Data Generator ~ the basics

What is Data Generator

Data generator is a library that helps in generating patterns based on a given model.

Those patterns can be used:

- In developing load tests
- In developing functional tests
- In developing UI automation tests, where the generated data can represent a user scenario
- To aid in development in situations where no data exists

Data Generator can be used to distribute the generation problem over the Hadoop system as we will see in the upcoming slides.

Data Generator ~ the basics

Data Generator is a Java library

To make best use of Data Generator, expect to write code that calls the Data Generator core library at minimum.

The core library provides tools that allow the user to:

- Load an parse a model
- Split the model into n smaller problems
- Execute each smaller problem in a separate thread
- Send the generated patterns to user defined class called ***a writer***.

Data Generator ~ the basics

How the Data Generator model looks like

Data Generator uses SCXML an XML based language that can represent complex state machines called **S**tate **C**hart e**X**tensible **M**arkup **L**anguage.

The model represents the data as states, which can set output variables to certain values. Transitions between the states can optionally contain conditions using which the user can control the data values.

Data Generator ~ the basics

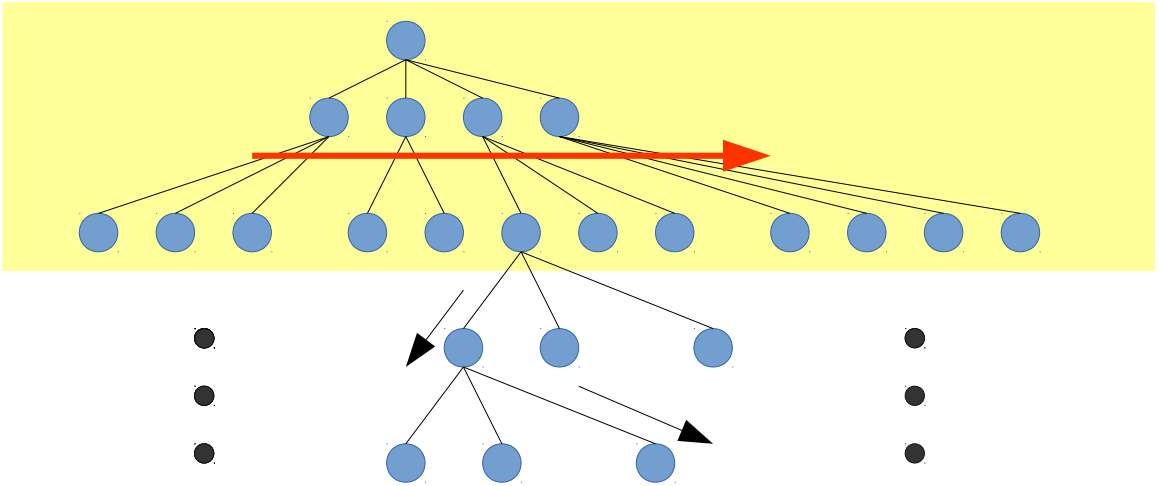
A sample SCXML model

```
<scxml xmlns="http://www.w3.org/2005/07/scxml"
       xmlns:cs="http://commons.apache.org/scxml"
       version="1.0"
       initial="start">

    <state id="start">
        <transition event="SETV1" target="SETV1"/>
    </state>

    <state id="SETV1">
        <onentry>
            <assign name="var_out_V1_1" expr="set:{A1,B1,C1}"/>
            <assign name="var_out_V1_2" expr="set:{A2,B2,C2}"/>
            <assign name="var_out_V1_3" expr="77"/>
        </onentry>
        <transition event="SETV2" target="SETV2"/>
    </state>

    <state id="SETV2">
        <onentry>
            <assign name="var_out_V2" expr="set:{1,2,3}"/>
            <assign name="var_out_V3" expr="#{customplaceholder}"/>
        </onentry>
        <transition event="end" target="end"/>
    </state>

    <state id="end">
        <!-- We're done -->
    </state>
</scxml>
```

```
SCXML Header




The start state


Create a state called SETV1
  Once the execution enters this state
    Set variables var_out_V1_1 & V1_2 to one value of the given

    V1_3 is always 77


  Unconditionally, transition to SETV2


Create a state called SETV2
  Once the execution enters this state
    Set variable var_out_V2 to one value of the given values
    Set var_out_V3 to a template, to be replaced in the java code


  Unconditionally, transition to END

Create a state called END


SCXML Closed tag
```

Data Generator ~ the basics

Engines: engines know about your model. For SCXML, we need to use an engine that knows SCXML. Engines will load and parse the model, convert it into a graph, walk the top of the graph using Depth First Search, and convert the result into a list of Frontiers that know its format – in our example an SCXMLFrontier.



Frontier: The frontiers gets serialized using a Gapper class into strings. Gappers need to know the format. In our example an SCXMLGapper is used.

Data Generator ~ the basics

Distributors: Send the String serialized Frontiers to their compute location.

Threads if you're executing locally (DefaultDistributor).

If you're using Hadoop, then they're sent to a Map job (HDFSDistributor in MR example).

Frontiers again: Once the Frontiers are reconstructed from Strings, they are asked to continue searching the graph using Depth First Search.

Once a frontier completes one round of Depth First Search, it places its findings on a Queue. The Distributor reads the Queue using a different thread and asks a list of provided transformers to operate on every item.

Transformers: Transformers convert custom place holders that you write in your model with their values. For example a value like #AccountNumber can be formatted using a transformer into 101X400.

Writers: Multiple writers provided by the user format the result and write them into multiple output files.

Data Generator ~ the basics

```
//will default to samplemachine, but you could specify a different file if you
choose to
InputStream is = CmdLine.class.getResourceAsStream("/" + (args.length == 0 ?
"samplemachine" : args[0]) + ".xml");

engine.setModelByInputFileStream(is);

// Usually, this should be more than the number of threads you intend to run
engine.setBootstrapMin(1);

//Prepare the consumer with the proper writer and transformer
DataConsumer consumer = new DataConsumer();
consumer.addDataTransformer(new SampleMachineTransformer());
consumer.addDataWriter(new DefaultWriter(System.out,
        new String[]{"var_out_V1_1", "var_out_V1_2", "var_out_V1_3",
"var_out_V2", "var_out_V3"}));

//Prepare the distributor
DefaultDistributor defaultDistributor = new DefaultDistributor();
defaultDistributor.setThreadCount(1);
defaultDistributor.setDataConsumer(consumer);
Logger.getLogger("org.apache").setLevel(Level.WARN);

engine.process(defaultDistributor);
```