Discrete Event Simulation: Implementation and Test

Principles of Reactive Programming

Martin Odersky

The Simulation Trait

All we have left to do now is to implement the `Simulation` trait.

The idea is to keep in every instance of the `Simulation` trait an *agenda* of actions to perform.

The agenda is a list of (simulated) *events*. Each event consists of an action and the time when it must be produced.

The agenda list is sorted in such a way that the actions to be performed first are in the beginning.

```
trait Simulation {
  type Action = () => Unit
  case class Event(time: Int, action: Action)
  private type Agenda = List[Event]
  private var agenda: Agenda = List()
```

Handling Time

There is also a private variable, `curtime`, that contains the current simulation time:

```
private var curtime = 0
```

It can be accessed with a getter function `currentTime`:

```
def currentTime: Int = curtime
```

An application of the `afterDelay(delay)(block)` method inserts the task

```
Event(curtime + delay, () => block)
```

into the `agenda` list at the right position.

Implementing AfterDelay

```
def afterDelay(delay: Int)(block: => Unit): Unit = {
  val item = Event(currentTime + delay, () => block)
  agenda = insert(agenda, item)
}
```

Implementing AfterDelay

```
def afterDelay(delay: Int)(block: => Unit): Unit = {
  val item = Event(currentTime + delay, () => block)
  agenda = insert(agenda, item)
}
```

The `insert` function is straightforward:

```
private def insert(ag: List[Event], item: Event): List[Event] = ag match {
  case first :: rest if first.time <= item.time =>
    first :: insert(rest, item)
  case _ =>
    item :: ag
}
```

The Event Handliong Loop

The event handling loop removes successive elements from the agenda, and performs the associated actions.

```
private def loop(): Unit = agenda match {
  case first :: rest =>
    agenda = rest
    curtime = first.time
    first.action()
    loop()
  case Nil =>
}
```

The Run Method

The `run` method executes the event loop after installing an initial message that signals the start of simulation.

```
def run(): Unit = {
  afterDelay(0) {
    println("*** simulation started, time = "+currentTime+" ***")
  }
  loop()
}
```

Probes

Before launching the simulation, we still need a way to examine the changes of the signals on the wires.

To this end, we define the function `probe`.

```
def probe(name: String, wire: Wire): Unit = {
  def probeAction(): Unit = {
    println(s"$name $currentTime value = ${wire.getSignal}")
  }
  wire addAction probeAction
}
```

Defining Technology-Dependent Parameters

It's convenient to pack all delay constants into their own trait which can be mixed into a simulation. For instance:

```
trait Parameters {
  def InverterDelay = 2
  def AndGateDelay = 3
  def OrGateDelay = 5
}

object sim extends Circuits with Parameters
```

Setting Up a Simulation

Here's a sample simulation that you can do in the worksheet.

Define four wires and place some probes.

```
import sim._
val input1, input2, sum, carry = new Wire
probe("sum", sum)
probe("carry", carry)
```

Next, define a half-adder using these wires:

```
halfAdder(input1, input2, sum, carry)
```

Launching the Simulation

Now give the value `true` to `input1` and launch the simulation:

```
input1.setSignal(true)
run()
```

To continue:

```
input2.setSignal(true)
run()
```

A Variant

An alternative version of the OR-gate can be defined in terms of AND and INV.

```
def orGateAlt(in1: Wire, in2: Wire, output: Wire): Unit = {
  val notIn1, notIn2, notOut = new Wire
  inverter(in1, notIn1); inverter(in2, notIn2)
  andGate(notIn1, notIn2, notOut)
  inverter(notOut, output)
}
```

Exercise

Question: What would change in the circuit simulation if the implementation of `orGateAlt` was used for OR?

0 Nothing. The two simulations behave the same.

0 The simulations produce the same events, but the indicated times are different.

0 The times are different, and `orGateAlt` may also produce additional events.

0 The two simulations produce different events altogether.

Summary

State and assignments make our mental model of computation more complicated.

In particular, we lose referential transparency.

On the other hand, assignments allow us to formulate certain programs in an elegant way.

Example: discrete event simulation.

- Here, a system is represented by a mutable list of *actions*.
- The effect of actions, when they're called, change the state of objects and can also install other actions to be executed in the future.