CHAPTER

ONE

TUTORIAL INTRODUCTION TO AML

This chapter provides a quick introduction to the Axini Modeling Language (AML). Our aim is to show the essential elements of the language with some small examples. We are not trying to be complete. We want to get you up to speed with AML as quickly as possible to the point where you can write useful models. We will concentrate on the basics of AML: processes, channels, stimuli, responses, states and variables.

AML is the modeling language of the Axini Modeling Platform (AMP). AML can be used to describe the behavior of reactive systems. Reactive systems – sometimes called control-oriented systems – are systems which react on input and evolve: outputs do not only depend on the current input but also on earlier inputs. Typically, reactive systems are meant to be executed non-stop and will never terminate. Examples of reactive systems are communication protocols and multi-threaded systems with several interfaces. Reactive systems are often specified using state-transition based notations. AML has a formal, mathematical semantics; therefore it can be used to automatically generate test cases.

A *model* of a reactive system describes the inputs that it should accept and the outputs it may, or should, send. A model is an abstract, high-level description of the System Under Test (SUT). A model describes the behavior of the system, and specifies *what* the system should do. Conversely, a computer program (i.e., the software) prescribes the system, and dictates *how* the system works. An AML model is a description of the SUT; the test tool (AMP) is responsible for generating test cases from the model.

1.1 Hello Tea

As with programming, the only way to learn a new modeling language is by writing models in it. The first model that we are going to write is a model for a machine which provides tea: after pressing a `button`, the machine offers a cup of `tea`. The following AML model describes such a system.

```
1  # Hello Tea: our first AML model!
2
3  external 'machine'
4
5  process('hello-tea') {
6    timeout 2.0
7
8    channel('machine') {
9      stimulus 'button'
10     response 'tea'
11   }
12
13   receive 'button'
14   send 'tea'
15 }
```

The model starts with a line comment: everything after a # until the end of the line is ignored. The model uses a single external channel called `'machine'`. An external channel represents an interface of the SUT to its environment. The model defines a single process `'hello-tea'` which describes the external behavior of the system: the interactions of the system with its environment.

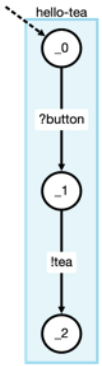
Within the process we have to declare the actions which the process can do to interact with the environment. On the external channel `'machine'` we define two actions: a stimulus `'button'` and a response `'tea'`. A stimulus is an input to the system and a response is an output from the system. The generic name for a stimulus or a response is *label*. An action is then the occurrence of a label.

We also need to define the `timeout` for this process: the maximum time in seconds that AMP will wait for a response to arrive. When testing, AMP will exit with the verdict *fail* if a response does not arrive in time.

After these definitions we specify the behavior of this process. First, the process will receive a `'button'` and then it will send a `'tea'`. Note that our first model is not really a reactive system: after a single `'button'` stimulus and a single `'tea'` response, the process ends.

Although a model is different from a computer program, the structure of a model and its processes follows the structure of a program: first we define the channels, the labels, etc., and then we use these definitions to define the behavior of the processes.

AML has a formal, mathematical semantics. Each process is mapped upon on a state-transition system, where each state may have outgoing transitions which carry labels of the process. When your AML model is syntactically correct, you can use AMP's VISUALIZE menu to show the state transition systems of the process of the model. As a matter of fact, the figure below is a screenshot of the `'hello-tea'` process that we defined above.



1.2 Choice and goto

Let's extend our previous model to describe a system which also serves coffee. Coffee is more expensive than tea, so this machine requires a coin before serving coffee. Furthermore, we want the system to continue to work after serving a beverage. The following AML model specifies such a system.

```
external 'machine'

process('tea-coffee') {
  timeout 2.0

  channel('machine') {
    stimulus 'button'
    stimulus 'coin'
    response 'tea'
    response 'coffee'
  }

state 'idle'
  choice {
    o {
      receive 'button'
      send 'tea'
```

(continues on next page)

(continued from previous page)

```
    }
    o {
      receive 'coin'
      send 'coffee'
    }
  }
  goto 'idle'
}
```

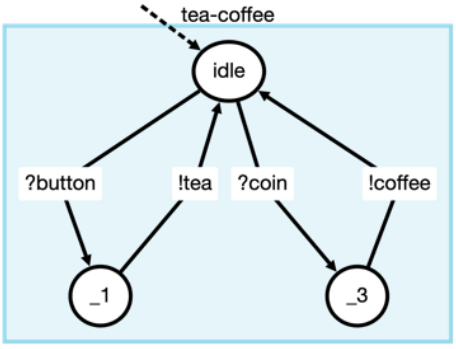
Again, we first have to declare the labels of the process.

We have added a state `'idle'` to the system: in this state, the system is waiting for input. In the state `'idle'` there is a *choice* between two inputs: either a press of the `'button'` or the insertion of a `'coin'`. AML provides the `choice` construct to specify a choice between several options: each option starts with an `o` (lowercase `o`, which makes the options of a choice look like an unordered, bulleted list), followed by a sequence of zero or more actions, enclosed in curly brackets (`{ }`).

A user only has to explicitly name a state of a process, if the process needs to jump to this state using a `goto`.

In this example model we can recognize some typical syntax rules of AML. AML uses curly brackets (`{}`) to group constructs together: the definitions and behavior of a process, the labels of a channel, the options of a choice, the options themselves, etc. Identifiers in AML are written as strings: the names of processes, channels, labels, states, etc. Such strings can be written between single quotes (`'name'`) or double quotes (`"name"`).

The semantics of the process `'tea-coffee'` is the following state-transition diagram:



1.2.1 Choose your options wisely

When there is a choice between stimuli, there is no priority between the various options: when testing, AMP can offer either of them to the SUT. For our model, when AMP chooses to offer a `'button'`, the choice is resolved and AMP will then wait to observe `'tea'`. If a `'coin'` is offered, AMP will wait to observe `'coffee'`. In both cases, after sending the beverages, the options are ended and we continue after the closing bracket of the `choice` construct: in this case the statement `goto 'idle'`. We will go back to the initial state where there is again the choice between the stimuli.

Note that AML's non-deterministic `choice` is a high-level construct that does not have a counterpart in a programming language: a computer program dictates precisely what instructions have to be executed, whereas the `choice` construct specifies that multiple options are possible.

Typically, the alternatives of a `choice` are all stimuli or all responses. Mixing stimuli and responses in a `choice` seems natural from a modelling point of view. A system might be able to accept a stimulus or do some output. However, with respect to testing it is not clear what this would mean. When should we do the stimulus? How long should we wait for the response? And although the syntax of AML allows the mixing of stimuli and responses in a `choice`, AMP will always give precedence to the responses: it will wait to observe the responses, and thus ignore the stimuli. The reason for this is subtle: if we were to choose the stimulus and later observe the response, we do not know whether the response is caused by the stimulus or whether the response is just late.

1.3 Repeat

Within programming languages, the `goto` statement is often considered harmful. When modelling reactive systems, though, the use of states and `goto` statements is natural: the behaviour of the SUT is often specified as a state-transition system in the specification of the SUT.

Still, AML offers a `repeat` statement as a looping construct. Like the `choice` construct, the `repeat` construct consists of a list of options, which specify the possible actions which can be repeated. When the end of an specific option is reached, control is transferred back to the beginning of the `repeat` statement.

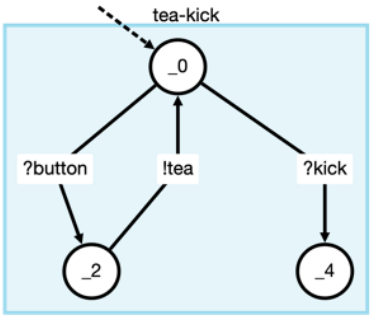
To illustrate the `repeat` construct, we will model a system which continually serves tea after pressing a button. But the system also accepts a `'kick'` from an impatient customer after which the system comes to a halt.

```
external 'machine'

process('tea-kick') {
  timeout 2.0

  channel('machine') {
    stimuli   'button', 'kick'
    responses 'tea'
  }

  repeat {
    o { receive 'button'; send 'tea'      }
    o { receive 'kick';   stop_repetition }
  }
}
```

In the `repeat` loop there are two options: either a press on the `'button'` or a `'kick'`. After the `'button'`, the system sends a `'tea'` and goes back to the beginning of the `repeat`. After the `'kick'`, though, the command `stop_repetition` ensures that `repeat` loop is ended and the process terminates. The `stop_repetition` mimicks the `break` statement often found in programming languages.

Usually, we place the statements of an AML model on different lines. Sometimes – when the statements are short – we put the statements on the same line. If we do so, the statements have to be delimited using semicolons (`;`).

The semantics of the process `'tea-kick'` is depicted below:



1.4 Label parameters

In the previous models, the observable labels were just abstract names: `'button'`, `'coin'`, `'kick'`, `'tea'` and `'coffee'`. In practice, however, stimuli and responses usually carry data. AML supports data through parameters on the labels.

In the definition of a label, its label parameters have to be specified as a list of `'name' => <type>` definitions (optionally enclosed in curly brackets). The name is a string and the <type> can be any of AML's *simple* types: `:integer`, `:boolean`, `:decimal`, `:string`, `:date` and `:time`, or a *complex* type like a list, struct or hash.

We are going to model a beverage machine which does not longer give out tea for free: every beverage has its price. A cup of tea requires a coin of 50 cents and the cup of coffee requires a coin of 100 or 200 cents. Moreover, the machine determines the amount of beverage to be poured on the basis of the coin inserted.

```
external 'machine'

process('tea-coffee-coins') {
  timeout 2.0

  channel('machine') {
    stimulus 'coin',   { 'value' =>  :integer }
    response 'tea',    { 'volume' => :integer }
    response 'coffee', { 'volume' => :integer }
  }

  repeat {
    o {
      receive 'coin', constraint: 'value == 50'
      send 'tea',     constraint: 'volume == 300'
    }
    o {
      receive 'coin', constraint: 'value == 100'
      send 'coffee',  constraint: 'volume == 200'
    }
    o {
      receive 'coin', constraint: 'value == 200'
      send 'coffee',  constraint: 'volume == 500'
    }
  }
}
```

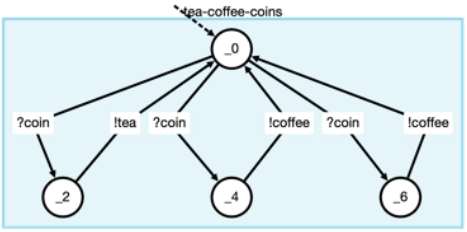
In the declaration of the labels, the stimulus `'coin'` has been attached an parameter `'value'` of type `:integer`. The responses `'tea'` and `'coffee'` both have an parameter `'volume'`, also of type `:integer`.

The behavior of the process is defined with a `repeat` loop. There are three options: one for a `'coin'` of 50, one for a `'coin'` of 100, and one for a `'coin'` of 200. When a `'coin'` of 50 is received, we expect to observe a `'tea'` with a volume of 300. Similarly for the other coins.

The handling of the values of the label parameters is done through the `constraint:` option of the action. The constraint is a string defining a boolean expression, which should evaluate to true for the action to be enabled. If the label is a *stimulus*, the constraint should constrain the values of all parameters of the stimulus, i.e., defining possible values for the parameters such that the constraint is true. If the label is a *response*, the constraint should also evaluate to true; the value of the label parameters are checked by the expression.

In our model, the value of the `'coin'` stimuli are restricted by equality expressions: there is only one way for which the expression evaluates to true, i.e., the assignment of `'value'` to the particular integer constant.

The semantics of the process `'tea-coffee-coins'` is the following.



1.5 Process variables

AML supports process variables – also known as state variables – to store and manipulate data within a process. This is similar to local variables of methods, as supported by most programming languages. Variables have a name (a string), a type (same as for label parameters) and a value. Whereas label parameters only have a value when the label occurs as an action (i.e., `send` or `receive`), process variables are accessible everywhere within the process. Variables have to be declared before they can be used:

```
var 'name', <type> [, <initial_value> ]
```

We will model a gambling system, which accepts `'money'` with a certain `'value'` and either returns nothing or a price which is twice the `'value'` which was bet.

```
external 'external'

process('gambling') {
  timeout 1.0

  channel('external') {
    stimulus 'money', { 'value' => :integer }
    response 'prize', { 'value' => :integer }
  }

  var 'bet', :integer, 0

  repeat {
    o {
      receive 'money',
          constraint: 'value > 0 && value <= 100',
              update: 'bet = value'

      choice {
        o { send 'prize', constraint: 'value == 0'       }
        o { send 'prize', constraint: 'value == 2 * bet' }
      }
    }
  }
}
```

The variable `'bet'` will hold the `'value'` of the `'money'` inserted. The variable `'bet'` is declared with the type `:integer` and initialized on 0. The constraint of the `'money'` action specifies that the `'value'` should be greater than 0 and at most 100. When testing, AMP will choose a value between 0 and 100 to make the constraint true.

Apart from the constraint, the `'money'` action has another option: `update`. The `update` option is a string containing assignments which update process variables. In our model the `'value'` of `'money'` is saved into the variable `'bet'`. When there are multiple assignments, they should be seperated by semicolons.

The model specifies that there are two possible `'price'` responses: one with a `'value'` of 0, and one with a `'value'` which is twice the `'bet'`.

It turns out that the behavior of the gambling machine is slightly more subtle than we modeled in the first place. The machine remembers the balance of the money which was inserted and the money given as a price. Only when the balance is positive – more money has been inserted than has been given away – it *might* give a price with twice the value bet. The more precise model is the following.

```
external 'external'

process('gambling-profit') {
  timeout 1.0

  channel('external') {
    stimulus 'money', { 'value' => :integer }
    response 'prize', { 'value' => :integer }
  }

  var 'bet',     :integer
  var 'balance', :integer, 0

  repeat {
    o {
      receive 'money',
          constraint: 'value > 0 && value <= 100',
              update: 'bet = value'

      choice {
        o {
          send 'prize',
              constraint: 'value == 0',
                  update: 'balance = balance + bet'
        }
        o {
          send 'prize',
              constraint: 'balance > 0 && value == 2 * bet',
                  update: 'balance = balance - 2 * bet'
        }
      }
    }
  }
}
```

The variable `'balance'` holds the balance between the inserted money and the money given as a price; initially this balance is 0. Note that in this model we did not initialize `'bet'`: initialization is not really needed as it will get its initial value when `'money'` is inserted.

The constraint on the doubled `'price'` now includes a constraint on `'balance'`: only when this is positive, this action is enabled. Both `'price'` actions need to update the `'balance'` variable.

1.6 Behaviors

When AML models grow larger it might become harder to understand and manage the complete behavior of the model. AML offers the `behavior` construct to structure an AML model into different sub-behaviors. An AML `behavior` is defined within a process, and may contain a series of AML statements.

An AML `behavior` is like a method or procedure of a programming language: it allows for local variables, you can pass values to a `behavior` and it may return a value. A `behavior` can access the process variables and labels of the process in which it is defined. There are two types of behaviors: terminating and non-terminating behaviors. *Terminating* behaviors are like methods: they will eventually terminate and control is transferred back to the callee. Terminating behaviors are called with `call` construct. *Non-terminating* behaviors are not meant

to terminate: when called, they behave like goto's: control is never transferred back to the callee. Transfer to a non-terminating behavior is done with the `behave_as` construct.

Let's illustrate the usage of `behavior` through an example.

```
external 'machine'

process('tea-coffee') {
  timeout 2.0

  channel('machine') {
    stimuli   'coin', 'button_tea', 'button_coffee'
    responses 'tea', 'coffee'
  }

  behavior('insert coin') {
    receive 'coin'
  }

  behavior('press button', :terminating, [], :string) {
    choice {
      o { receive 'button_tea';    exit_with "'tea'"    }
      o { receive 'button_coffee'; exit_with "'coffee'" }
    }
  }

  behavior('offer drink', :terminating, ['drink' => :string]) {
    choice {
      o { send 'tea',    constraint: "drink == 'tea'"    }
      o { send 'coffee', constraint: "drink == 'coffee'" }
    }
  }

  behavior('main', :non_terminating) {
    var 'drink_chosen', :string

    call 'insert coin'
    call 'press button', [], into: 'drink_chosen'
    call 'offer drink', ['drink_chosen']

    behave_as 'main'
  }

  behave_as 'main'
}
```

This model is another model of a beverage machine. The machine first accepts a `'coin'`. Afterwards, an user can either press `'button_tea'` or `'button_coffee'`. The machine delivers the beverage of the button pressed.

In this model, three terminating and one non-terminating behaviors are defined. Only when a terminating behavior uses parameters and/or returns a value, one has to specify that the behavior is `:terminating`. The behavior `'insert coin'` does not have parameters and does not return a value, so the `:terminating` parameter is not needed.

The terminating behavior `'press button'` does not have any parameters (hence the empty list `[]`), but returns a value of type `:string`. The body of the behavior consists of choice between either `'button_tea'` and `'button_coffee'`. Depending on the choice, the behaviors exits with a different value using the `exit_with` statement. This statement has a string argument which will be evaluated, like the `constraint:` and `update:` options of actions, that we saw earlier.

The terminating behavior `'offer drink'` is passed a parameter of type string. Within the behavior this parameter can be accessed as variable `'drink'`. This behavior does not return any value, so the return type can be left out in de definition.

Finally, the non-terminating behavior `'main'` specifies the complete behavior of the machine. It calls the three terminating behaviors, and then calls itself (or better: behaves as itself): this essentially leads to an endless loop of offering tea or coffee. The behavior uses a local variable `'chosen_drink'` which stores the button pressed from the behavior `'press button'`.

A non-terminating behavior can be seen as an advanced `state`. The `behave_as` is then the `goto` for such behavior. As such, it fundamentally differs from (recursive) methods calls in programming languages. As a non-terminating behavior never returns to the callee there is no need to retain the information of this callee.

We agree that the syntax of specifying a `behavior` is not elegant, and needs some time to get used to. The cumbersome syntax is due to the fact that (the current version of) AML is defined as a Ruby DSL.

1.7 Multiple processes

So far, our AML models consisted of a single process. AML models, however, typically consist of several processes which execute independently of each other. Processes can communicate with each other over `internal` channels, which have to be declared outside the processes. As with the `external` channels, the labels of the `internal` channels have to be defined in the processes which use these channels. The processes have to define matching pairs of internal stimuli and responses.

Let's look at another variation on our beverage machine.

```
external 'user', 'display'
internal 'internal'

process('beverages') {
  timeout 10.0

  channel('user') {
    stimulus 'coin',     { 'money' => :integer }
    response 'beverage', { 'drink' => :string }
  }
  channel('internal') {
    response '_beverage', { 'drink' => :string, 'price' => :integer }
  }

  var 'DRINKS', {:integer => :string}, {
    50  => 'tea', 100 => 'coffee', 200 => 'soda'
  }

  var 'inserted_coin', :integer
  var 'corresponding_drink', :string

  repeat {
    o {
      receive 'coin',
          constraint: "money in keys(DRINKS)",
              update: 'inserted_coin = money;
                       corresponding_drink = DRINKS[money]'

      send '_beverage',
          constraint: 'drink == corresponding_drink &&
                       price == inserted_coin'

```

(continues on next page)

(continued from previous page)

```
      send 'beverage',
          constraint: 'drink == corresponding_drink'
    }
  }
}

process('display') {
  timeout 1.0

  channel('display')  {
     response 'message', { 'beverage' => :string, 'total' => :integer }
  }
  channel('internal') {
    stimulus '_beverage', { 'drink' => :string, 'price' => :integer }
  }

  var 'current_drink', :string
  var 'total_money', :integer, 0

  repeat {
    o {
      receive '_beverage',
          update: 'current_drink = drink;
                   total_money += price'

      send 'message',
          constraint: "beverage == current_drink &&
                       total == total_money"
    }
  }
}
```

The machine that is modelled here has a separate display which will show the beverage which is currently being prepared *and* the total of money that has been inserted. The external channel `'user'` carries the `'coin'` and `'beverage'` labels. The external channel `'display'` shows a `'messsage'` with the beverage and the total. The process `'beverages'` services the `'user'` channel and the process `'display'` manages the `'display'` channel. The process `'beverages'` sends an internal message to `'display'` on the coin that has been inserted, and the beverage that will be delivered.

Before discussing the internal communication, let's have a closer look at the `display` process. We use variables `'inserted_coin'` and `'corresponding_drink'` to store the value of the coin and corresponding drink. The variable `'DRINKS'` is a hash variable which maps `:integer` keys upon `:string` values. It represents the relationship between the inserted coin and the associated beverage. As AML does not (yet) support constants, we have put `'DRINKS'` in uppercase to remind us that its value will never change. When receiving a `'coin'`, we use the AML builtin function `keys` to get the list of all keys of `DRINKS` and let AMP choose one of them. After AMP has chosen one of these keys, we get the corresponding drink using the index operator `[]` on `DRINKS`.

Both processes communicate with each other over the `'internal'` channel: the process `'beverages'` sends a message `'_beverage'` and the process `'display'` receives this message. When the two processes communicate, it is a synchronized, handshake communication: it happens as a single step. The label parameters of the `'_beverage'` label are the same for both processes. We are free to name internal channels and the labels which are send over these channels. By convention we often start the *internal* labels with an underscore (_) to distinguish them from the observable labels of the *external* channel(s).

Internal communication is much like the external communication that we have seen before: we can use the `constraint:` option to specify constraints on the labels or variables and can use the `update:` option to update variables. There is one important difference though: the sender (in this case the `'beverages'` process) must specify the values for all label parameters. The receiving process (here the `'display'` process) receives the label

and its parameters and may use the `constraint` to check the values.

A internal communication action is *not* an observable action. Although internal communication is visible in the state diagram of a process, it will *not* appear in the test case of AMP: it is an invisible internal action, often referred to as a tau-step.

Note that both processes have different `timeout` values: the message on the display should be there within one second, but the delivering of the drink may take ten seconds.

1.8 Advanced Features

This ends our tutorial introduction on AML. Several advanced and useful aspects of AML have not been discussed, though. Just to give you a taste, we list some of them below:

- the use of `include` to organize the model,
- the use of Ruby as a pre-processor to (conditionally) define the model,
- `function` definitions for complex computations on testing-time,
- `urgent` transitions to enforce internal communication,
- timed testing: the keywords `after` and `before` and the variable `clock`,
- `expedited` stimuli, and
- more details on structured data types for parameters and variables: lists, structures, and hashes.

Please refer to AMP's on-line help and/or AML's Reference Manual to learn about these advanced features.