

UC Berkeley computer science

Subject: XML front end for ArchSim
Date: September 2005
Authors: Igor Benko
Number: UCIB# 2005-ib03

References:

SML# 2005-0103: ArchSim structure, Igor Benko, February 2005
SML# 2005-0254: GASP Model in ArchSim, Igor Benko, March 2005

MOTIVATION

This memo describes the first version of XML front-end for ArchSim simulator. ArchSim is an event-based simulator that we intend to use as one of our core tools for exploring Fleet architectures. In our initial testing of ArchSim, we assembled simulation models programmatically. That is, we would write a program that assembled components into the structure we wanted to simulate. Such an approach is very flexible, but is also slow and error-prone. From the outset, our intention was to assemble simulation models via a graphical user interface. Specifically, our target was integration with the VLSI design tool Electric and, more precisely, its schematic editor. The first version of this integration is now operational: We can use Electric to draw schematics representing simulation models, export the models into ArchSim and run simulations. To close the loop, Electric can read the results of simulations and provide us with a graphical view and playback of the simulation.

Electric now has the capability to export an XML file from a schematic. ArchSim can read the XML file, build the corresponding simulation model, and run desired simulations. Electric can read simulation journal and play it back, showing activity of connections on both the schematic and in a wave-front form.

Below we detail the XML front-end to ArchSim, describe what functionality we put in place to provide this front-end, and describe the process of converting an XML file to an internal ArchSim model.

XML INPUT

XML input to ArchSim consists of several files. The most notable is the file that defines a model by listing the components and their connections. Below is the list of components and connections for a two-ship Fleet.

This document is a product of a collaboration between Sun Microsystems and the University of California at Berkeley. The ideas contained herein are freely available for any academic purpose.

```

<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE model SYSTEM "ArchSimModel.dtd">

<model name= "Two-Ship-Fleet">

<component name= "InstrGen" type= "InstructionGenerator2" />
<component name="InstrHorn" type= "InstructionHorn2" />
<component name= "SourceFunnel" type= "SourceFunnel2"/>
<component name= "DestHorn" type= "DestinationHorn2"/>
<component name= "Ship0" type= "TestShip" />
<component name= "Ship1" type= "TestShip" />

<connection name= "InstructionTrunk">
    <from component="InstrGen" terminal="Out" />
    <to component="InstrHorn" terminal="In" />
</connection>

<connection name="InstructionToken0" >
    <from component="InstrHorn" terminal="Out0" />
    <to component="SourceFunnel" terminal="In0" />
</connection>

<connection name="InstructionToken1" >
    <from component="InstrHorn" terminal="Out1" />
    <to component="SourceFunnel" terminal="In1" />
</connection>

<connection name="SrcData0" >
    <from component="Ship0" terminal="Out" />
    <to component="SourceFunnel" terminal="SrcIn0" />
</connection>

<connection name="SrcData1" >
    <from component="Ship1" terminal="Out" />
    <to component="SourceFunnel" terminal="SrcIn1" />
</connection>

<connection name="DestData0" >
    <from component="DestHorn" terminal="Out0" />
    <to component="Ship0" terminal="In" />
</connection>

<connection name="DestData1" >
    <from component="DestHorn" terminal="Out1" />

```

```

    <to component="Ship1" terminal="In" />
</connection>

<connection name="Trunk" >
    <from component="SourceFunnel" terminal="Out" />
    <to component="DestHorn" terminal="In" />
</connection>

</model>

```

The root element in the XML structure above is `model` and it contains two elements – `component` and `connection`. Each component element specifies one instance of a component in an ArchSim model. For each instance of a component, a `component` element provides two attributes, the component `name` and component `type`. Components are distinguished by names, thus each component name must be unique. The component type corresponds to a class that can be loaded by Java VM when executing ArchSim. Additionally, the class corresponding to the component type must be a subclass of `Component`, which we will discuss below, where we also provide more detail on how we establish the connection between the type name and the actual class.

The second element in the XML file, `connection`, specifies a connection between components. ArchSim translates each connection into two states. One state has attached data path, and the other state corresponds to a state-wire only. The arrangement is similar to the two states that connect subsequent actions in a FIFO – one state carries the data, so when that state is active, the connection between two actions in the FIFO is “full”. When the other state becomes active, this can be seen as an acknowledgment that the data has been copied and that new data can be deposited in the connection. That is, when the second state is active, a subsequent datum can be “deposited” in the connection.

Each `connection` has two kinds of sub-elements – `from` and `to`. We see these two elements as specifications of connection points. Each connection point is attached to a terminal at a component. Attribute `component` provides the name of the component, and attribute `terminal` provides the name of the terminal.

The distinction between `from` and `to` connection points is only important when we consider data path. That is, when data is moved to a connection, the data can only originate from connection points labeled as `from`, and the data will only be passed on through connection points labeled with `to`. At this time, we decided against having connection points that can be both sources and destinations for data. This decision was made for the sake of simplicity and may be revisited when the need arises. The distinction is important for the simulator – when firing of an action is modeled, the

simulator must know what terminals will serve as origins of data and what terminals will serve as destinations for data.

Recall that ArchSim translates each of the connections into a pair of states. Also recall that a state can be connected to an arbitrary number of actions. The same holds for connections described in the XML file. For each connection we can have multiple “from” end points and multiple “to” end points as illustrated in the following example:

```
<connection name="TWO-BY_TWO" >
  <from component="Src0" terminal="DATA" />
  <from component="Src1" terminal="DATA" />
  <to component="Dest0" terminal="DATA" />
  <to component="Dest1" terminal="DATA" />
</connection>
```

XML DTD

To simplify error handling, we require a DTD (document type definition) file to be provided with the XML file. A DTD file contains a “grammar”, which defines XML elements that can be used in the XML file. The DTD file that defines the format of files containing ArchSim models, contains the following:

```
<?xml version='1.0' encoding='us-ascii'?>
<!-- DTD for an archsim model -->

<!ELEMENT model ((component | connection)+)>
<!ATTLIST model
  name CDATA #REQUIRED
>

<!ELEMENT component ( comment? ) >
<!ATTLIST component
  name CDATA #REQUIRED
  type CDATA #REQUIRED
>

<!ELEMENT connection (( from | to )+ )>
<!ATTLIST connection
  name CDATA #REQUIRED
>

<!ELEMENT from (comment?) >
<!ATTLIST from
```

```

    component CDATA #REQUIRED
    terminal CDATA #REQUIRED
>

<!ELEMENT to ( comment? ) >
<!ATTLIST to
    component CDATA #REQUIRED
    terminal CDATA #REQUIRED
>

<!ELEMENT comment (#PCDATA) >

```

That is, the root element `model` contains one or more of the elements `<component>` or `<connection>`. A component element can contain an optional `<comment>` element, and must contain two attributes, a component `name` and a component `type`. We note that inclusion of the optional `<comment>` element allowed us to circumvent a bug in the XML parser that does not allow definitions of empty elements.

Each `<connection>` contains a non-empty sequence of `<from>` and `<to>` elements. Both, the `<from>` and the `<to>` element define an end point of a connection. Thus, both elements have two attributes, the `component` and the `terminal` name for the connection end point.

The XML tutorial at <http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/TOC.html> contains more information about XML and DTD files.

COMPONENTS

How does ArchSim recognize components that are listed in the “netlist” XML file produced by Electric? Above, we mention that the “type” attribute of a component elements corresponds to a class that can be loaded by a VM when executing ArchSim. The component “type” must, therefore, somehow be linked to the full class name. ArchSim establishes this mapping with another XML file that represents the library of components. For each component, the component library file provides its type and the full name of the corresponding class. Here is an example of a simple component library:

```

<?xml version='1.0' encoding='utf-8'?>
<!-- Test for ClassRegistryLoader -->
<!DOCTYPE component_list SYSTEM "ComponentList.dtd">

<component_list>

<component>
<type>InstructionGenerator</type>

```

```

<class> com.sunlabs.archsim.fleet.components.TestInstrGen
</class>

</component>
<component>
<type>InstrGen4</type>
<class> com.sunlabs.archsim.fleet.components.TestInstrGen4
</class>
</component>

<component>
<type>InstructionHorn4</type>
<class> com.sunlabs.archsim.fleet.components.InstructionHorn4
</class>
</component>

<component>
<type>SourceFunnel4</type>
<class> com.sunlabs.archsim.fleet.components.SourceFunnel4
</class>
</component>

<component>
<type>DestinationHorn4</type>
<class> com.sunlabs.archsim.fleet.components.DestinationHorn4
</class>
</component>

<component>
<type>TestShip</type>
<class> com.sunlabs.archsim.fleet.components.TestShip </class>
</component>

</component_list>

```

ArchSim reads the component library file and uses it to build a class registry. The component library file must satisfy two conditions: First, JVM must be able to load the class. That is, the corresponding class file must be in the JVM's classpath. And second, the class must be a subclass of class `Component`. Failure to meet any of these two conditions causes an exception that will currently cause ArchSim to terminate.

Let us remark that one can load multiple class library files. Components from each subsequent file are added to the class registry. If a type repeats, the class loader throws an exception that causes ArchSim to terminate.

Note that the component library file also relies on a DTD for ease of parsing. The DTD specifies that each component element must have two sub elements, one to specify the type followed by one for the class:

```
<?xml version='1.0' encoding='us-ascii'?>

<!--
  DTD for a list of components used in archsim models
-->

<!ELEMENT component_list (component*)>
<!ELEMENT component (type, class)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT class (#PCDATA)>
```

SETTINGS

ArchSim has configurable settings such as delays, the log (journal) level and so on. To establish values of these settings, ArchSim can read an XML file from which it builds an internal `Globals` class, which, in turn, component can use to obtain these values. Here is an example of a file with settings:

```
<?xml version='1.0' encoding='us-ascii'?>

<settings>

  <log_level value= "1" />
  <journal_state value="false" />
  <journal_action value="false" />
  <journal_terminal value="false" />

  <state_to_action_delay value="10" />
  <action_to_state_delay value="10" />
  <action_fired_delay value="0" />

  <arbiter_delay value="0" />
  <merge_arbiter_delay value="0" />
  <merge_pass_delay value="0" />
  <source_sync_pass_data_dela value= "0" />
  <source_sync_ack_delay value= "0" />

  <test_ship_source_delay value= "10" />
  <test_ship_sink_delay value= "10" />
```

```
</settings>
```

Note that each XML element in the file, apart from the root element `settings` is an empty element. The name of the element is recorded in the `Globals` class (a singleton) as the name of the parameter, and the `value` attribute is recorded as the value of the parameter.

The `Globals` class provides an interface for obtaining values of parameters on the basis of the parameter name. The value can be returned as an `Object`, in which case the calling object must take care of making sure that the value is of the correct data type. The `Globals` object returns the value `null` for unknown parameter names. In such a case it is the responsibility of the caller to react appropriately, for example by loading a default value.

Alternatively, the `Globals` object can provide a parameter of one of the following types: `Integer`, `Boolean`, `Double` or `String`. If the parameter name is unknown, or if the value provided could not be transformed to the desired type, a `null` object is returned. Again, it is the caller's responsibility to deal with the null object appropriately. A `Globals` object also provides method `hasParam`, which allows the caller to determine whether a certain parameter has been loaded into `Globals`.

Note that no DTD is used for settings. One can freely add parameters without making any coding changes. When the parameters are loaded into the `Globals` object, no name or data type checking is performed. Such an approach simplifies syntax and helps with maximizing flexibility. Component that require values of these parameters would then use a `Globals` object to access required values, or would use default values when the parameter is not present in `Globals`.

Let us also remark that parameters can be specified in multiple files that can be loaded in succession. When a parameter is specified more than once, `Globals` class stores the final (last) value loaded. We decided to allow this overwriting to enable modification of the model "on the fly".

OPERATION "SCRIPT"

The simulator is driven through a list of operations listed in a file in an XML format. The file name is the only command-line argument the simulator accepts. If the file name is not provided in the command line, the simulator uses "script.xml". Here is a simple example of such a file:

```
<?xml version='1.0' encoding='utf-8'?>
```

```

<!-- test for ModelLoader -->

<!DOCTYPE operation_list SYSTEM "OperationList.dtd">

<operation_list>
<output file="test.asj" />
<load_parameters file="globals.xml" />
<load_components file="FleetComponents.xml" />
<print_components/>
<load_model file="Four-Ship-Fleet.xml" />
<build_GASP_model/>
<self_check/>
<print_model_description />
<print_model />
<run time="100"/>
</operation_list>

```

Note that the root element **operation_list** contains a list of operations. Each operation is specified with an empty element, where element attributes provide parameters for the operations. The “script” file above specifies the following operations:

1. Output of the simulator is directed to file test.asj. The output consists of error messages, information messages, and simulation log. These three kinds of output can be directed to separate files. Default for the output is the standard output (terminal) for the simulation log and information messages, and standard error (terminal) for error messages.
2. Global parameters are loaded from the file named “globals.xml”.
3. A component library is loaded from the file named “FleetComponents.xml”.
4. The contents of the library is printed as the informational output.
5. A model for the simulation is loaded from the file named “Four-Ship-Fleet.xml”.
6. The simulator constructs an internal GASP model. We decided to make this step separate from loading the model, so that we can have an option to build different internal simulation models from the same “netlist”.
7. A self check is ran on the simulation model.
8. The model description is produced as an informational output. The model description contains the list of components and their connections. One can use this output to verify whether the simulation model indeed corresponds to the “netlist”.
9. Internal representation of the model in terms of States, Actions, and Terminals is printed.
10. A simulation runs for 100 time units.

Also note that the format of the “script” file is governed by a DTD file, which reads as follows:

```

<?xml version='1.0' encoding='us-ascii'?>

<!--
  DTD for operations list
-->

<!ELEMENT operation_list (
  ( load_parameters
  | load_components
  | load_model
  | build_GASP_model
  | output
  | journal
  | error
  | info
  | run
  | step
  | reset
  | self_check
  | print_components
  | print_model
  | print_model_description )* ) >

<!ELEMENT load_parameters (comment* ) >
<!ATTLIST load_parameters
  file CDATA #REQUIRED
>

<!ELEMENT load_components (comment* ) >
<!ATTLIST load_components
  file CDATA #REQUIRED
>

<!ELEMENT load_model ( comment* ) >
<!ATTLIST load_model
  file CDATA #REQUIRED
>

<!ELEMENT build_GASP_model (comment*) >

<!ELEMENT output ( comment* ) >
<!ATTLIST output
  file CDATA #REQUIRED
>

```

```

<!ELEMENT journal ( comment* ) >
<!ATTLIST journal
    file CDATA #REQUIRED
>

<!ELEMENT error ( comment* ) >
<!ATTLIST error
    file CDATA #REQUIRED
>

<!ELEMENT info ( comment* ) >
<!ATTLIST info
    file CDATA #REQUIRED
>

<!ELEMENT run ( comment* ) >
<!ATTLIST run
    time CDATA #REQUIRED
>

<!ELEMENT step ( comment* ) >
<!ATTLIST step
    number CDATA #IMPLIED
>

<!ELEMENT self_check ( comment* ) >

<!ELEMENT reset ( comment* ) >

<!ELEMENT print_components ( comment* ) >

<!ELEMENT print_model ( comment* ) >

<!ELEMENT print_model_description ( comment* ) >

<!ELEMENT comment (#PCDATA) >

```

More precisely, the following operations are supported:

- **load_parameters**: Loads global parameters, the attribute “file” provides the name of the file to load. Note that this operation can be repeated if we are to instruct the simulator to read parameters from multiple files.
- **load_components**: Loads a component library, the attribute “file” provides the name of the file to load. Note that this operation can be repeated if we are to instruct the simulator to load component libraries from multiple files.

- **load_model**: Loads a “netlist”, the attribute “file” provides the name of the file to load. This operation can, too, be repeated and multiple “netlists” can be loaded in an additive manner. The attribute “file” provides the name of the file to load.
- **build_GASP_model**: Builds an internal representation of loaded “netlists”. Refer to SML 2005-0254 for more details on the GASP model.
- **output**: Set the output file for the simulation journal, informational messages, and error messages. The file attribute provides the name for the target file. If the target file already exists, it is overwritten.
- **journal**: Set the output file for the simulation journal. The file attribute provides the name for the target file. If the target file already exists, it is overwritten.
- **error**: Set the output file for the error messages. The file attribute provides the name for the target file. If the target file already exists, it is overwritten.
- **info**: Set the output file for informational output of the simulator. The file attribute provides the name for the target file. If the target file already exists, it is overwritten.
- **run**: Run a simulation for the specified number of time units. The attribute “time” specifies the number of time units to run.
- **step**: Step the simulation for the specified number of events. The attribute “number” specifies the number of events to simulate.
- **reset**: Reset the simulation: clear the simulation queue, reset the simulation time to zero.
- **clear_all**: This operation is equivalent to restarting ArchSim. The model, component registry, global settings, and the queue are deleted, and the simulation time is reset to zero.
- **self_check**: Run the self-check on the internal model. The self-check verifies that proper connections are established and, for the GASP model, that each Action has a function attached to each output terminal. See SML 2005-0254 for more details.
- **print_components**: Prints the component library as informational output.
- **print_model**: Prints the “netlist” as the informational output.
- **print_model_description**: Prints the description of the internal model. For the GASP model, this means printing the details for each Action, Terminal, and State.

