A Modeling and Simulation Package for Classic Hierarchical DEVS

Jean-Sébastien Bolduc (jseb@cs.mcgill.ca)
Hans Vangheluwe (hv@cs.mcgill.ca)

July 2002

Abstract

Discrete Event system Specification (DEVS) were first introduced by Zeigler in 1976 as a rigorous basis for discrete-event modelling. In this work we first present the classic formalism. An early prototype of a *DEVS Modeling and Simulation Package* is then introduced. The work then finishes with examples that illustrate the use of the package and shows some limitations of the classic formalism.

1 Design and Implementation

This version of the *DEVS Modeling and Simulation Package* has been implemented using *Python*, an interpreted, very high-level, object- oriented programming language. With built-in high-level data types and a simple yet elegant syntax, *Python* is an ideal tool for rapid application development in many areas. The interpreter is freely available for all major platforms from the Python Language Website (at http://www.python.org), and can be freely distributed. Extensive documentation on the language can be found on or through the same site.

The package consists of two files, the first of which (`DEVS.py`) provides a class architecture that allows hierarchical DEVS models to be easily defined. The simulation engine (SE) itself is implemented in the second file (`Simulator.py`). Based on the DEVS simulator described in [1], it uses the same message-passing mechanism. A detailed description of both the *modelling architecture* and the SE follows.

We should emphasize that the package described here is still an early prototype with important limitations. For instance a model cannot be thoroughly validated using the present methods, and the modeler is consequently encouraged to write model descriptions in dedicated files rather than using the interpreter in an interactive manner. More importantly, the SE offers limited means to terminate a simulation and provides no easy model-reinitialization possibilities yet. Further versions of the package will also support sets declarations and extended type-checking as well as information methods to help debugging. Other improvements to increase the tool's robustness and conviviality will be suggested along the way.

1.1 Modelling Architecture

The *modelling architecture* implemented in `DEVS.py` is a canvas from which hierarchical DEVS models can be easily described. It essentially consists in a number of classes arranged in such a way as to capture the essence of hierarchical DEVS: a model is described in a dedicated file by deriving coupled- and/or atomic-DEVS *descriptive* classes from this architecture, and arranging them in a hierarchical manner through composition. Methods and attributes form the standard interface that allows a SE — such as the one described in the next sub-section — to interact with the instantiated DEVS model. Our main concerns in writing the modelling architecture were twofold: remain as consistent as possible with the original hierarchical DEVS

definition, and maintain a flexible approach to DEVS description so as to encourage model reusability through parameterization.

The class architecture is represented in Figure 1: `BaseDEVS` is the root class which provides basic functionalities common to both atomic- and coupled- DEVS. It consists essentially in ports-related attributes and methods, which is reasonable since sets of input and ouput ports are present in both formalisms. In a further implementation, we project to derive the `BaseDEVS` class from a general `MSO` (Modeling and Simulation Object) class that will mainly implement the extended type-checking. `Sets` and `Ports` classes will also be derived from this broad class (the former class, inexistent in the current implementation, will allow the modeler to define sets; the second class is implemented as an independant entity in the current version).

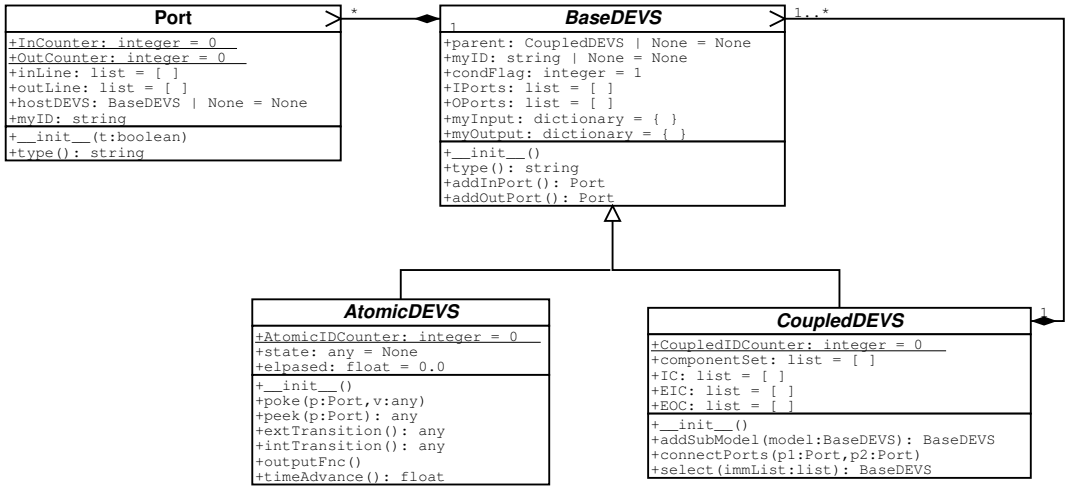

Figure 1: Modelling class architecture

Methods `addInPort` and `addOutPort` are used to add (through composition) an input or output port to the DEVS: they take no parameter and return a reference to the added port, which shall typically be saved in the DEVS object local dictionary. For example, the following line of code in a DEVS' descriptive class constructor will add an input port to the DEVS, locally known as `IN`:

```
self.IN = self.addInPort()
```

Whether the returned value is saved or not, a reference to the added port is appended to the attribute list `IPorts` (or `OPorts`, in case of an output port). One will recognize in those lists the *set of input* (*output*) *ports* of the DEVS' definition.[1] The other port-related attributes, `myInput` and `myOutput`, are dictionaries of the form $\{port\_reference : message\_on\_port\}$ storing the DEVS' latest input and output configurations (an unkeyed port is interpreted as the port being idle, *i.e.*, holding the "null" message, or $\emptyset$). In the current version the actual message transfer from a DEVS to another is the SE's reponsibility; however this might change as we intend to review the transfer mechanism in order to simplify the model–simulator interface.

The `parent` attribute is a reference to the parent coupled-DEVS in the hierarchical representation of the model (remains `None` for a root-DEVS). `myID` is an identification string assigned upon instantiation, consisting of a unique integer preceded by the descriptive character `A` or `C` (atomic- or coupled-DEVS, respectively); the `type` method checks this character to return the type-string, either "`ATOMIC`" or "`COUPLED` ". Finally, `condFlag` is a boolean condition flag unused for the moment, but that should indicate whether the DEVS is active or idle.

Referring back to Figure 1, we see that two classes are inherited from `BaseDEVS` to deal with the specifics of atomic- and coupled-DEVS formalisms. These three classes are all abstract in that they cannot be directly

[1] Although the lists are accessible to the modeler, she will usually prefer the use of a local attribute, as in the previous example.

instantiated: a model is rather described by deriving *descriptive classes* from either the `AtomicDEVS` or the `CoupledDEVS` class, providing them with a suitable constructor and eventually overriding the default interface methods. Note that the constructors at every level up the class hierarchy have an active role, hence a descriptive class' constructor should always start by calling the parent class' constructor.

The constructor of the `AtomicDEVS` class merely initializes the `myID` attribute and provides a default initial value for the DEVS' *total state*, through the `state` and `elapsed` attributes. The rest of the class definition consists mainly in default method declarations for the DEVS' interface functions δ_{ext} (`extTransition`), δ_{int} (`intTransition`), ta (`timeAdvance`) and λ (`outputFnc`). These methods expect no parameter, and it is up to the modeler to be consistent with the corresponding functions' domain when overriding the methods (*i.e.*, all but the first method will typically access *only* the local `state` state attribute). Except for `outputFnc` (which uses the `poke` method as described below), all the methods shall return a value compatible with the corresponding function range.

Since default values are provided for both attributes and methods, the minimal atomic-DEVS descriptive class is empty:

```
class MinimalAtomicDEVS(AtomicDEVS):
  pass
```

This atomic-DEVS is passive — it just remains in its default state forever. A more interesting example is a *generator*, which sends a message (the integer 1 in this case) through its unique output port at constant time-interval:

```
class SimpleGenerator(AtomicDEVS):
  def __init__(self, n = 1):
    AtomicDEVS.__init__(self)
    self.interval = n
    self.message = 1
    self.OUT = self.addOutPort()
  def outputFnc(self):
    self.poke(self.OUT, self.message)
  def timeAdvance(self):
    return self.interval
```

Note how the constructor is parameterizable, allowing the time- interval to be specified upon model instantiation. The constructor first calls the parent class' constructor, defines two local attributes and adds an output port to the DEVS. Since the total state is not overridden, the DEVS starts with the default `state` and `elapsed` attributes (`None` and `0.`, respectively). The DEVS having no input port, the `extTransition` function will never be called and is not overridden: the default `intTransition` function is also used, which merely returns the current state, and the `timeAdvance` method is specified to return the message-interval time. As mentionned above, the `outputFnc` returns no value; instead it relies on the `poke` method to send `message` (second parameter) through the `OUT` output port (first parameter). The companion method, `peek`,[2] returns the message on the input port that is given as a unique parameter, and is used exclusively in the `extTransition` function. Both `poke` and `peek` methods are defined in the `AtomicDEVS` class, and should not be overridden.

Whereas the main work in defining an atomic-DEVS descriptive class involves overriding interface methods, the `CoupledDEVS` class only has one such interface method, namely the tie-breaking `select` function which takes as parameter a list of imminent sub-models (sorted in lexicographic order of their `myID` attributes) and returns one of them. As opposed to atomic-DEVS, the role of coupled-DEVS is mainly "structural", first by occupying the inner nodes[3] in the hierarchical representation of a DEVS model (see figure 2), and by

[2] Those who have ever worked on old Commodore machines know where these names come from...

[3] Formally, a coupled-DEVS could occupy a leaf node in the hierarchical representation, but we cannot think of any reason why this would be meaningful.

defining the couplings between its children's and its own ports. Hence the heart of coupled-DEVS definition is in the descriptive class' constructor, where *submodels* and their *couplings* are described.

To help in this, the class has an interface attribute `componentSet` which corresponds to the *set of components* $\{M_d \mid d \in \mathcal{D}\}$ and is a list of references to (atomic- or coupled-) sub-DEVS. The *coupling sets* attributes `IC`, `EIC` and `EOC` (respectively, *internal couplings*, *external input couplings* and *external output couplings*) each consists in sets of pair of pairs: the first pair referencing the component and the port the coupling is beginning at, and the second referencing the component and the port the coupling is ending at.

The `addSubModel` method allows the modeler to add a sub-model (either an atomic- or coupled-DEVS) to the coupled-DEVS. The method takes as a parameter an instance of the sub-model, sets that instance's root DEVS' `parent` attribute to itself, and adds a reference to the instance in the local `componentSet`. A reference is also returned which, as for ports, shall typically be saved in the local dictionary. Doing so provides the modeler with an implicit set of *component references* $\mathcal{D}$.

Coupling of ports is performed through the `connectPorts` method: it takes as a first parameter the port the coupling is to begin at, and as a second parameter the port the coupling is to end at. The ports either belong to the coupled-DEVS itself or to one of its sub-DEVS', and some requirements must be met for a coupling to be valid: namely, the method checks that

1. at least one of the DEVS the ports belong to is a child of the coupled-DEVS, while the other is either the coupled-DEVS itself or *another* of its children (to avoid loops). The DEVS' "parenthood relationship" uniquely determine the type of coupling, *i.e.*, in which coupling set the coupling will be stored;
2. the types of the ports (either input or output) are consistent with the "parenthood" of the associated DEVS. This validates the coupling determined above.

A coupling is rejected and an error message issued were a coupling invalid. Otherwise the coupling information is stored in the appropriate coupling set. Of course, output ports can have any number of outgoing couplings: but it might come as a surprise that input ports also can have more than a single incoming coupling. The reason why it is possible is that whenever a pair of DEVS shall undergo an internal transition synchronously (and thus send messages at the same time), the DEVS formalism specifies that the transitions are actually performed sequentially at the same simulation time, by means of the `select` method.[4]

Although consistent with Zeigler's definition, the coupling sets `IC`, `EIC` and `EOC` are not used by the simulator, which instead relies on class `Port`'s `inLine` and `outLine` attributes to figure out couplings. These are lists of references to the (possibly many) ports from which the port receives its messages, or to the (possibly many) ports which receive messages from the port, respectively (while an atomic-DEVS output or input port would only need to declare `outLine` or `inLine`, the dual nature of coupled- DEVS ports require both attributes). The other `Port` attributes are `hostDEVS`, a reference to the DEVS the port belongs to, and `myID` which is a unique identification string assigned upon instantiation similar to `BaseDEVS`'s eponymous attribute: the unique integer in that case is preceded by the descriptive string "`IN`" or "`OUT`". Accordingly, the `Port` class also has a `type` method which either returns the string "`INPORT`" or "`OUTPORT`".

`Port` is a rather simple class, but this shall change in further versions: we noted earlier that the class shall be derived from a general `MSO` class that implements extended type-checking. This will allow us to implement the actual message transfer at the `Port` level (rather than at the SE level). Also, the port's state, which are currently stored in input- and output- dictionaries (once again at the SE level) shall obviously be stored in the ports objects themselves. We also project to implement couplings as objects, which would allow a modeler to specify output-input transfer functions, a feature that should greatly enhance model reusability.

As for the atomic-DEVS, the minimal coupled-DEVS descriptive class is of no practical interest. So let's consider as an example instead the situation illustrated in Figure 2; we have a descriptive class `SomeDEVS` for a DEVS (either atomic- or coupled-) with an input and output port locally known as `IN` and `OUT`. We

[4]The situation where the same DEVS has more than one of its output ports connected to another DEVS' input port is of course a degenerate case.

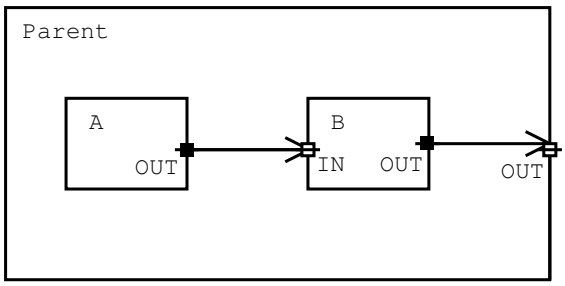

Figure 2: Simple Coupled-DEVS

want to connect the input port to the output port of the `SimpleGenerator` atomic-DEVS described above: both DEVS must of course be children of the same coupled-DEVS for the coupling to be performed. The coupled-DEVS descriptive class is defined below.

```
class Parent(CoupledDEVS):}
  def __init__(self, x = 1):
    CoupledDEVS.__init__(self)
    self.OUT = self.addOutPort()
    self.A = self.addSubModel(SimpleGenerator(x))
    self.B = self.addSubModel(SomeDEVS())
    self.connectPorts(self.A.OUT, self.B.IN)
    self.connectPorts(self.B.OUT, self.OUT)
```

Note that the parameter to the constructor is used to parameterize the `SimpleGenerator` atomic-DEVS. As for atomic-DEVS, the constructor first calls the parent class' constructor. Note also that the coupled-DEVS itself has an output port: the first coupling is an *internal coupling*, while the second is an *external output coupling*. This is a complete definition for a coupled-DEVS descriptive class: while the `select` function could very well be customized, the example here uses the default method which merely returns the first DEVS in the imminent list.

Once all the descriptive classes in the hierarchical DEVS model have been defined, the whole model can be build by instantiating the root DEVS (we will see in the next subsection that the instantiation usually takes place when the simulator is itself being instantiated). This is possible since the hierarchical representation of the model is build by *composition* rather than *aggregation*. Of course, nothing prevents the modeler to use aggregation as in the following partial code:

```
class SomeAtomicDEVS(AtomicDevs):
  # ...

AtomicDEVSInstance = SomeAtomicDEVS()

class SomeCoupledDEVS(CoupledDEVS):
  def __init__(self):
  CoupledDEVS.__init__(self)
  # ...
  self.A = self.addSubModel(AtomicDEVSInstance)
  # ...
```

But it is a good habit to avoid such constructs, since it would then be easy to destroy the tree structure by using the same object twice (descriptive classes, on the other hand, can be instantiated as many times as needed). As a final warning we will note that recursive definitions are illegal, since they would be incompatible with a tree structure: as a trivial example, a coupled-DEVS' descriptive class `SomeCoupledDEV` cannot call in its constructor the `addSubModel` method with an instance of `SomeCoupledDEV`. This is mentionned since such recursive constructs will not be detected.

1.2 Simulation Engine

The *Simulation Engine* (SE) is the tool that simulates the behavior of a system given its model as specified in the previous subsection. Since the SE is independent from the modelling architecture presented above, a clean interface between the model and the simulator must be defined, which will allow alternative SEs (*i.e.*, parallel implementations) to be written. The particular SE we describe here is loosely based on the DEVS simulator described in [1]. We emphasized at the beginning of section 1 that this prototype SE provides no model-reinitialization methods: this implies that the simulator will always start a simulation using the current state of a model (*i.e.*, the set of the total states of all the DEVS in the hierarchical model), regardless of the model's past history.

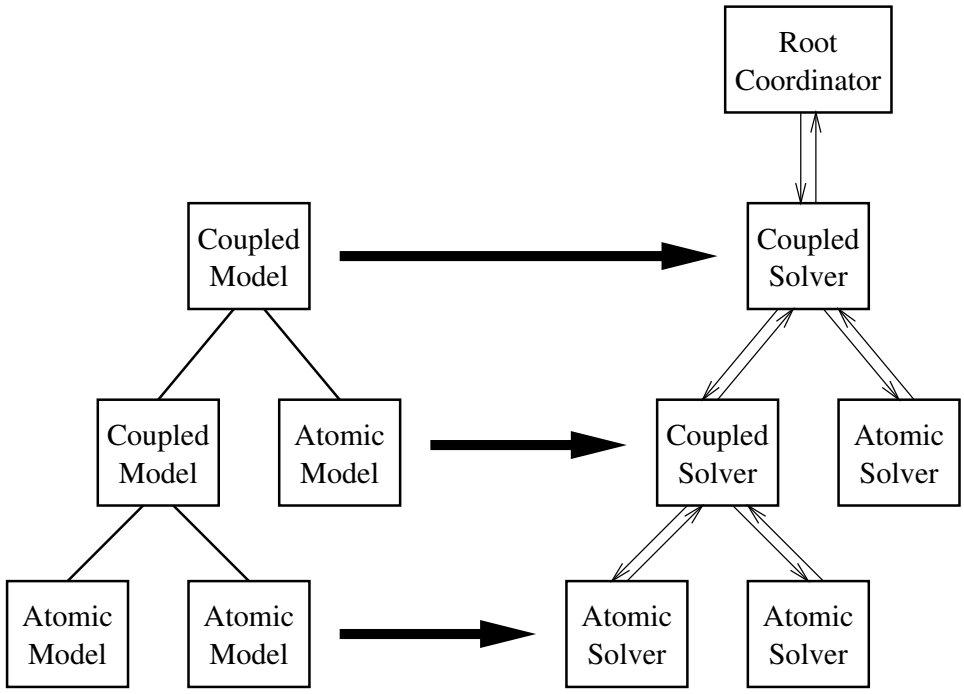

Figure 3: Mapping a hierarchical model onto a hierarchical simulator

The main challenge in writing a SE for a hierarchical DEVS model consists in determining, at run-time, the sequence (atomic- or coupled-) DEVS activation. As it turns out, the hierarchical representation of the model (Figure 3), although it gives no information on the couplings, is an appropriate view to tackle the problem by means of a message-passing mechanism[5]: in Zeigler's words [1],

> A hierarchical simulator for hierarchical DEVS coupled model models consists of *devs-simulators* and *devs-coordinators* and uses four types of messages. An initialization method (i, t) is sent at

[5]The *transfer* of messages (data) from port to port shall not be confused with the *passing* of messages (control) from solver to solver.

the initialization time from the parent simulator object to all its subordinates, to synchronize their clocks. The scheduling of events is done by the internal state transisiton message $(*, t)$ and are sent from the coordinator to its imminent child. An output message (y, t) is sent from the subordinates to their parents to notifiy them of output events. The input message (x, t) is sent from the coordinator to its subordinates to cause external events.

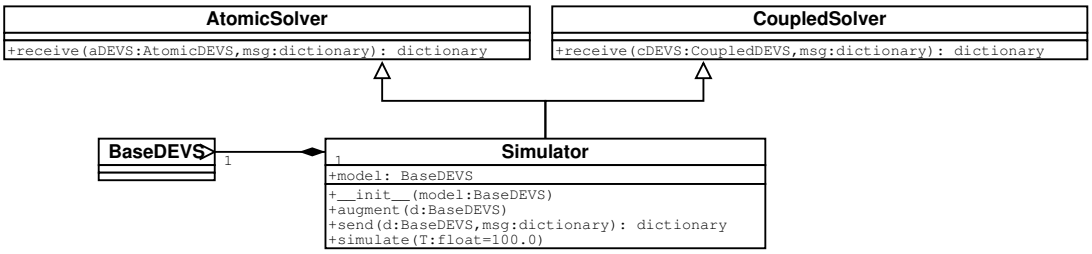

Figure 4: Simulation class architecture

We note that the t in the messages is the time stamp of that message. The other part of the message is either just a command flag (i or $*$), or an input or output dictionary (x and y). Whereas Zeigler calls the coupled-DEVS and atomic-DEVS solvers *coordinators* and *simulators* respectively, we feel that *simulator* is a term that should be reserved for the whole SE: consequently we will prefer the terms *atomic-solvers* and *coupled-solvers*. DEVS objects being passive by themselves, the role of the corresponding solvers is to operate on them when they are activated (*i.e.*, call the DEVS' methods and modify the DEVS' attributes). This being said, we have to find a mechanism to either *augment* each DEVS in the hierarchical model into a solver, or to *associate* the DEVS with an appropriate solvers.

We first considered augmentation of DEVS into solvers by means of mixin classes, but found the approach to be not memory-efficient. Associating a solver object with a DEVS object suffers from the same drawback. Hence we opted for a compromise, which consists in adding only the simulation-related attributes to the DEVS objects, and storing the solver methods in two distinct solver classes `AtomicSolver` and `CoupledSolver`. The actual `Simulator` class inherits from these two as in Figure 4, and the actual model object is added either by aggregation or composition to the simulator object upon instantiation: for instance, coupling the hierarchical DEVS described above with a simulator could be done using the following line of code (aggregation):

```
S = Simulator(Parent(1))
```

Upon instatiation, the simulator class calls its `augment` method which traverses the hierarchical model to effectively augment the DEVS objects with required attributes: in the current implementation these consist of `timeLast` and `timeNext`, that respectively hold the simulation time when the last transition occured within the given DEVS, and the scheduled time for the next internal transition. In case of coupled-DEVS, an `eventList` attribute is also added which is the list of pairs (tn_d, d), where d is a reference to a sub-model of the coupled-DEVS and tn_d is set to the `timeNext` attribute of that model (hence for a coupled DEVS `timeNext` is set to the *minimal* tn_d in its `eventList`).

The simulation is started by calling the `simulate` method:

```
S.simulate(30)
```

where the unique parameter is the length of the simulation (simulated time). The `simulate` code corresponds to that of Zeigler's *root- coordinator*: it starts by sending an initialization message (i, t) to the root-DEVS, which will cascade through the whole model. The main loop then just repeatedly sends internal state transition messages $(*, t)$ to the root DEVS until the simulation clock reaches the length of the simulation. Messages

are passed from DEVS to DEVS by means of the simulators's `send` method, which dispatches the messages to either the `AtomicSolver.receive` or the `CoupledSolver.receive` method as required. The progress of a simulation is dumped to the standard output, in a format that will be described in the next section.

A final note is in order to stress a difference between our message-passing pattern and that described in Zeigler's view. Since (y, t) messages are sent back to a parent coupled-DEVS only in response of its sending an $(*, t)$ message, the former message is "sent back" as a returned value rather than by means of the `send` method. Hence the receive methods in both solver classes treat only three kinds of messages, namely (i, t), $(*, t)$ and (x, t). This design choice, which does not impact the behavior of the simulated model, is motivated by the fact that extra-coding would otherwise have been required to distinguish between (x, t) and (y, t) messages. As a matter of fact, these messages are implemented as pairs whose first member is either an input- or output-dictionary — which are essentially indistinguishable.

2 Examples

In this section we will go through the details of modelling and simulating a system of p parallel identical processors each capable of processin jobs. The description of the system is given below:

1. jobs are created by a generator at time-interval randomly chosen in $[I_a, I_b[$ (uniform distribution), where I_a and I_b are both positive integers;
2. jobs are characterized by their size, which corresponds to the amount of time the jobs requires to be processed. Sizes are randomly chosen in $[S_a, S_b[$ (uniform distribution), where S_a and S_b are both positive integers;
3. The generator sends its job to the first available processor if any (a processor is available if it is idle or if its queue is not full), otherwise the job is discarded. Processors are cascaded as described further on;
4. each processor has a finite FIFO queue of capacity n, where unprocessed jobs can be stored. Once the queue is full, arriving jobs are discarded;
5. a soon as a processor becomes idle (*i.e.*, the current job is finished), it starts processing the first message in its queue if any;

We will model the system as a coupled DEVS, with the generator DEVS and the p processor DEVS atomic DEVS sub-models. But first of all, we represent jobs as instances of a `Job` class, which is defined in the following piece of code. The code also imports required modules, plus the `randint` function which returns random integers uniformly distributed between two positive integers:

```
from DEVS import *
from Simulator import *

from whrandom import randint

class Job:
  """ A job has a positive integer id number, and a positive integer 'size'.
  """
  IDCounter = 0
  def __init__(self, szl, szh):
    self.ID = Job.IDCounter = Job.IDCounter + 1
    self.size = randint(szl, szh)
  def __str__(self):
    return "(job %d, size %d)" % (self.ID, self.size)
```

The constructor for the `Job` class merely sets the `size` and `ID` attributes. `IDCounter` is a class attribute which gets incremented with each instantiation. The `__str__` method is called when the `print` keyword is used on an instance of the class: it is in no way compulsory, but makes the simulator's output more readable. As you might already known, the text included within triple quotes is the optional class' "documentation string".

We now describe the generator atomic-DEVS. The `Generator` class differs slightly from the `SimpleGenerator` class described before. First of all, let's examine the `timeAdvance` method: since the time to the next job is determined by calling the random number generator, we want the method to be consistent and not call the `rand` function every time it is activated. The `timeFlag` attribute signals the method when to call the random function: it is set to -1 when the generator has just sent a job, and it is set to the time until the next job otherwise. The `intTransition` method of the generator resets the `timeFlag` attribute when necessary. We also note that the `outputFnc` method sends an instance of the `Job` class through the single output port:

```
class Generator(AtomicDEVS):
  """ Generates jobs.
  The interval between job is an integer randomly chosen in the
  interval [ia, ib[ (uniform distribution).
  The sizes of the jobs is an integer randomly chosen in the
  interval [sa, sb[ (uniform distribution).
  """
  def __init__(self, ia, ib, sa, sb):
    AtomicDEVS.__init__(self)
    self.ia = ia; self.ib = ib
    self.sa = sa; self.sb = sb
    self.OUT = self.addOutPort()
    self.timeFlag = -1
  def intTransition(self):
    self.timeFlag = -1
    return self.state
  def outputFnc(self):
    p = Job(self.sa, self.sb
    self.poke(self.OUT, p)
  def timeAdvance(self):
    if self.timeFlag == -1:
      self.timeFlag = randint(self.ia, self.ib)
    return self.timeFlag
```

The bounds for both the time-interval and the job size distributions are specified as parameters to the class. We now implement the processor descriptive class: as all p processors are identical, we need only one prototype class which will be instantiated as many times as required.

Obviously, a processor will have an output port, and input port connected to the generator's output. But we are facing a problem if we want the processors to be in parallel: when the generator sends a job, we want it to be sent to just one available processor, not all of them. Hence we would requirer some kind of a dispatcher. Although this could be implemented, there exists a simpler solution to our problem (refer to Figure 5). The idea is to connect only one processor to the generator's output, and provide each processor with an extra output port (`DISCARD`): if the head processor is available when a job is received, the job is enqueued and everything goes as expected. If however the head processor is not available, the jobs it receives are immediately sent through the `DISCARD` port, which is connected to the next processor's input port. This way a job will cascade through all processors until it finds one available, or will definitely be discarded otherwise. We adopt this design even though it is suboptimal in that jobs are enqueued in the first processor whenever possible, even if all other processors are idle. Of course, the actual couplings are not described at the atomic-DEVS level, and the coupling pattern is exposed here only to motivate the descriptive class design.

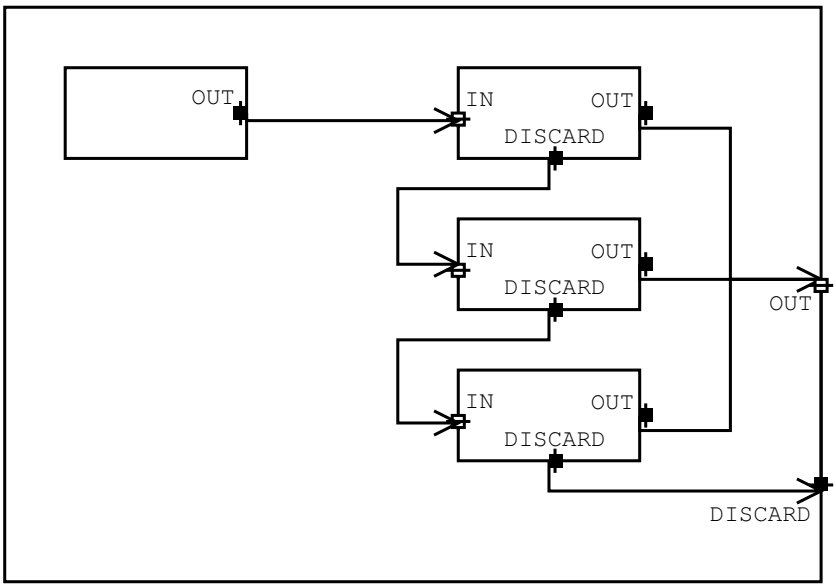

Figure 5: Parallel processors

The size of the queue is specified as a parameter to the class. The state of the processor DEVS is a triple: the first item is a flag that indicates whether the DEVS is idle (0), busy (1) or about to discard a job (2). In case the DEVS is not idle, the second item references the job currently being processed. The third item holds the time spent so far on the current process: this is necessary as the `elapsed` attribute is reset after an external event occurs (*i.e.*, when a new job is received from the generator), and we don't want the processor to be reset when a job is appended to the queue. The `timeAdvance` method uses this item to return the time until the next scheduled internal transition when the DEVS is busy. If the DEVS is busy but ready to discard a job, the method returns 0 as we want the action to be instantaneous. A large number that stands for infinity is returned if the DEVS is idle. The other methods also base their actions on the flag value in `state[0]`:

```
class Processor(AtomicDEVS):
  """Processes jobs.
  The processor has a queue of size n to store jobs.
  Jobs are processes for a time equal to their size attribute.
  When a job is finished being processed, it is sent through OUT output port.
  If the queue is full, incoming jobs are sent through DISCARD output port.
  """

  def __init__(self, n):
    AtomicDEVS.__init__(self)
    self.n = n
    self.queue = []
    self.state = [0, None, 0]
    self.IN = self.addInPort()
    self.OUT = self.addOutPort()
    self.DISCARD = self.addOutPort()

  def extTransition(self):
    p = self.peek(self.IN)
    self.queue.append(p)  # Enqueue incoming message
    if self.state[0] == 0:  # If idle, make busy with head of queue
```

```
      tempJob = self.queue[0]
      self.queue = self.queue[1:]  # trim the queue
      return [1, tempJob, 0]
    else:  # If not idle, update last field of the state
      tempElapsed = self.state[2] + self.elapsed
      if len(self.queue)>self.n:
        return [2, self.state[1], tempElapsed]
      else:
        return [1, self.state[1], tempElapsed]

  def intTransition(self):
    if self.state[0] == 1:
      if len(self.queue) > 0:  # remains busy
        tempJob = self.queue[0]
        self.queue = self.queue[1:]  # trim the queue
        return [1, tempJob, 0]
      else:  # becomes idle
        return [0, None, 0]
    elif self.state[0] == 2:
      self.queue = self.queue[:-1]  # discarded job removed from queue
      if len(self.queue)>self.n:  # another job to discard
        return [2, self.state[1], self.state[2]]
      else:  # normal busy state
        return [1, self.state[1], self.state[2]]
    elif self.state[0] == 0:  # shall not happen
      print "Something went wrong!"

  def outputFnc(self):
    if self.state[0] == 1:  # send processed job through output port
      self.poke(self.OUT, self.state[1])
    elif self.state[0] == 2:  # send discarded job through discard port
      self.poke(self.DISCARD, self.queue[-1])
    elif self.state[0] == 0:  # shall not happen
      print "Something went wrong!"

  def timeAdvance(self):
    if self.state[0] == 1:
      return self.state[1].size - self.state[2]  # time to next int trans
    elif self.state[0] == 2:
      return 0  # instantaneous action
    elif self.state[0] == 0:
      return 10000  # infinity
```

The final step of the model description is to define the coupled-DEVS descriptive class `Root`. We provide the DEVS with an output port `OUT` connected to the `OUT` port of every processor, and an output port `DISCARD` connected to the last processor's `DISCARD` port. The for loop which performs the connections is the heart of the DEVS definition. The parameters to the class all have default values and correspond respectively to the number of processors (`np`), the processors' queue size (`qs`) and the bounds for the random distributions. Note that since the number of processors is not known in advance, references to the processors cannot be saved in the local dictionary: however, the modeler could always access them through the `componentSet` attribute if need be:

```
class Root(CoupledDEVS):
  """ Wraps the generator and processors.
```

```
  """
  def __init__(self, np = 3, qs = 1, ia = 1, ib = 3, sa = 5, sb = 10):
    CoupledDEVS.__init__(self)
    self.OUT = self.addOutPort()
    self.DISCARD = self.addOutPort()
    self.GDEVS = self.addSubModel( Generator(ia,ib,sa,sb) )
    prevPORT = self.GDEVS.OUT
    for I in range(0,np):
      tmp = self.addSubModel(Processor(qs))
      self.connectPorts(prevPORT, tmp.IN)
      self.connectPorts(tmp.OUT, self.OUT)
      prevPORT = tmp.DISCARD
    self.connectPorts(prevPORT, self.DISCARD)
```

Once the model has been defined as above, it can be simulated as follows:

```
Model = Root()
S = Simulator(Model)
S.simulate(20)
```

Here, the default parameter values for the model are used, and the simulation is performed for 20 seconds (simulated time). We emphasize that if we were to call the `simulate` method a second time in this same environment, the second simulation would try to start where the first one has left: however the results would be imprevisible since some jobs might have been erased after the first simulation ended.

The output has the following format (refer to the appendix for an output of the above simulation):

- a first line indicates the current value of the clock;
- the next line indicates in which atomic-DEVS the internal transition took place at that time. Since the DEVS are represented by their `myID` attributes, it might be difficult to figure out which DEVS is meant if the order in which they were instantiated is not obvious: In our example, `A1` corresponds to the generator DEVS, and `A2` through `A4` correspond to the processors (with `A2` the head processor);
- the next few lines indicate the new state of the atomic-DEVS *after* the transition, the jobs it is sending through its output ports and the next scheduled internal transition time;
- next, a report of the external transition is printed for each of the atomic-DEVS influenced by the above DEVS: this includes the jobs received through input ports, the new state after the external transition and finally the next scheduled transition time;
- finally, the output port configuration of the root DEVS is printed.

You might have noticed that the jobs addresses that are printed with the input and output port configurations vary for a same job: for instance the job 1 is first reported at address `1a82b8` at the generator's output port, and at address `1a8270` at the head processor's input port. This makes perfect sense since the formalism specifies that *copies* of messages are to be transfered from DEVS to DEVS, rather than the messages themselves (just consider the scenario where a DEVS wants to modify the attributes of a message it *shares* with other DEVS).

When they occur, collisions are reported right after the first line. The coupled-DEVS where the collision took place is indentified, along with the involved sub-DEVS (atomic- or coupled-). The DEVS chosen by the `select` function is also shown. Let us illustrate the possible effect of collisions with two examples each involving two processors as defined above. The first example consists of two processors with 0 queue length connected in series (*i.e.*, `Processor1`'s output is connected to `Processor2`'s input):

```
class TwoProcessors(CoupledDEVS):
  def __init__(self):
    CoupledDEVS.__init__(self)
    self.OUT = self.addOutPort()
    self.Processor1 = self.addSubModel(Processor(0))
    self.Processor2 = self.addSubModel(Processor(0))
    self.connectPorts(self.Processor1.OUT, self.Processor2.IN)
    self.connectPorts(self.Processor2.OUT, self.OUT)
    self.Processor1.state = [1, Job(1,1), 0]
    self.Processor2.state = [1, Job(1,1), 0]
```

The coupled-DEVS' constructor sets the two processors' states to make sure a collision will occur. The standard `select` method is used, which returns the first item in the imminent list: in our example that will always be `Processor1` since, being instantiated first, it has a lexicographically smaller `myID` attribute. Under those circumstances, `Processor1` will undergo an internal transition first, sending its job to `Processor2`: but the job gets discarded since the latter processor is still busy with its own job. As a matter of fact, were the two internal transitions to occur *at the same time* as they should, the second processor would simultaneously finish its first job and accept the second. This same behavior could be achieved if the `select` method would be modified to choose `Processor2` first. This illustrates our point that the choice of the `select` method might impact the simulated behavior. But now consider another example.

This is exactly the same model as above, except that we add an extra coupling from `Processor2`'s output to `Processor1`'s input, thus effectively creating a loop. We would expect the two processors to swap their messages continuously: however, for the same reason as in the previous example, one of the two jobs will irremediably be lost at the very beginning of the simulation. This stresses an important weakness of the classic DEVS formalism. *parallel-DEVS* [1], which will be treated in a further work, were developped precisely to solve that problem.

3 Conclusion

In this we introduced an early prototype of a *DEVS Modeling and Simulation Package* implemented in *Python*. We emphasized that the package still suffers at this point from important limitations: one of the most important being the lack of a general method to save and load the state of a model, making it difficulty to re-initialize a model after a simulation run. But despite this situation, we have seen that the package still provides an easy way to model and simulate hierarchical DEVS. The examples also pointed to an inherent drawback of the classic DEVS formalism.

The next step of our work is to fill the gaps in the package to make it a more robust and convivial tool. We also wish to include in the simulator the possibility to gather elaborate statistics on DEVS models as a simulation progresses. In a further work parallel-DEVS, which allow internal transitions to happen synchronously in many DEVS models, will be considered.

References

[1] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of modeling and simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, San Diego, CA, second edition, 2000.

A Output From a Simulation Run

```
* * * * * * * * * *  CLOCK: 1.000000


   INTERNAL TRANSITION: A1
     New State: None
     Output Port Configuration:
       port0: (job 1, size 8) at <__main__.Job instance at 1a82b8>
     Next scheduled internal transition at time 4.000000

   EXTERNAL TRANSITION: A2
     Input Port Configuration:
       port0: (job 1, size 8) at <__main__.Job instance at 1a8270>
     New State: [1, <__main__.Job instance at 1a8270>, 0]
     Next scheduled internal transition at time 9.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 4.000000


   INTERNAL TRANSITION: A1
     New State: None
     Output Port Configuration:
       port0: (job 2, size 7) at <__main__.Job instance at 1a82b8>
     Next scheduled internal transition at time 7.000000

   EXTERNAL TRANSITION: A2
     Input Port Configuration:
       port0: (job 2, size 7) at <__main__.Job instance at 1a7a20>
     New State: [1, <__main__.Job instance at 1a8270>, 3.0]
     Next scheduled internal transition at time 9.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 7.000000


   INTERNAL TRANSITION: A1
     New State: None
     Output Port Configuration:
       port0: (job 3, size 10) at <__main__.Job instance at 1a82b8>
     Next scheduled internal transition at time 9.000000

   EXTERNAL TRANSITION: A2
     Input Port Configuration:
       port0: (job 3, size 10) at <__main__.Job instance at 1a82e8>
     New State: [2, <__main__.Job instance at 1a8270>, 6.0]
     Next scheduled internal transition at time 7.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 7.000000


   INTERNAL TRANSITION: A2
     New State: [1, <__main__.Job instance at 1a8270>, 6.0]
     Output Port Configuration:
       port0: None
       port1: (job 3, size 10) at <__main__.Job instance at 1a82e8>
```

```
      Next scheduled internal transition at time 9.000000

   EXTERNAL TRANSITION: A3
     Input Port Configuration:
       port0: (job 3, size 10) at <__main__.Job instance at 1a82d0>
     New State: [1, <__main__.Job instance at 1a82d0>, 0]
     Next scheduled internal transition at time 17.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 9.000000


   Collision occured in C1, involving:
         A1
         A2
     select chooses A1

   INTERNAL TRANSITION: A1
     New State: None
     Output Port Configuration:
       port0: (job 4, size 8) at <__main__.Job instance at 1a82b8>
     Next scheduled internal transition at time 10.000000

   EXTERNAL TRANSITION: A2
     Input Port Configuration:
       port0: (job 4, size 8) at <__main__.Job instance at 1a8258>
     New State: [2, <__main__.Job instance at 1a8270>, 8.0]
     Next scheduled internal transition at time 9.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 9.000000


   INTERNAL TRANSITION: A2
     New State: [1, <__main__.Job instance at 1a8270>, 8.0]
     Output Port Configuration:
       port0: None
       port1: (job 4, size 8) at <__main__.Job instance at 1a8258>
     Next scheduled internal transition at time 9.000000

   EXTERNAL TRANSITION: A3
     Input Port Configuration:
       port0: (job 4, size 8) at <__main__.Job instance at 1a8240>
     New State: [1, <__main__.Job instance at 1a82d0>, 2.0]
     Next scheduled internal transition at time 17.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 9.000000


   INTERNAL TRANSITION: A2
     New State: [1, <__main__.Job instance at 1a7a20>, 0]
     Output Port Configuration:
       port0: (job 1, size 8) at <__main__.Job instance at 1a8270>
       port1: None
     Next scheduled internal transition at time 16.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: (job 1, size 8) at <__main__.Job instance at 1a8270>
```

```
       port1: None

* * * * * * * * * *  CLOCK: 10.000000


   INTERNAL TRANSITION: A1
     New State: None
     Output Port Configuration:
       port0: (job 5, size 7) at <__main__.Job instance at 1a82b8>
     Next scheduled internal transition at time 11.000000

   EXTERNAL TRANSITION: A2
     Input Port Configuration:
       port0: (job 5, size 7) at <__main__.Job instance at 1a8360>
     New State: [1, <__main__.Job instance at 1a7a20>, 1.0]
     Next scheduled internal transition at time 16.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 11.000000


   INTERNAL TRANSITION: A1
     New State: None
     Output Port Configuration:
       port0: (job 6, size 8) at <__main__.Job instance at 1a82b8>
     Next scheduled internal transition at time 12.000000

   EXTERNAL TRANSITION: A2
     Input Port Configuration:
       port0: (job 6, size 8) at <__main__.Job instance at 1a7a08>
     New State: [2, <__main__.Job instance at 1a7a20>, 2.0]
     Next scheduled internal transition at time 11.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 11.000000


   INTERNAL TRANSITION: A2
     New State: [1, <__main__.Job instance at 1a7a20>, 2.0]
     Output Port Configuration:
       port0: None
       port1: (job 6, size 8) at <__main__.Job instance at 1a7a08>
     Next scheduled internal transition at time 16.000000

   EXTERNAL TRANSITION: A3
     Input Port Configuration:
       port0: (job 6, size 8) at <__main__.Job instance at 1a8258>
     New State: [2, <__main__.Job instance at 1a82d0>, 4.0]
     Next scheduled internal transition at time 11.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 11.000000


   INTERNAL TRANSITION: A3
     New State: [1, <__main__.Job instance at 1a82d0>, 4.0]
     Output Port Configuration:
       port0: None
       port1: (job 6, size 8) at <__main__.Job instance at 1a8258>
```

```
      Next scheduled internal transition at time 17.000000

   EXTERNAL TRANSITION: A4
     Input Port Configuration:
       port0: (job 6, size 8) at <__main__.Job instance at 1a8318>
     New State: [1, <__main__.Job instance at 1a8318>, 0]
     Next scheduled internal transition at time 19.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 12.000000


   INTERNAL TRANSITION: A1
     New State: None
     Output Port Configuration:
       port0: (job 7, size 6) at <__main__.Job instance at 1a82b8>
     Next scheduled internal transition at time 14.000000

   EXTERNAL TRANSITION: A2
     Input Port Configuration:
       port0: (job 7, size 6) at <__main__.Job instance at 1a8378>
     New State: [2, <__main__.Job instance at 1a7a20>, 3.0]
     Next scheduled internal transition at time 12.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 12.000000


   INTERNAL TRANSITION: A2
     New State: [1, <__main__.Job instance at 1a7a20>, 3.0]
     Output Port Configuration:
       port0: None
       port1: (job 7, size 6) at <__main__.Job instance at 1a8378>
     Next scheduled internal transition at time 16.000000

   EXTERNAL TRANSITION: A3
     Input Port Configuration:
       port0: (job 7, size 6) at <__main__.Job instance at 1a7990>
     New State: [2, <__main__.Job instance at 1a82d0>, 5.0]
     Next scheduled internal transition at time 12.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 12.000000


   INTERNAL TRANSITION: A3
     New State: [1, <__main__.Job instance at 1a82d0>, 5.0]
     Output Port Configuration:
       port0: None
       port1: (job 7, size 6) at <__main__.Job instance at 1a7990>
     Next scheduled internal transition at time 17.000000

   EXTERNAL TRANSITION: A4
     Input Port Configuration:
       port0: (job 7, size 6) at <__main__.Job instance at 1a82e8>
     New State: [1, <__main__.Job instance at 1a8318>, 1.0]
     Next scheduled internal transition at time 19.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
```

```
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 14.000000


   INTERNAL TRANSITION: A1
     New State: None
     Output Port Configuration:
       port0: (job 8, size 9) at <__main__.Job instance at 1a82b8>
     Next scheduled internal transition at time 16.000000

   EXTERNAL TRANSITION: A2
     Input Port Configuration:
       port0: (job 8, size 9) at <__main__.Job instance at 1a8300>
     New State: [2, <__main__.Job instance at 1a7a20>, 5.0]
     Next scheduled internal transition at time 14.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 14.000000


   INTERNAL TRANSITION: A2
     New State: [1, <__main__.Job instance at 1a7a20>, 5.0]
     Output Port Configuration:
       port0: None
       port1: (job 8, size 9) at <__main__.Job instance at 1a8300>
     Next scheduled internal transition at time 16.000000

   EXTERNAL TRANSITION: A3
     Input Port Configuration:
       port0: (job 8, size 9) at <__main__.Job instance at 1a8270>
     New State: [2, <__main__.Job instance at 1a82d0>, 7.0]
     Next scheduled internal transition at time 14.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 14.000000


   INTERNAL TRANSITION: A3
     New State: [1, <__main__.Job instance at 1a82d0>, 7.0]
     Output Port Configuration:
       port0: None
       port1: (job 8, size 9) at <__main__.Job instance at 1a8270>
     Next scheduled internal transition at time 17.000000

   EXTERNAL TRANSITION: A4
     Input Port Configuration:
       port0: (job 8, size 9) at <__main__.Job instance at 1a7b58>
     New State: [2, <__main__.Job instance at 1a8318>, 3.0]
     Next scheduled internal transition at time 14.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 14.000000


   INTERNAL TRANSITION: A4
     New State: [1, <__main__.Job instance at 1a8318>, 3.0]
     Output Port Configuration:
```

```
        port0: None
        port1: (job 8, size 9) at <__main__.Job instance at 1a7b58>
      Next scheduled internal transition at time 19.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
        port0: None
        port1: (job 8, size 9) at <__main__.Job instance at 1a7b58>

* * * * * * * * * *  CLOCK: 16.000000


   Collision occured in C1, involving:
         A1
         A2
     select chooses A1

   INTERNAL TRANSITION: A1
     New State: None
     Output Port Configuration:
       port0: (job 9, size 8) at <__main__.Job instance at 1a82b8>
     Next scheduled internal transition at time 17.000000

   EXTERNAL TRANSITION: A2
     Input Port Configuration:
       port0: (job 9, size 8) at <__main__.Job instance at 1a8168>
     New State: [2, <__main__.Job instance at 1a7a20>, 7.0]
     Next scheduled internal transition at time 16.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
        port0: None
        port1: None

* * * * * * * * * *  CLOCK: 16.000000


   INTERNAL TRANSITION: A2
     New State: [1, <__main__.Job instance at 1a7a20>, 7.0]
     Output Port Configuration:
       port0: None
       port1: (job 9, size 8) at <__main__.Job instance at 1a8168>
     Next scheduled internal transition at time 16.000000

   EXTERNAL TRANSITION: A3
     Input Port Configuration:
       port0: (job 9, size 8) at <__main__.Job instance at 1a7a08>
     New State: [2, <__main__.Job instance at 1a82d0>, 9.0]
     Next scheduled internal transition at time 16.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
        port0: None
        port1: None

* * * * * * * * * *  CLOCK: 16.000000


   Collision occured in C1, involving:
         A2
         A3
     select chooses A2

   INTERNAL TRANSITION: A2
     New State: [1, <__main__.Job instance at 1a8360>, 0]
     Output Port Configuration:
       port0: (job 2, size 7) at <__main__.Job instance at 1a7a20>
       port1: None
     Next scheduled internal transition at time 23.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
```

```
       port0: (job 2, size 7) at <__main__.Job instance at 1a7a20>
       port1: None

* * * * * * * * * *  CLOCK: 16.000000


   INTERNAL TRANSITION: A3
     New State: [1, <__main__.Job instance at 1a82d0>, 9.0]
     Output Port Configuration:
       port0: None
       port1: (job 9, size 8) at <__main__.Job instance at 1a7a08>
     Next scheduled internal transition at time 17.000000

   EXTERNAL TRANSITION: A4
     Input Port Configuration:
       port0: (job 9, size 8) at <__main__.Job instance at 1a83d8>
     New State: [2, <__main__.Job instance at 1a8318>, 5.0]
     Next scheduled internal transition at time 16.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 16.000000


   INTERNAL TRANSITION: A4
     New State: [1, <__main__.Job instance at 1a8318>, 5.0]
     Output Port Configuration:
       port0: None
       port1: (job 9, size 8) at <__main__.Job instance at 1a83d8>
     Next scheduled internal transition at time 19.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: (job 9, size 8) at <__main__.Job instance at 1a83d8>

* * * * * * * * * *  CLOCK: 17.000000


   Collision occured in C1, involving:
         A1
         A3
     select chooses A1

   INTERNAL TRANSITION: A1
     New State: None
     Output Port Configuration:
       port0: (job 10, size 8) at <__main__.Job instance at 1a82b8>
     Next scheduled internal transition at time 19.000000

   EXTERNAL TRANSITION: A2
     Input Port Configuration:
       port0: (job 10, size 8) at <__main__.Job instance at 1a8138>
     New State: [1, <__main__.Job instance at 1a8360>, 1.0]
     Next scheduled internal transition at time 23.000000

   ROOT DEVS' OUTPUT PORT CONFIGURATION:
       port0: None
       port1: None

* * * * * * * * * *  CLOCK: 17.000000


   INTERNAL TRANSITION: A3
     New State: [1, <__main__.Job instance at 1a8240>, 0]
     Output Port Configuration:
       port0: (job 3, size 10) at <__main__.Job instance at 1a82d0>
```

```
      port1: None
    Next scheduled internal transition at time 25.000000

  ROOT DEVS' OUTPUT PORT CONFIGURATION:
      port0: (job 3, size 10) at <__main__.Job instance at 1a82d0>
      port1: None

* * * * * * * * * *  CLOCK: 19.000000


  Collision occured in C1, involving:
        A1
        A4
    select chooses A1

  INTERNAL TRANSITION: A1
    New State: None
    Output Port Configuration:
      port0: (job 11, size 5) at <__main__.Job instance at 1a82b8>
    Next scheduled internal transition at time 21.000000

  EXTERNAL TRANSITION: A2
    Input Port Configuration:
      port0: (job 11, size 5) at <__main__.Job instance at 1a8150>
    New State: [2, <__main__.Job instance at 1a8360>, 3.0]
    Next scheduled internal transition at time 19.000000

  ROOT DEVS' OUTPUT PORT CONFIGURATION:
      port0: None
      port1: None

* * * * * * * * * *  CLOCK: 19.000000


  Collision occured in C1, involving:
        A2
        A4
    select chooses A2

  INTERNAL TRANSITION: A2
    New State: [1, <__main__.Job instance at 1a8360>, 3.0]
    Output Port Configuration:
      port0: None
      port1: (job 11, size 5) at <__main__.Job instance at 1a8150>
    Next scheduled internal transition at time 23.000000

  EXTERNAL TRANSITION: A3
    Input Port Configuration:
      port0: (job 11, size 5) at <__main__.Job instance at 1a7b70>
    New State: [1, <__main__.Job instance at 1a8240>, 2.0]
    Next scheduled internal transition at time 25.000000

  ROOT DEVS' OUTPUT PORT CONFIGURATION:
      port0: None
      port1: None

* * * * * * * * * *  CLOCK: 19.000000


  INTERNAL TRANSITION: A4
    New State: [1, <__main__.Job instance at 1a82e8>, 0]
    Output Port Configuration:
      port0: (job 6, size 8) at <__main__.Job instance at 1a8318>
      port1: None
    Next scheduled internal transition at time 25.000000

  ROOT DEVS' OUTPUT PORT CONFIGURATION:
      port0: (job 6, size 8) at <__main__.Job instance at 1a8318>
```

```
        port1: None
}
```