Discrete Event Simulation: API and Usage

Principles of Reactive Programming

Martin Odersky

How to Make it Work?

The class `Wire` and the functions `inverter`, `andGate`, and `orGate` represent a small description language of digital circuits.

We now give the implementation of this class and its functions which allow us to simulate circuits.

These implementations are based on a simple API for discrete event simulation.

Discrete Event Simulation

A discrete event simulator performs *actions*, specified by the user at a given *moment*.

An *action* is a function that doesn't take any parameters and which returns `Unit`:

```
type Action = () => Unit
```

The *time* is simulated; it has nothing to with the actual time.

Simulation Trait

A concrete simulation happens inside an object that inherits from the trait `Simulation`, which has the following signature:

```
trait Simulation {
  /** The current simulated time */
  def currentTime: Int = ???

  /** Registers an action ‘block‘ to perform after a given delay
   *  relative to the current time */
  def afterDelay(delay: Int)(block: => Unit): Unit = ???

  /** Performs the simulation until there are no actions waiting */
  def run(): Unit = ???
}
```

Class Diagram

The `Wire` Class

A wire must support three basic operations:

`getSignal: Boolean`

Returns the current value of the signal transported by the wire.

`setSignal(sig: Boolean): Unit`

Modifies the value of the signal transported by the wire.

`addAction(a: Action): Unit`

Attaches the specified procedure to the *actions* of the wire. All of the attached actions are executed at each change of the transported signal.

Implementing Wires

Here is an implementation of the class `Wire`:

```
class Wire {
  private var sigVal = false
  private var actions: List[Action] = Nil
  def getSignal: Boolean = sigVal
  def setSignal(s: Boolean): Unit =
    if (s != sigVal) {
      sigVal = s
      actions foreach (_())
    }
  def addAction(a: Action): Unit = {
    actions = a :: actions
    a()
  }
}
```

State of a Wire

The state of a wire is modeled by two private variables:

`sigVal` represents the current value of the signal.

`actions` represents the actions currently attached to the wire.

The Inverter

We implement the inverter by installing an action on its input wire.

This action produces the inverse of the input signal on the output wire.

The change must be effective after a delay of `InverterDelay` units of simulated time.

The Inverter

We implement the inverter by installing an action on its input wire.

This action produces the inverse of the input signal on the output wire.

The change must be effective after a delay of `InverterDelay` units of simulated time.

We thus obtain the following implementation:

```
def inverter(input: Wire, output: Wire): Unit = {
  def invertAction(): Unit = {
    val inputSig = input.getSignal
    afterDelay(InverterDelay) { output setSignal !inputSig }
  }
  input addAction invertAction
}
```

The AND Gate

The AND gate is implemented in a similar way.

The action of an AND gate produces the conjunction of input signals on the output wire.

This happens after a delay of `AndGateDelay` units of simulated time.

The AND Gate

The AND gate is implemented in a similar way.

The action of an AND gate produces the conjunction of input signals on the output wire.

This happens after a delay of `AndGateDelay` units of simulated time.

We thus obtain the following implementation:

```
def andGate(in1: Wire, in2: Wire, output: Wire): Unit = {
  def andAction(): Unit = {
    val in1Sig = in1.getSignal
    val in2Sig = in2.getSignal
    afterDelay(AndGateDelay) { output setSignal (in1Sig & in2Sig) }
  }
  in1 addAction andAction
  in2 addAction andAction
}
```

The OR Gate

The OR gate is implemented analogously to the AND gate.

```
def andGate(in1: Wire, in2: Wire, output: Wire): Unit = {
  def andAction(): Unit = {
    val in1Sig = in1.getSignal
    val in2Sig = in2.getSignal
    afterDelay(AndGateDelay) { output setSignal (in1Sig & in2Sig) }
  }
  in1 addAction andAction
  in2 addAction andAction
}
```

The OR Gate

The OR gate is implemented analogously to the AND gate.

```
def orGate(in1: Wire, in2: Wire, output: Wire): Unit = {
  def orAction(): Unit = {
    val in1Sig = in1.getSignal
    val in2Sig = in2.getSignal
    afterDelay(OrGateDelay) { output setSignal (in1Sig | in2Sig) }
  }
  in1 addAction orAction
  in2 addAction orAction
}
```

Exercise

What happens if we compute `in1Sig` and `in2Sig` inline inside `afterDelay` instead of computing them as values?

```
def orGate2(in1: Wire, in2: Wire, output: Wire): Unit = {
  def orAction(): Unit = {
    afterDelay(OrGateDelay) {
      output setSignal (in1.getSignal | in2.getSignal) }
    }
  in1 addAction orAction
  in2 addAction orAction
}
```

```
O   ‘orGate‘ and ‘orGate2‘ have the same behavior.
O   ‘orGate2‘ does not model OR gates faithfully.
```