

DEVS Trace Plotter

(Technical Report)

Hongyan (Bill) Song

McGill University

Winter 2006

Contents

Purposes and Motivations.....	3
Architecture.....	4
The XML Trace DTD.....	6
Requirements for Models.....	8
Design and Implementation.....	8
Trace Parser.....	8
ModelParser.....	10
TmpEvent and DevsEvent.....	11
StateParser and SimpleStateParser.....	11
The Visual Event Plotter	12
Two Different Plotters.....	14
An Example of Using Trace Plotter.....	14
Model Level Support.....	14
The Job Class.....	14
The Generator State.....	15
The Processor State.....	16
Simulation Level Support.....	18
Run the Trace Plotter.....	19
The splotter.....	19
The cplotter.....	20
An Example of The State Parser.....	20

Purposes and Motivations

The general goals of doing simulation are to verify the correctness of models and analyze the performance of algorithms. These tasks can both be done at modeling level, which means that the modelers should provide the necessary information for model verification and performance analysis when they build models. This is possible, however it is not very efficient. If the simulator builder can provide some functionality to facilitate the process, things may become easier.

Because this functionality is an bonus from tool builders to final users, and it has nothing to do with DEVS specification itself. Different existing DEVS simulators deal with this issue differently. DEVSJava provides the functionality of visualizing the behavior of message passing among the connected components, through which users can see the process of messages passing from one component to another when they run the simulation. Python DEVS dumps out the whole model simulation trace to screen. By reading the trace output the user can analyze the model behavior. ADEVS does not provide any explicit output to facilitate this process by the simulator level. It totally depends on the users themselves to provide the information they need at model level.

No matter what way you take to finish this job, the simulation trace provides valuable information for model verification and performance analysis. So the key issue is how to effectively analyze the simulation trace. Trace animation is a intuitive and direct way to see the result and effect of the simulation, but it does not help much for detailed analysis. You can only see the instant effect of the simulation. It is hard to see the true relation between one event and another. Dumping out all the simulation trace including the state information is a good idea. However, for a complex dynamic system, the trace file of the simulation could be huge. Without proper tools support, reading this kind of file needs great energy and courage.

In order to facilitate the process of the trace analysis, we built a generic trace plotter, which can visualize textual DEVS trace in an easily understanding way. This plotter was built based on PythonDEVS. We call it a generic trace plotter, because it provides the possibility of plotting trace file generated from any DEVS simulator if it follows certain format rules.

A big challenge for building such a tool is how to format the trace file. Because we want the plotter be open and generic, the file format first should be open, easy to be understood, and easy to be accepted. After a careful study and discuss, we decided to save the trace file in XML format. There are many benefits of using XML format. Firstly, XML as standardized way for data storage has been widely accepted by people both in industry and academic area. There would be less resistance and no big curve for learning. Secondly, XML can be defined by DTD or XML Schema, you can validate a XML file against its DTD or XML Schema. This makes it easier for validating a trace file before you plot it. Thirdly, with the XSLT support, the update and evolution of a trace file will be not a big issue in the future.

Another challenge comes from DEVS. At present, there is no a commonly accepted standard for DEVS. The currently available DEVS simulators use variant terms to describe models. How to represent the trace information using different terms in an easily understanding and accepting way is not a small problem. An important term in DEVS model is the sequential state. Sequential state plays an important role in defining DEVS models, similarly, it is the key for trace analysis. The same event occurring at different sequential state will have very different consequences. The ironic is that DEVS does not distinguish the sequential state clearly from the system state or the component state. So when confusion arise, people use different terms to refer it, such as partial state(DSDEVS), phase

(DEVSTJava) and so forth. Because there is no clear definition, it is impossible for the plotter to determine which attribute in the dumped system state represent this term automatically. The plotter does not know this information, however the user certainly know which attributes represent the sequential state and which attributes they interest. So we make it open to the user to make the decision at run time.

For the convenience of users with different level of computer programming knowledge and users with different level of plotting requirements, we provided two plotters which are called simple plotter, `splotter.py`, and customized plotter, `cplotter.py`. The simple plotter assumes that the sequential state of a DEVS model is determined by one attribute of the model state. So the user just needs to select the model interested, and the attribute determining the sequential state, the plotter will visualize the trace automatically. The customized plotter is more powerful and complex. It provides a programming interface, by which the user can program themselves to determine how the sequential state is calculated by from the whole state, and which part of the state attributes should be shown and how to shown in the plotter. In the simple plotter, the state attributes shown on the plotter are taken from the trace file directly, which is generated by the simulator using the state's string conversion function specified by the modeler.

Though we use the trace file generated from the simulator, but the trace cannot come from nowhere. We need some kind of support from the modeler. The modeler should write two functions where they build the model. One function tells the simulator how to change the model state into string, and the second instructs the simulation how to converse the state into XML format. After the modeler finishes these two functions, the simulator will take care other things.

The following part of the report is organized as this. First, we will discuss the design architecture of the plotter, by which we can get a general idea of the design. Then I will explain each part of the architecture in more details. After that, I will give an example to see how to use the simple and customized plotters.

Architecture

The architecture of the trace plotter is shown as the figure 1. The simulator generates a XML format trace file. The trace parser parses the XML trace file into an intermediate trace file. A dynamic parser takes request from the trace plotter and parses the intermediate trace file into a format that can be recognized by the plotter according to the request, and return the result to the plotter. The dynamic parser parses the intermediate trace file every time when the plotting criteria change.

The reason for existence of intermediate trace file and dynamic parser is purely a performance consideration. We save all the trace of a simulated system in one trace file. If every time the user changes the filter we parse all the trace file again, it will be time consuming. So we use two parsers to make the plotting more efficient. The first one trace parser takes the raw XML trace file, half parse them and separate the trace file according to the models the events belong to. We say half parse because the all other parts of the trace are parsed excepts the state. Because the trace are half parsed and separated, we need only parse the state part and traverse the trace based on the model the user interests, which can save a lot of time for parsing and traversing.

For example, if we plot a trace file for a coupled DEVS model including four sub-models. Without the

intermediate trace, every time you what plot the trace of one sub-model, you need to traverse and parse all the traces to get the trace information for the specific sub-model. When with the intermediate trace, you will get the sub-model trace immediately, and only need to parse the state parse of the trace any time when you plot a model trace.

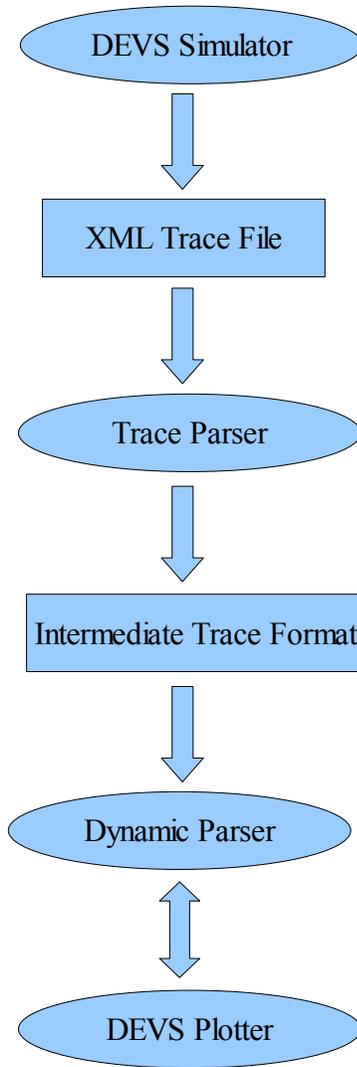


Figure 1. the Architecture of DEVS Trace Plotter

The intermediate trace format is an in-memory data structure, we did not output it into disk. So every time a trace file loaded into the memory we need parse the it into intermediate format one time. If the trace file is really huge, outputting the intermediate trace to the hard drive could not be a bad choice.

The XML Trace DTD

As we mentioned earlier, one of the benefits for using XML is the capability for file validation. The validation is done by checking whether an XML file conforms to a DTD or XML Schema. XML Schema and DTD serve for a same purpose, the major difference is that the expressiveness of XML Schema is more powerful than DTD. Correspondingly, the XML Schema is more complex than DTD. For simplification reason, we used DTD rather than XML Schema to specify our trace file. The DTD we used for specifying the XML trace is as below.

1. `<!ELEMENT trace (event+)>`
2. `<!ELEMENT event (model, time, kind, port*, state)>`
3. `<!ELEMENT model (#PCDATA)>`
4. `<!ELEMENT time (#PCDATA)>`
5. `<!ELEMENT kind (IN|EX|#PCDATA)>`
6. `<!ELEMENT port (message)>`
7. `<!ELEMENT message (#PCDATA)>`
8. `<!ELEMENT state (attribute+)>`
9. `<!ELEMENT attribute (name, type, value+)>`
10. `<!ELEMENT name (#PCDATA)>`
11. `<!ELEMENT type (#PCDATA)>`
12. `<!ELEMENT value (#PCDATA|attribute)*>`
13. `<!ATTLIST port name CDATA #IMPLIED>`
14. `<!ATTLIST port category (I|O) #REQUIRED>`
15. `<!ATTLIST attribute category (P|C | PC | CC) #REQUIRED>`
16. `<!ENTITY title "DEVS Simulation Trace">`
17. `<!ENTITY publisher "MSDL">`
18. `<!ENTITY copyright "Copyright 2006 MSDL">`

The meanings of some elements are explained as below.

Line 1 means the root element in a XML trace file is the XML element called trace, and a trace element includes one or more event elements. In XML format, it will be written like:

```
<trace>
  <event> ... </event>
  <event> ... </event>
  .....
  <event> ... </event>
</trace>
```

Line 2 means that an event element is composed of a model element, a time element, a kind element, and zero or more port element, and a state element. So an XML format event could be like this:

```
<event>
  <model> ... </model>
  <time> ... </time>
  <kind> ... </kind>
  <port> ... </port> * can be absent
  <state> ... </state>
</event>
```

Here the model element means the full qualified name of a model. A full qualified model name means that the hierarchal relation among models can be seen from the model name. For example, if we simulate a coupled model A, the structure of the model A is shown as Figure 2, then the full qualified names for models B, C, D, and E are A.B, A.C, A.B.D, and A.B.E respectively. The time of an event means the logical time at which the event happens.

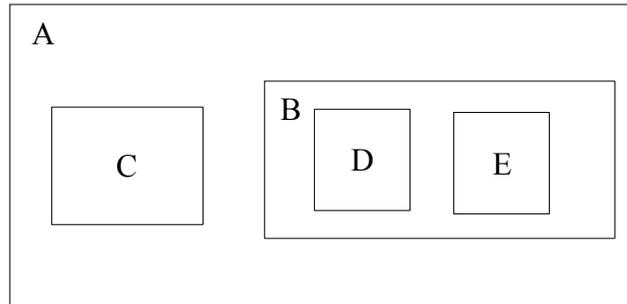


Figure 2. Example of Full Qualified Name

Line 5 specifies the data allowed for the kind element. For an atomic model, the kind for an event can be the value of 'IN' or 'EX', where 'IN' means internal event, 'EX' means external event. For an coupled model, the kind can be empty or anything allowed in XML PCDATA. The reason we specify the value of kind for atomic models is that we need this information to distinguish an external event and an internal event when we show it in the visual plotter.

Line 6, 7, 13 and 14 specify the structure of the port element. The port element has two attributes, name and category and one element. The name attribute is just the text name of the port in question. The category attribute of a port element can be 'I' or 'O', where 'I' means input port and 'O' means output port. The message element is the current message or event in the port. The represent of the message is in textual format, if the message structure is complex, the modeler should provide a str function to transform it into text format.

An valid port element can be like this:

```

<port name="in", category="I">
  <message>Job id=1, size=3</message>
</port>
  
```

Line 8, 9, 10, 11, 12, and 15 specify the structure of the state element. A state is composed of one or more attributes. Each attribute has an attribute called category, and three elements name, type, and value. The category attribute of the attribute element can be 'P', 'C', 'PC', or 'CC', where 'P' means primitive type, 'C' means customized type, 'PC' means a collection of values of a primitive type, and 'CC' means a collection of values of a customized type. The type for an attribute can be any text that meaningful as a type of an attribute.

The below is an example of customized attribute. The attribute category is "C", and name is currentJob, the type is Job which is a user defined type. The Job type has two attributes, ID and size. In this case, the currentJob's value is ID=1 and size= 4.26872848822 .

```

<attribute category="C">
  <name>currentJob</name>
  <type>Job</type>
  <value>
    <attribute category="P">
      <name>ID</name>
      <type>Integer</type>
      <value>1</value>
    </attribute>
    <attribute category="P">
      <name>size</name>
      <type>Float</type>
      <value>4.26872848822</value>
    </attribute>
  </value>
</attribute>

```

Requirements for Models

In order to get the information needed for trace plotting. There are some extra requirements for modelers when they build a DEVS model. Each model must have a attribute call state, which is a class or structure variable that includes all the information you want to plot. This state variable must have a function called toXML, which is responsible for transforming the model state into XML format. Optionally, the state variable has a str function, which can transform the model state into pure text format.

Design and Implementation

Trace Parser

The task of the TraceParser class is to traverse and parse a whole raw XML trace file into lists of TmpEvent, and get the following information: the set of models included in the trace file, the events of each model, the state information of each model, the type of each model, and the components included in each coupled model.

After the parsing, the set of models is saved in modelSet, in which each model is identified by its full qualified name. For coupled model A in Figure 2, the modelSet is like this {A, A.B, A.C, A.B.D, A.B.E}. The events of each model are saved into a map structure called modelEvents, in which model names are used as keys and a list of model events as value of a corresponding key. For a trace file like Figure 3. (A), the key and value pairs of modelEvents will be like Figure 3. (B).

TraceParser
<pre> -fileName: String -modelSet: Set -mdoelEvents: Map (Dictionary) -modelState: Map (Dictionary) -modelType: Map (Dictionary) -coupleComp: Map (Dictionary) </pre>
<pre> +init(fileName:String) -initialize(): void -parseEvents(): void +getModelType(modelName:String): String +getSubComponents(modelName:String): List +getModels(): Set +getModelEvents(modelName:String): List +getModelStateAttributes(modelName:String): List </pre>

```

<trace>
  <event><model>A.C</model>e1</event>
  <event><model>A.B.E</model>e2</event>
  <event><mdoel>A.B.D</model>e3 </event>
  <event><model>A.C</model>e4</event>
  <event><model>A.B.E</model>e5</event>
  <event><model>A.B.E</model>e6</event>
</trace>

```

(A)

Keys	Value
A	e1, e2, e3, e4, e5, e6
A.B	e2, e3, e5, e6
A.C	e1, e4
A.B.D	e3
A.B.E	e2, e5, e6

(B)

Figure 3. Trace file parsing example

Though both the events for coupled models and events for atomic models are all saved in the same map structure. The structure of a coupled model event and the structure of an atomic model event are different. Events for coupled models are parsed into DevsEvents, however, events for atomic models are parsed into TmpEvents. The reason for this difference is that we do not know which part of the state will be used for plotting, so we should keep the all the state information for a atomic model, by which way we can dynamic parse the state when users change their plotting criteria.

The modelState is also a map structure, which saves the state information about each model indexed by the model name. The state information for each model is simply a list of attribute names of the model

state. Because the coupled models have no state information, it only works for atomic models. ModelType is used to indicate a model is an atomic model or coupled model. An atomic model is indicated as “A”, and a coupled model is represented by “C”. For coupled models, the map structure coupleComp maps from a model name to a list of its atomic models. For the model A in Figure 2, the coupleComp will look like this.

Keys	Value
A	A.C, A.B.D, A.B.E
A.B	A.B.D, A.B.E

Figure 4. Example of coupleComp

The operations of the TraceParser class just return the values as that are indicated by function names. The meanings are obvious.

ModelParser

The model parser is the one who plays the role to the dynamic parser in the Figure 1. This class takes a list of TmpEvents and a state parser as parameters. The list of TmpEvents are the events of the model that the user wants to plot using the specific state parser. The rules of how to generate a sequential state from the model state are specified in the state parser. The task of the ModelParser it to translate the TmpEvents into DevsEvent that can be plotted by the visual plotter using the specified state parser.

ModelParser
-stateParser: StateParser -events: List -states: List
+init(modelEvents:List, stateParser:StateParser) +getStateParser(): StateParser +getEvents(): List +getStates(): List

Besides the constructor, there are three public functions of this class. The getStateParser returns the current StateParser used, the getEvents returns the generated DevsEvents, and the getStates returns the set of sequential states generated for the events in question.

TmpEvent and DevsEvent

TmpEvent
<code>-time: Float</code> <code>-type: String</code> <code>-portInfo: String</code> <code>-xmlState: String</code>
<code>+init(time:Float,type:String="",portInfo:String="",state:String="")</code> <code>+str(): String</code> <code>+getTime(): Float</code> <code>+getType(): String</code> <code>+getPortInfo(): String</code> <code>+getXmlState(): String</code>

DevsEvent
<code>-time: Float</code> <code>-type: String</code> <code>-state: String</code> <code>-content: String</code>
<code>+init(time:Float,type:String="",state:String="",content:String="")</code> <code>+str(): String</code> <code>+getTime(): Float</code> <code>+getType(): String</code> <code>+getState(): String</code> <code>+getContent(): String</code>

The TmpEvent and DevsEvent contain the same information but in different format and for different purposes. The reason for the existence of the TmpEvent is purely a performance consideration. The meanings of the time and type attributes in TmpEvent and DevsEvent are the same. The state attribute of a DevsEvent is the sequential state generated from a corresponding TmpEvent, and the content of a DevsEvent is the concatenation of portInfo and parsed xmlState of the TmpEvent.

StateParser and SimpleStateParser

StateParser
<code>+getString(xmlState:String): String</code> <code>+getSeqState(xmlState:String): String</code>

SimpleStateParser
-attrName: String
+init(attrName:String) +getString(xmlState:String): String +getSeqState(xmlState:String): String

The StateParser and the SimpleStateParser are almost the same. We use two classes for different purposes. The getString function returns the textual representation of a model state specified by the xmlState parameters, and getSeqState returns the sequential state of the xmlState. The StateParser is an interface for users. The user can specify what they want to show the state in textual format in the getString function and how to generate a sequential state in the getSeqState function. The SimpleStateParser adds a limitation on the functionalities of the StateParser. It assumes that the sequential state is generated based on the value of an attribute of the xmlState, and the textual representation of the xmlState is specified by modelers when the model is built. So it returns the value of the attrName as sequential state, and the default textual information generated in the simulation as textual representation of the state. If there were no default str function for the model state, the getString may return empty string.

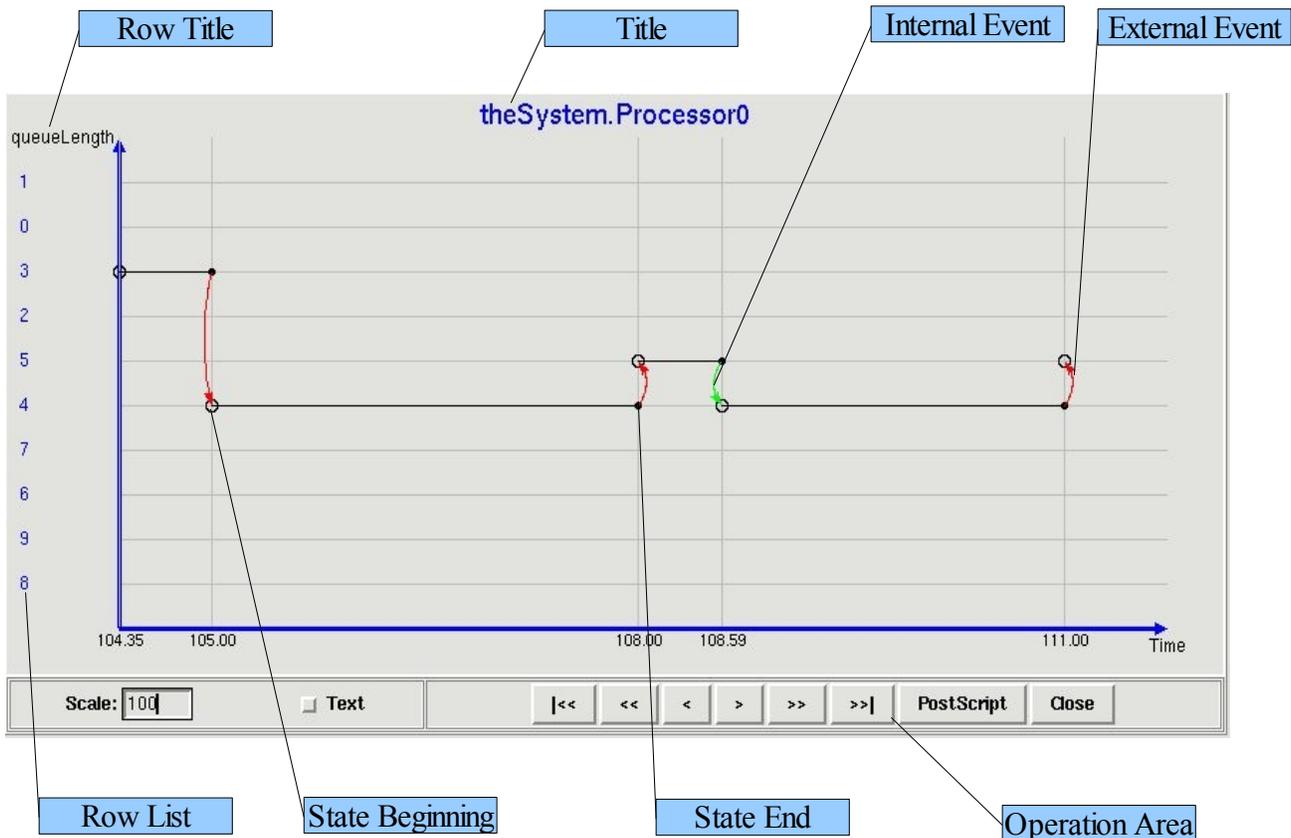
The Visual Event Plotter

EventPlotter
+parent: Frame +rowList: List +eventList: List +title: String +rowTitle: String +scale: Float
+init(parent:Frame=None,rowList:List=[], eventList:List=[],title:String='DEVS Plotter', rowTitle:String='state',plotterScale:Float=1.0) +makeOperationArea(): void +showTraceText(): void +onDoubleClick(event:Event): void +onRightClick(event:Event): void +setScale(event:Event): void +initPanel(event:Event): void +updateDisplay(): void +previousOne(): void +nextOne(): void +toBegin(): void +toEnd(): void +previousPage(): void +nextPage(): void +saveToPostScript(): void +closeWindow(): void

The purpose of the event parsing is for event plotting. The plotting task is done by the visual event plotter. The structure of the event plotter looks like below.

The parent is a Frame window in which the plotter will be displayed. The meaning of rowList, title, and rowTitle are shown on the figure in next page. The eventList is a list of DevsEvents to be plotted. The scale is the parameter specifies how to map the pixel of the screen to the time unit of events. The default is 1.0, which means that one pixel on the screen corresponds to one time unit. The trick for DEVS event plotting is on the values of the rowList. Originally, the rowList only includes the sequential states of the model being plotted. However, the nature of DEVS that a model can only change its state when an event occurs. So, besides the sequential state, the values of any state variable can be used as rowList to be plotted.

When the sequential states are used as rowList, we can see the sequential states change along the occurring time of events. While the values of a state variable or combination of state variables, we can see when the values change, and which events cause the value change. The below is a screen shot of the event plotter with some marks.



We can see from the figure above that the values of the state variable queueLength are used as the rowList. So we can read from the picture that, the values of the queueLength varies from 0 to 9 in this simulation. At time 104.35, queueLength changes to 3, and it keeps this value until 105.00, at which an external event occurs, and then the queueLength changes to 4, and the value 4 is kept until time 108.00, when another external event happens, the value changes to 5, and then at time 108.59, an internal event

occurs, the value goes to 4. The value of `queueLength` changes at irregular time periods like this, until the end of the simulation.

The meanings of some shapes are shown on the marks. An hollow circle represents the beginning of a state or value; the solid circle means the end of a state or value; the red curve with an arrow means an external event occurs at a specific time, and the arrow indicates the where the new state or value goes; the green curve with an arrow means an internal event happens at a certain time, and the arrow has the same meaning as that of the external event.

Two Different Plotters

In order to make it easier for use, we provided two different versions of trace plotter, a simple one, `splotter.py`, and a complex or customized one, `cplotter.py`. The functionalities of these two plotters are the same. The difference lies on the way that you specify the plotting criteria. In the simple plotter, the plotting filter is specified by the value of the property you select. you do not and cannot specify a state parser for the simple plotter. In the `cplotter`, the user needs to specify the rules to generate the textual representation of the state and the way to generate the sequential state from the XML format of the state.

An Example of Using Trace Plotter

As we mentioned earlier, in order to generate the XML trace file we need some support both at modeling level and simulation level. The below is an example of how to generate the XML trace file from Python DEVS.

Model Level Support

We need two functions for a model state at model level. One function to generate a textual representation of the state and another function to generate the XML representation of the state. In Python, there is a default function called `__str__` used by Python to generate textual representation of an object. So we expect that a state object of any DEVS model should have a `__str__` function. There is no default function for generating XML representation. We gave the function name as `toXML`, and also we expect the state object have such a function.

The Job Class

```
class Job:
    """ A job has a unique, strictly positive integer ID number,
        and a positive 'size', in this case directly indicating
        the time a processor needs to process this job.
    """
    IDCounter = 0 # class variable, globally unique
```

```

def __init__(self, szl, szh):
    """ The job size is drawn from the interval [szl, szh[
        (uniform distribution)
    """
    Job.IDCounter += 1
    self.ID = Job.IDCounter

    self.size = uniform(szl, szh)

def __str__(self):
    return "(job %d, size %f)" % (self.ID, self.size)

def toXML(self):
    return ( "<attribute category='P'>\n"
            + "<name>ID</name>\n"
            + "<type>Integer</type>\n"
            + "<value>%s</value>" % self.ID
            + "</attribute>\n"
            + "<attribute category='P'>\n"
            + "<name>size</name>\n"
            + "<type>Float</type>\n"
            + "<value>%s</value>" % self.size
            + "</attribute>\n"
            )

```

The Generator State

```

class GeneratorState:
    """ A class to hold the state of a Generator Atomic DEVS.

```

```

    Not really required, but allows us to define
    the __str__ method for pretty printing.
    """

```

```

def __init__(self, first):
    # keep track whether this is the first
    # time an internal transition is invoked
    self.first = first

```

```

def __str__(self):
    return "\n\
        first      = %s" % self.first

def toXML(self):
    return ( "<attribute category='P'>\n"
            + "<name>first</name>\n"
            + "<type>bool</type>\n"
            + "<value>" + str(self.first) + "</value>\n"
            + "</attribute>\n"
            )

```

The Processor State

```

IDLE   = 0
BUSY   = 1
DISCARD = 2

```

```

class ProcessorState:

```

```

    """ A class to hold the state of a Processor Atomic DEVS.

```

```

    Not really required, but allows us to define
    the __str__ method for pretty printing.

```

```

    """

```

```

def __init__(self, queue=[],
              processorStatus=IDLE,
              currentJob=None,
              timeSpent=0):
    self.queue = queue
    self.processorStatus = processorStatus
    self.currentJob = currentJob # None if the processor is idle
    self.timeSpent = timeSpent
    # timeSpent is the time spent so far on the current process. This is
    # necessary as the DEVS elapsed time is reset after an external event
    # has occurred (i.e., when a new job is received from the generator) !

    # To be able to calculate average processing time performance
    # metric, need to add extra state variables.

```

```

self.currentTime = 0      # as seen by this Processor
self.processingStartTime = 0 # of current Job
self.processingEndTime = 0  # of current Job
self.totalProcessingTime = 0 # of all Jobs, by this Processor
self.numberOfJobsProcessed = 0 # by this Processor

def __str__(self):
    str_repr = "\n"
    if len(self.queue) == 0:
        str_repr += "    queue    = []\n"
    else:
        str_repr += "    queue    = [\n"
        for i in range(len(self.queue)-1):
            str_repr += "                %s,\n" % self.queue[i]
            str_repr += "                %s\n" % self.queue[-1]
        str_repr += "                ]\n"
    str_repr += "    processorStatus = %s\n" \
        % ("IDLE", "BUSY", "DISCARD")[self.processorStatus]
    str_repr += "    currentJob    = %s\n" % self.currentJob
    str_repr += "    timeSpent     = %s\n" % self.timeSpent
    str_repr += "\n"

    # Note: we don't print the extra state variables used
    # to calculate performance measures (average processing time
    # in this case).

    return str_repr

def toXML(self):
    stateInfo = ""
    stateInfo += ( "<attribute category='P'\>\n"
        + "<name>processorStatus</name>\n"
        + "<type>String</type>\n"
        + "<value>%s</value>\n" % ("IDLE", "BUSY", "DISCARD")[self.processorStatus]
        + "</attribute>\n"
        )
    stateInfo += ( "<attribute category='P'\>\n"
        + "<name>queueLength</name>\n"
        + "<type>Integer</type>\n"
        + "<value>%s</value>\n" % (len(self.queue)%10)
    )

```

```

        + "</attribute>\n"
    )

    if(self.currentJob!=None):
        stateInfo += ( "<attribute category=\"C\">\n"
            + "<name>currentJob</name>\n"
            + "<type>Job</type>\n"
            + "<value>%s</value>\n" % self.currentJob.toXML()
            + "</attribute>\n"
        )
    else:
        stateInfo += ( "<attribute category=\"C\">\n"
            + "<name>currentJob</name>\n"
            + "<type>Job</type>\n"
            + "<value>None</value>\n"
            + "</attribute>\n"
        )

    return stateInfo

```

One thing needs to be mentioned is that we these two functions, but it does not mean that we need you write these two functions to transform all the state information to textual or XML representation. You can only transform part of the state that you are interested into text or XML.

Simulation Level Support

The requirement for the simulation level support is very simple. You just need to turn the trace generation switch on when you run the simulator.

```

sim.simulate(termination_condition=terminate_whenEndTimeReached,
             verbose=False, trace=True)

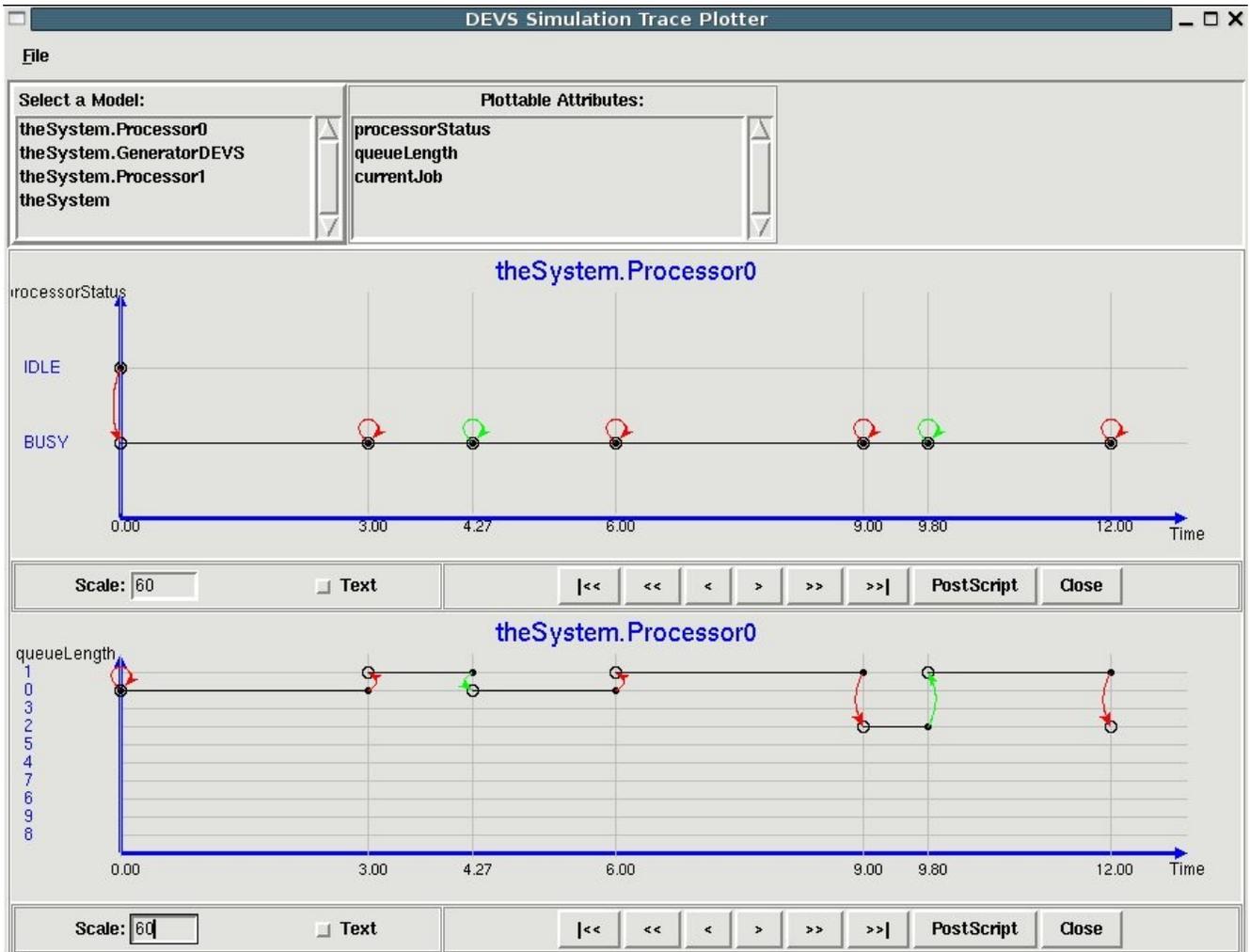
```

You set the 'trace' parameter's value to 'True', the simulator will generate a XML trace file called 'devstrace.xml'.

Run the Trace Plotter

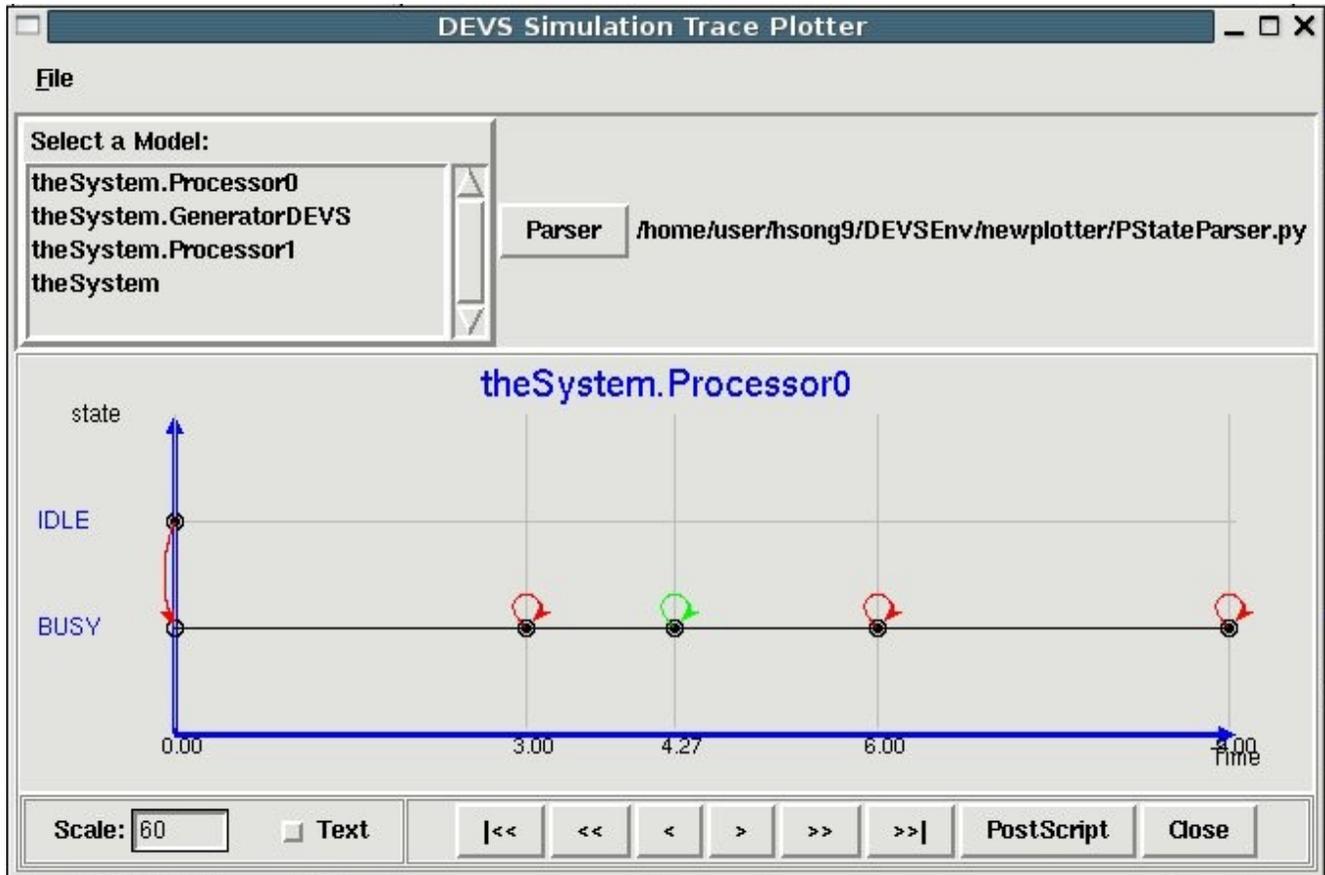
The plotter

1. Type 'python plotter.py'
2. Open the devstrace.xml file in the plotter
3. Select a model from models list and an attribute from the state attributes list
4. Play on the graphical screen



The cplotter

1. Type 'python cplotter.py'
2. Open the devstrace.xml file in the plotter
3. Select a model from models list and locate you customized state parser
4. Play on the graphical screen



An Example of The State Parser

The below is the PstateParser.py used in the figure above.

```
from xml.dom.minidom import *

class PStateParser(object):
    def getString(self, xmlState):
        for node in xmlState.childNodes:
            if node.nodeType == Node.CDATA_SECTION_NODE:
                return node.nodeValue #node.childNodes[0].data
        return "No state string"

    def getSeqState(self, xmlState):
        attributes = xmlState.getElementsByTagName("attribute")
        for attribute in attributes:
            attrName = attribute.getElementsByTagName("name")[0]
            txtName=attrName.childNodes[0].data
```

```
#print txtName, "--", self.attrName
if(txtName=="processorStatus"):
    attrValue = attribute.getElementsByTagName("value")[0]
    txtValue=attrValue.childNodes[0].data
    return txtValue
return "Error"
```