Real-Time pythonDEVS (pythonDEVS-RT)

Spencer Borland
Documentation
July 2002

Modeling, Simulation & Design Lab
McGill University

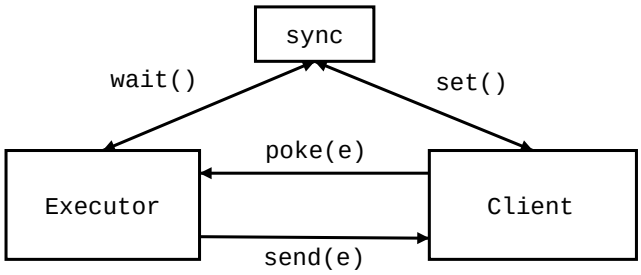

Figure 1: The `pythonDEVS-RT` threading model.

1 Abstract

This document outlines the motivation and basics of creating a real-time DEVS execution engine. An explanation is given as to how the previous Simulator was adapted to become an execution backend for DEVS models. This documentation ends with an example of a DEVS model and its subsequent execution.

2 Motivation

Instead of simulating DEVS models where the implicit time variable is simulation time, we wish to execute a model where time units are equivalent to real-world time. The reason for this is to be able to execute real-time programs with the DEVS real-time engine driving the dynamics of the application. In particular, graphical programs are an example of this desired framework. The graphical application has entities such as buttons which generate events. These events are sent to the real-time DEVS engine which may send a return message. The graphical application can then change its state based on the received message.

3 `pythonDEVS-RT`

Firstly the `pythonDEVS-RT` `Executor` class as well as the threading model are explained. Following, is a discussion of `pythonDEVS-RT` Executor's wait-dispatch loop. Finally, the interface between the Executor and client applications is outlined for graphical applications.

3.1 Threading Model

When an application uses a DEVS engine as its underlying dynamics, both the client and the engine are started in separate threads. This is illustrated in figure 1. The `sync` block is a global `Event` object taken from the `threading` library. This is the mechanism which synchronizes activities between the

client and the Executor. The Executor is in an unending wait-dispatch loop which utilizes calls to the `wait(x)` function. The method, `wait(x)`, will return if a call to `set()` is made from another thread or if x time has passed. Thus, when a client wishes to send an event to the DEVS engine, it will first `poke()` the event and then `set()` the `sync` object. The engine will then wake up, and process the event which was poked.

3.2 The Wait-Dispatch Loop

The Executor uses the same method of collecting information from all atomic DEVS components. However, some of this information is not needed anymore. In particular, the time of the next internal transition, t_N, is irrelevant information in a real-time framework. Instead, the smallest time advance of all the atomic DEVS in the model is required. The Executor must explicitly wait for this amount of time before the next internal transition. Once the wait stage is passed, the Executor must decide what type of message to send to its sub-components. It searches all its input ports and checks if any of them correspond to components which are sending events. If there exists such an event, it is considered an incoming external event and is passed to the DEVS model's corresponding input port. If there are no external events, a timeout must have occurred and an internal event signal is sent to the DEVS model.

The code below is the Executor's wait-dispatch loop, taken from `RT_DEVS.py`.

```
self.send(self.model, (0, [], 0))

from time import time
initialRT = time()

# Main loop repeatedly sends $(*,\,t)$
# messages to the model's root DEVS.
while 1:

  # if the smallest time advance is infinity, wait forever
  # if the smallest time advance is x, then wait(x)
  # don't wait at all if the smallest time advance is 0
  ta = self.model.smallestTimeAdvance
  if ta == INFINITY:
    self.sync.wait()
  elif ta > 0:
    self.sync.wait(ta)
  self.sync.clear()
  if self.done:
    return
```

```
# calculate how much time has passed so far
clockRT = time() - initialRT
print "* " * 10, "CLOCK (RT): %f" % clockRT

# pass incoming external events to their respective
# ports from the GUI if there are no external events
# then we must do an internal transition
immChildren = self.model.immChildren
external = 0
for inport in self.model.IPorts:
  for sourceport in inport.inLine:
    if len(sourceport.host.myOutput) != 0:
      external = 1
      for e in sourceport.host.myOutput.values():
        self.send(self.model, [{inport:e}, immChildren, clockRT])
      sourceport.host.myOutput.clear()
if not external:
  self.send(self.model, (1, immChildren, clockRT))
```

3.3 Executor Interface For Graphical Applications

When developing a graphical application, it must extend the `DEVS_GUI_APP` class found in `RT_DEVS.py`. This class basically emulates a coupled DEVS since it is responsible for connecting ports. The `connectPorts` method is used for this effect. The `DEVS_GUI` class must be extended for building the actual graphical program itself. This class inherits from the `Tk` class `Frame` as well as `Thread`. `DEVS_GUI` has methods `addOutPort` and `addInPort` which allow the graphical application to send events though different ports. The `send` method is used from a DEVS model to send events back to the GUI. These events are captured in the `recv` method which should be overridden.

4 Crosswalk Example

This example can be downloaded from:
`http://msdl.cs.mcgill.ca/people/sborla/pythonDEVS-RT/`.

The model in figure 2 is a model of a crosswalk and traffic light. The light alternates between RED, YELLOW and GREEN. If the light is RED and the crosswalk button is pressed the light will turn GREEN faster. A 'Police Interrupt' will cause the RED light to flash repeatedly only until the 'Police Interrupt' is pressed again. After the 'Police Interrupt' is pressed a second time, the light will turn the same color it was before the first time this button was pressed. Moreover, the entire system can be turned on and

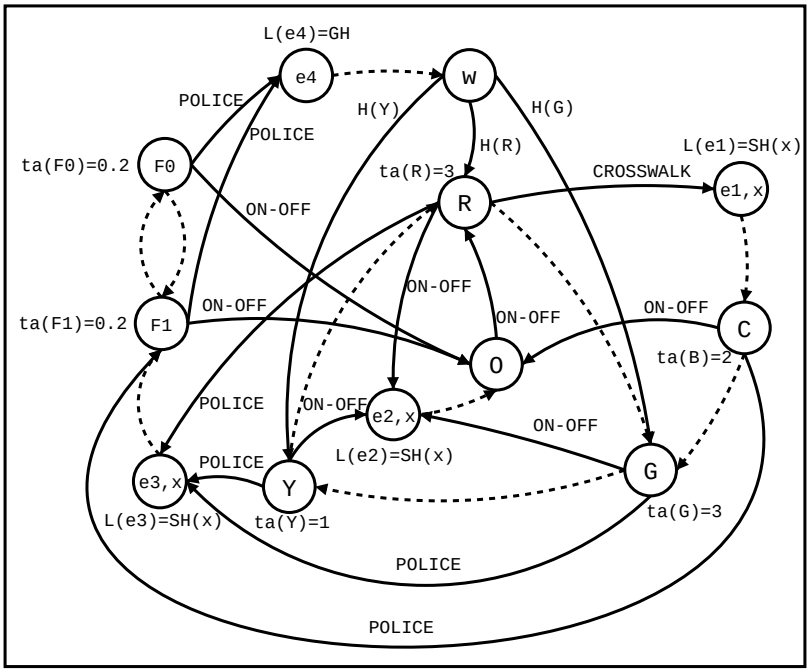

Figure 2: The DEVS crosswalk model.

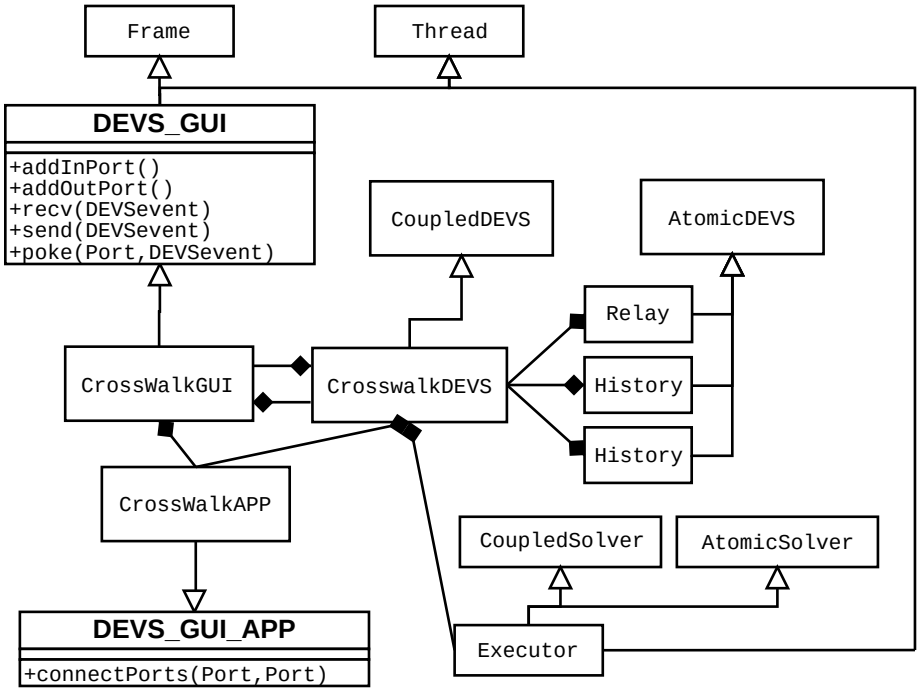

Figure 3: The crosswalk UML class diagram.

off. Once turned off, the system forgets all information, and once turned on the system will always start in the RED state.

The program is run by typing "`python CrossWalkAPP.py`" at the shell prompt. The class diagram for the application can be seen in figure 3. The program starts by creating a new application object which instantiates and initializes the `tk` graphical objects as well as the underlying DEVS engine.

One common problem and its solution are as follows. If an external event is generated but has no effect on the current state, s, it is desirable to return a time advance value which is the difference between $ta(s)$ and the elapsed time since the last state change. In the crosswalk example, if the current state of the traffic light is GREEN and the 'Pedestrian Crossing' button is pressed, the system to schedule the next internal transition as 9 seconds. Since, the 'Pedestrian Crossing' button has no effect on the current state while the light is GREEN, the system should return a time advance of $9 - e'$, where e' is the elapsed time since the last state change.