Triana Conditional Looping Tutorial

Ian Wang

2nd February 2003

1 Introduction

In this tutorial we describe the mechanisms in Triana for conditionally looping over subsections of a workflow. The default method is to use the Loop tool, a standard Triana tool located in the Common.Logic toolbox. The Loop tool enables `for`/`while` loops to be created with exit conditions based on variables including the number of iterations executed and the parameters of other tasks. In addition, in this tutorial we discuss both extending the Loop tool to apply custom exit conditions and the creation of completely new control tools. This tutorial assumes a full installation of Triana, which is available at `www.trianacode.org`.

2 Getting Started

Once a workflow has been constructed there are two methods for creating a loop over tasks within that workflow using the Loop tool. The more illustrative but less elegant method is to insert the Loop tool directly into the workflow, while the preferred method is to create a group of the tasks and to use the Loop tool as a hidden control task for the group. We look at these looping methods in Sections 2.1 and 2.2 respectively, and discuss the exit conditions for both methods in Section 3.

2.1 Looping using the Loop Tool Directly

In Figure 1 we show the Loop tool inserted directly into a simple workflow for generating, incrementing and displaying a constant. As mentioned before, the Loop tool is located in the Common.Logic toolbox.

The Loop tool is initialized in a *pre-loop* state. In this state it waits for data on its first input node (from ConstGen in Figure 1). When data is received on this node the user defined exit condition for the loop is evaluated (see Section 3). If the exit condition is met then the data is output by the Loop tool on its first output node (to ConstView), and the Loop tool remains in a *pre-loop* state. If the exit condition is not met then the data is output

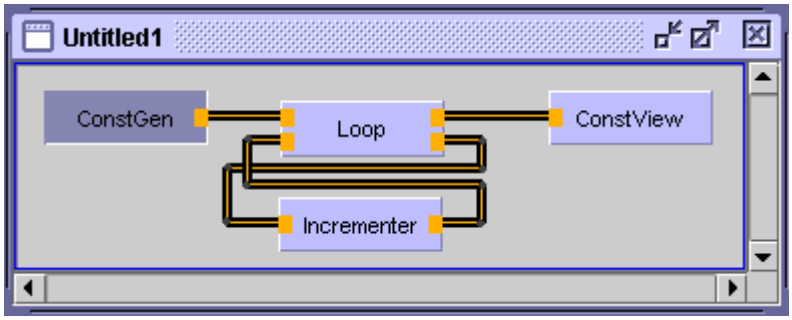

Figure 1: The Loop tool inserted directly into a simple workflow.

by the Loop tool on its second output node (to Incrementor) and the Loop tool enters an *in-loop* state.

An *in-loop* state is the same as the *pre-loop* state except that the Loop tool waits for data on its second input node (from Incrementor in Figure 1). As with the *pre-loop* state, in an *in-loop* state the exit condition for the loop is evaluated when the data is received, and if the condition is met then the data is output by the Loop tool on its first node (returning the Loop tool to a *pre-loop* state). If the condition is not met then the data is output by the Loop tool on its second node (remaining the Loop tool in an *in-loop* state). We illustrate the *pre-loop* and *in-loop* states in Figure 2.

2.2 Looping using the Hidden Control Task

A more elegant solution than inserting a loop directly into the workflow is to use a hidden control task to loop over a group of tasks. When a group is created from one or more tasks (using *Edit→Group*) a hidden control task is automatically attached to handle the input/output from the group; by default this task is an instance of Loop tool. As shown in Figure 3, the topology of the cables attached to this task is identical to that when a loop

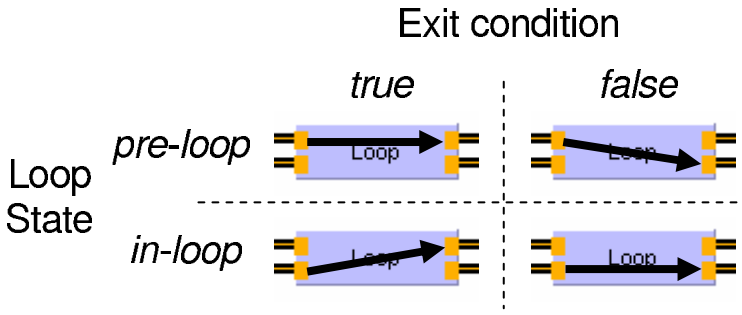

Figure 2: The data flow through the Loop tool based on the loop state and exit condition.

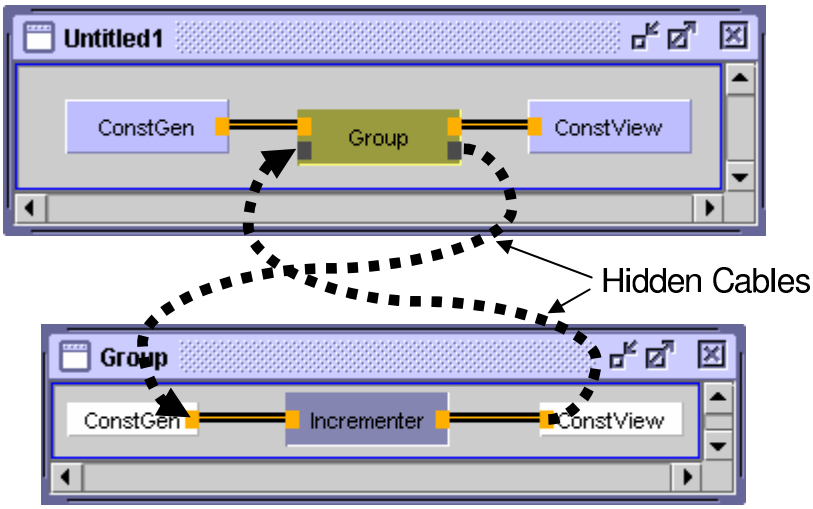

Figure 3: Looping over a group of tasks using the Loop tool as a hidden control task.

is inserted directly into the workflow (Figure 1) except that the cables and control task are concealed from the user.

The obvious benefit of looping using a control task is that concealing the cables makes the workflow cleaner and less confusing. Using a group also ensures that the inpoints/outpoints to the loop are well defined, but conversely this restricts more complex interactions with tasks within the loop. Another disadvantage from looping using a control task is that parameter nodes cannot be attached to the control task, so it is not possible to pipe out the iteration count for example. If parameter input/output from the control task is required then this can be done by inserting the task directly into the workflow.

Although an instance of Loop tool is used as the control task for a group by default, it is possible to change this to an instance of a different Triana tool. This is done using the *Control* panel in the *Group Editor* (right click on the background of the open group). We discuss creating custom instances of Loop tool in Section 3.5 and creating new control tools in Section 4.

2.3 Multiple Loop Inputs/Outputs

In the previous section we only considered using a hidden control task to loop over groups of tasks that have a single data input/output, however it is acceptable to loop over groups with multiple data inputs/outputs. In a situation where the number of data inputs to the group is equal to the number of outputs there are no difficulties as the data output from one iteration exactly provides the input to the next iteration, as shown in Figure 4.

When the number of outputs from a group is greater than the number

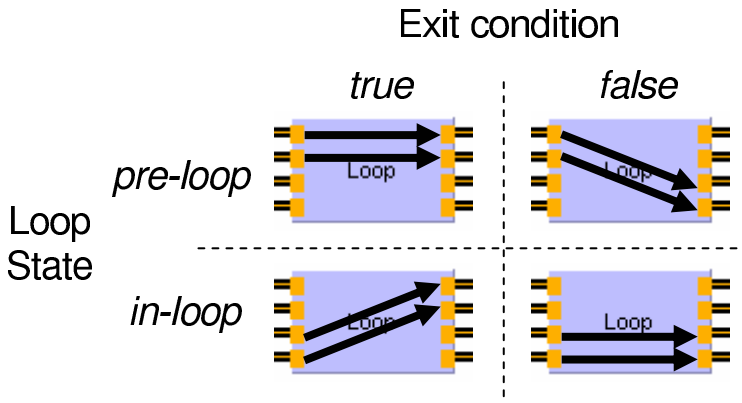

Figure 4: Loop tool data flow when the number of data inputs equals the number of data outputs (2 inputs, 2 outputs).

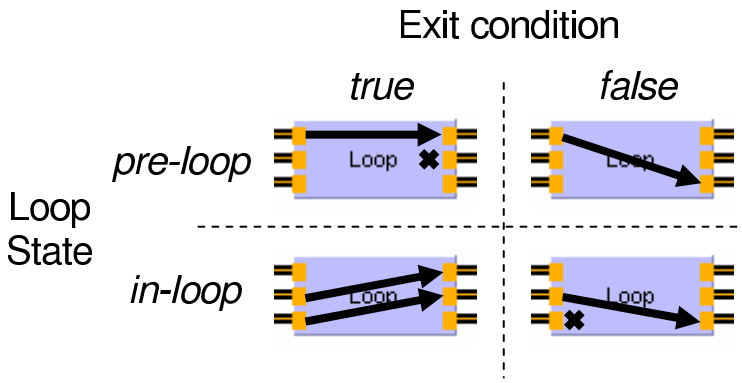

Figure 5: Loop tool data flow when there is a mismatch between the number of data inputs and the number of data outputs (1 input, 2 outputs).

of inputs then the data output from one iteration of the group exceeds that required by the next. In this situation the extra data items output from one iteration are simply not passed through to the next, as illustrated in Figure 5. Although the extra data items are not passed through to further iterations they are still useful in that they are output when the loop finishes (shown in Figures 6) and can be used in the exit conditions for the loop. For example, in Figure 6 the output from the additional Random task could be used in the exit condition to stop the loop when a random number over 0.9 is generated (`$data1 > 0.9`). We discuss exit conditions further in the Section 3.

When the number of data inputs to a group is greater than the number of data outputs then there is insufficient data output from one iteration to provide the input to the next. In such a situation the workflow will hang as tasks wait for data that does not exist, and it is therefore not recommended

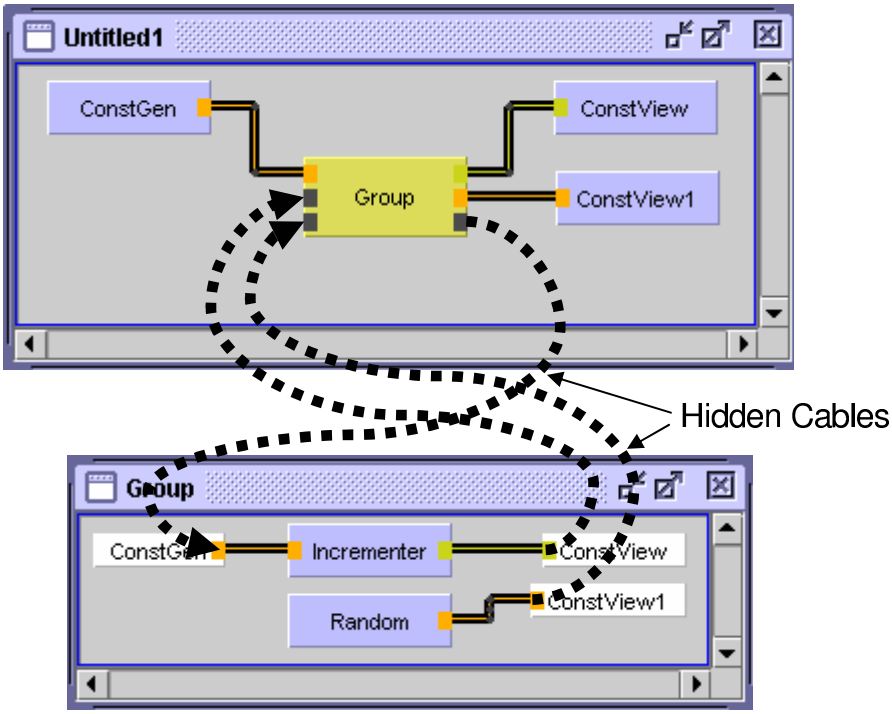

Figure 6: Looping over a group of tasks with mismatched data inputs/outputs (1 input, 2 outputs).

to employ this setup.

As with looping using a hidden control task, when using the Loop tool directly in a workflow it is acceptable to have multiple inputs to/outputs from the loop. Increasing the number of inputs/outputs to a Loop tool can either be done using the *Node Editor* (right click on the tool) or using the little *plus* icons on the tool. When the number of nodes is changed it is important that the number of inputs nodes and output nodes remain equal, with the first half of each assumed to be input to/output from the loop and the second half of each handling the output from/input to each iteration. If there is an odd number of input/output nodes then it is assumed that the extra node is an additional output from the loop; for example, the behaviour assumed for a Loop tool with 3 input/output nodes is the same as that shown in Figure 5.

As with multiple input/output nodes, looping over a group of tasks with zero input/output nodes is acceptable. In fact, looping with zero input nodes is very useful as it allows selected inputs to a workflow to be executed multiple times. For example, if a workflow requires a sequence of 100 random numbers to be generated then this can be achieved by making a group from the Random task and then looping over that group for 100 iterations.

Looping over a group with zero inputs and zero outputs is also useful for executing an entire workflow (encompassed in the group) multiple times.

3 Exit Conditions

Once inserted into a workflow (either directly or as a control task), the exit condition for the Loop tool must be set. This is done using the parameter panel for the Loop task. If the Loop tool is being used directly then this panel is displayed by right-clicking on the task and selecting *Properties*. If the Loop tool is being used as a hidden control task then this panel is displayed by right-clicking on the group and selecting *Contol Properites*.

By default the Loop tool is disabled, in which case a single iteration of the loop is executed when data is received (this is the expected behaviour when the user is unaware/ignoring the control task). To enable conditional looping simply check the *Enable Conditional Looping* box on the Loop panel. Once this is done the exit conditions for the loop can be set using either basic exit conditions or using advanced exit conditions. We discuss these two methods in Sections 3.1 and 3.2 respectively.

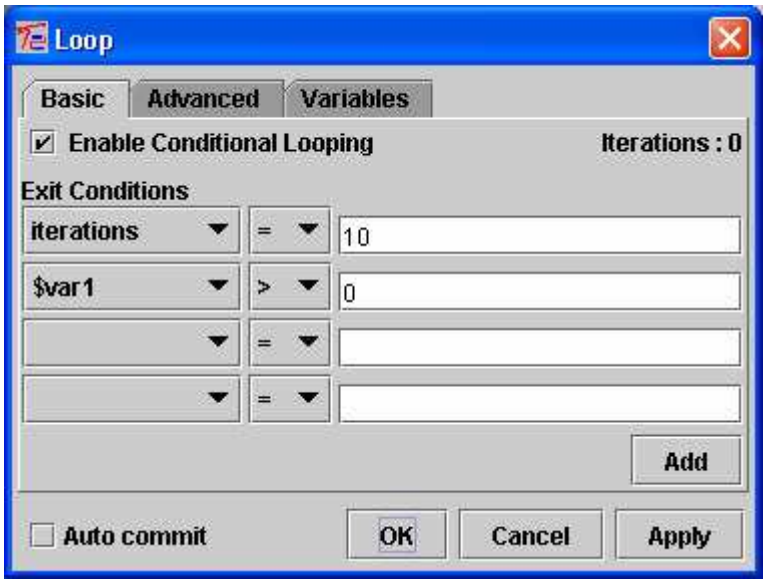

Figure 7: The Loop tool panel for inputting basic exit conditions.

3.1 Basic Exit Conditions

The basic exit condition panel (Figure 7) allows conditions to be expressed in the form <*variable*> <*comparison*> <*equation*>. For example, to exit the loop after 5 iterations:

```
iterations = 5
```

The variable `iterations` is a parameter of the Loop tool that counts the number of iterations of the loop that have executed. This count is reset every time the loop finishes (returns to a *pre-loop* state). In addition to `iterations`, there are a number of other variable types that can be used in exit conditions:

`total iterations` **-** The total iterations that the loop has executed. Unlike `iterations` this counter is not reset every time the loop exits, only when the workflow is reset by the user.

`$data0 $data1` **etc. -** Input data variables. We discuss input data variables in Section 3.3.

`$var0 $var1` **etc. -** User defined variables. We discuss user defined variables in Section 3.4.

`task.parameter` **-** Parameters of tasks within the loop. These take the form *TaskName.ParameterName*, for example ConstGen.constant or Wave.frequency.

When multiple exit conditions are specified in the basic exit conditions panel then all the conditions must be met in order for the loop to exit (`<condition1> && <condition2> && ...`). For example, the following only exits the loop when at least than 10 iterations have been executed and ConstGen.constant is less than 0.5:

```
iterations >= 0
ConstGen.constant < 0.5
```

In addition, the *equation* section of an exit conditions can reference other variables and include mathematical operators, for example:

```
iterations >= total iterations / 2
ConstGen1.constant < ConstGen2.constant + 2 / 3
$data0 = iterations % (10 + $var2)
$var1 = $data0 + $data1
```

Although basic exit conditions can be used in most situations, they do not allow for certain operations (such as `OR` and `NOT`). If extra flexibility is required then advanced exit conditions can be used; we outline advanced exit conditions in the next section.

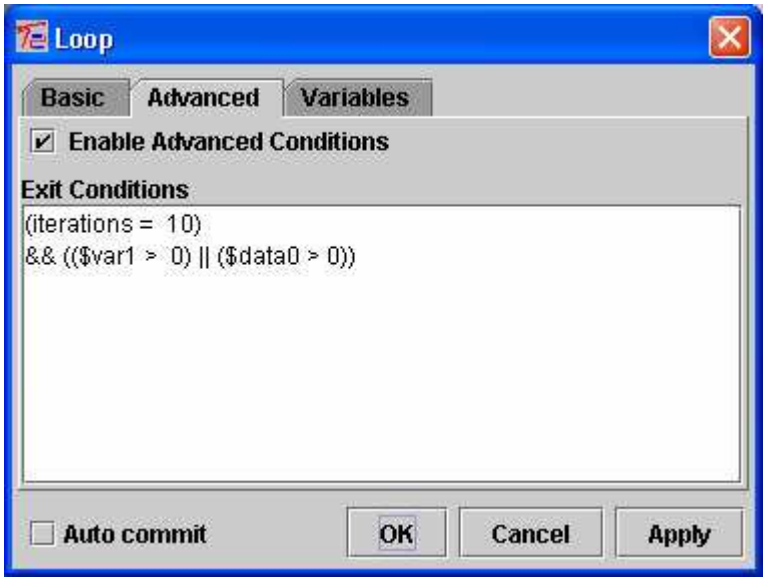

Figure 8: The Loop tool panel for inputting advanced exit conditions.

3.2 Advanced Exit Conditions

The advanced exit condition panel (Figure 8) allows exit conditions that it is not possible to specify using the basic exit condition panel to be entered, such as exit conditions that include `OR` and `NOT` operations. Exit conditions entered using the advanced exit condition panel can include all the variable types discussed in the previous section as well as `&&`, `||` and `!` operators (`AND`, `OR` and `NOT` respectively). For example, the condition to exit a loop after 10 iterations or when ConstGen.constant = 5 would be specified as:

```
(iterations = 10) || (ConstGen.constant = 5)
```

3.3 Input Data Variables

Using the Loop tool it is possible for exit conditions to be based on the value of the data being passed around the loop. This is done using the input data variables (`$data0`, `$data1` etc.). The index of the data variable refers to the index of the node it was received on in the context of the current loop state. For example, when looping over a group with a single data input/output then there is only a single input data variable (`$data0`). In a *pre-loop* state `$data0` represents the initial data item received on the first node of the Loop tool, while in an *in-loop* state it represents the data item returned from the loop and received on the second node of the Loop tool.

Using the example shown in Figure 6, in a *pre-loop* state `$data0` represents the data received from ConstGen, while in an *in-loop* state `$data0`

represents the data received from Incrementor. As in this example only a single data item is received in the *pre-loop* state `$data1` is set to `Double.NaN`[1] when the exit condition is first evaluated, while in an *in-loop* state `$data0` represents the data received from Random.

From the example shown in Figure 6, the exit condition to exit when a random number greater than 0.9 is generated would be:

```
$data1 > 0.9
```

Equality comparisons (`=`, `!=`) for input data variables are done using the `equals` method for the data's Java class. If the class implements the `Comparable` interface then ordering comparisons (`>`, `>=`, `<`, `<=`) can also be done.

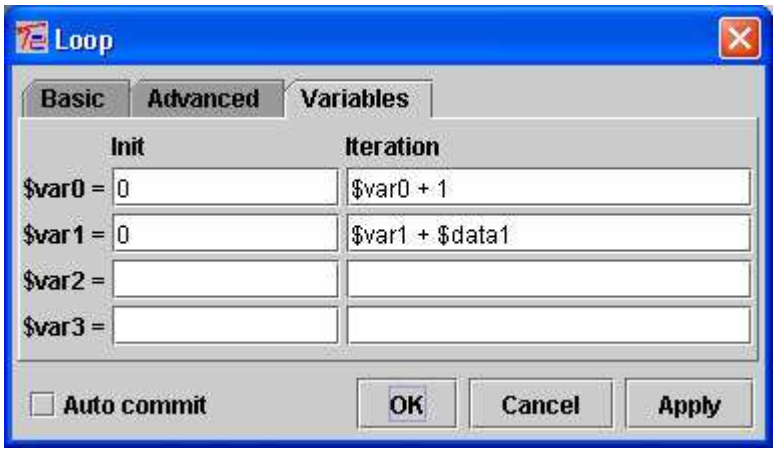

Figure 9: The Loop tool panel for defining user variables.

3.4 User Defined Variables

The user defined variables panel (Figure 9) allows users to specify variables (`$var1`, `$var2` etc.) that are initialized when the Loop receives its initial input data, and then updated every iteration. For example, a user variable that counts the number of iterations in same way as the `iterations` variable discussed in Section 3.1 can be specified as:

```
init:      $var1 = 0
iteration: $var1 = $var1 + 1
```

The init and iteration equations for user defined variables can include any of the variable types (iterations, task parameters etc.) described in Section 3.1.

[1] `Double.NaN` is a constant holding a double of value not-a-number. A data variable can be tested as being not-a-number using `$data0 = NaN` etc..

3.5 Custom Exit Conditions

Although the standard exit conditions discussed in the previous sections can handle most general looping situations, if a specific functionality is required then a custom exit condition can be attached to the Loop tool. A custom exit condition is simply a Java class that implements the Common.Logic.ExitCondition interface. The ExitCondition interface is contains three key methods:

`init()` - This method is called when the initial input data is received by the Loop tool (before isExitLoop is called), and should perform any initialization required by the exit condition.

`iteration()` - This method is called after each iteration of the loop (before isExitLoop is called for that iteration), and should perform any iteration based updates required by the exit condition.

`isExitLoop(Object[] data)` - This is the most important method. It is called when the Loop tool receives initial input data and after every iteration, and returns a boolean indicating whether the loop should exit. The data parameter is an array of the data input to the Loop tool for the current iteration (the equivalent of `$data0`, `$data1` etc. described in Section 3.3).

Once a custom exit condition is written and compiled (using *Tools→Compile Unit* for example) then it needs to be attached to a copy of the Loop tool. To make a copy of the Loop tool right-click on the tool in the tool tree, select *copy*, and then paste the copied tool either back into the Common.Logic toolbox under a different name or into a separate toolbox. To attach the custom exit condition right-click on the copied tool and select *Edit XML Definition*, and then in the XML editor change the value of the conditionUnit parameter to the location of the custom exit condition (the original value of the conditionUnit parameter is Common.Logic.DefaultExitCondition). As well as the exit condition, the parameter panel class for the custom loop can also be changed (by changing the value of the paramPanelClass parameter).

As mentioned in Section 2.2, it is possible to change the hidden control task for a group to a customized Loop tool. This is done using the *Control* panel in the *Group Editor* (right click on the background of the open group).

4 Custom Control Tools

In this tutorial we have talked about using the Loop tool as a hidden control task for looping over a group of tasks; however the Loop tool is just a standard Triana unit and it is possible write a completely new unit for use as a hidden control task. As a complete description of writing of new units

is beyond the scope of this tutorial we will just mention a few points which should be considered when writing a custom control unit:

- All custom control units are connected into the workflow in the same way as shown for the Loop tool in this tutorial (see Figures 3 and 6).
- To check whether the custom unit is being used as a control task use:

  `TaskGraphUtils.isControlTask(getTaskInterface()).`

- To retrieve the number of data inputs/outputs for the group that the control unit is acting for use:

  ```
  getTaskInterface().getParent().getGroupTask().getDataInputNodes()
  getTaskInterface().getParent().getGroupTask().getDataOutputNodes()
  ```

- It is probably best to set the default node requirement to optional by inserting `setDefaultNodeRequirement(OPTIONAL)` in the `init` method for the unit. This means that the `process` method for the unit is called when data is received on any node, and the unit can then use `isInputAtNode(int nodeindex)` to pool whether data has been received at all the nodes it is currently interested in.
- Each time an iteration of the group is executed a call to `getControlInterface().runGroup()` should be made to execute all those tasks in the group without data inputs. Alternatively, `getControlInterface().runTask(TaskInterface task)` can be used just to run specific tasks within the group.

For more hints the source code for the standard Loop tool is a good place to start. Once written and compiled a custom control tool can be attached to a group using the *Control* panel in the *Group Editor* for that group (right click on the background of the open group).