**Model Based Testing** with the
**Axini Modeling Platform**
– LABORATORY –

# Introduction

In this laboratory, you will carry out some practical exercises with a state-of-the-art Model Based Testing (MBT) tool: the Axini Modeling Platform (AMP). AMP is an industrial-strength MBT tool, developed by Axini[1] during the last decade.

The modeling language used in AMP is the Axini Modeling Language (AML). The semantics of AML models is defined upon Symbolic Transitions Systems (STS), a data-extension of Labeled Transition Systems (LTS). For this laboratory we will mostly use the LTS part of AML.

AMP is an integrated development Platform for AML models. Models can be edited, visualized, simulated, and most importantly, tested against a system under test (SUT). All test results are automatically saved. AMP uses an *online testing* approach, which means that test cases are generated on-the-fly while testing the SUT, rather than beforehand.

To become familiar with AMP, you will first do an introductory exercise with a Coffee Machine model and SUT. This will give you some hands-on experience with modeling and testing. Afterwards, you will be ready to do a small project with AMP, where you will model a remote controlled door (SMARTDOOR) from scratch using a specification document and test various implementations of that system.

[1] https://www.axini.com/

# 1. The Coffee Machine

In this exercise you will play around with a simple model of a Coffee Machine which dispenses coffee, tea and lemonade. You will see the core features of AMP, extend a small AML model and test an implementation of the Coffee Machine using your model.

AMP is a web application. For this laboratory, an online version of AMP is available at
`https://course02.axini.com`.
Only the browsers *Google Chrome* and *Firefox* are fully supported, but other modern browsers should work too.

## 1.1 Exploring AMP

When you log in for the first time, you will be placed in the TEST page with the message 'This test set has no test runs yet.'

Your environment should contains two *projects* to work in: '`Coffee Machine`' and '`SmartDoor`'. Select the '`Coffee Machine`' project using the navigation bar at the top of the screen. In this navigation bar, you can also select a *test set*, which are used to manage different testing configurations. The '`Coffee Machine`' project contains only a single test set.

We will start by exploring the main features of AMP.

1. MODEL. Press the MODEL button in the left pane to enter the model editor. This is the integrated modeling environment of AMP where you can manage your AML models. To get you started, your repository already contains a beginning of the model of the Coffee Machine in the model part `coffee_machine.aml`.

   Inspect this model carefully. Do you understand everything?

2. VISUALIZE. Press the menu VISUALIZE. This will open a new tab in your browser. In this new window, the STS of the model is shown. Can you relate the visualized STS to the textual model?

3. EXPLORE. Select the main tab of AMP again and press EXPLORE. This will start the explorer, a simulator for the model. AMP will offer a menu of actions it can perform. In the initial state, two actions are possible: we can select the `?[controller]button_coffee` stimulus or let time pass using the `tick` action. Under 'State vectors (1)' the current state of the process `coffee machine` is shown: `coffee machine_start`, the initial state. Select the `?[controller]button_coffee` stimulus and press the 'Advance' button in the top of the window to simulate pressing this button. AMP will *advance* to the next state and will offer a new set of actions to choose from. At the bottom of the page, the 'Explored Trace' shows the actions executed.

   Note that the explorer only uses the model: the actual Coffee Machine implementation is not used yet.

4. TEST. Now we are ready to test an implementation of the Coffee Machine. Press the TEST button. This brings us back to the test page. Press the 'Start' button to test whether the SUT conforms to the model. You will see a graph which shows the *transition coverage* of the model. Below the graph, a summary of the test cases that have been executed is shown. This project is configured in such a way that a test run consists of three test cases, each of ten steps each. All three test cases should have passed.

5. TEST CASE DETAILS. Select one of the test cases on the test page. A detailed trace of the steps is shown. Note that the trace is presented in the same way as an exploration of the explorer. This test case is derived from the model. Can you relate the two?

6. COVERAGE. After the test, AMP can give information on the *state coverage* and *transition coverage* of the test run. Press the COVERAGE button on the left. The page shows that the test run has covered all states and transitions of the model. Press the VISUALIZE button, which opens a page with the STS of the model where the states and transitions are either green (visited) or red (not visited). Due to the 100% coverage, all states and transitions are green. Note that behavior of SUT not included in the model is not tested and cannot be reported on.
7. HELP. Finally, if you want to learn more about AMP or AML, the tool includes a help page, which can be accessed in the top right corner.

## 1.2 Extending the model

The model currently only describes how to get coffee. The actual Coffee Machine also dispenses tea when `'button_tea'` is pressed. As you can see in the generated test cases, this behavior is therefore not tested either [2].

□ Extend the model with the behavior to dispense tea. When done, check the visualization of your model. Does it look as expected?

*Hint: you can use the* `choice` *keyword from* AML *to model alternative paths.*

## 1.3 Testing the SUT

□ Test the SUT again with your extended model. The test run should pass. If not, inspect your model carefully and fix any mistakes.

□ Inspect the generated test cases and the coverage page to confirm that the added behavior is tested.

## 1.4 When life gives you lemons

The Coffee Machine can also dispense *lemonade* when the corresponding button is pressed. Let us add this behavior to the model as well.

□ We have not yet defined labels in the model for stimulus `'button_lemonade'` and response `'lemonade'`. Add them now. Note that the label names cannot be chosen freely; these label names are implemented in the adapter we are using to test the actual Coffee Machine SUT. Therefore, the model must use the same label names [3].

□ When you are done, test the SUT again. Inspect the executed test cases carefully. What does the model expect after `'button_lemonade'` is pressed? How does the SUT actually react?

□ Inspect a number of different test cases. Can you discern a pattern in the behavior of the SUT?

[2] We allow our models to be *underspecified*, which means not all possible behavior has to be included. This is a very useful property when testing, but also requires care to ensure we test everything we want to.

[3] When you make your model before developing the adapter it will be the other way around. In practice, both are built around the same time when testing a new system.

# 2. SMARTDOOR

Now that you are more familiar with AMP, we can continue to the main event of this laboratory. You will do a small MBT project in which you will model and test various implementations of a remote controlled door called SMARTDOOR. This project should give you a good feel for how MBT works in practice.

In AMP, you can select the SMARTDOOR project using the navigation bar at the top.

## 2.1 Modeling

In this exercise, you will write a model in AML from scratch.

Informally, the door has the following features: it can be opened, closed and locked, and has passcode that is set when the door is locked. The provided specification document provides all the details.

In the following sections, we will guide you through modeling the behavior of SMARTDOOR. After each step, you are recommended to check the visualization to see if the model ended up the way you intended. Moreover, the SMARTDOOR implementation by Axini has already been tested with MBT. You can consider this implementation to be correct and use it to validate your model.

### 2.1.1 The adapter

The adapter needed to test the various SMARTDOOR implementations has been provided for you. This adapter can translate all messages of the SMARTDOOR protocol back and forth between the symbolic representation of the model and the technical formats used by the different implementations.

Since the adapter translates symbolic labels from the model, it requires built-in knowledge of the names of channels, labels and label parameters used in the model. Please follow any naming instructions carefully. Moreover, the adapter also defines which interactions with the SUT are possible. As you will see, some cases cannot be tested with the given adapter.

The adapter also ensures the SUT is in a known state when a test case is started. The door will always begin *closed* and *unlocked*.

### 2.1.2 Skeleton

First, we will write a skeleton for the model using what you have learned during the Coffee Machine exercise.

- □ Define a *process* for the door and an external *channel* named `'door'`.
- □ Define the labels needed for the model. The label and channel names cannot be chosen freely as they they must be known by the adapter used to the SUT. The configured adapter implements the labels as written in the specification on the channel named `'door'`. Start with the stimuli `'open'` and `'close'` and the responses `'opened'` and `'closed'`.
- □ Optionally, you could also define the other stimuli and responses at this stage. This might save some time later on.
- □ Set a global `timeout` for responses.

### 2.1.3 Open and close: good weather

When modeling a new system, we generally start small and extend the model over time. The SMARTDOOR model is well suited for this approach.

□ Let us start with modeling some of the *good weather* behavior. This is the intended usage of the system. Add the AML fragment that describes the opening and closing behavior of the door. You can use the same modeling concepts used in the Coffee Machine model.

### 2.1.4 Open and close: bad weather

Now we will model how the SUT should react to *bad weather* behavior of the environment. Bad weather behavior is concerned with behavior which should not be offered by well behaved clients of the SUT. These are typically stimuli which are not allowed in certain states or invalid data values.

□ Extend the model so that it can receive an `'open'` stimulus when the SMARTDOOR is already opened and it can receive a `'close'` stimulus when the SMARTDOOR is already closed. What are the expected responses in these cases?

*Hint: you will need to define a new response in your model. Otherwise, you will see the following error:* `No label is defined that matches with X.`

*Hint: you can use the* `repeat` *keyword instead of the* `choice` *keyword to model staying in the same state without the need of* `goto`*.*

### 2.1.5 Locking and unlocking

Now we are going to add the locking and unlocking behavior of the specification to the model. Here, you will see the data aspect of symbolic transition systems come into play.

□ Model the new labels, states and transitions for locking and unlocking the door. You can ignore the passcode at this stage. Do not forget to model the relevant bad weather behavior.

□ Define a *label parameter* named `'passcode'` of type `:integer` for the `'lock'` and `'unlock'` stimuli. The adapter recognizes this label parameter name and uses it when locking and unlocking the door [4].

□ Use the `constraint:` option of the `receive` keyword to constrain the value of the passcode on the relevant transitions. Read the specification carefully to ensure that all possible behavior is covered. For example, what should happen when the door is locked with an invalid passcode?

*Hint: You can use a* state variable *of type* `:integer` *to store the selected passcode in your model. The value of this variable can be referenced later to accept or reject the passcode of the* `'unlock'` *command.*

[4] The adapter will translate the integer representation of the passcode (e.g. `42`) into the four digit format (e.g. `'0042'`). This makes it easy to model the domain of passcodes. However, as a result the case where the given passcode is too short cannot be tested.

## 2.2 Testing

When your model is completed, you can start testing the various SMARTDOOR implementations. Within the `SmartDoor` project, you will find a test set for each manufacturer.

The following implementations are available for testing:

1. Axini
2. Besto
3. Logica
4. OnTarget
5. quickerr
6. SmartSoft
7. TrustedTechnologies
8. univerSolutions
9. XtraSafe

As stated above, the implementation of Axini is intended to be correct, so you can use it to validate your model. All test cases should pass when testing this SUT.

When a test case fails, AMP has detected a difference between the observed and the modeled behavior. It is up to you to analyze both and conclude whether the model or the SUT is in the wrong. When in doubt, the specification can often provide the answer. If you conclude that the SUT is in the wrong, you have found a bug in the implementation. Note that implementations might contain multiple bugs, so do your analysis carefully.

When we find bugs in the real world, we report them in such a way that a product owner has sufficient information to evaluate our findings, and developers have enough information to fix them. This often involves snippets from the log file to catch the SUT red handed.

For you, it is enough if you can answer the following questions.

1. In what state(s)/situation(s) does the bug occur?
2. What is the expected behavior?
3. What behavior is observed instead?
4. What is your hypothesis of how the bug works?

*Tip: Since you are testing multiple implementations, it is recommended to track the found bugs carefully. You can use the tag system of* AMP *to categorize (failed) test cases for easier analysis.*

The *Effective AML* guide found on the help page of AMS can provide additional tips on dealing with bugs and making alternative behavior togglable in your model using Ruby macros.