axini

Model Based Testing
with the
Axini Modeling Platform

Theo Ruys

theo.ruys@axini.com

UT Software Testing
Spring 2025

axini

Axini Lecture 1

Introduction to AMP and AML
Coffee Machine
SmartDoor
Plugin Adapters

Monday, 24-Mar-2025

MBT of NewsFeed with AMP

- Goal: apply Model-Based Testing (**MBT**) to test **NewsFeed** using the Axini Modeling Platform (**AMP**).
- To get there:
  1. **Learn** the Axini Modeling Language (**AML**).
  2. **Play** with **AMP** using an example SUT: **SmartDoor**.
     - This part will be **graded** per group.
  3. Create an **adapter** to connect **AMP** to **NewsFeed**.
  4. Create an **AML model** of NewsFeed.
  5. Use AMP to **test** NewsFeed.

Mon 24-Mar
Tue 1-Apr

Study Material

*The **AML documentation** is also available from within **AMP**.*

- Tutorial Introduction to **AML.**
- **AML** Quick Reference Card.
- **AMP Laboratory**
  - "**roadmap**" descriptions of the AMP assignments: Coffee Machine, SmartDoor and NewsFeed.
  - **specifications** of **SmartDoor** and **NewsFeed**: blueprints for model and implementation.

\+ **slides** of the two lectures

Should be self contained (not available from within AMP).

Axini lectures/support

Mon 24-Mar	13.45-15.30	**Lecture #1**: introduction to AML
~~Thu 27-Mar~~	~~16.00-17.00~~	~~**Q&A** on SmartDoor (online, via Microsoft Teams!)~~
Mon 31-Mar	09.30-10.30	**Q&A** on SmartDoor (online, via Microsoft Teams!)
Tue 1-Apr	13.45-15.30	**Lecture #2**: more on AML, adapters
Thu 3-Apr	13.45-15.30	**Q&A** on NewsFeed, adapters, etc.

Proposal for replacement of Q&A of Thu 27 Mar:
Wed 26-Mar 16.00-17.00

Lecture 1

- Introduction to **Axini**
- Model-Based Testing (**MBT**), revisited
- Basic Axini Modeling Language (**AML**)
- Demo of Axini Modeling Platform (**AMP**): Coffee Machine
- Some more **AML**
- SmartDoor exercise
- Axini Plugin Adapters (**PA**)

*Probably an **early break** to hand-out usernames and accounts for AMP.*

axini

Introduction into Axini, MBT, AMP and AML

Theo Ruys

Axini

- start-up / **scale-up**:
  - 17+ years
  - 20+ employees
    - developers – Ruby on Rails
    - test architects / consultants
- product: Axini Modeling Platform (formerly AMS, TestManager)
  - based on conformance testing theory of Tretmans et.al.
- AMP is used to test
  - technical applications (e.g., client: ProRail, Thermo Fisher)
  - financial applications (e.g., client: Achmea)

But simply called '**Axini**' by our users.

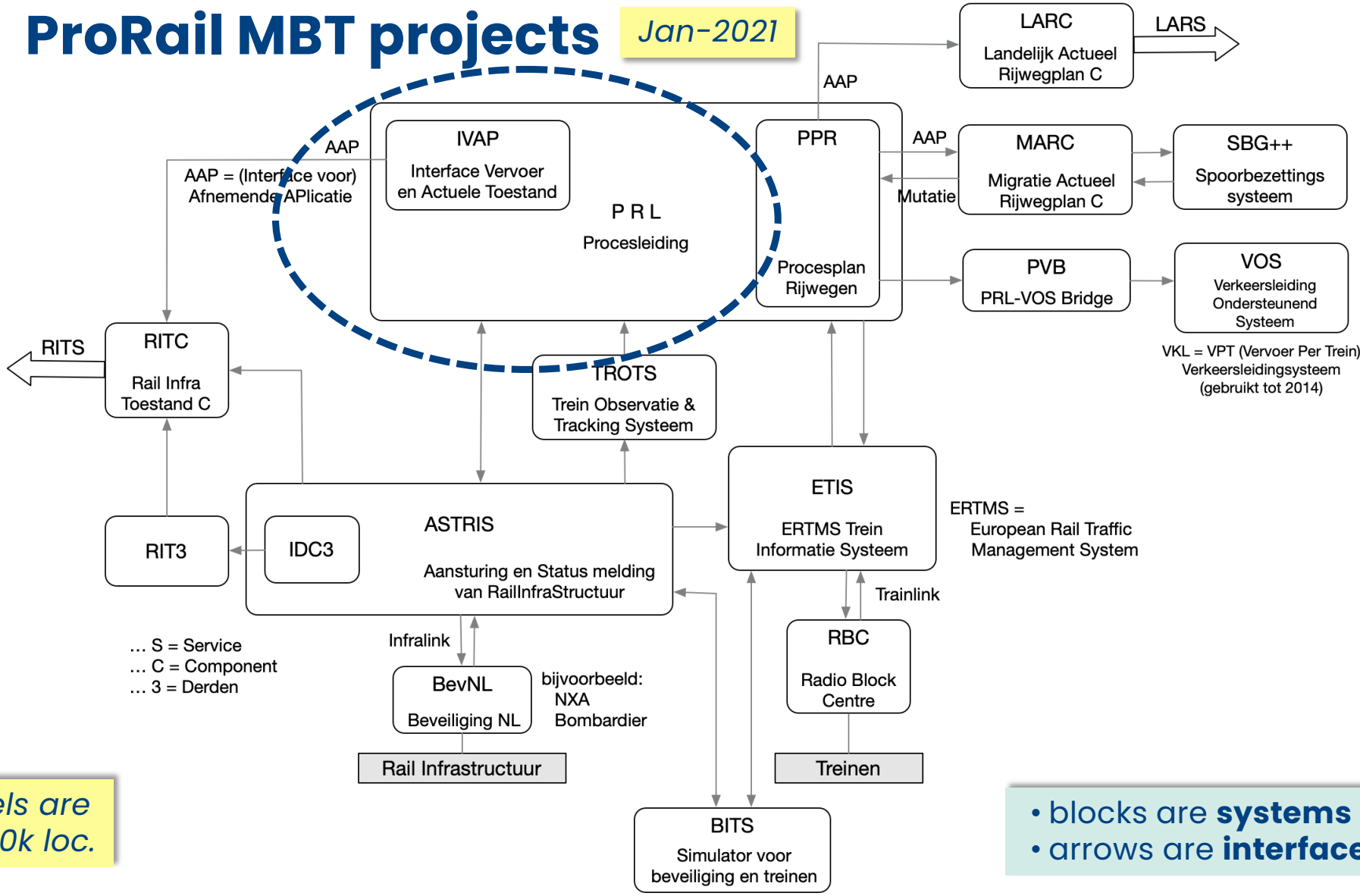
*Our **goal** is to help clients deliver:*
- *software products that **work** when it matters,*
- ***on time**, without concessions on quality, using our **unique testing platform**.*

Theo C. Ruys

- 1995 **MSc** FMT@UT on "Code Generation"
- 2001 **PhD** FMT@UT on "Model Checking"
- 2000-2010: Assistant Professor in **FMT@UT**
- 2015-now: **Axini**
  software engineer, consultant, teacher

ProRail MBT projects

Jan-2021



Some models are more than 10k loc.

- blocks are **systems**
- arrows are **interfaces**

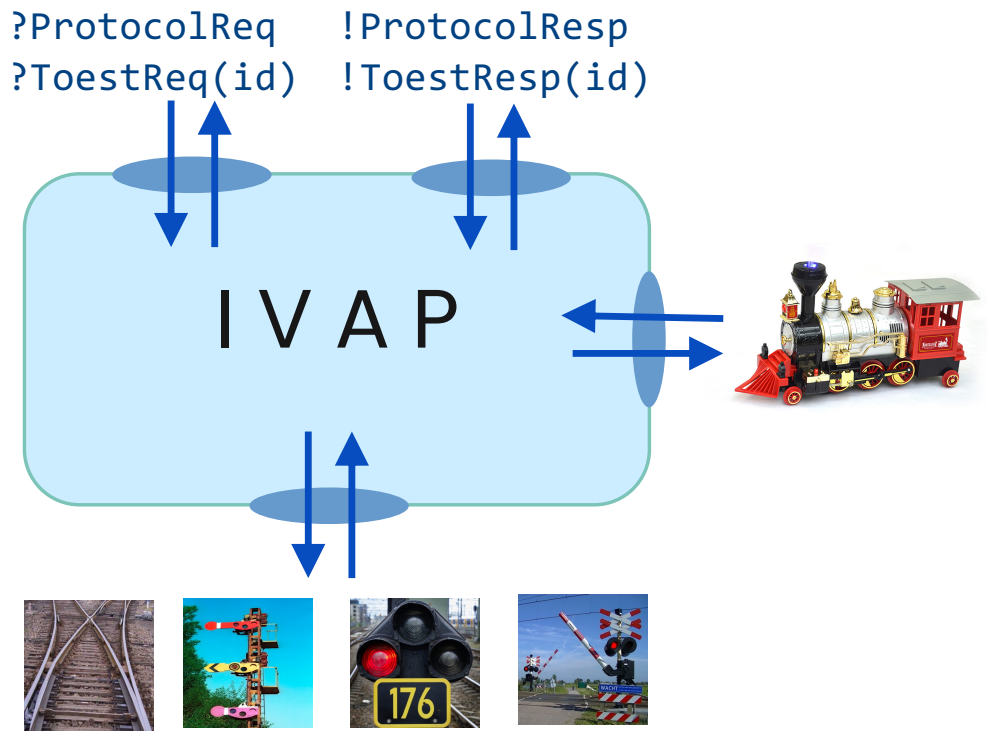
IVAP-AAP

IVAP = **I**nterface **V**ervoer en **A**ctuele Toestand voor **P**ost

AAP = (interface voor) **A**fnemende **Ap**plicatie

IVAP is a typical ProRail application:

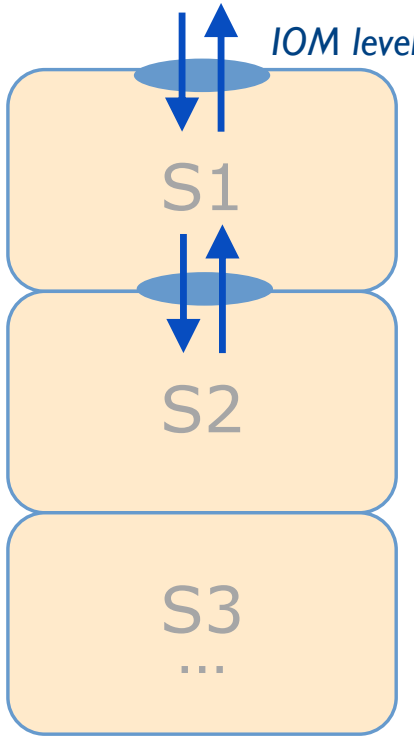
- **parallel system** with several interfaces
- message services: **EMS/JMS**
- **XML** messages



ThermoFisher: many subsystems

Incremental MBT

- One **large system**, consisting of **many subsystems**.
- Each **subsystem** is constructed and maintained by a team of 4-10 persons.
- A subsystem's **provided interface** can be used by other subsystems.
- On **integration time**, many **bugs** are found which could have been found during development.

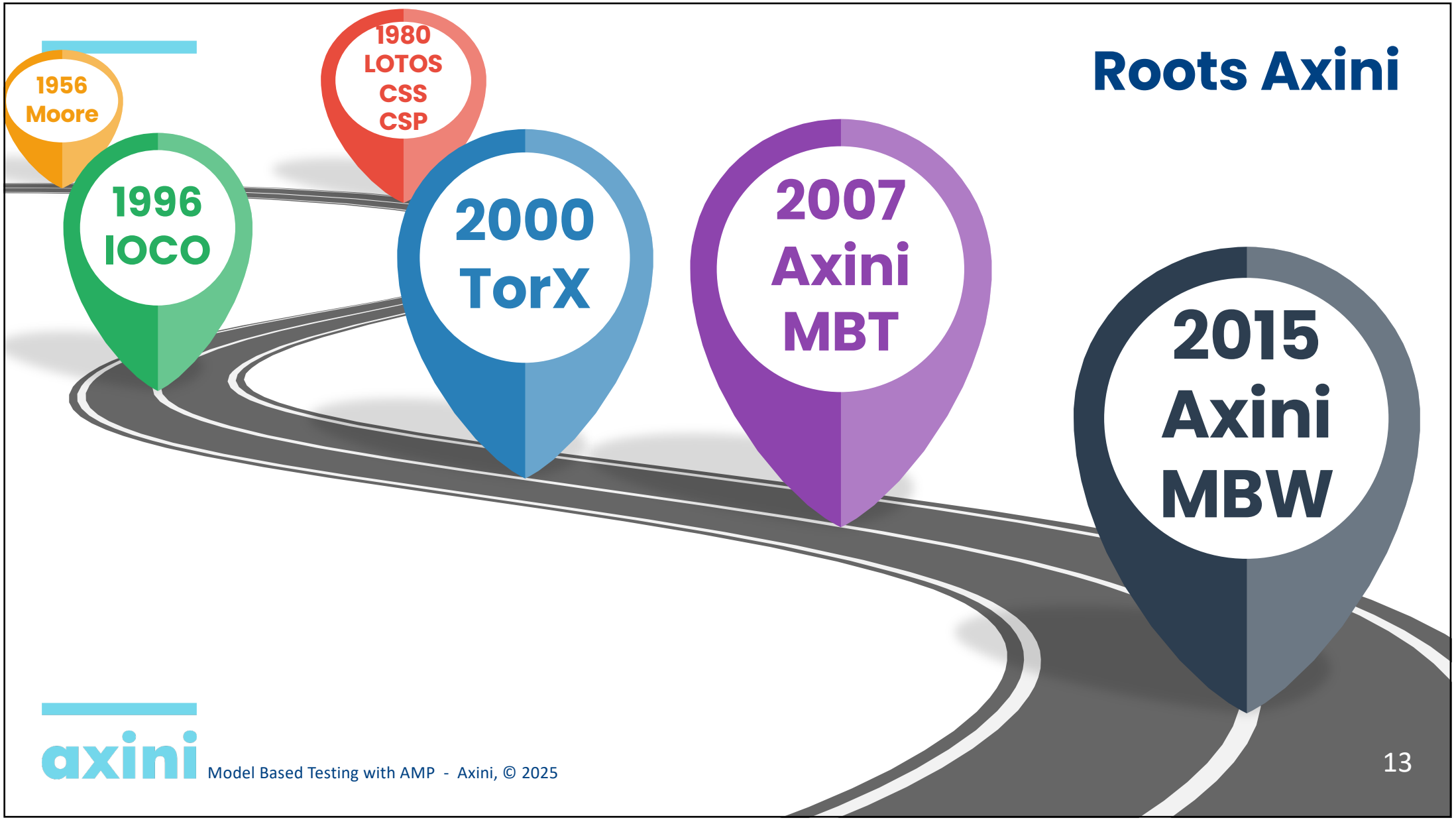


*S1 can use **different versions** of S2 (e.g., simulator, production)*

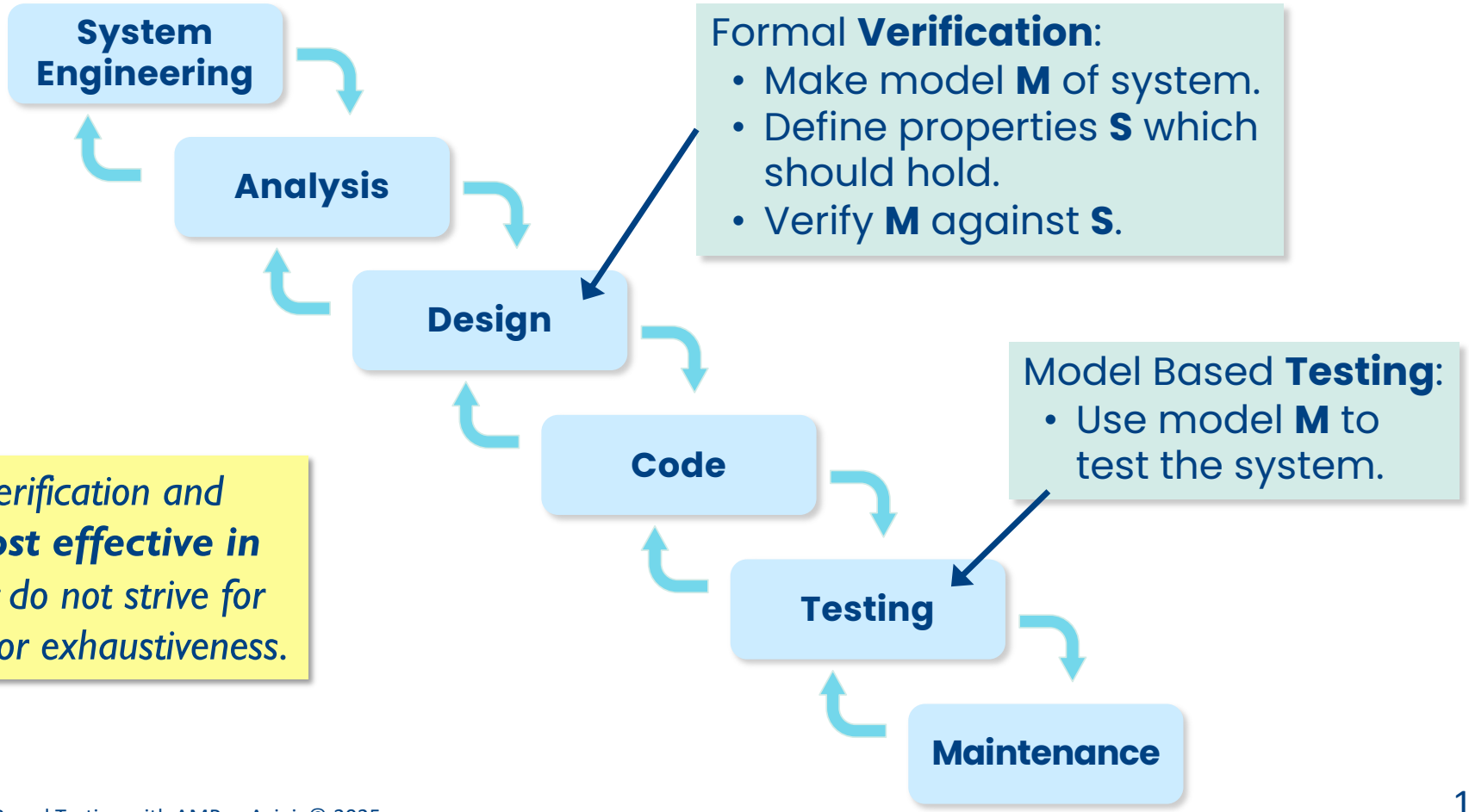
*S2 can have **other** clients than S1.*

A **method** is represented by **two labels**
- **start** of the method + **parameters**
- **end** of the method + **return value** of the method

Roots Axini


1956
Moore
1980
LOTOS
CSS
CSP
1996
IOCO
2000
TorX
2007
Axini
MBT
2015
Axini
MBW

Waterfall … and Formal Methods

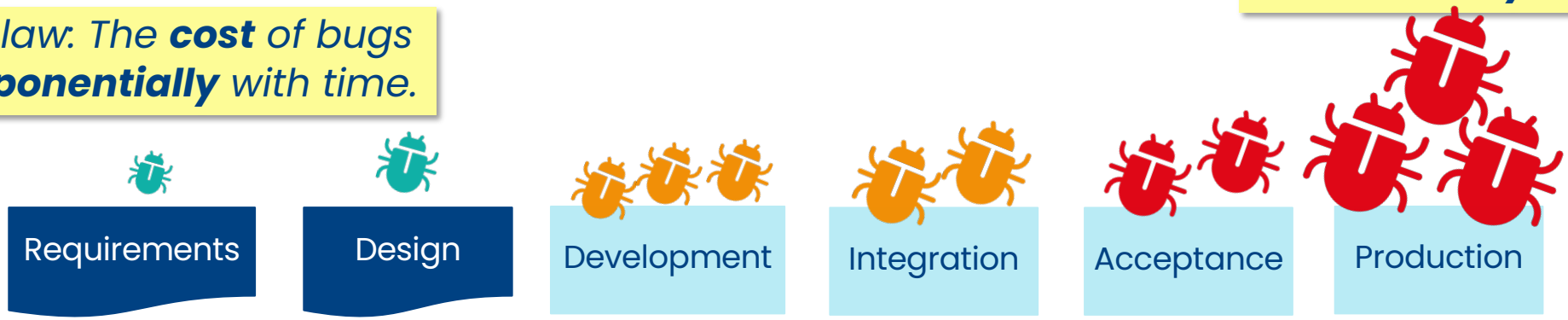


*Observation: verification and testing are **most effective in finding bugs**; do not strive for completeness or exhaustiveness.*

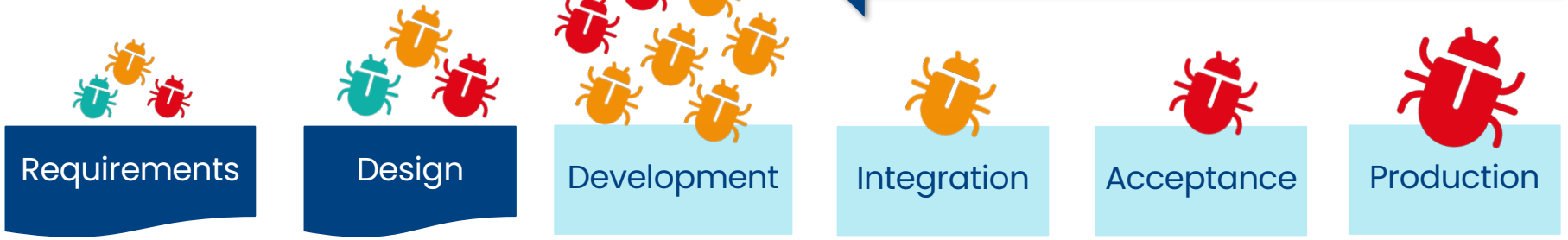
Shift Left: find bugs earlier

***Bigger** bugs are more **costly**.*

***Boehm**'s law: The **cost** of bugs grows **exponentially** with time.*



with MBT:



*First **models** can (and should!) be created at requirements- capturing time.*

Many different testing approaches

orthogonal and complementary

- unit testing
- test/behavior driven development (BDDs)
- input-output testing
- mutation testing
- classification testing
- interface testing
- integration testing
- non-functional testing
- regression testing
- user acceptance testing
- maintenance testing
- **model based testing**

*Except for MBT, for all these approaches the test cases have to be **written manually**. With **MBT, test cases** are automatically **generated**.*

Best choice for testing:
- reactive, control-oriented systems
- several interfaces
- multi-threaded, parallel processes

Black-box testing: we do not have the source of the System under Test (SUT)!

axini

Model Based Testing

is an **automated** technical process,

... performed by **automatically** executing/experimenting

... with a **software** system, in a **controlled environment**,

... following prescribed behavior of a **formal model**,

... with the intent of **measuring** one or more **characteristics** or the **quality** of the product

... by demonstrating the **deviation** of the actual status of the product from the required status/**specification**.

*The 'Model' is an **abstract, high-level** and **formal** description of the SUT.*

Test tool comparison


Manual step


Automated

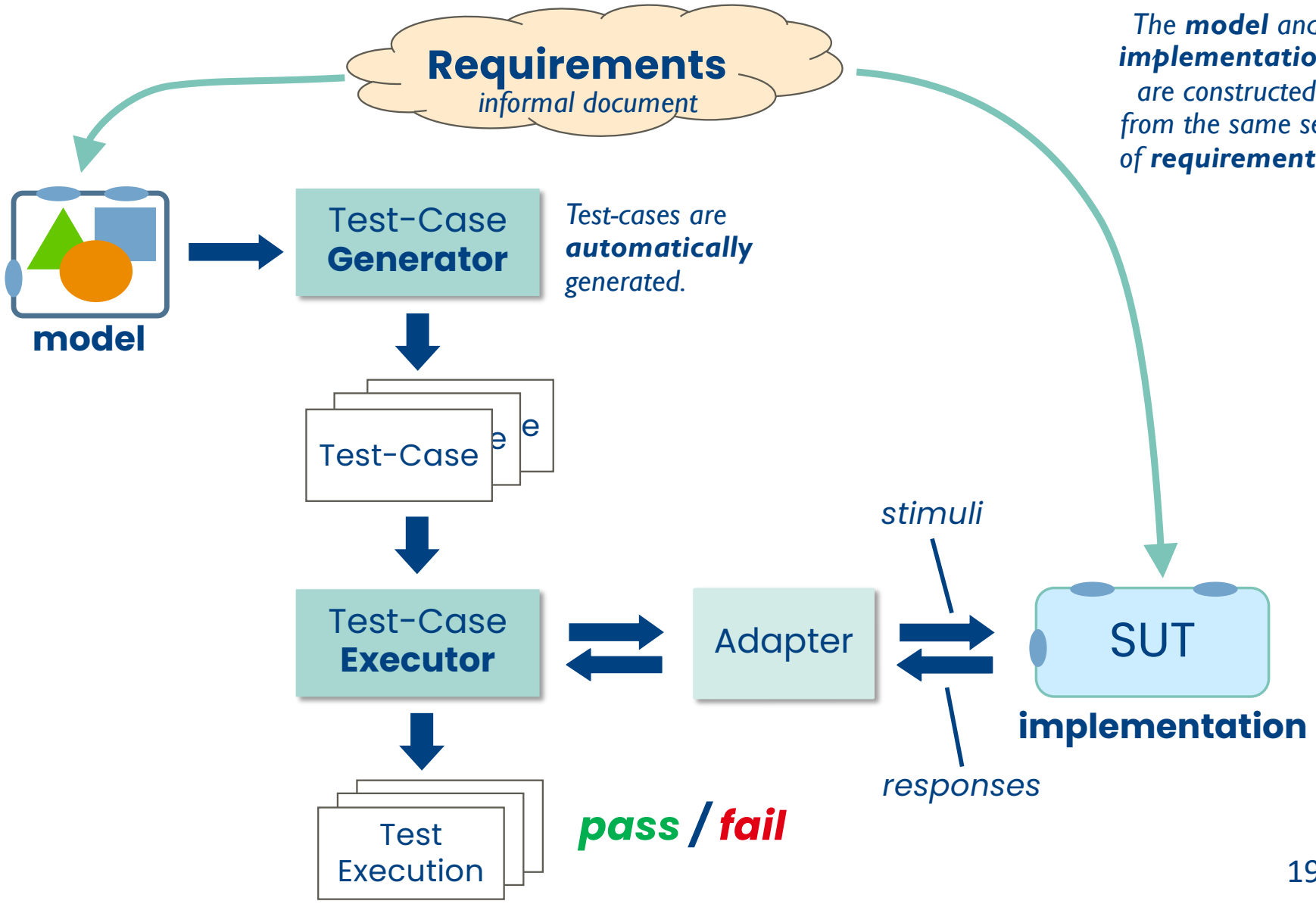
	Process	Manual	BDD	Axini
Design	Make specification			
	Make model			
Test	Make test			
	Predict outcome			
	Script test			
	Execute test			
	Evaluate outcome			

more coverage = more certainty

100% auto-mation

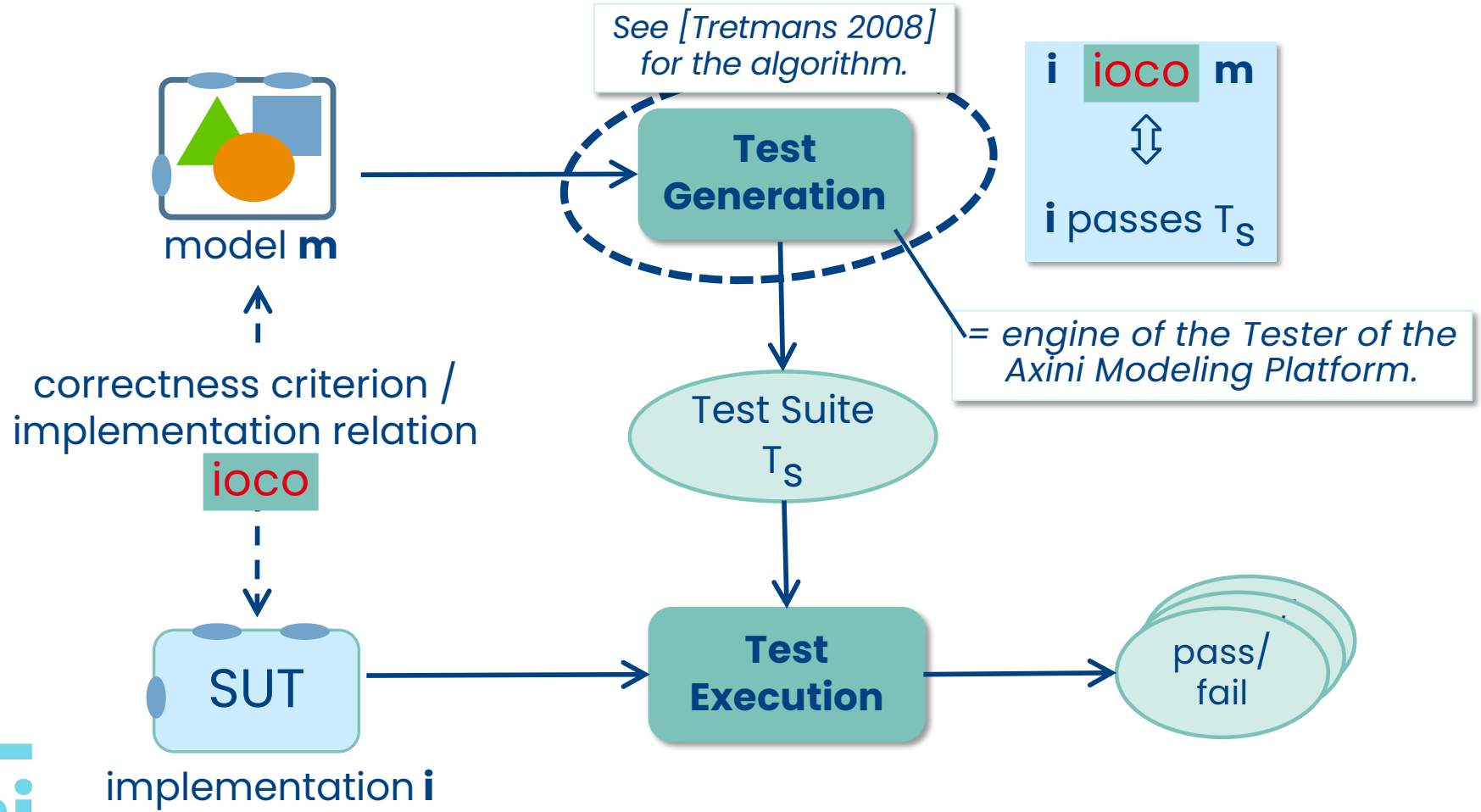
axini

Model Based Testing



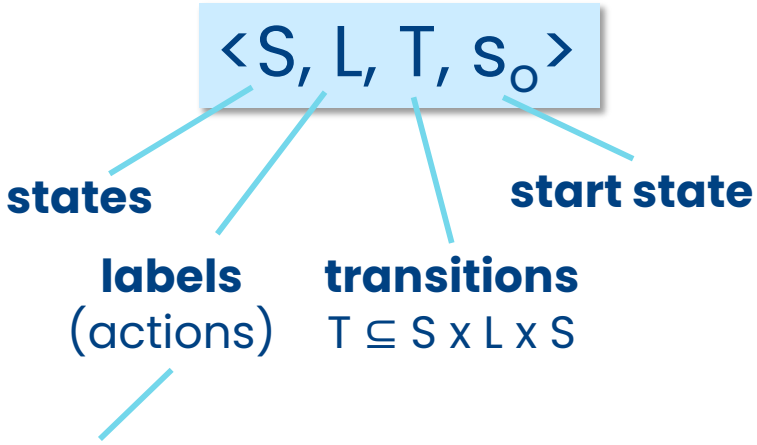
axini

Formal Conformance Testing

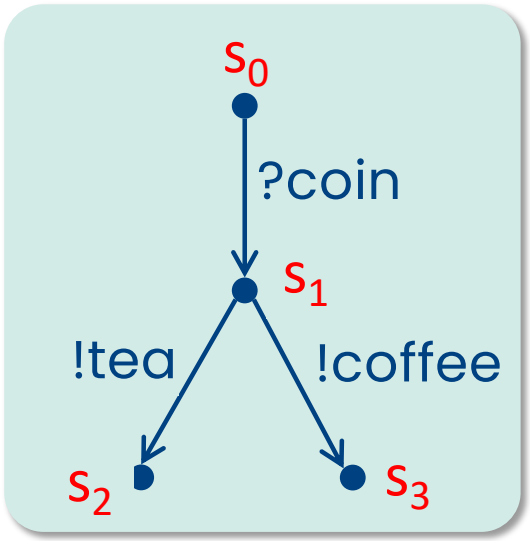


axini

Labeled Transition System (LTS)



actions:
- observable
  - stimulus (?) ∈ L_I
  - response (!) ∈ L_O
- internal action: τ



trace σ: a sequence of labels,
e.g., ⟨?coin, !coffee⟩
(often written as ?coin!coffee)

ioco

≈ *any output of* **i** *has been foreseen by* **m**

Consequently, a ***failed test case*** *in AMP is caused by a* ***response*** *by the SUT (or* ***quiescence****) which was* ***not expected*** *by the model.*

out(s) = {λ ∈ L_O ∪ {δ} | s –λ→ }
s **after** σ = {s' | s =σ⇒ s'}
Straces(s) = {σ ∈ L_δ^* | s =σ⇒ s'}

- **ioco** = **i**nput-**o**utput **co**nformance
  - model m is an LTS
  - assumption: implementation **i** can be represented by an IOTS (i.e., always input enabled)
- **i** ioco-conforms to **m**, iff
  - if **i** *produces output* ***x*** after trace **σ**, then **m** ***can*** *produce* ***x*** after **σ**
  - if **i** ***cannot*** *produce* any output after trace **σ**, then **m** ***cannot*** *produce* any output after **σ**

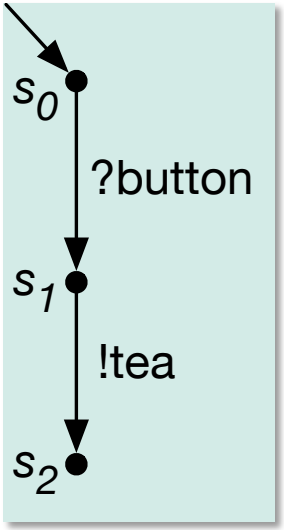
δ

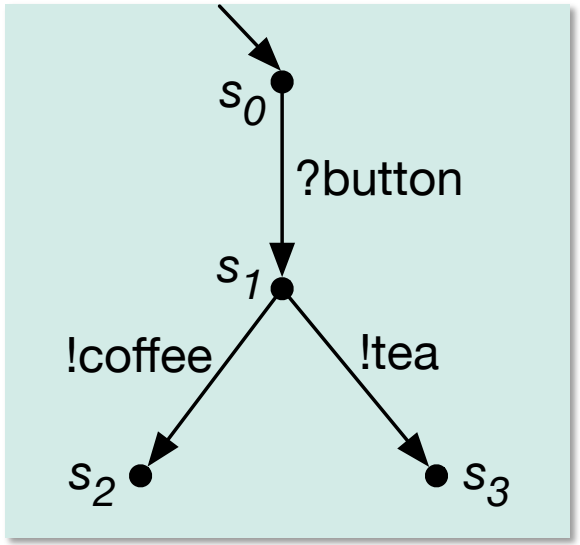
- i **ioco** m =$_{def}$ ∀σ ∈ **Straces**(m): out(i **after** σ) ⊆ out(m **after** σ)

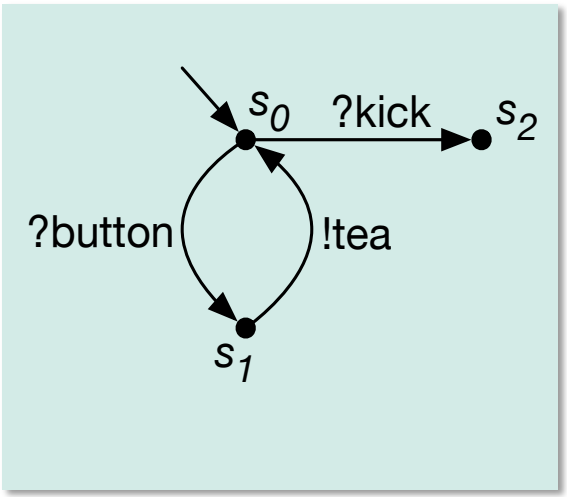
axini

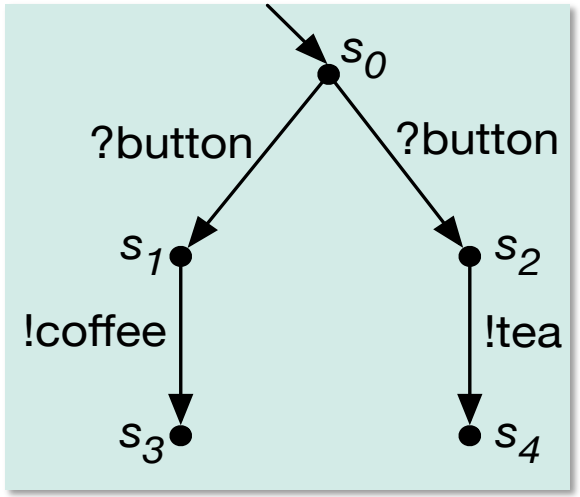
Labeled Transition Systems

labels *are external* ***actions****, events*









Small vocabulary, but ***precise****!*

axini

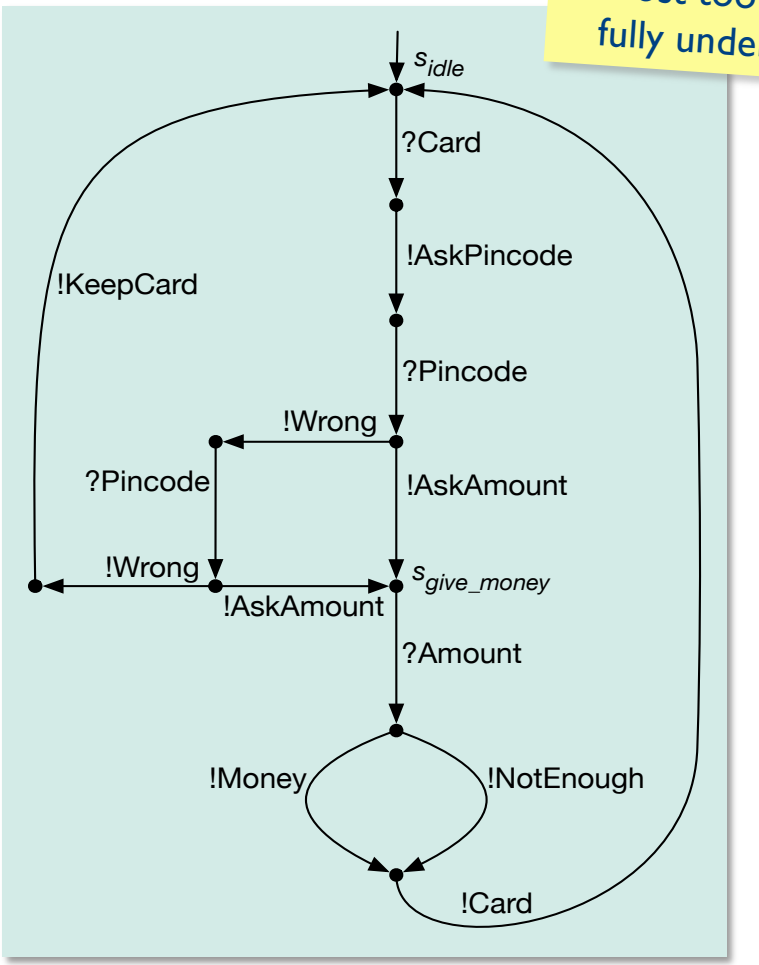
Coverage

As the SUT is a **black box**, the **coverage criteria** for MBT are only on the **model**.

Used (and reported) by AMP.

- **state** coverage: percentage of states of the LTS
- **transition** coverage: percentage of transitions of the LTS
- **path** coverage: subpaths of length n
- **data** coverage: percentage of data values seen

Almost too big to be fully understood.



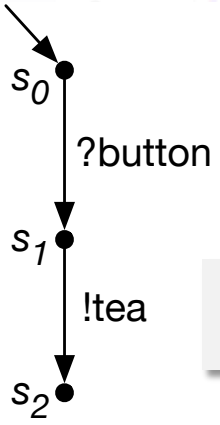
*Sometimes, the SUT can provide **statement** and/or **branch coverage** after the test has finished.*

AMP: Axini Modeling Platform

*LTS is the **model**... but not very convenient for writing large models.*

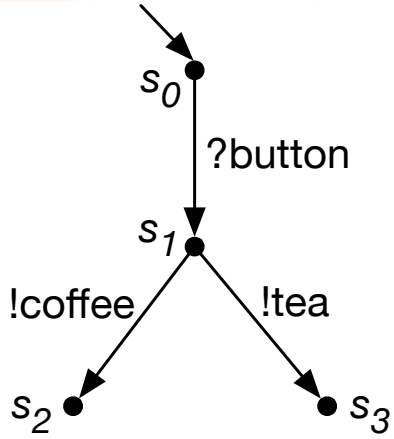
- **AMP**: tool to edit, explore, visualize, test AML **models**
  - **Test-cases** are automatically generated. *Technically, these are **traces**.*
- Models are written in Axini Modeling Language (**AML**)
  - **behavior**: mapped upon **labeled transition systems (LTSs)**
  - **data**: modern, powerful, static typed 'programming' language

Examples of LTSs



AML

```
receive 'button'
send 'tea'
```



AML

```
receive 'button'
choice {
  o { send 'coffee' }
  o { send 'tea'    }
}
```

? *= input (stimulus)*
! *= output (response)*

Axini Modeling Language (AML)

- process + data language
  - inspired by **Promela** (SPIN) and **LOTOS**
- model consists of **parallel processes**
- communication over **hand-shake channels**
  - *external*: communication with SUT
  - *internal*: communication between processes

Largest AML model: >10k loc.

behavioral part:
- **stimuli** (inputs)
- **responses** (outputs)
- (non-deterministic) **choice**
- **repeat**
- states / **goto**

first things first

data part:
- Ruby-like, **strictly typed**
- messages can be **received** and **send**
  - **label** (name)
  - **parameters** (attributes)
- process can have **variables**

Semantics

A process in AML is mapped upon a (symbolic) labelled transition system.

*AML is implemented as a **Ruby DSL**: Ruby can be used as preprocessor.*

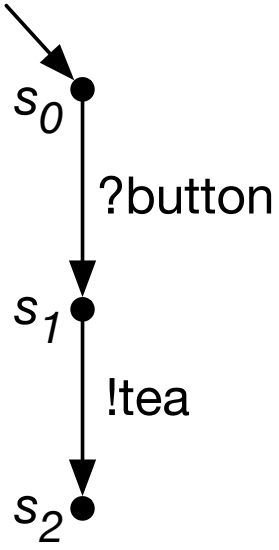
axini

Hello, Tea Machine

```
# Hello Tea: our first AML model!
external 'machine'
process('hello-tea') {
  # declarations of labels, variables
  timeout 10.0

  stimulus 'button', on: 'machine'
  response 'tea', on: 'machine'

  # behavior of the process
  receive 'button'
  send 'tea'
}
```

name of the external channel

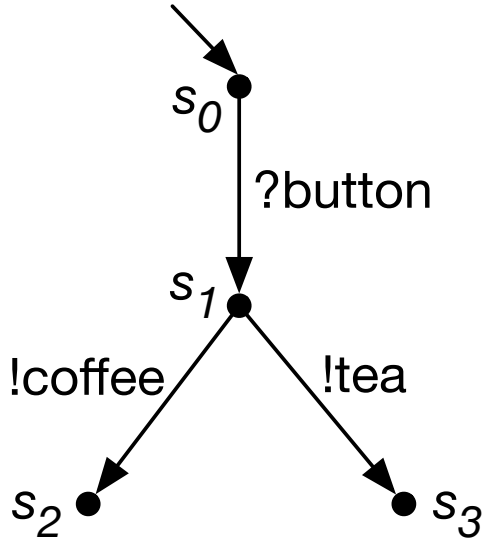
timeout for all responses



choice

```
external 'extern'
process('tea-or-coffee') {
  # declarations of labels, variables
  timeout 10.0
  channel('extern') {
    stimulus 'button'
    responses 'tea', 'coffee'
  }

  # behavior of the process
  receive 'button'
  choice {
    o { send 'tea' }
    o { send 'coffee' }
  }
}
```

declarations keep (more-or-less) the same

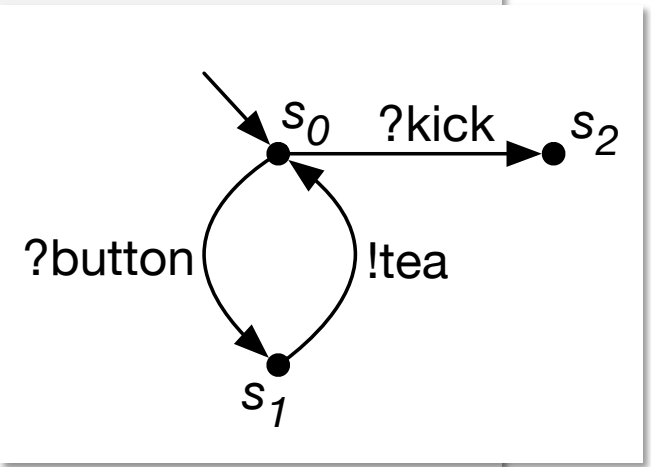


axini

states and goto

convention: states start in column 1

```
external 'extern'
process('tea-or-kick-goto') {
  timeout 10.0
  channel('extern') {
    stimuli 'button', 'kick'
    responses 'tea'
  }

state 's0'
  choice {
    o { receive 'kick' }
    o { receive 'button'; send 'tea'; goto 's0' }
  }
}
```

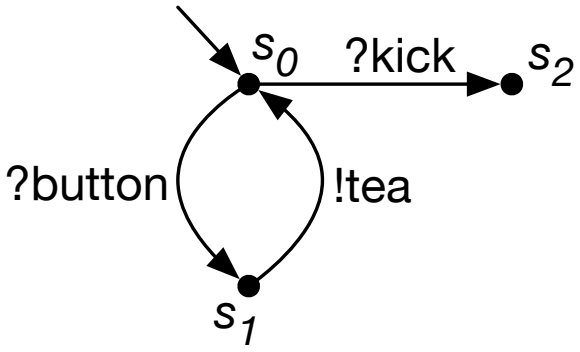


*In programming languages the **goto** statement might be not-done. In modelling languages (for reactive systems) it is a powerful mechanism to model **state-transition systems**, such as **protocols**.*

axini

repeat

```
external 'extern'
process('tea-or-kick-repeat') {
  timeout 10.0
  channel('extern') {
    stimuli 'button', 'kick'
    responses 'tea'
  }

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



break out of the loop

axini

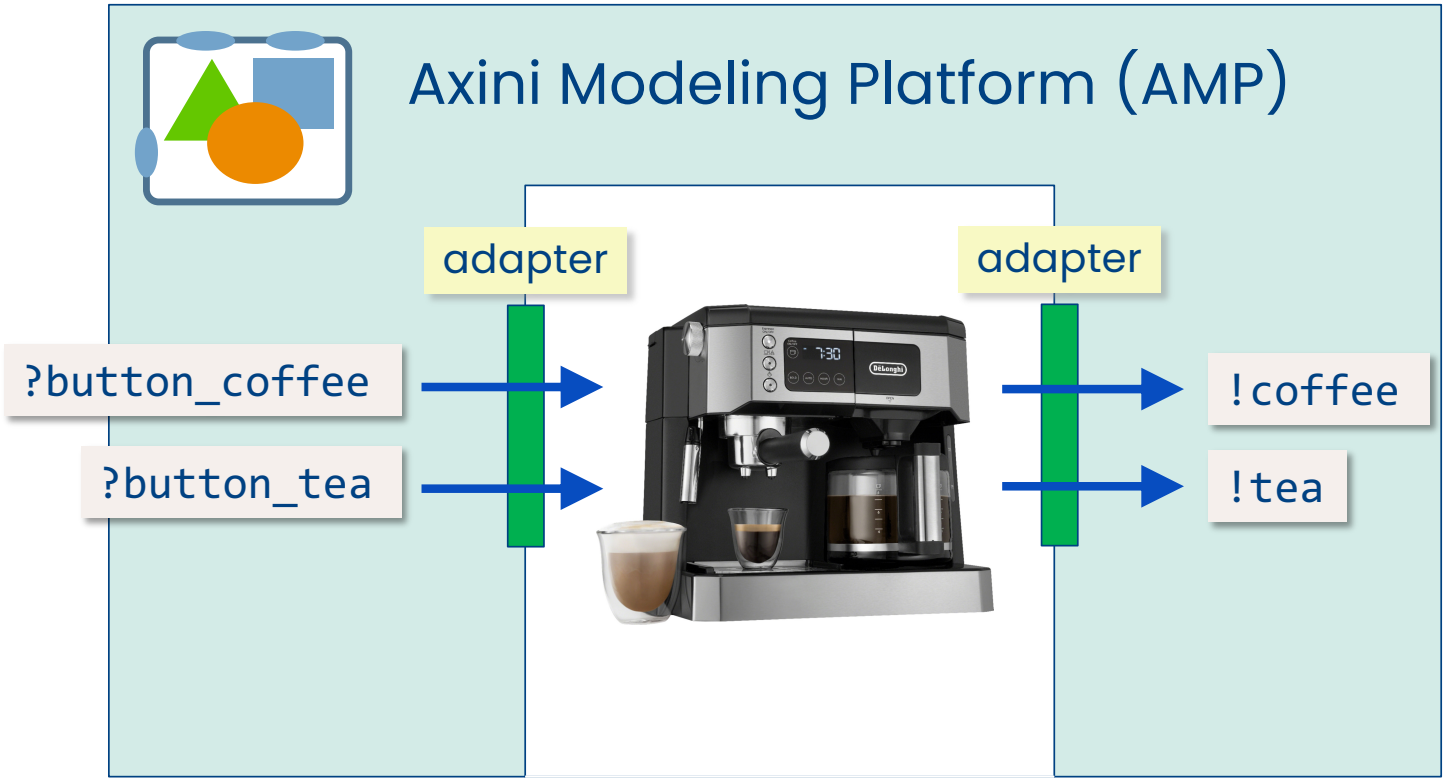
Coffee Machine

Theo Ruys

System Under Test

Challenge: **model** and **test** that the beverage machine delivers **coffee**, **tea**, and … **lemonade**.

AMP: horse shoe around SUT



AMP + model + adapters
'**simulate**' the **environment** of the SUT.

Laboratory: Coffee Machine

*Let's do it **together** to get **introduced** to **AMP**.*

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 the model you created.

1.1 Exploring AMP

When you log in for the first time, you will arrive at the test page with th
no test runs yet'.

Your environment should contain multiple projects to work in. Select th
using the navigation bar at the top of the screen. In this navigation bar,
set, which are used to manage different testing configurations. The Coffe
only a single test set 'Coffee Machine'.

We will start by exploring the main features of AMP.

1.1.1 Model

Press the 'Model' button in the sidebar 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 model 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?

1.1.2 Visualize

Press the 'Visualize' button. This will open the visualization page in a new window in your browser. In this new window, the STS of the model is shown. Can you relate the visualized STS to the textual model?

Follow **instructions**:

1.1 Exploring AMP

1.2 Extending the model

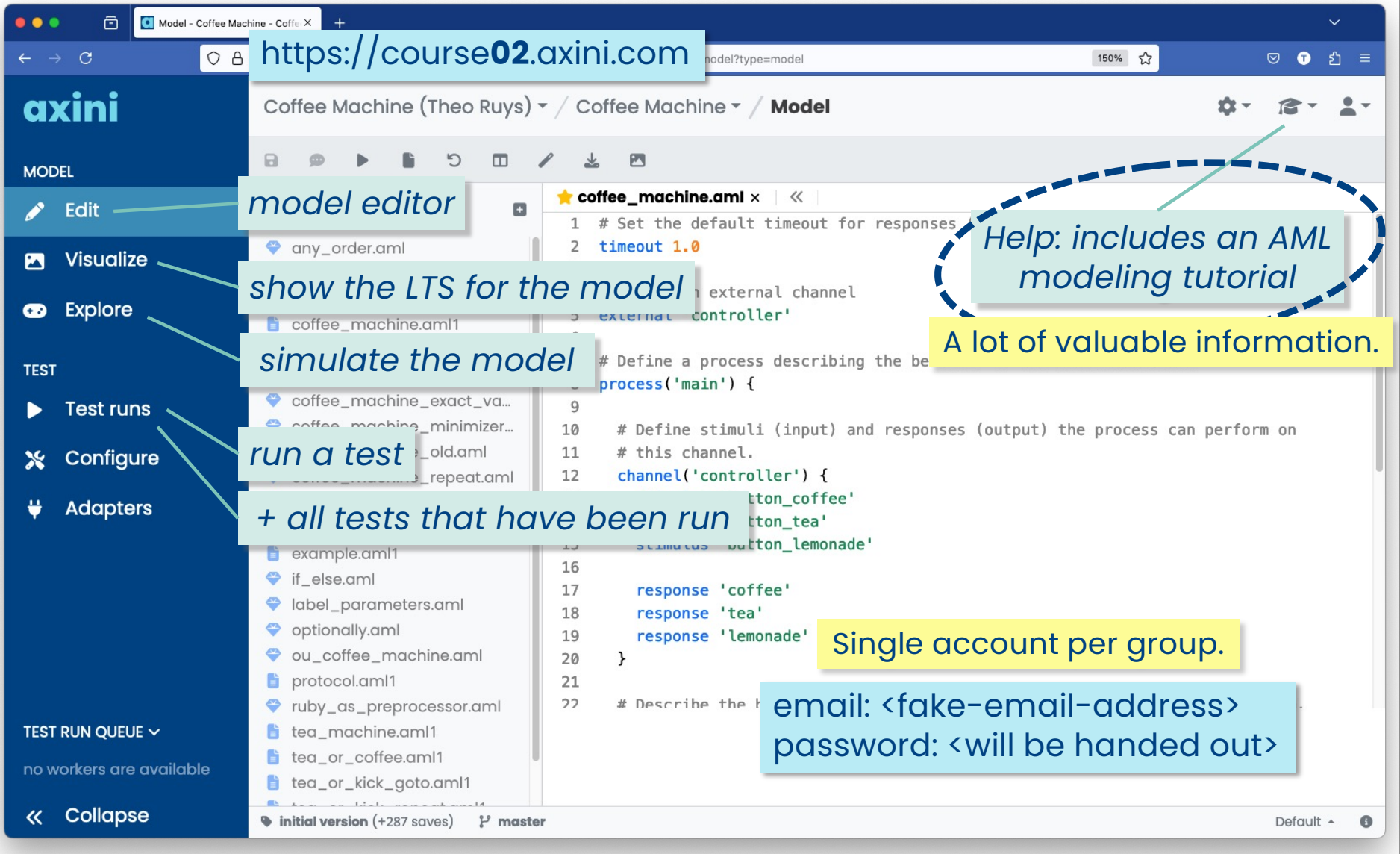
1.3 Testing

1.4 Lemonade?!

*A **non-typical** exercise: we do **not** have **requirements** of the SUT!*

What is the **'exact' behavior** of the **SUT**?

AMP


https://course02.axini.com
model editor
show the LTS for the model
simulate the model
run a test
+ all tests that have been run
Help: includes an AML modeling tutorial
A lot of valuable information.
Single account per group.
email: <fake-email-address>
password: <will be handed out>
axini
Coffee Machine (Theo Ruys) / Coffee Machine / Model
MODEL
Edit
Visualize
Explore
TEST
Test runs
Configure
Adapters
TEST RUN QUEUE
no workers are available
Collapse
coffee_machine.aml
initial version (+287 saves)
master
Default

axini

Test Sets: working together on same account

> ***Test Sets*** *can be seen as* ***different views*** *on same the project. For example: Master model, development branches, Bad Weather, Replaying bugs, etc.*

- **Projects** for each UT group:
  - `Coffee Machine`
  - `SmartDoor`
  - `SmartDoor PA`
  - `NewsFeed PA`
- A project can have **multiple Test sets.**
- For each **Test set** you have to modify:
  - **root model part**: the "main" AML file
  - **configuration** (number of test cases, steps, strategy, etc.)
  - **adapter** (for PA projects)
- **Git** can also be used (versions and branches), but is not needed.

`Axini Piet`

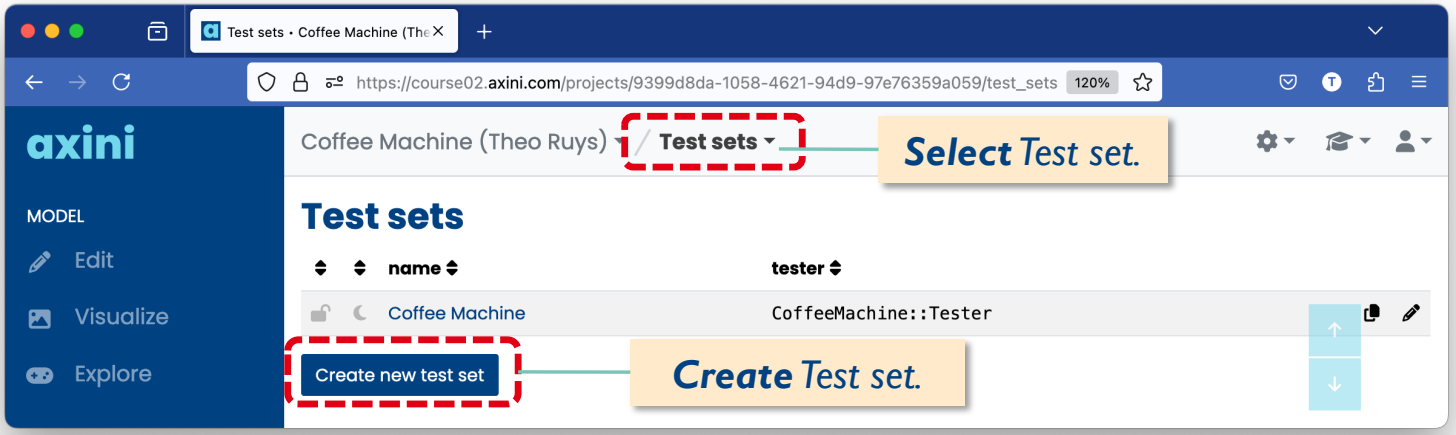
`smartdoor-piet.aml`

`Axini Hein`

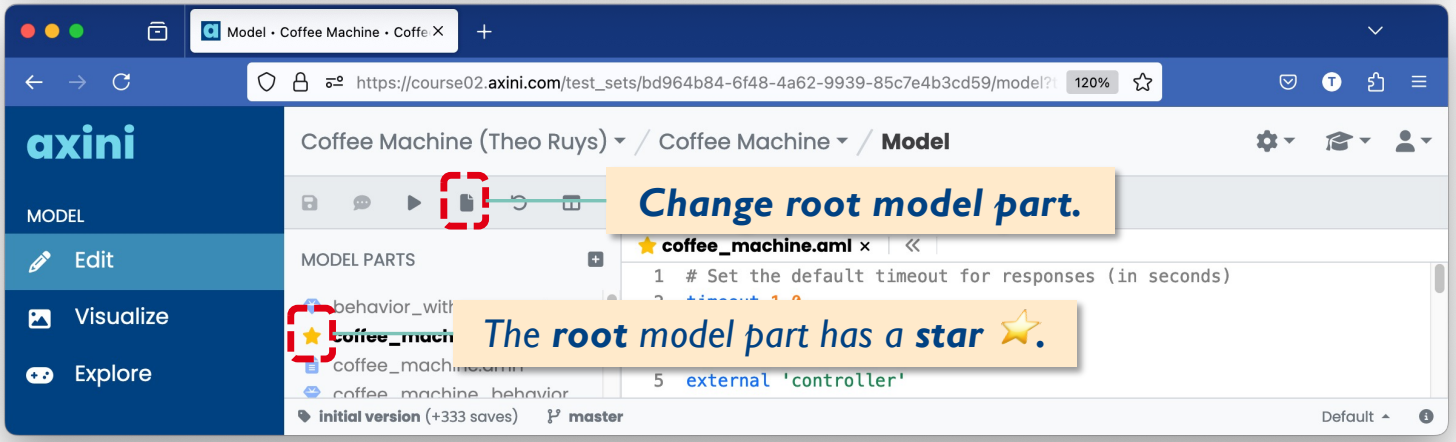
`smartdoor-hein.aml`

Test Sets: working together on same account

1 Create or Select **Test set.**



2 Set the **root model part** for the Test set.



*You can also **right-click** on a file to make it the **root model part** of the current Test Set.*

*+ (optionally) set the **adapter***
*+ (optionally) change the **configuration***

axini

Coffee Machine

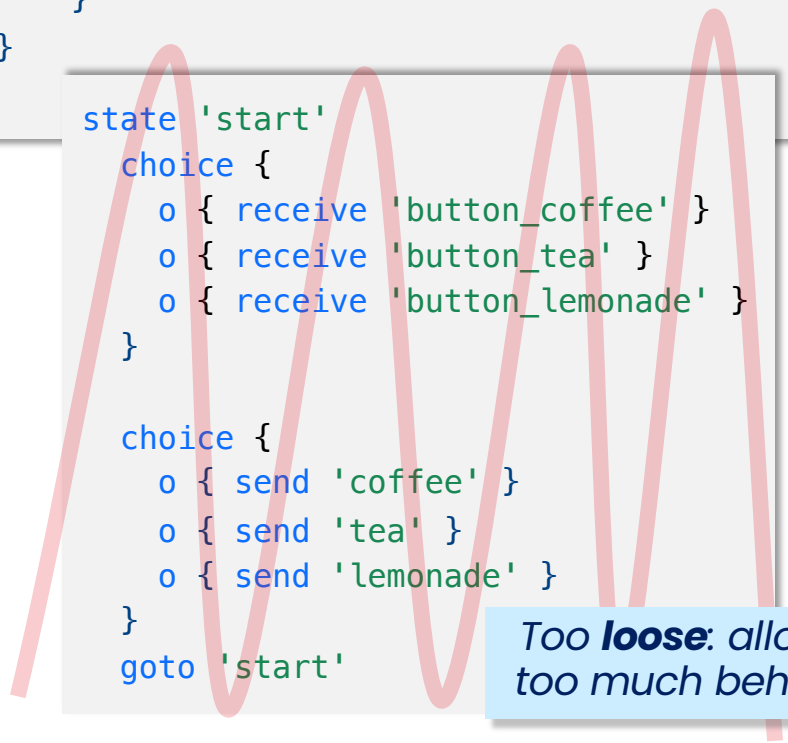
```
state 'start'
  choice {
    o { receive 'button_coffee'   ; goto 'coffee'   }
    o { receive 'button_tea'      ; goto 'tea'      }
    o { receive 'button_lemonade' ; goto 'lemonade' }
  }

state 'coffee'
  send 'coffee'
  goto 'start'

state 'tea'
  send 'tea'
  goto 'start'

state 'lemonade'
  choice {
    o { send 'lemonade' }
    o { send 'coffee' }
    o { send 'tea' }
  }
  goto 'start'
```

After ?button_lemonade, we observe either !lemonade, !coffee, or !tea.

Using repeat instead of states/goto.

```
repeat {
  o { receive 'button_coffee'; send 'coffee' }
  o { receive 'button_tea';    send 'tea'    }
  o { receive 'button_lemonade'
      choice {
        o { send 'coffee' }
        o { send 'tea' }
        o { send 'lemonade' }
      }
  }
}
```

```
state 'start'
  choice {
    o { receive 'button_coffee' }
    o { receive 'button_tea' }
    o { receive 'button_lemonade' }
  }

  choice {
    o { send 'coffee' }
    o { send 'tea' }
    o { send 'lemonade' }
  }
  goto 'start'
```



*Too **loose**: allowing too much behavior.*

Coffee Machine (exact?)

Observation: after ?button_lemonade, the last given beverage is observed again.

```
state 'start'
  choice {
    o { receive 'button_coffee' ;   send 'coffee'    ; goto 'coffee_last' }
    o { receive 'button_tea' ;      send 'tea'       ; goto 'tea_last' }
    o { receive 'button_lemonade' ; send 'lemonade'  ; goto 'start' }
  }

state 'coffee_last'
  choice {
    o { receive 'button_coffee' ;   send 'coffee'    ; goto 'coffee_last' }
    o { receive 'button_tea'    ;   send 'tea'       ; goto 'tea_last' }
    o { receive 'button_lemonade' ; send 'coffee'    ; goto 'coffee_last' }
  }

state 'tea_last'
  choice {
    o { receive 'button_coffee' ;   send 'coffee'    ; goto 'coffee_last' }
    o { receive 'button_tea'    ;   send 'tea'       ; goto 'tea_last' }
    o { receive 'button_lemonade' ; send 'tea'       ; goto 'tea_last' }
  }
```

Only at the start, !lemonade is observed.

axini

Coffee Machine (exact?)

*Using a **state variable**, which remembers the last beverage.*

```
var 'last', :string, ''

repeat {
  o { receive 'button_coffee'; send 'coffee', update: "last = 'coffee'" }
  o { receive 'button_tea';    send 'tea',    update: "last = 'tea'"    }
  o {
    receive 'button_lemonade'
    choice {
      o { send 'lemonade', constraint: "last == ''"       }
      o { send 'tea',      constraint: "last == 'tea'"    }
      o { send 'coffee',   constraint: "last == 'coffee'" }
    }
  }
}
```

*We also have to use the **update** and **constraint** options of a label here.*

*Final note: the coffee machine is the **only** AML model where we have **different labels on the same line**, separated by a **semicolon**.*

axini

Some more AML

Theo Ruys

non-terminating behavior ≈ state

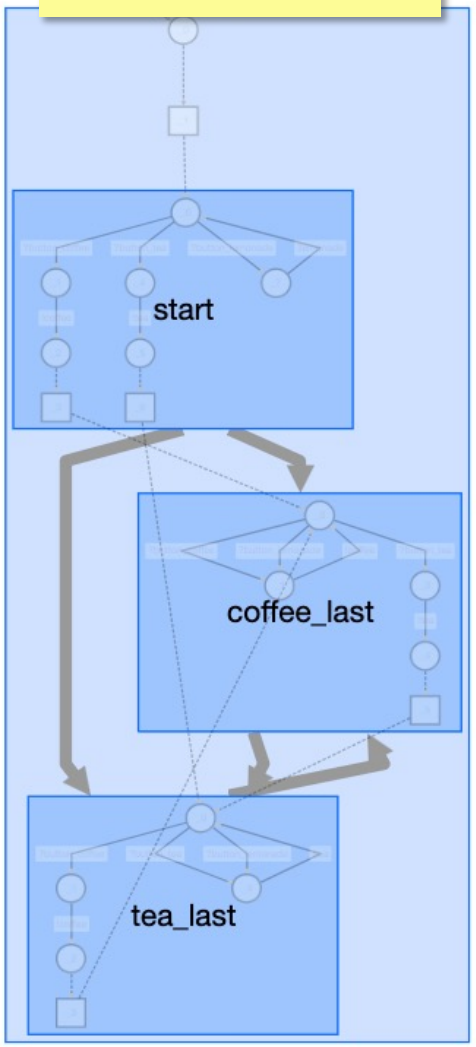
```
behavior('start', :non_terminating) {
  choice {
    o { receive 'button_coffee' ;   send 'coffee'   ; behave_as 'coffee_last' }
    o { receive 'button_tea' ;      send 'tea'      ; behave_as 'tea_last' }
    o { receive 'button_lemonade' ; send 'lemonade' ; behave_as 'start' }
  }
}

behavior('coffee_last', :non_terminating) {
  choice {
    o { receive 'button_coffee' ;   send 'coffee'   ; behave_as 'coffee_last'
    o { receive 'button_tea'    ;   send 'tea'      ; behave_as 'tea_last'
    o { receive 'button_lemonade' ; send 'coffee'   ; behave_as 'coffee_las
  }
}

behavior('tea_last', :non_terminating) {
  choice {
    o { receive 'button_coffee' ;   send 'coffee'   ; behave_as 'coffee_last' }
    o { receive 'button_tea'    ;   send 'tea'      ; behave_as 'tea_last' }
    o { receive 'button_lemonade' ; send 'tea'      ; behave_as 'tea_last' }
  }
}

behave_as 'start'
```

behave_as ≈ goto

***Visualization** of non-terminating behaviors is nicer than states.*

*By some of our clients, **non-terminating behaviors** are **preferred** over states/goto.*

*Behaviors get their own **'blocks'**.*



Axini Modeling Language (AML)

- process + data language
  - inspired by **Promela** (SPIN) and **LOTOS**
- model consist of **parallel processes**
- communication over **hand-shake channels**
  - *external*: communication with SUT
  - *internal*: communication between processes

Largest AML model: >10k loc.

behavioral part:
- **stimuli** (inputs)
- **responses** (outputs)
- (non-deterministic) **choice**
- **repeat**
- states / **goto**

data part:
- Ruby-like, **strictly typed**
- messages can be **received** and **send**
  - **label** (name)
  - **parameters** (attributes)
- process can have **variables**

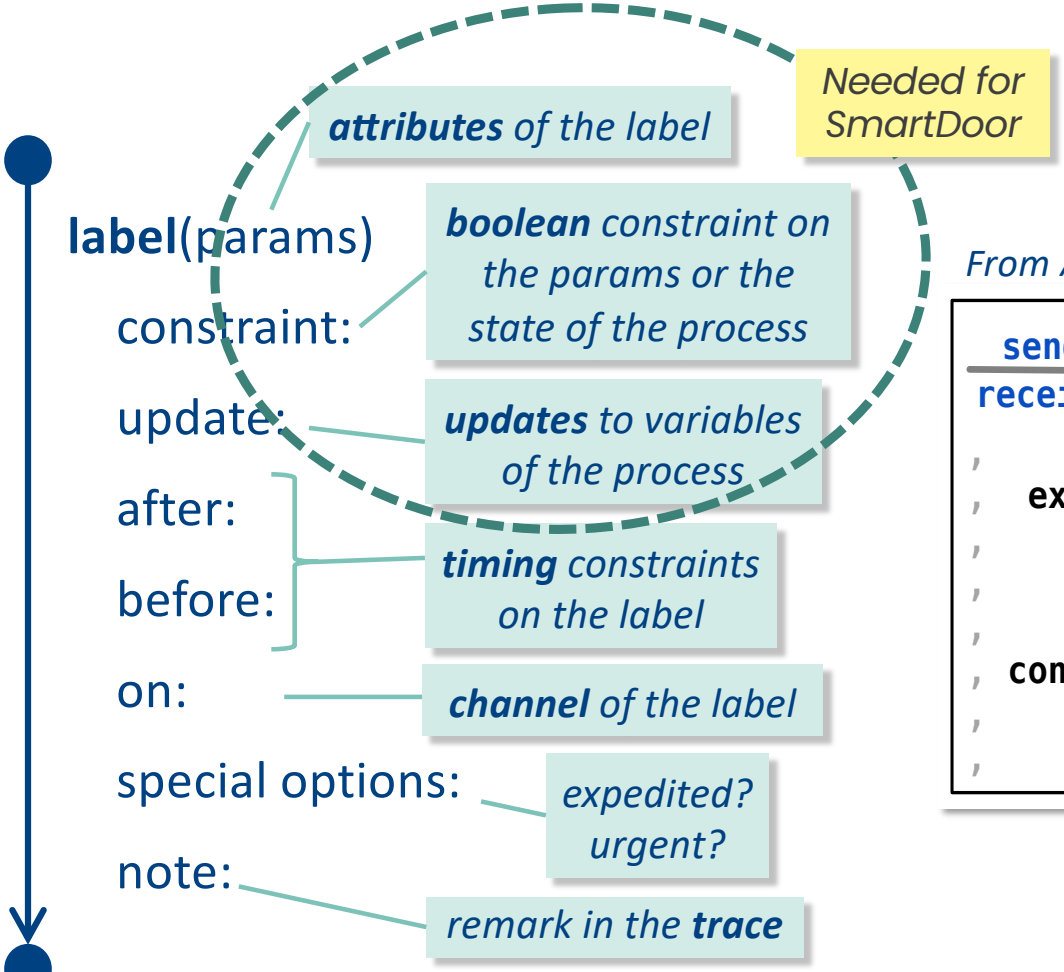
Semantics

A process in AML is mapped upon a (symbolic) labelled transition system.

*AML is implemented as a **Ruby DSL**: Ruby can be used as preprocessor.*

Label in AML

*A label is **more** than an **observable action**.*



From AML Quick Reference Card:

all label options are optional

```
send
receive   label_name
,          on: channel_name
,  expedited: <boolean>
,     urgent: <boolean>
,      after: <time_expr> | <decimal_expr>
,     before: <time_expr> | <decimal_expr>
, constraint: <bool_expr>
,     update: <assignment_expr>
,       note: <string> | [ <string>* ]
```

axini

Label parameters

*For a label to be **enabled**, its **constraint** has to evaluate to **true**.*

```
external 'extern'
process('coin-tea-parameters') {
  timeout 10.0
  channel('extern') {
    stimulus 'coin', {'value'  => :integer}
    response 'tea',  {'volume' => :integer}
  }

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

*parameters have a **name** and **type***

***creates** stimulus with specific parameter*

***checks** parameters of the response*

AMP will generate values for the label parameters such that the constraint is true.

axini

State variables

*Often called "**process**" variables.*

```
external 'extern'
process('coin-tea-total') {
  timeout 10.0
  channel('extern') {
    stimulus 'coin',     {'value'  => :integer}
    stimulus 'stop'
    response 'tea',      {'volume' => :integer}
    response 'display',  {'number' => :integer}
  }
  var 'total', :integer, 0

  repeat {
    o {
      receive 'coin', constraint: 'value  == 50',
                         update: 'total = total + value'
      send 'tea',     constraint: 'volume == 200'
    }
    o {
      receive 'coin', constraint: 'value  == 100',
                         update: 'total = total + value'
      send 'tea',     constraint: 'volume == 500'
    }
    o { receive 'stop'; stop_repetition }
  }
  send 'display', constraint: 'number == total'
}
```

state variable

update of state variables

axini

variables & types

*All **identifiers** in AML are **strings** (between **single** or **double** quotes).*

- **label parameters**

curly brackets are optional

```
stimulus '<name>', { '<name>' => <type>, ... }
response '<name>', { '<name>' => <type>, ... }
```

- **state variables** – local to a process

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

*The initial value is **optional**. If not provided, the value will be **:void** (equivalent to null/nil).*

- **types**
  - simple types `:integer :decimal :string :boolean :time :date`
  - structured types
    - array `[ <type> ]`
    - struct `{ '<name>' => <type>, ... }`
    - hash `{ <type> => <type> }`

*AML does **not** support global variables, **by design**.*

axini

variables: usage

- **constraint** of a transition/action:

response

```
send 'value', constraint: 'x > 3 && y < 4'
```

  - **x**, **y** can be *label parameters* of `'value'`, or *state variables*; send is only enabled when the constraint evaluates to **true**.

stimulus

```
receive 'dice', constraint: 'x > 3'
```

  - If **x** is a *label parameter* of `'dice'`, AMP will **generate** a value for **x** such that the expresion `'x > 3'` is **true**.
  - If **x** is a *state variable*, the expression `'x > 3'` most evaluate to **true** for the receive to be enabled.

- **update** of a transition/action:

stimulus/receive

```
send 'prize', update: 'prize_won = true'
```

  - variable `'prize_won'` must be a boolean *state variable*.

axini

stimuli: label parameters

*AMP requires that **all label parameters** of a **stimulus** are **constrained**.*

```
external 'extern'
process('stimulus-example') {
  timeout 10.0
  channel('extern') {
    stimulus 'stimulus',  { 'x' => :integer }
    response 'response',  { 'y' => :integer, 'z' => :string }
  }

  receive 'stimulus', constraint: 'x != :void'

  receive 'stimulus', constraint: 'x == 42'

  receive 'stimulus', constraint: 'x >= 0 && x <= 10'

  receive 'stimulus', constraint: 'x in [0, 2, 4, 6, 8]'

  send 'response',    constraint: 'y == 42'
}
```

nil value

any value (but AML will probably choose 0, ... the rirst time)

fixed value

interval

from a list

*response: check on **y**, but not on **z***

axini

Example

```
external 'external'
process('parameters-example') {
  timeout 10.0
  channel('external') {
    stimulus 'aap',  { 'y' => :integer }
    response 'noot', { 'z' => :integer }
  }

  var 'x', :integer, 2
  var 'y_rcv', :integer
  var 'z_snd', :integer

  receive 'aap',
      constraint: 'y > 0 && y < 10 && x > 0',
          update: 'y_rcv = y; x += 10'

  send 'noot',
      constraint: 'z > y_rcv && x > 10',
          update: 'z_snd = z'
}
```

*variables to **store** the values of label parameters*

*constraint on **label parameter***

*constraint on **state variable***

*AMP will thus **generate** a value for **y** between 0 and 10.*

***saving** the value of the label parameter into a state variable*

axini

Some of these will be discussed on Tue 1-Apr.

AML – Advanced Features

- **structured data types** (lists, structs, hashes)
- **behavior definitions** to encapsulate behavior
- **internal communication** between processes
- **timed testing** features (delay and force labels)
- **function definitions** for complex computations
- **urgent transitions** to enforce internal communication
- use of **Ruby** as preprocessor to structure the model

… see the "Modeling Tutorial" in the Help of AMP.

+ **Reference material**:
- AML Reference Manual
- AML Quick Reference

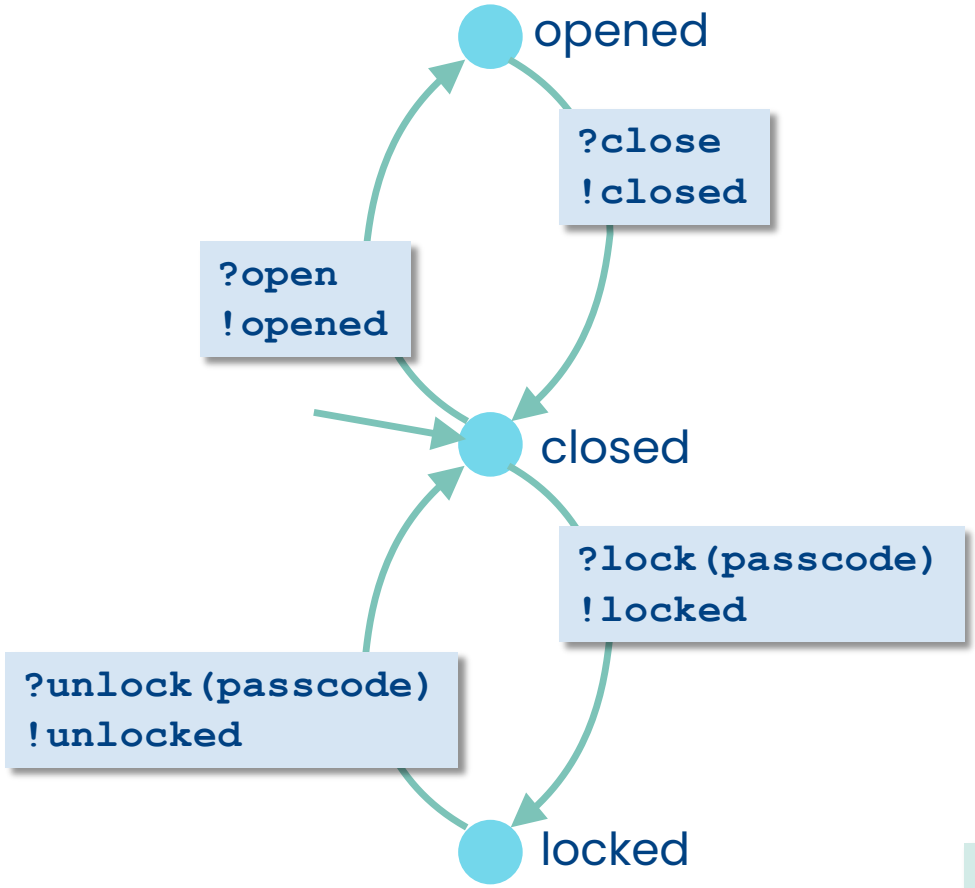
axini

SmartDoor

Theo Ruys

Laboratory: SmartDoor

***Happy Flow** behavior of the SmartDoor: all 'invalid' and 'incorrect' labels are **not included**.*



*Can also be modelled with three **non-terminating behaviors**.*

*After three incorrect passcodes, the SmartDoor will **'shut off'**.*

Laboratory: SmartDoor

- **SmartDoor**
  - develop a **model** on basis of a specification
  - using a (correct) SUT to develop and **test** the model
  - **testing** several (incorrect?) SUTs
  - **find all the bugs** in the incorrect implementations

- **Material**
  - SmartDoor **specification**: formal specification
  - **Laboratory text**: roadmap to approach the exercise

*Carefuly study the **specification** before following the **Laboratory steps**.*

Remember: goal is to make a **model of the SUT**.
- alternative, **high-level abstraction** of the SUT
- **direction of messages** (stimuli, responses) is from the point of view of the SUT.

axini

Laboratory: SmartDoor – HOWTO

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 controlled door called SmartDoor. This project should give you a good feel for how MBT practice.

In AMP, you can select t… top. This pr… tains multiple Test Sets… e SmartDoor… choose the Axini test se…

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. Wh… is locked, a passcode is set. A separate *SmartDoor Interface Requirements Specification* … ment is provided that contains more details.

In the following sections, we will guide you through modeling the behavior of SmartDoo… step, we recommended checking the visualization to see if your model ended up the wa… tended. Moreover, the SmartDoor implementation by Axini has already been tested with… can consider this implementation to be correct and use it to validate your model.

Follow the **step-wise instructions** in the exercise **description**.

Be sure to model **all requirements** of the "official" **specification**.

SMARTDOOR
Interface Requirements Specification

SmartDoor Solutions

IRS Version 2.0
September 2020

*Warning: in certain places, the requirements may **not be precise** enough… as with specifications in practice.*

Laboratory: Report

*Report of the bugs should **not be longer** than a single **A4** paper.*

- Write a small **report** on the **bugs** found in the various SUTs.
  - Do **not** go **overboard** here! We are only **interested** in the **bugs found** and a concise and precise **description** of these bugs.

Section 2.6 of the Laboratory text.

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?

*Formulate the bug in **one** (or two) **sentences** (with a reference to the violated requirement).*

For example:

```
Micro$oft SUT: when locked, after an ?open stimulus, the SUT
responds with a !unlocked response, whereas it should have
responded with an !invalid_command (see: BEHAVR-01 and BEHAVR-05).
```

Include your **final AML model** of the SmartDoor (which has been used for **all** the tests).

Laboratory: Grading

Grading of the SmartDoor will be done as follows.

- (**4** points) **Modeling**: we will grade your AML model on whether it includes all behavior of the SmartDoor.
- (**3** points) **Testing and Analysis**: find, describe and perform analysis on all **easy bugs** and analyze what might have caused them.
- (**3** points) **Testing and Analysis**: find, describe and perform analysis on all **hard bugs** and analyze what might have caused them.

The online documentation might still mention 4 here.

A bonus point is awarded for teams that find **all bugs** in the SmartDoor implementations.

Only a single group (UvA 2024) has found all bugs.

axini

SmartDoor: Some AML Tips

Theo Ruys

AML Quick Reference Card

For the 'SmartDoor' exercise, only the following statements are needed:

- **send & receive**
- **choice**
- **repeat**
- **state & goto**

+ variables, constraints, and updates

```
external channel_name [, type: :duplex]
internal channel_name
...
process(process_name) {
  timeout <decimal_value>
  channel(channel_name) {
    stimulus label_name ,
        param_name  => <type> ,
        ...
    response label_name, ...
  }
  var var_name, <type> [, <initial_value>]
  ...
  # behavior of process: AML statements
}

process(process_name) {
  ...
}
```

channel_name has to be declared globally

AML constructs are separated by newlines or ;

```
include 'model_part.aml'
```

can be used anywhere

```
<type>
:integer
:decimal
:boolean
:string
:date
:time
```

```
[ <type> ]                          list
{ <type> => <type> }                hash
{
  field_name => <type>,
```

```
send
receive
,
, expedit
, urge
, aft
, befo
, constrai
, upda
, no
```

```
<statements> ::
  | { <state
```

```
choice {
  o  <statements>
  o  <statements>
  ...
}
```

```
repeat {
  o  <statements>
  o  <statements>
  ...
}
```

```
state state_name
goto state_name
```

```
stop_repetition
next_repetition
```

deterministic choice

```
_if <bool_expr>,
_then { ... } [,
_else { ... } ]
```

deterministic loop

```
_while(<bool_expr>) {
  ... # body
}
```

```
optionally { <statements> }
```

internal constraint

```
constraint <bool expr>
```

internal action

```
update <assignment_expr>
```

Ruby code

```
def method(params)
  ...
end
```

```
if <ruby_expr>
  # AML fragments
else
  # AML fragments
end
```

```
"...#{<ruby_expr>}..."
```

```
behavior(behavior_name, :terminating
                        :non_terminating
  [, [param_name => <type>, ...]] [, <return_type>] ) {
  # AML statements
}
```

terminating behaviors return value with `exit_with`

```
call terminating_name [, [<expr>, ...]] [, into:<var_name>]
```

```
r>, ...]]
```

```
<type>) {|p, ...|
```

xpressions: constraints or updates

- Names of labels, states, variables are 'strings' in quotes.
- Curly brackets { … } are used to group statements.
- Statements are separated by newlines (or ';').

fixed font	keywords
*_name	string, e.g. "Notify"
<*_expr>	string, e.g. "x > 5"
<foo>	grammar placeholder
...	repetition / placeholder
[...]	optionally

Some common mistakes

- The SmartDoor SUTs start in a **closed** + **unlocked** state.
- Only use the label names as mentioned in the **text**. So:

```
stimuli   'open', 'close'
stimulus  'lock', { 'passcode' => :integer }
responses 'opened', 'closed', 'invalid_command'
```

- Constraints and updates are **strings**: '…' or "…"

```
receive 'lock', constraint: 'passcode >= 0 && passcode <= 9999' ,
                update: "entered_passcode = passcode"
```

- Only a **single** constraint and update per send/receive.
  - not: `send 'foo', update: 'x = 1', update: 'y = 2'`
  - but: `send 'foo', update: 'x = 1; y = 2'`
- Deterministic choice

```
_if 'x > 0' ,
_then { send 'positive' } ,
_else { send 'negative' }
```

Both the _if and _then and the _then and _else should be separated by a comma.

For the SmartDoor, there is only one situation where _if _then _else is appropriate.

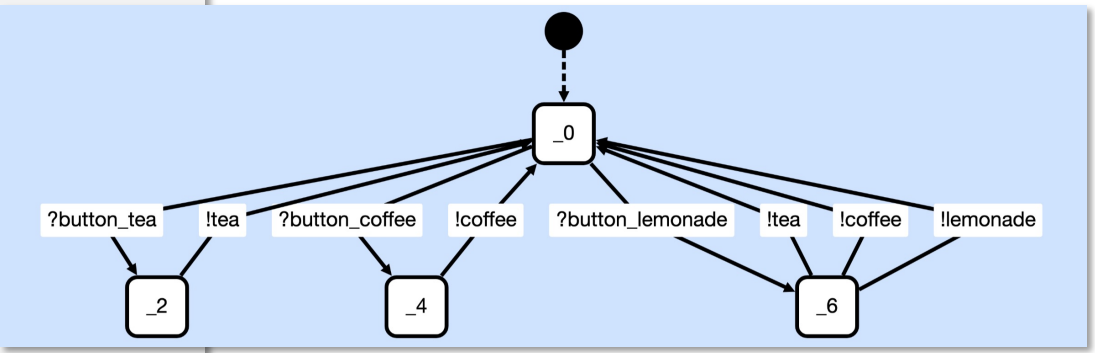
Using Ruby as preprocessor

```
config = {
  :include_lemonade => true
}

process('main') {
  channel('controller') {
    stimuli 'button_tea', 'button_coffee', 'button_
    responses 'tea', 'coffee', 'lemonade'
  }

  repeat {
    o { receive 'button_tea'    ; send 'tea' }
    o { receive 'button_coffee' ; send 'coffee' }
if config[:include_lemonade]
    o {
      receive 'button_lemonade'
      choice {
        o { send 'tea' }
        o { send 'coffee' }
        o { send 'lemonade' }
      }
    }
end
  }
}
```

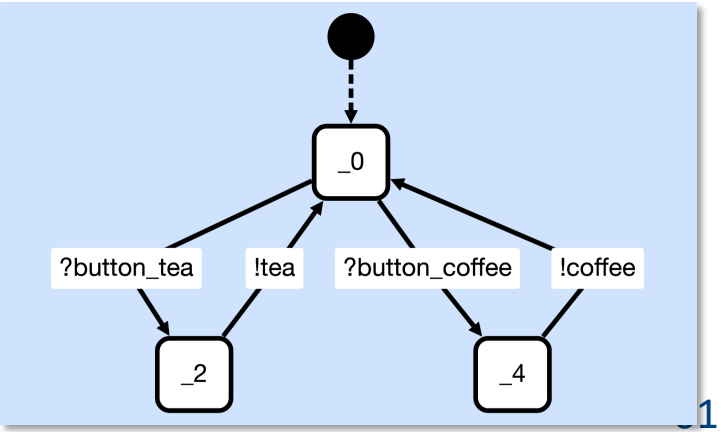
Ruby fragment

Similar to the CPP preprocessor which is used in C/C++: ***#define #if #ifdef,*** *etc.*

`config[:include_lemonade] == true`



`config[:include_lemonade] == false`



SmartDoor – some modeling tips (1/2)

To **include/exclude** model **fragments**:

*Add the following **Ruby hash** to the preamble of your model:*

```
config = {
  :besto_1  => false,
  :besto_2  => true,
  :logica_1 => false,
}
```

… or the equivalent:

```
config = {
  besto_1:  false,
  besto_2:  true,
  logica_1: false,
}
```

Within the model itself:

```
if config[:besto_1]
  ... # AML fragments to include
end

...

if config[:logica_1]
  ... # AML fragments: logica bug
else
  ... # AML fragments: correct behavior
end
```

*This ensures that we have a **single AML model** which contains all expected behavior and all bugs found.*

SmartDoor – some modeling tips (2/2)

- To *observe* '**quiescence**' in the model: *needed for SECLOC-7*

  *Add '**virtual**' response 'quiescence' to the process.*

```
response 'quiescence', on: 'external'
```

  *This response will **never arrive**, so AMP will wait for 2.0 seconds.*

```
optionally { send 'quiescence', timeout: 2.0 }
```

```
choice {
  o { send 'quiescence',
              timeout: 2.0 }
  o {}
}
```

- Use **notes** to emphasize specific labels in a test case (trace):

```
send 'opened', note: 'opened'
```

```
send 'locked', note: ['locked', '$entered_passcode #red']
```

*inserts the contents of state **variable** 'entered_passcode' into the **note***

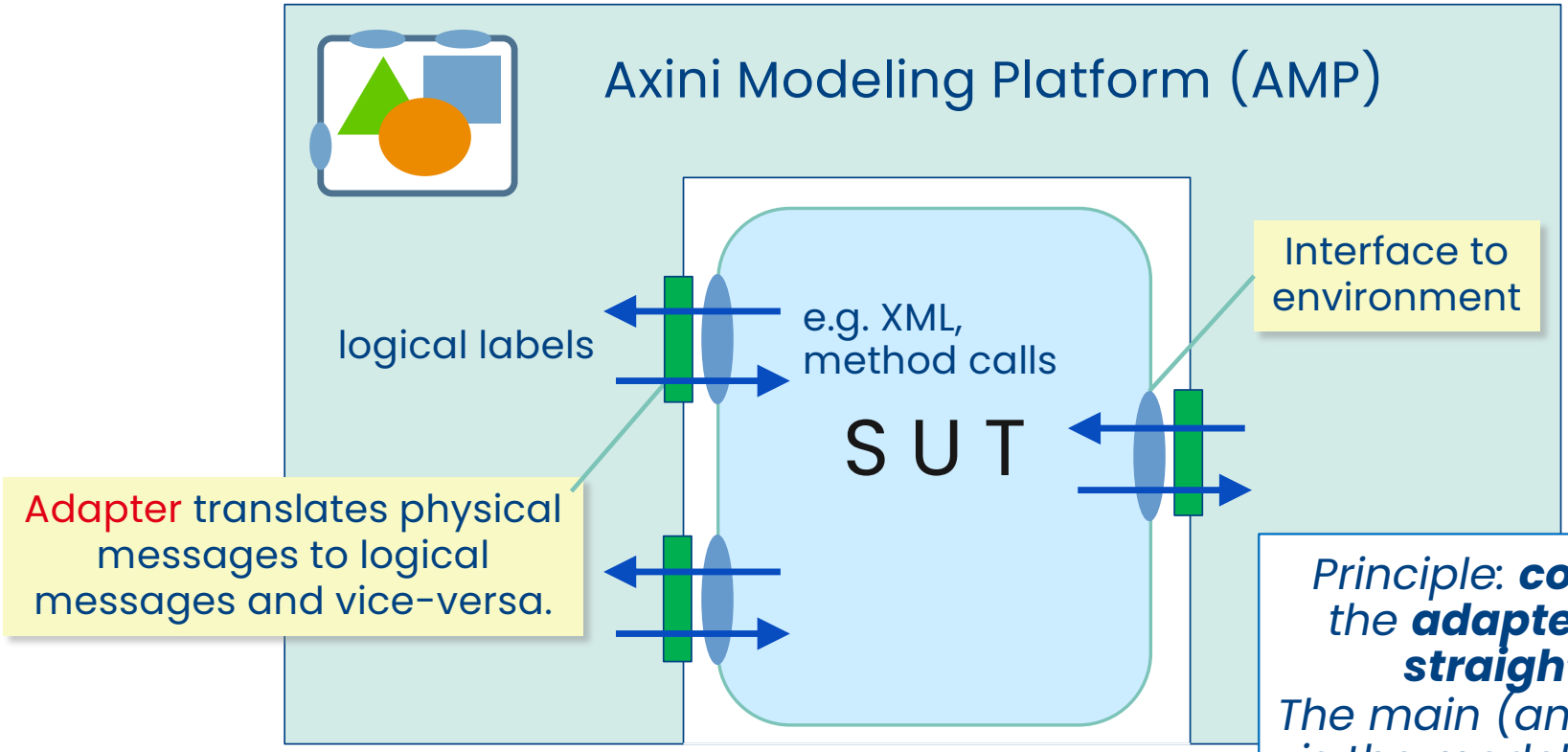
*You can also use **label parameters** of the label here.*

axini

Plugin Adapters

Theo Ruys

AMP: horseshoe around SUT

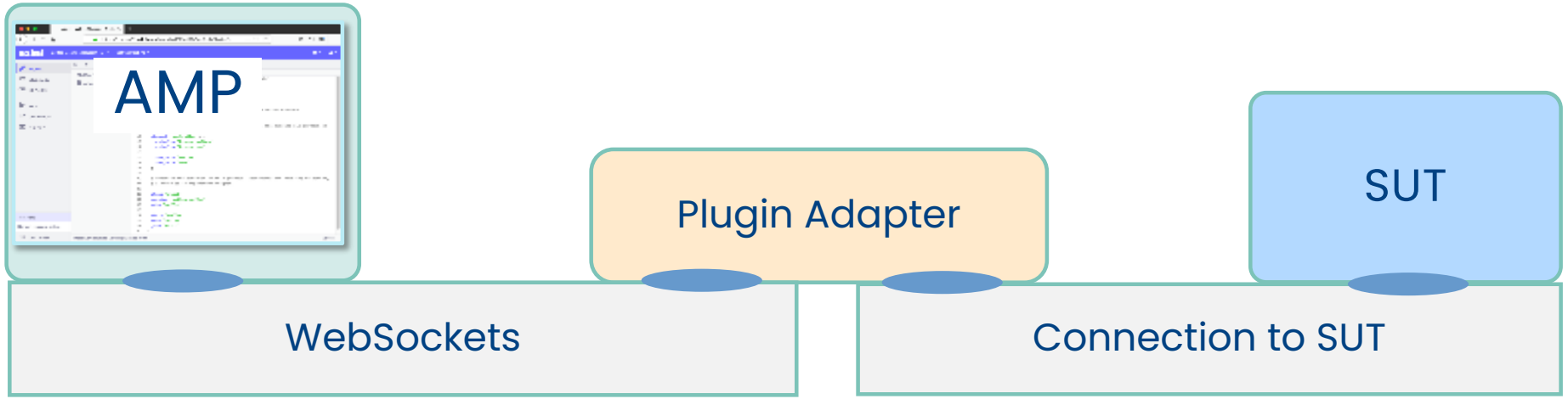


axini

AMP – Plugin Adapters

*Objective: **third parties** should be able to develop adapters for AMP.*

- Programming Language **independence**.
- **WebSockets** over HTTP for transport.
- Google **Protobuf** to represent **labels**.
- AMP acts as **server** for incoming adapers.



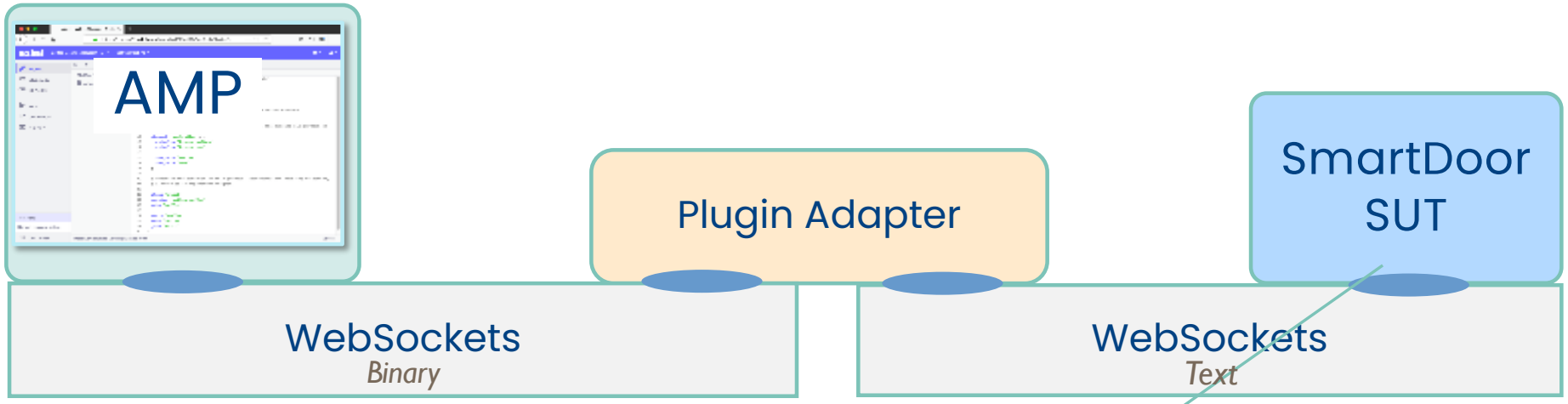
*Plugin adapter can be run more **closely** to the SUT, which allow for more accurate **timestamps**.*

Axini Plugin Adapters

https://github.com/Axini

- **plugin-adapter-protocol**
  - .proto files
- **standalone-smartdoor-java**
  - Java implementation of a SmartDoor application
- example adapters for SmartDoor application:
  - smartdoor-adapter-**java**
  - smartdoor-adapter-**cpp**
  - smartdoor-adapter-**ruby**
  - smartdoor-adapter-**python**

Adapter for SmartDoor SUT



The SmartDoor SUT is a **standalone** SUT from the 'SmartDoor' laboratory exercise.

- commands & responses, e.g. OPEN, OPENED, CLOSE, INVALID_COMMAND
- LOCK with passcode

SmartDoor SUT talks in UPPERCASE!

Homework [for Tuesday, 1-Apr]

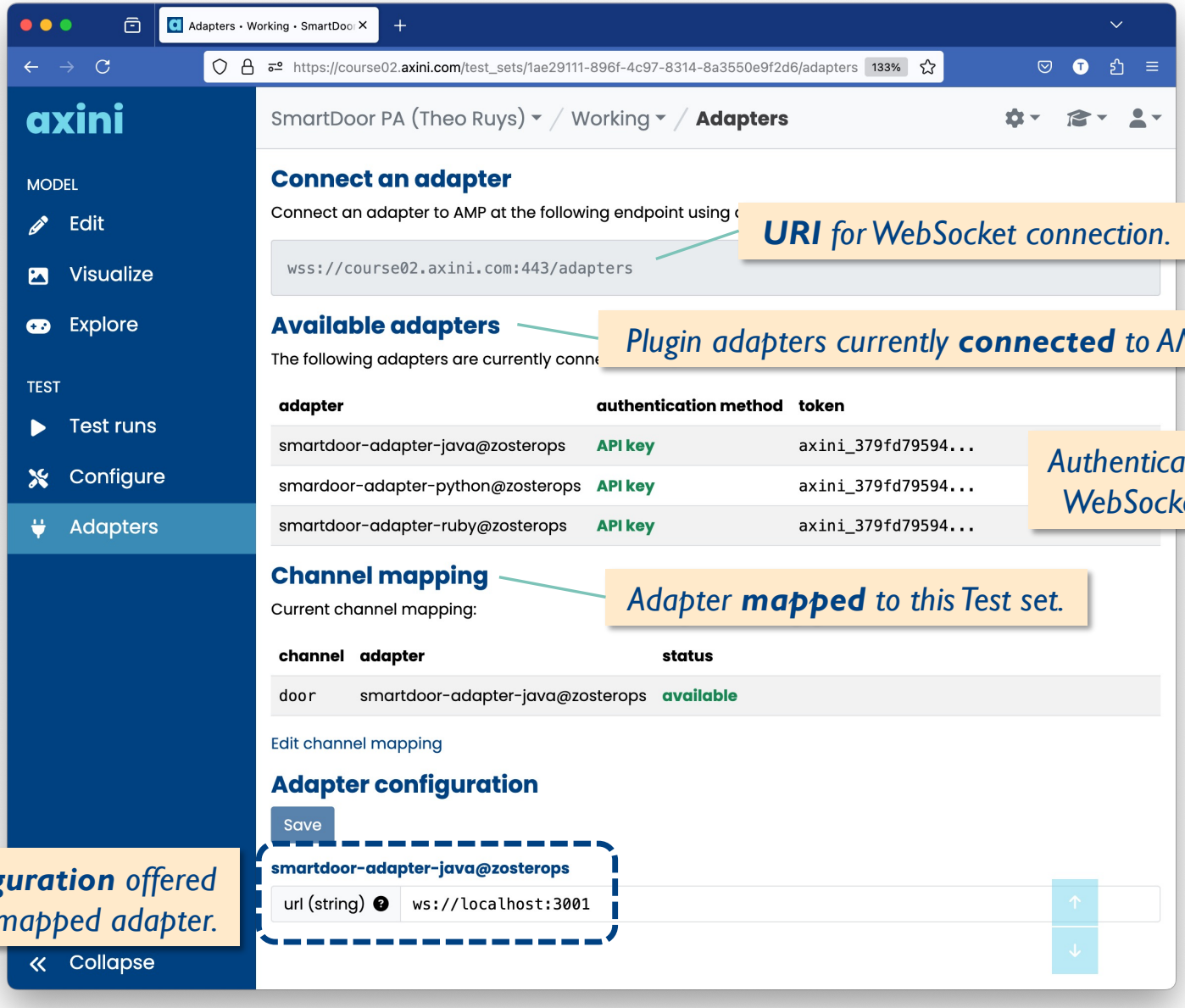
*The SmartDoor adapter will be the **starting point** for your NewsFeed adapter.*

1. Download **example** SmartDoor plugin adapter (for your **favorite** programming language) from Axini's **GitHub** page.
2. **Install** all necessary third-party **dependencies** (WebSockets, ProtoBuf, logging, etc.): see README.md.
   - **Ensure** that you are able to **connect** to AMP at `course02.axini.com`.
   - Use your "SmartDoor PA" project for this.
3. **Download standalone** SmartDoor SUT (in Java).
   - This SUT is correct (like Axini's SUT).
4. Use your own SmartDoor AML model to **test** the **plugin adapter** (and the SmartDoor standalone SUT).

*For the "SmartDoor PA" and "NewsFeed PA" projects, you have to create an **initial Test set** (e.g., `"Working"`) and **root model part** (e.g., `main.aml`).*

*Please **report** any **errors** and/or **mistakes** in the plugin adapters to us.*

AMP: Adapters page



Connecting your adapter

- Plugin adapter needs:
  - **URI address** of the AMP server (i.e., for `course02`)
  - **adapter token** (your personal API key token)
  - **name**: give your adapter a **unique** name
    - For example with your **group number** included, such that you can select the correct one.

*Should be changed in the **source** of the plugin adapter, or overwritten on the **command line**.*

*For example for the **Python** plugin adapter:*

```
% python3.11 plugin_adapter.py \
-n zosterops \
-u wss://course02.axini.com:443/adapters \
-t axini_379fd79594d694...
```

API key tokens

UT-Group-01	axini_9d9e4133f24e54f798b9bc305c7809f15078e5468b5cac896da26d3618de3fd3422e
UT-Group-02	axini_f98ca345d47a7343dff03944636f6a9b5fe45d79bd8f6fbbd6768be3624f7b9bd660
UT-Group-03	axini_4103ccf34807c131f417e3df3f574937c2635f658e62f6562eb99d8357fe913a5610
UT-Group-04	axini_5f266ed4b949eba600c959be50a15aec1af3135603e17d40bd423fa70a6a1bc242e9
UT-Group-05	axini_d58938ad1e51aaf43be2813317b3753aef85058567f471a1748f56d4bad695ac7420
UT-Group-06	axini_81341f64971dcb4eaacdb88293f5758515a01b2bf6fb26661e772d9dfc2f4c04fed5
UT-Group-07	axini_744d39339cb6352efecc1e549cd41557892ef723fa737f8971819ea2afaed2ac9303
UT-Group-08	axini_4f57e9f7a0d5a8a5734a7da6e0d4eaec2f4516e8b89cde59f27dfff869adcfdc6640
UT-Group-09	axini_d9613df600a56d5ebc4f3e5a78aa20b294d1d1ebf3305a33a4f85011d8e82b01efb8
UT-Group-10	axini_8436c7b4d2aaee1b59693792c7dce6924ff0c8f8cffbf99db50af66c9b1c56cae8fe
UT-Group-11	axini_6ffa081c23c10713d59ee7b111ac1a1742652c3bff2f3d7cefa2d3f6efa57f93cfb8
UT-Group-12	axini_4c71c38a5374c65629f31d50cb65dbfef5d3472cadf836c61fef2934c5987ef7b7ec
UT-Group-13	axini_d68d6e7ee353bcfa739c2cba2f70705254ccea3020156fbc3ea4f775dfb4a3bba170
UT-Group-14	axini_d5873a42d8f28f01ce15a3b7550b8329ee0e4ea9c900019817c9c8c0597262757a00
UT-Group-15	axini_4d2fce5f47d25c13278167e1c08e0c988aa0ffbe065aac6e9ee943cf888122c57cb1
UT-Group-16	axini_95cbf74c37082783c2a8f9b060f0797aa85adf6f45251e35764a65b4e008e7f5aea7
UT-Group-17	axini_c92225e9520c99ac7ad1eca78c622ea2f013c250c7a2acf0847e85e51f0c20232a9a
UT-Group-18	axini_9c043181ebcfbecc5c63855d071f4d85aa1233a1486d18c049c93eed1c2b84ed0962
UT-Group-19	axini_cd716f0463e7f2289a4725599654053fe166729e6a56786d126cc55f9fd6259d87a8
UT-Group-20	axini_4993b5056b2bd03d597048138307dd7f392a18eec66144d3659d94e41983ea9f26f8
UT-Group-21	axini_0e95bdd0ed0798188dce21d675154f439017b93b3878db22df4db10be39a2b58464f
UT-Group-22	axini_bd2c03d4bdda18d64933109b9659e3b85897e6cb73eb175bf72e3d4f822edfa434f2
UT-Group-23	axini_b78ab2b178ddc67e97b64a94f306cff212c7571a78b032395af099e5b2afeb80c4e0
UT-Group-24	axini_b95ece054f20dee7026d01a3db7d4f32ed7578ea515755b51d550cf6a8dda9563717
UT-Group-25	axini_20ec39c53d23dab9474d175bde8824f979919a5256a0d0e58c0b7e031f7dfc7d39b7

axini

axini

Conclusions

Theo Ruys

Day 1 – Mon 24 Mar

- Introduction to **MBT** and **AMP**
- **Basic AML**
  - labels: stimulus, response
  - choice
  - repeat
  - states, goto
- **Coffee Machine** exercise
- **SmartDoor** exercise
  - Create the **model** on the basis of a specification.
  - Use AMP and your model to **find bugs** in **erroneous** SUTs.

HOMEWORK

Deadline for SmartDoor bugs and model ***Mon 31 Mar 23:59h****.*

+ make sure that you have a ***working plugin adapter*** *for the standalone SmartDoor SUT.*

If you have any ***questions****, you can send an email to Theo Ruys <theo.ruys@axini.com>.*