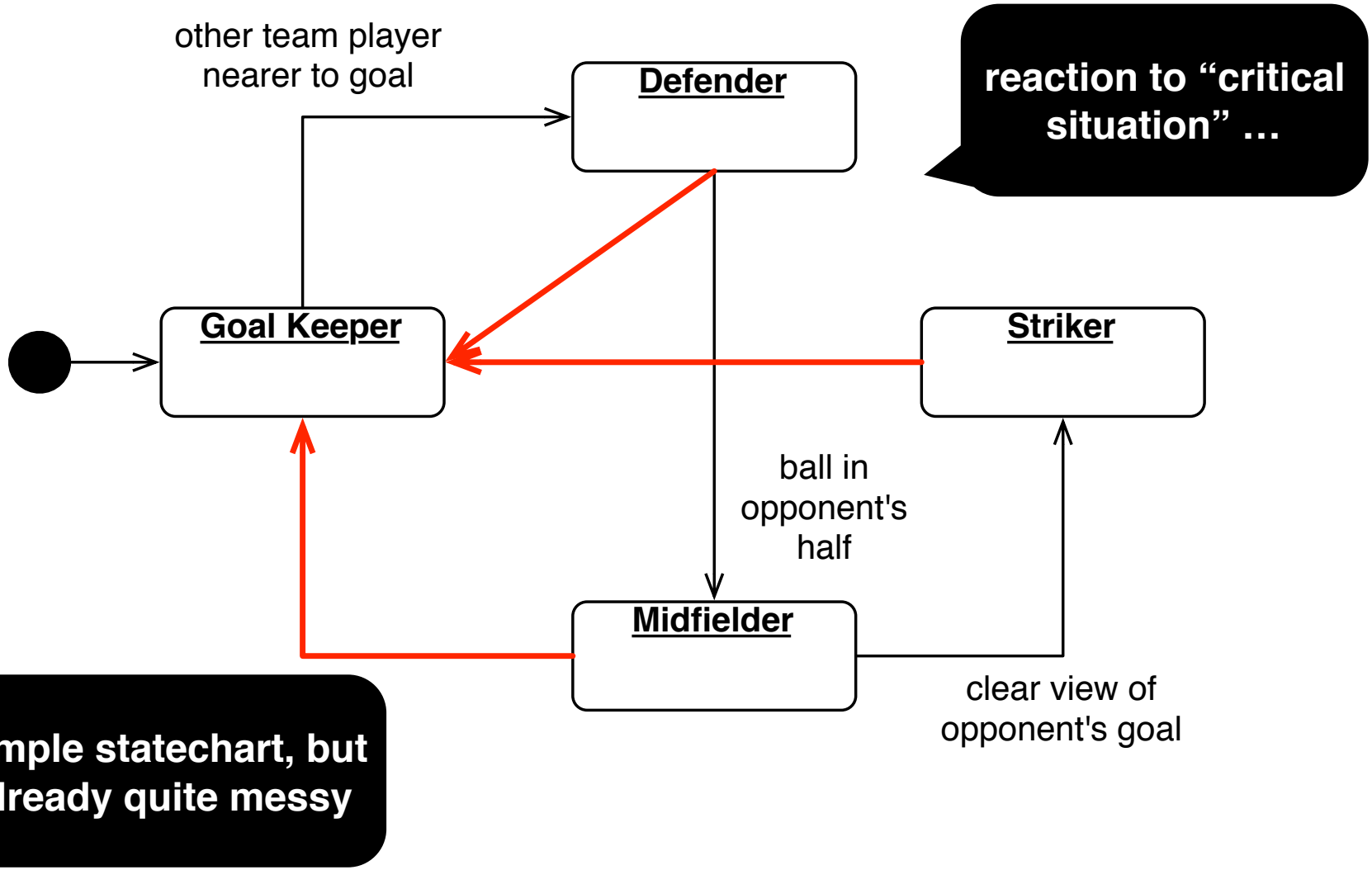
UML Statecharts



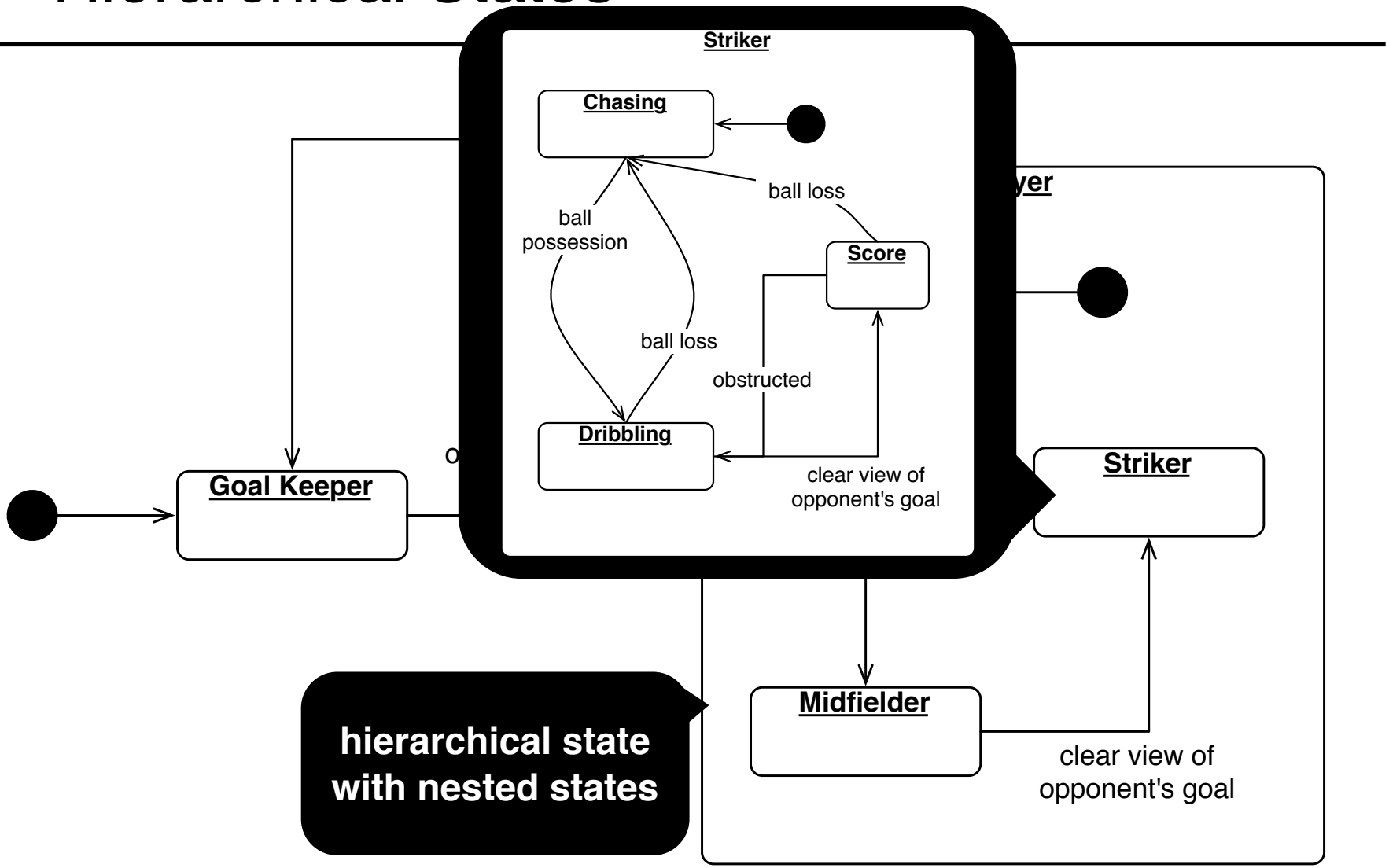
Hierarchical States



Hierarchical States

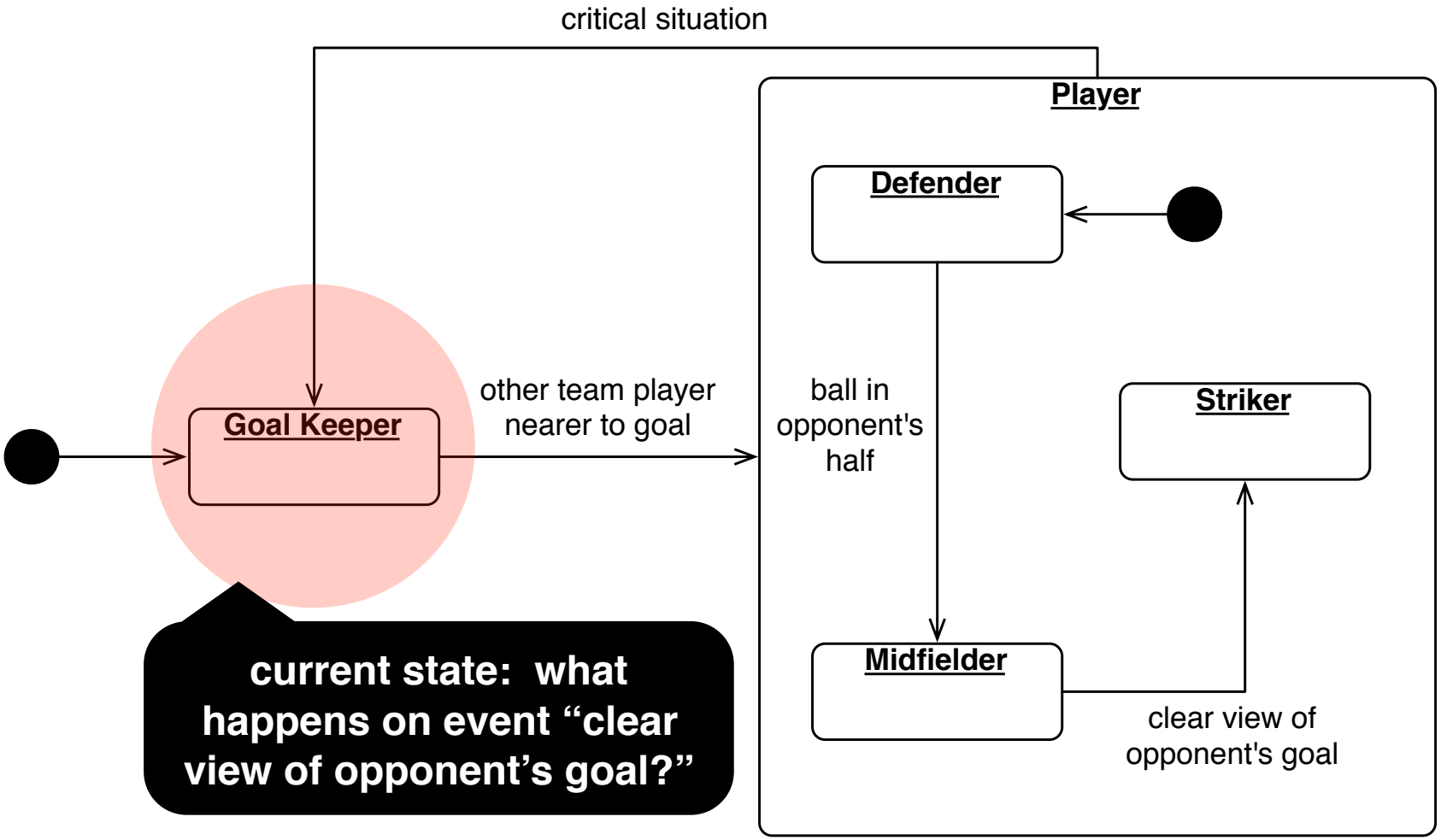


Hierarchical States

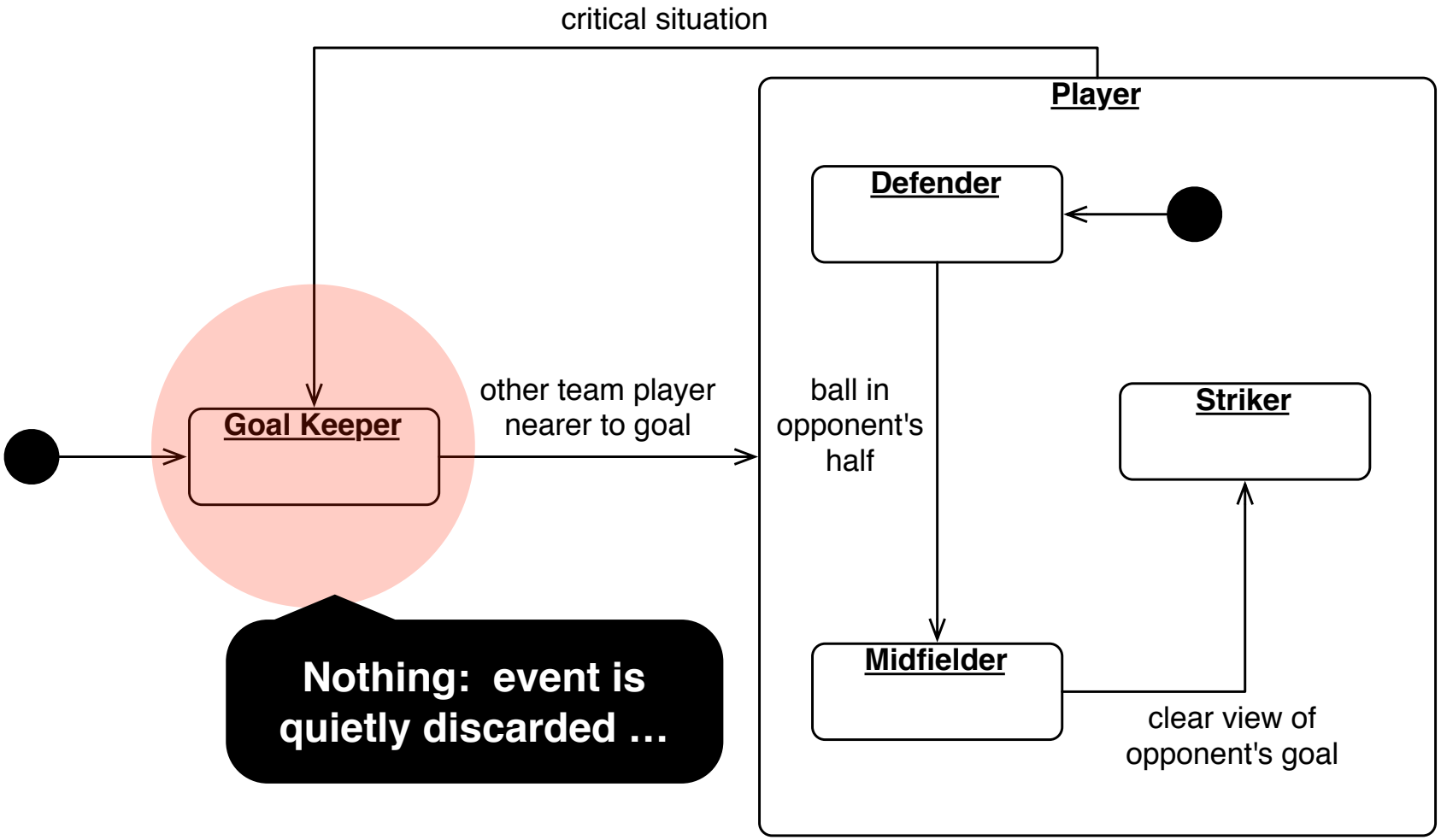


SEMANTICS

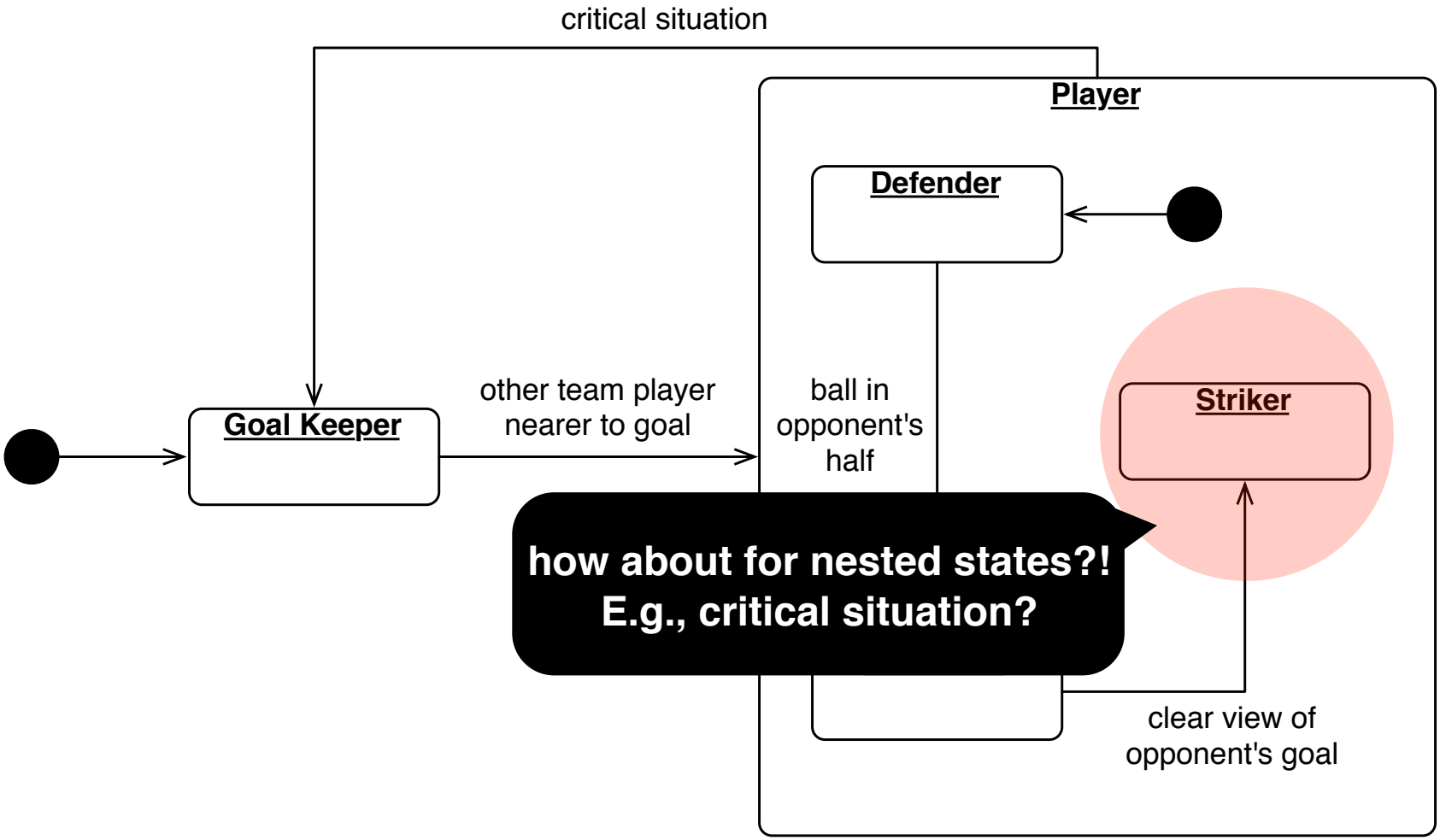
“Missing” Transitions?



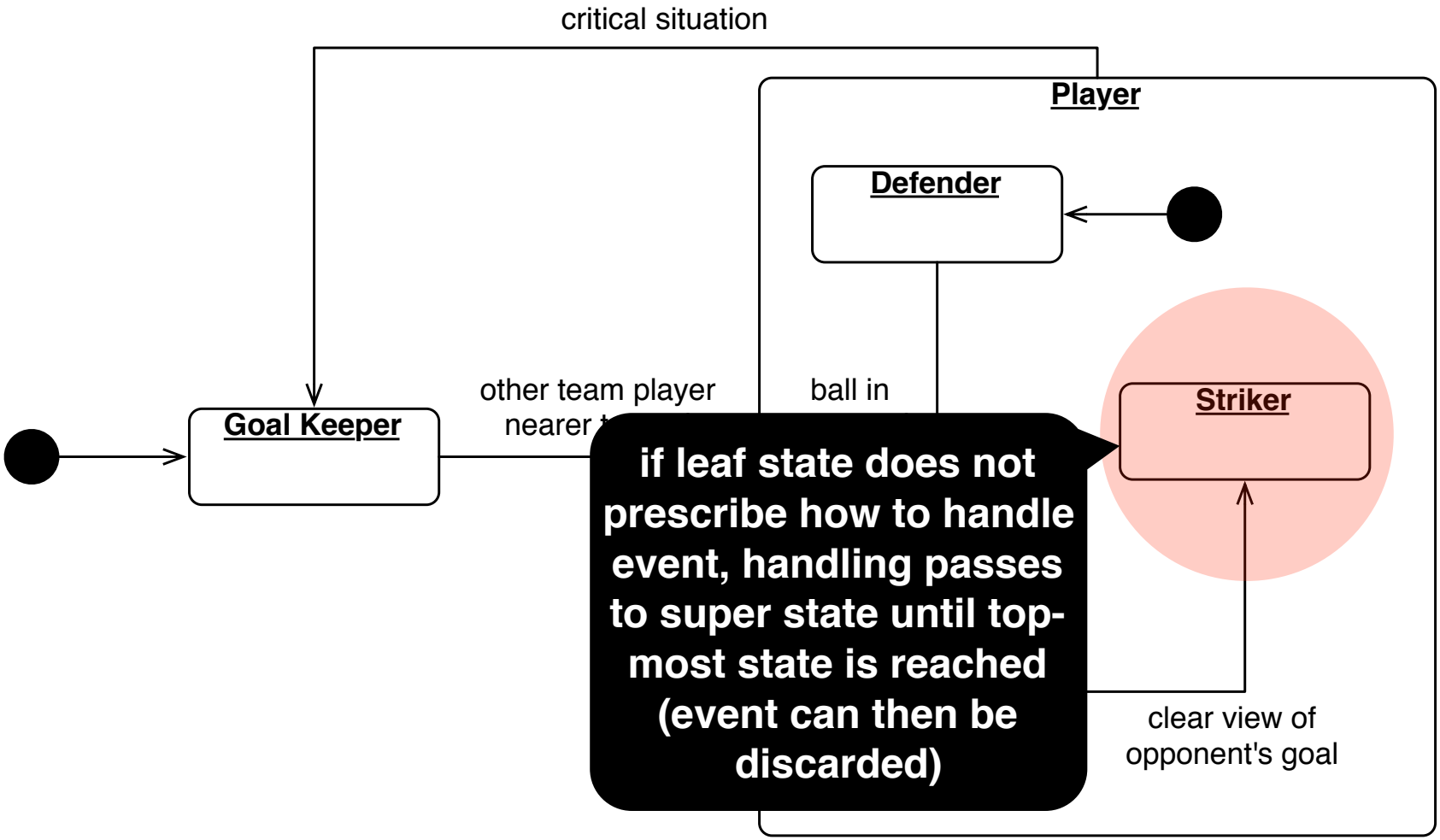
“Missing” Transitions?



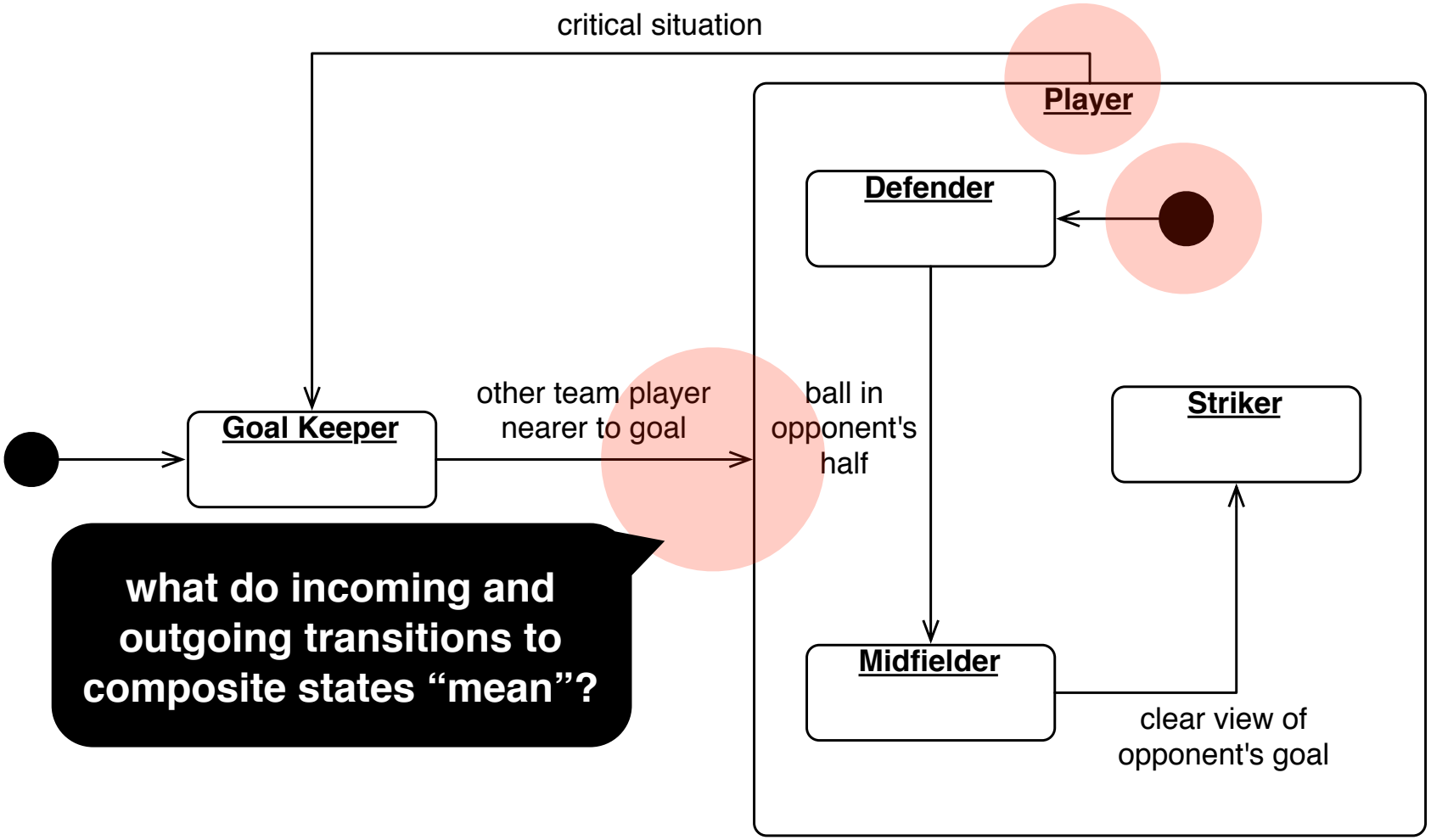
“Missing” Transitions?



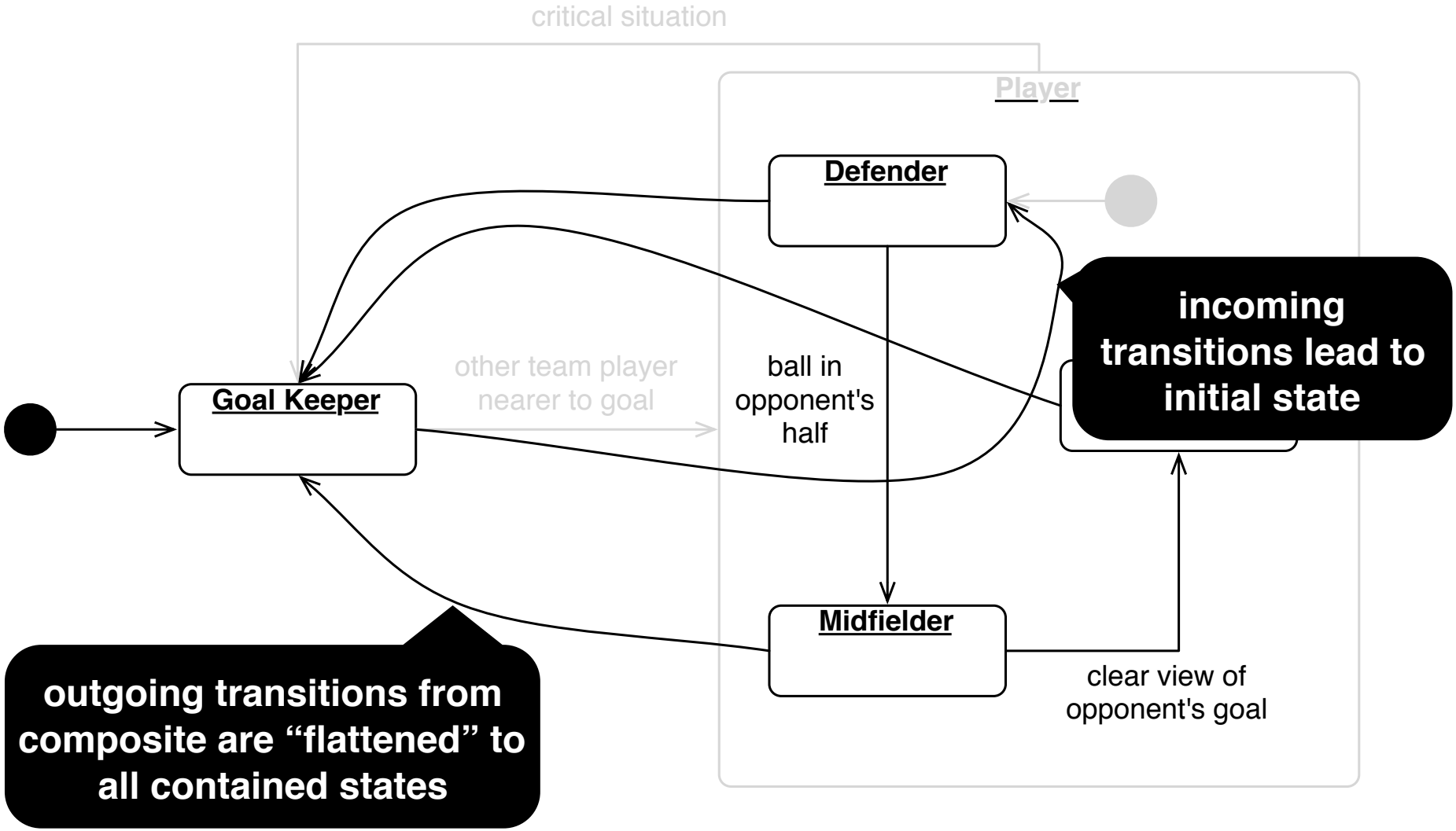
“Missing” Transitions?



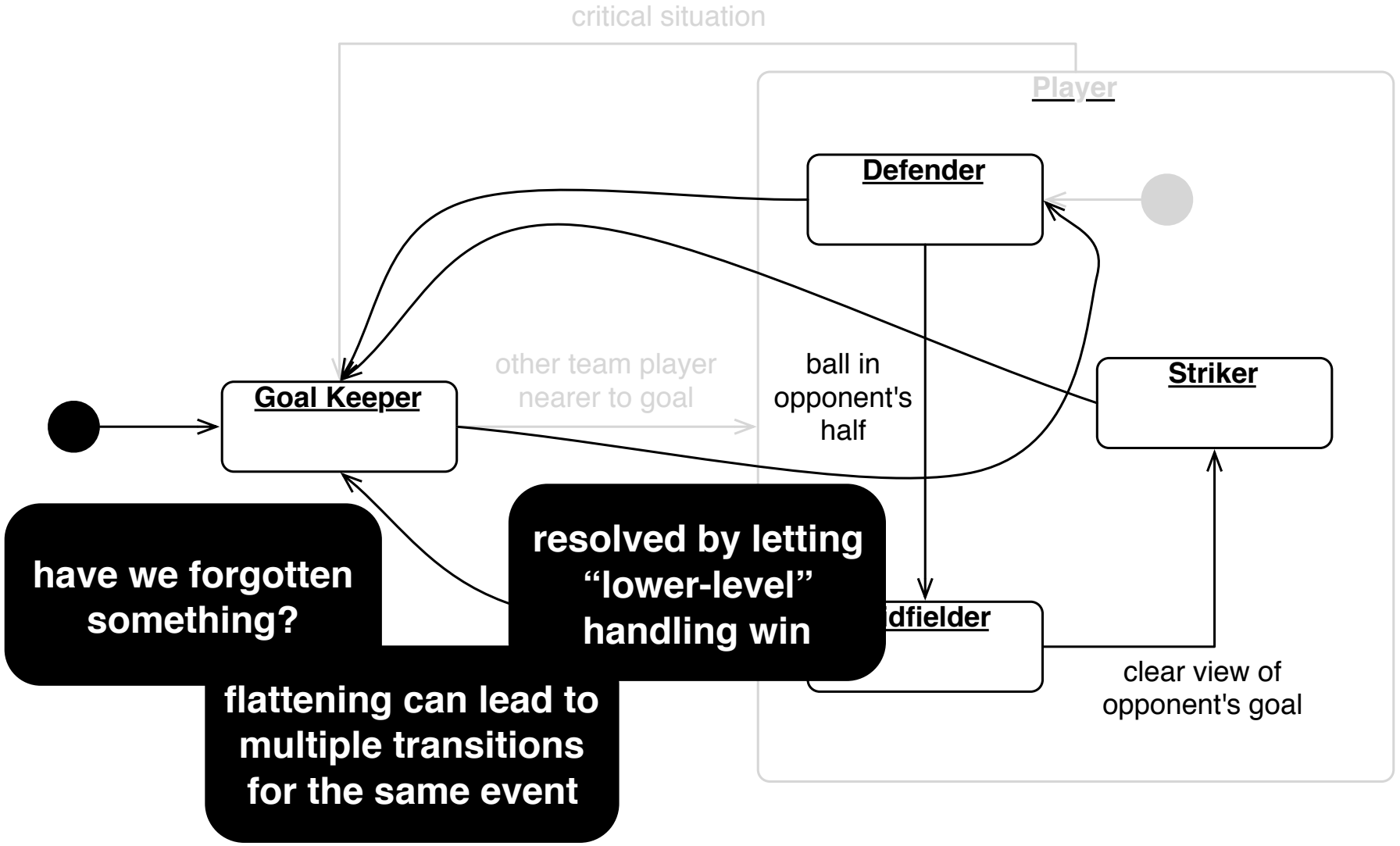
“Hierarchical” States?



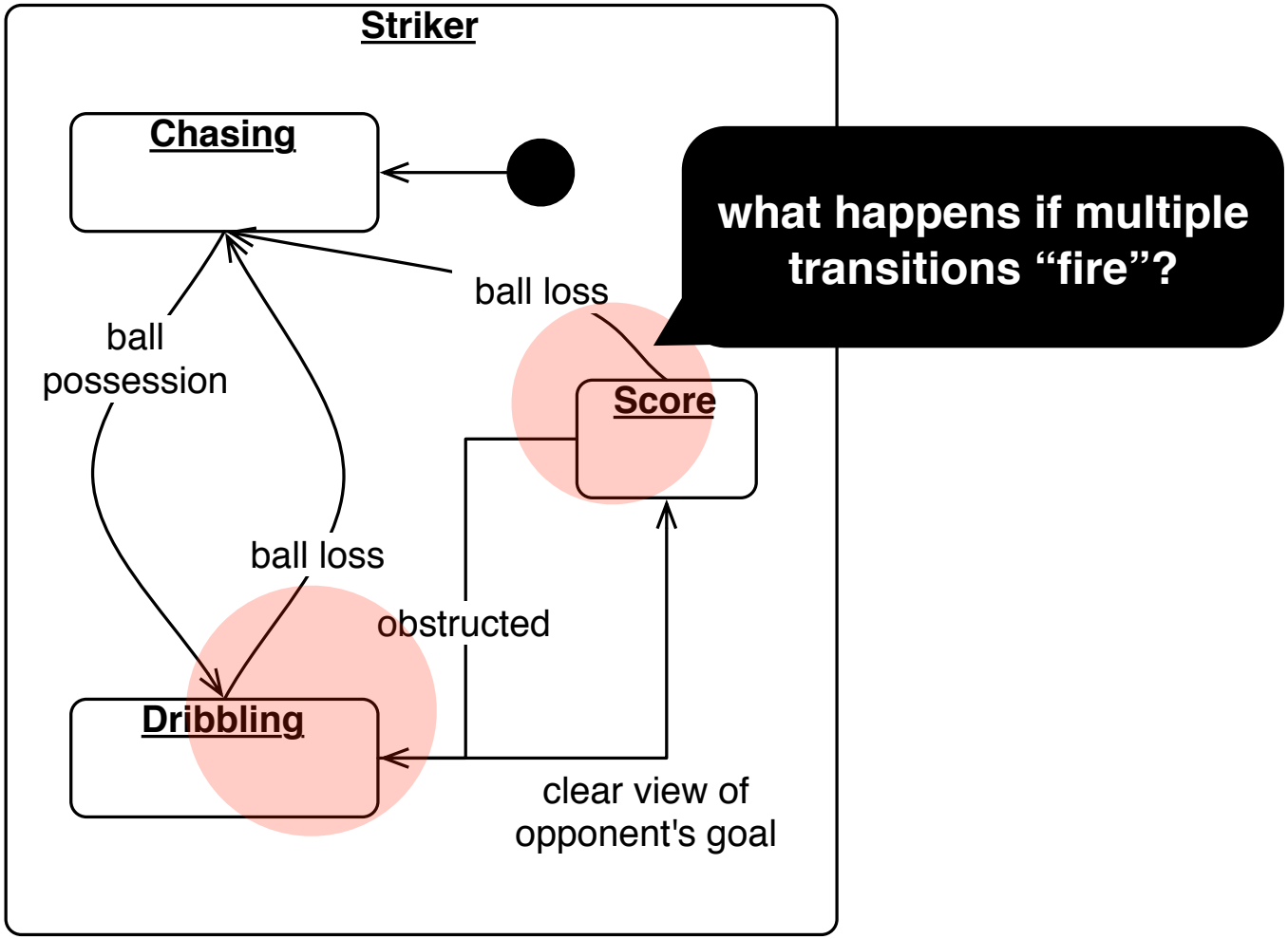
“Hierarchical” States?



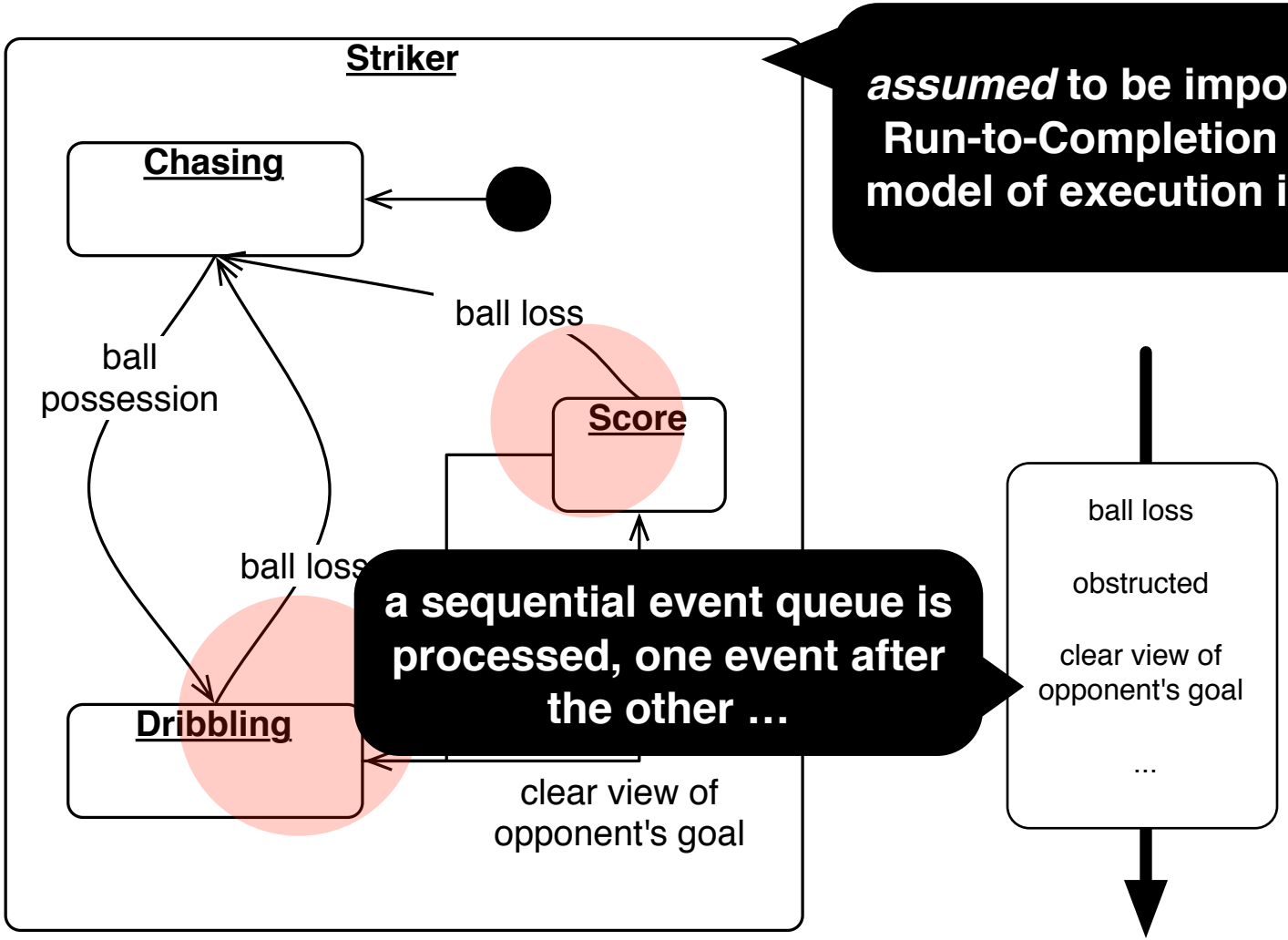
“Hierarchical” States?



Multiple Events?



Multiple Events?



Other Language Features

- Actions on transitions: TRIGGER / ACTION
- Actions on entry or exit of states

- Usage of variables (extended states)
- Guards on transitions: TRIGGER [GUARD] / ACTION
- Simple control flow (branching on variable values)

- Orthogonal regions (and vs. or decomposition)
- Internal vs. external vs. local transitions

- Networks of communicating state machines
- …

non-trivial (perhaps even unclear) semantics in some cases, can easily lead to “spaghetti code”!

APPLICATIONS OF STATE MACHINES

Modelling, Programming and Verification

The Boost Statechart Library

nblumhardt / **stateless**

Rubygem state_machine

oxo42 / **stateless4j**

some libraries for using state machines directly in your favourite programming language

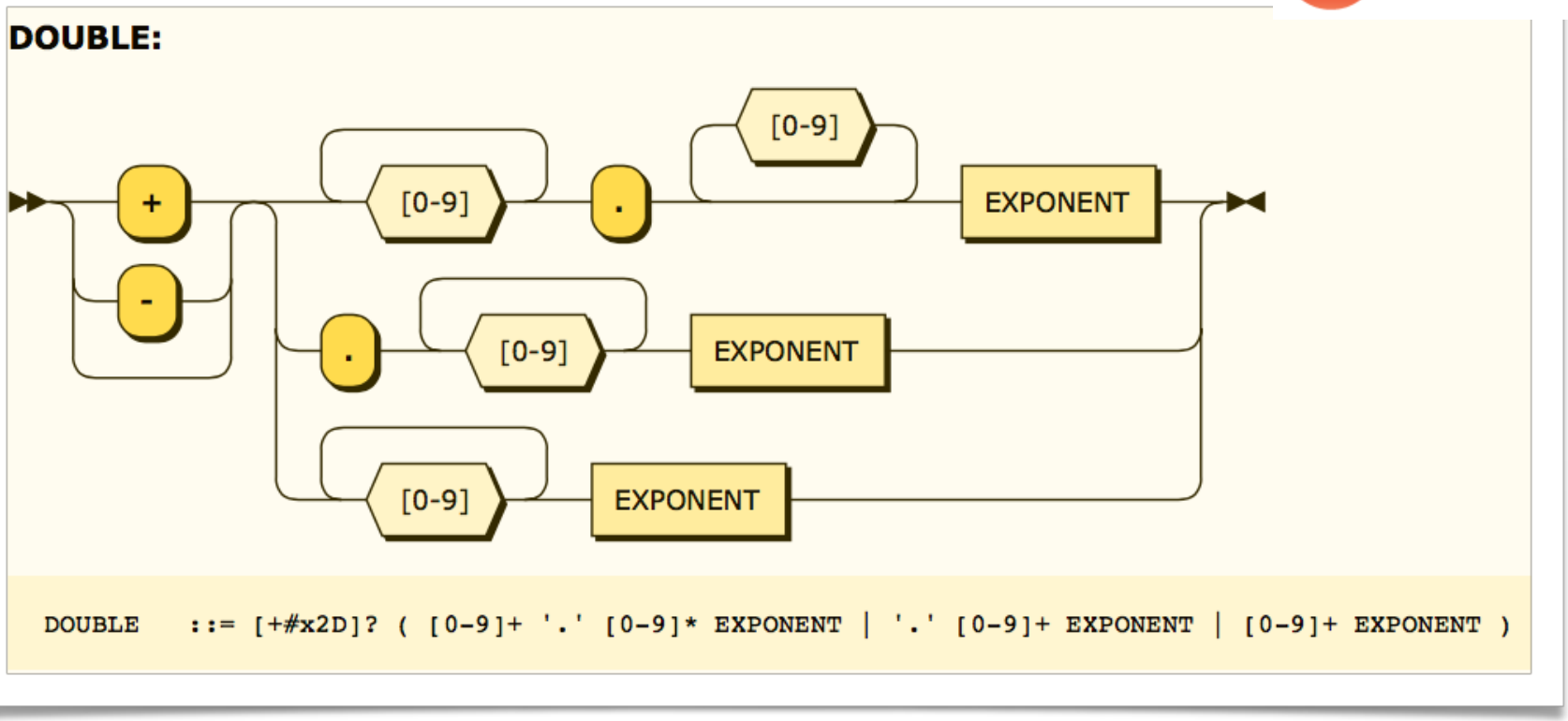
mbeddr

seamless extension of C with (amongst other things) verifiable state machines

UPPAAL

verification of real-time systems modelled as networks of timed automata

Language Recognition



Model-Based Testing

Are the test cases “good”?
Are they redundant?
Are they enough?
Are they up-to-date?

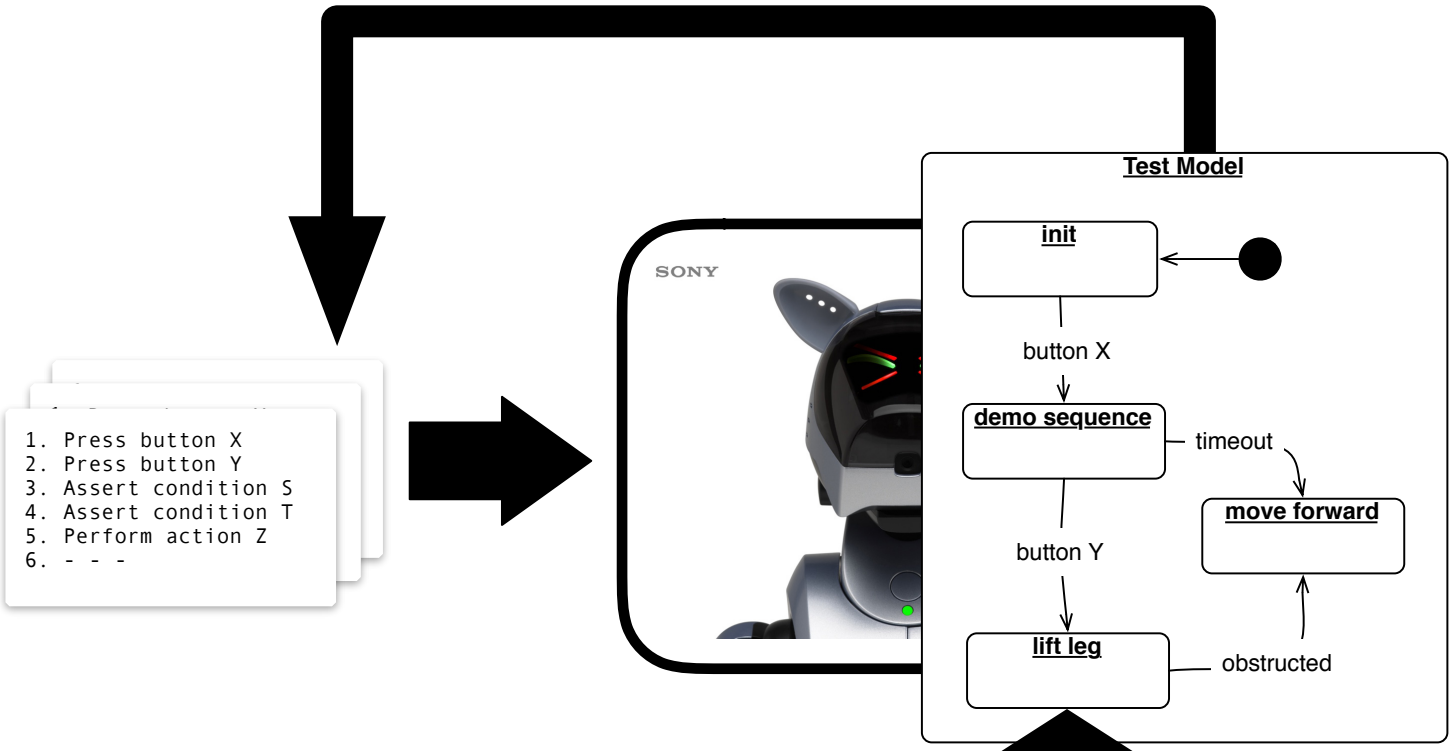
Test Cases for the SUT

```
1. Press button X
2. Press button Y
3. Assert condition S
4. Assert condition T
5. Perform action Z
6. - - -
```

System Under Test (SUT)

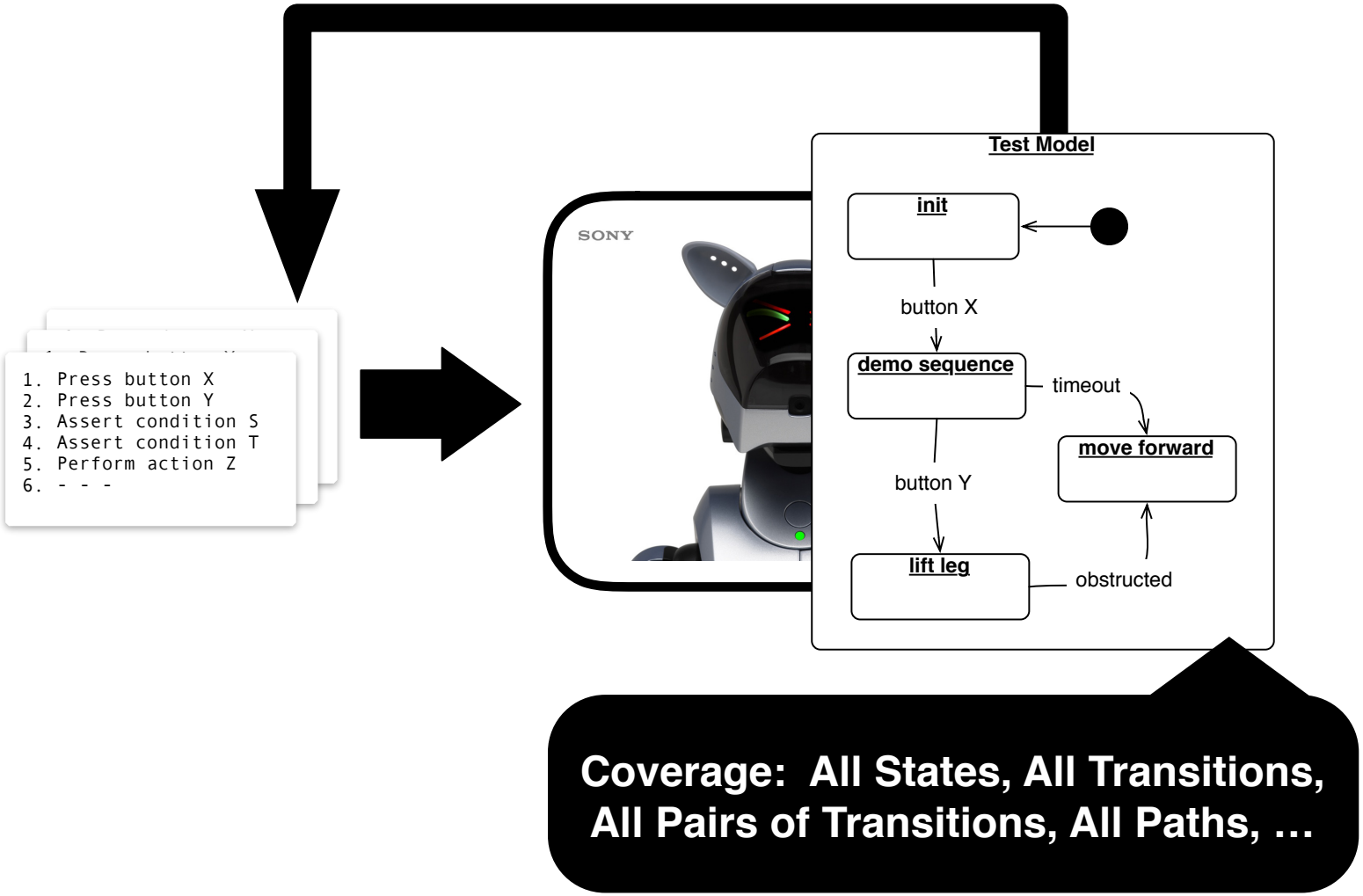
Model-Based Testing

use test model to generate tests
(and/or answer all our questions)



specify **expected** behaviour of
system as a state machine
(Test Model)

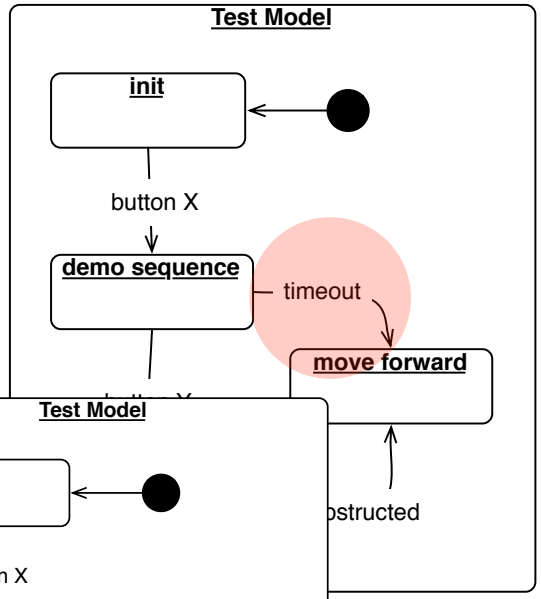
Model-Based Testing: Coverage

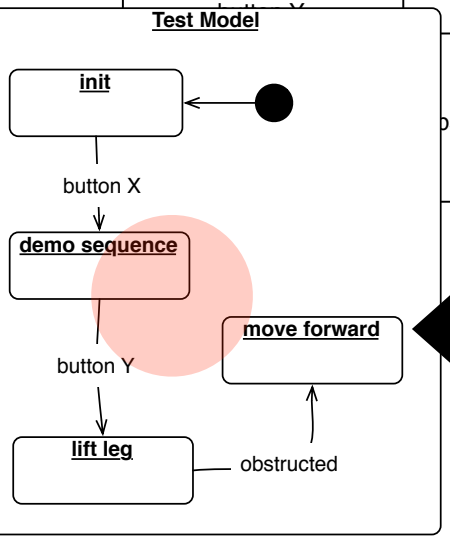


Model-Based Testing: Mutation Analysis

1. Run the tests on the test model, they should of course pass

```
1. Press button X
2. Press button Y
3. Assert condition S
4. Assert condition T
5. Perform action Z
6. - - -
```





2. Mutate the test model, systematically introducing faults (representing possible faults in the SUT)

3. A test is “good”, if it fails on (kills) as many mutants as possible

GO FORTH AND APPLY (STATE MACHINES)!