MarGotAspect - XML Configuration File User Manual

Arcari Leonardo, Galimberti Andrea

June 27, 2017

1 About

This document is intended to be used as a reference for the elements of the XML configuration file. It describes the use and meaning of each element providing examples. MarGotAspect is intended as a code generator for the mARGOt autotuning framework, therefore it uses concepts and constructs for which to refer to the mARGOt framework documentation.

2 Document Type Definition

```
<!ELEMENT margot (aspect*)>
<!ELEMENT aspect (function-monitor*, region-monitor*, goal-tuner*, state-tuner*)>
<!ATTLIST aspect block_name CDATA #REQUIRED>

<!ELEMENT function-monitor (function-name, return-type, argument*, configure-call)>
<!ELEMENT function-name (#PCDATA)>
<!ELEMENT return-type (#PCDATA)>
<!ELEMENT argument (type, name)>
<!ATTLIST argument sw-knob CDATA #REQUIRED>
<!ELEMENT type (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT configure-call (#PCDATA)>

<!ELEMENT region-monitor (argument+, configure-call)>

<!ELEMENT goal-tuner (control-var, goal-name, rule+)>
<!ELEMENT control-var (type, name)>
<!ELEMENT goal-name (#PCDATA)>
<!ELEMENT rule ((predicate|and|or|not), value)>
<!ELEMENT predicate (#PCDATA)>
<!ATTLIST predicate type CDATA #REQUIRED>
<!ELEMENT and ((predicate|and|or|not), (predicate|and|or|not))>
<!ELEMENT or ((predicate|and|or|not), (predicate|and|or|not))>
<!ELEMENT not ((predicate|and|or|not))>
<!ELEMENT value (#PCDATA)>

<!ELEMENT state-tuner (control-var, rule+)>
```

3 Aspect

The most external element *margot* can contain multiple *aspect* elements, each with a block_name attribute corresponding to a block declared in the mARGOt framework configuration files.

Each *aspect* element can contain any number of function monitors, region monitors, goal tuners and state tuners, and will be mapped to a single aspect (with its related pointcuts and advices) inside the generated aspect file.

It has to be noted that, aside from the XML configuration file contents and the related aspects that will be generated, a "special" aspect is always created containing an advice such that a margot::init() call is performed at the start of the main function of the application program.

4 Monitors

Monitors can either be function monitors or region monitors.

The *argument* elements correspond to variables which can be configured in the mARGOt framework as knobs, or be standard parameters passed to the monitored function; this is specified with the *sw-knob* attribute of each argument, which can take values "*yes*" or "*no*".

The optional *configure-call* subelement corresponds to a function that has to be called whenever the values of the knobs are modified; this function has to be written in the XML document as:

```
function_name(arg1_type arg1_name, arg2_type, arg2_name, ...)
```

4.1 Function Monitor

A function monitor is characterized by the name (*function-name*), the return type (return-type*)* and the arguments (*argument*) of the function that has to be monitored.

The generated aspect code will then start and stop the monitors for the corresponding knob parameters of the function, as configured in the mARGOt framework configuration files; logging activity for the monitors is also performed.

4.1.1 Example

Let's say we want to monitor the execution time of the function *do_work*. It takes as arguments two parameters, one is a normal parameter, the other one is a software knob in the **mARGOt** framework. This is the functional code the user wrote:

```
int main() {
  int trials = 100; // Normal argument
  int knob = 2; // Sofware knob
  for (int i = 0; i < N; ++i) {
    do_work(trials, knob);
  }
} // Functional code
```

So we write an XML file describing our intention to monitor a function named *do_work* taking two parameters, one of which is a *software knob* in the block named *foo*.

```
<margot>
  <aspect block_name="foo">
    <function-monitor>
      <function-name>do_work</function-name>
      <return-type>void</return-type>
      <argument sw-knob="no">
```

```
      <type>int</type>
    <name>trials</name>
    </argument>
    <argument sw-knob="yes">
      <type>int</type>
    <name>knob</name>
    </argument>
   </function-monitor>
  </aspect>
</margot>
```

And here is the AspectC++ code that **MarGotAspect** generates for us. Note that it performs a check on the value of our software knob and then starts the monitor before calling *do_work()*. Lastly, after *do_work()* returned, it stops the monitor and logs the results.

```
aspect GeneralAspect {
  pointcut main_exec() = execution("int main(...)");
  advice main_exec() : before() {
    margot::init();
  }
};
aspect fooAspect {
  pointcut do_work_exec(int trials, int knob) = execution("void do_work(...)") &&
      args(knob);
  advice do_work_exec(trials, knob) : before(int knob) {
    if (margot::foo::update(knob)) {
      margot::foo::manager.configuration_applied();
    }
    margot::foo::start_monitor();
  }

  advice do_work_exec(knob) : after(int knob) {
    margot::foo::stop_monitor();
    margot::foo::log();
  }
};
```

4.2 Region Monitor

Not only function calls can be monitored, but also well-delimited blocks of code (i.e., the regions of interest); the block delimitation has to be manually performed inside the application code, by applying at the start of the block the function call:

`margot_<block_name>_start_roi(knob1_name, knob2_name, ...)`

where all the knobs to monitor are listed as its parameters and <block_name> corresponds to the attribute with the same name for the aspect element, and at the end of the block the function call:

`margot_<block_name>_end_roi()`

4.2.1 Example

Let's stay that we want to monitor a region of code. So the user marks it calling the above function pair. Without any loss of generality let us assume that the user code looks like this:

```
int main() {
  int knob_one = 10;
  int knob_two = 10;
  margot_bar_start_roi(knob_one, knob_two);
  // Whatever
    for (int j = 0; j < inner_loop_size; ++j) {
    // code, like
    }
  // anything
    margot_bar_end_roi();
} // Functional code
```

So this time we define a *region-monitor* block telling **MarGotAspect** that in this **mARGOt** block named *foo* there are 2 *software knobs*.

```
<margot>
  <aspect block_name="bar">
    <region-monitor>
      <argument sw-knob="yes">
        <type>int</type>
        <name>knob_one</name>
      </argument>
      <argument sw-knob="yes">
        <type>int</type>
        <name>knob_two</name>
      </argument>
    </region-monitor>
  </aspect>
</margot>
```

And here is the AspectC++ code that **MarGotAspect** generates for us. As in *function-monitor* case it performs a check on the values of our software knobs and then starts the monitor right before our region-of-interest marker function. When it reaches the end of the region-of-interest it stops the monitor and logs the results.

```
aspect barAspect {
  pointcut bar_start_roi(int knob_one, int knob_two) = execution("void
      margot_bar_start_roi()") && args(knob_one, knob_two);
  pointcut bar_end_roi() = execution("void margot_bar_end_roi()");

  advice bar_start_roi(knob_one, knob_two) : before(int knob_one, int knob_two) {
    if (margot::bar::update(knob_one, knob_two)) {
      margot::bar::manager.configuration_applied();
    }
    margot::bar::start_monitor();
  }
  advice bar_end_roi() : before() {
    margot::bar::stop_monitor();
    margot::bar::log();
  }
};
```

In this case **MarGotAspect** generates another file, the C++ code defining the two marker functions:

```
// roiHeaders.h
void margot_bar_start_roi(int knob_one, int knob_two) { }
void margot_bar_end_roi() { }
```

5 Tuners

Goal tuners and state tuners corresponds to advice code which allows to change at run-time the goal values or the state of the **mARGOt** framework, according to some variable (not just a software knob, as already automatically possible for the framework).

Both of them share the concept of rule, which allows associating each particular state or goal value to a logical predicate to be verified at run-time.

5.1 Rule

Rules are a key aspect for both tuners, as they couple together predicates and goal/state values.

Each *rule* element has indeed an associated *value* element, which corresponds to a possible value for either a state or a goal, and a predicate element.

Predicates can be either simple predicates (*predicate*) or composite predicates (*and*, *or*, *not* elements).

Simple predicates are assigned a predicate value (a numerical value, or a boolean) and they can be of "*eq*", "*neq*", "*gt*", "*gte*", "*lt*", "*lte*" types, as reported in the corresponding *type* attribute; the simple predicate will be satisfied when the associate variable, i.e., the *control-var* element, will have a value such that the <*control-var*> <*predicate-type*> <*predicate-value*> expression is true.

Composite predicates instead use the *and*, *or* and *not* elements to compose (both simple and composite) predicates with the corresponding boolean logic functions.

5.2 Goal Tuner

To each goal tuner, a goal previously defined in the **mARGOt** configuration files is associated through the *goal-name* element.

In the context of the goal tuners, the simple predicate element values correspond to possible values of this goal, and the value of the goal is updated to the value of some rule whose predicate is verified.

5.2.1 Example

This time let us say we want to introduce a set of rules according to which we set the **mARGOt** goal in a block named *foo*. These rules should be applied when a control variable in our code is assigned to some values. Let us consider a piece of functional code that performs assignments to the control variable *goal_condition*. Let's assume that we had previously defined in **mARGOt** a goal named *my_execution_time_goal*.

```
int goal_condition = 1;
int main() {
  // Whatever
  for (int i = 0; i < N; ++i) {
    if (goal_condition >= 8) goal_condition = 1;
    else goal_condition = goal_condition + 1;
    }
  // code
} // Functional code
```

To accomplish our requirements we define a *goal-tuner* block telling **MarGotAspect** that the control variable is named *goal_condition* and of type *int*. Then we explicit our set of rules. Let us assume that in case *goal_condition = 1* we want our goal set to *80000*. Again in case *3 < goal_condition < 6 or 7 <= goal_condition <= 8* we want our goal set to *300000*. We can describe all of this in the XML file.

```
<margot>
  <aspect block_name="foo">
    <goal-tuner>
      <control-var>
        <type>int</type>
        <name>goal_condition</name>
      </control-var>
      <goal-name>my_execution_time_goal</goal-name>
      <rule>
        <predicate type="eq">1</predicate>
        <value>80000</value>
      </rule>
      <rule>
        <predicate type="eq">2</predicate>
        <value>200000</value>
      </rule>
      <rule>
        <not>
          <predicate type="neq">3</predicate>
        </not>
        <value>50000</value>
      </rule>
      <rule>
        <or>
          <and>
            <predicate type="gt">3</predicate>
            <predicate type="lt">6</predicate>
          </and>
          <and>
            <predicate type="gte">7</predicate>
            <predicate type="lte">8</predicate>
          </and>
        </or>
        <value>300000</value>
      </rule>
    </goal-tuner>
  </aspect>
</margot>
```

And here is the AspectC++ code that **MarGotAspect** generates for us. Quite short this time. In this case we execute a function named *tune_my_execution_time_goal* passing the new value of *goal_condition* as argument.

```
aspect fooAspect {
  pointcut goal_condition_set() = set("int ...::goal_condition");
  advice goal_condition_set() : after () {
    tune_my_execution_time_goal(*tjp->entity());
  }
```

```
};
```

tune_my_execution_time_goal is a function generated as well by **MarGotAspect** that implements the check on the new value of *goal_condition*:

```
// margotAspect.h
#include <margot.hpp>

void tune_my_execution_time_goal(int goal_condition) {
  if (goal_condition == 1) {
    margot::foo::goal::my_execution_time_goal.set(80000);
  } else if (goal_condition == 2) {
    margot::foo::goal::my_execution_time_goal.set(200000);
  } else if (!(goal_condition != 3)) {
    margot::foo::goal::my_execution_time_goal.set(50000);
  } else if (((goal_condition > 3 && goal_condition < 6) || (goal_condition >= 7 &&
      goal_condition <= 8))) {
    margot::foo::goal::my_execution_time_goal.set(300000);
  }
}
```

5.3 State Tuner

In the context of the state tuners, the simple predicate element values correspond to the names of states as defined in the **mARGOt** configuration files, and the state of the framework is updated to the value of some rule whose predicate is verified.

5.3.1 Example

This case is similar to the previous one. We still need a set of rules according to which we set the **mARGOt state** in a block named *foo*. These rules should be applied when a control variable in our code is assigned to some values. Let us consider a piece of functional code that performs assignments to the control variable *state_condition*. Let's assume that we had previously defined in **mARGOt** two states named *my_optimization* and *my_optimization_two*.

```
bool state_condition = true;
int main() {
  // Whatever
  for (int i = 0; i < N; ++i) {
    if (i % 2 == 0) state_condition = false;
    else state_condition = true;
    }
  // code
} // Functional code
```

To accomplish our requirements we define a *state-tuner* block telling **MarGotAspect** that the control variable is named *state_condition* and of type bool. Then we explicit our set of rules. This time we case the boolean value of *state_condition*. If it's *true* we want *my_optimization* set as **state** otherwise we want *my_optimization_two*.

```
<margot>
  <aspect block_name="foo">
    <state-tuner>
```

```
      <control-var>
        <type>bool</type>
        <name>state_condition</name>
      </control-var>
      <rule>
        <predicate type="eq">true</predicate>
        <value>my_optimization</value>
      </rule>
      <rule>
        <predicate type="eq">false</predicate>
        <value>my_optimization_two</value>
      </rule>
    </state-tuner>
  </aspect>
</margot>
```

The AspectC++ code that **MarGotAspect** generates for us is basically the same of *goal-tuner*. In this case we execute a function named *tune_foo_state* passing the new value of *state_condition* as argument.

```
aspect fooAspect {
  pointcut state_condition_set() = set("bool ...::state_condition");
  advice state_condition_set() : after () {
    tune_foo_state(*tjp->entity());
  }
};
```

tune_foo_state is a function generated as well by **MarGotAspect** that implements the check on the new value of *state_condition*:

```
// margotAspect.h
#include <margot.hpp>

void tune_foo_state(bool state_condition) {
  if (state_condition == true) {
    margot::foo::manager.change_active_state("my_optimization");
  } else if (state_condition == false) {
    margot::foo::manager.change_active_state("my_optimization_two");
  }
}
```