axini

Axini Lecture 2

SmartDoor: discussion
Some more AML
Plugin Adapters

Tuesday, 1-Apr-2025

Lecture 2

- AMP: analysing fails
- **SmartDoor**: discussion
- AML
  - **choices**
  - behaviors
  - internal synchronisation
- Axini **Plugin Adapters** (**PA**)

*Did every group manage to get the **plugin adapter** for the **standalone** SmartDoor SUT working?*

REVISIT DAY 1

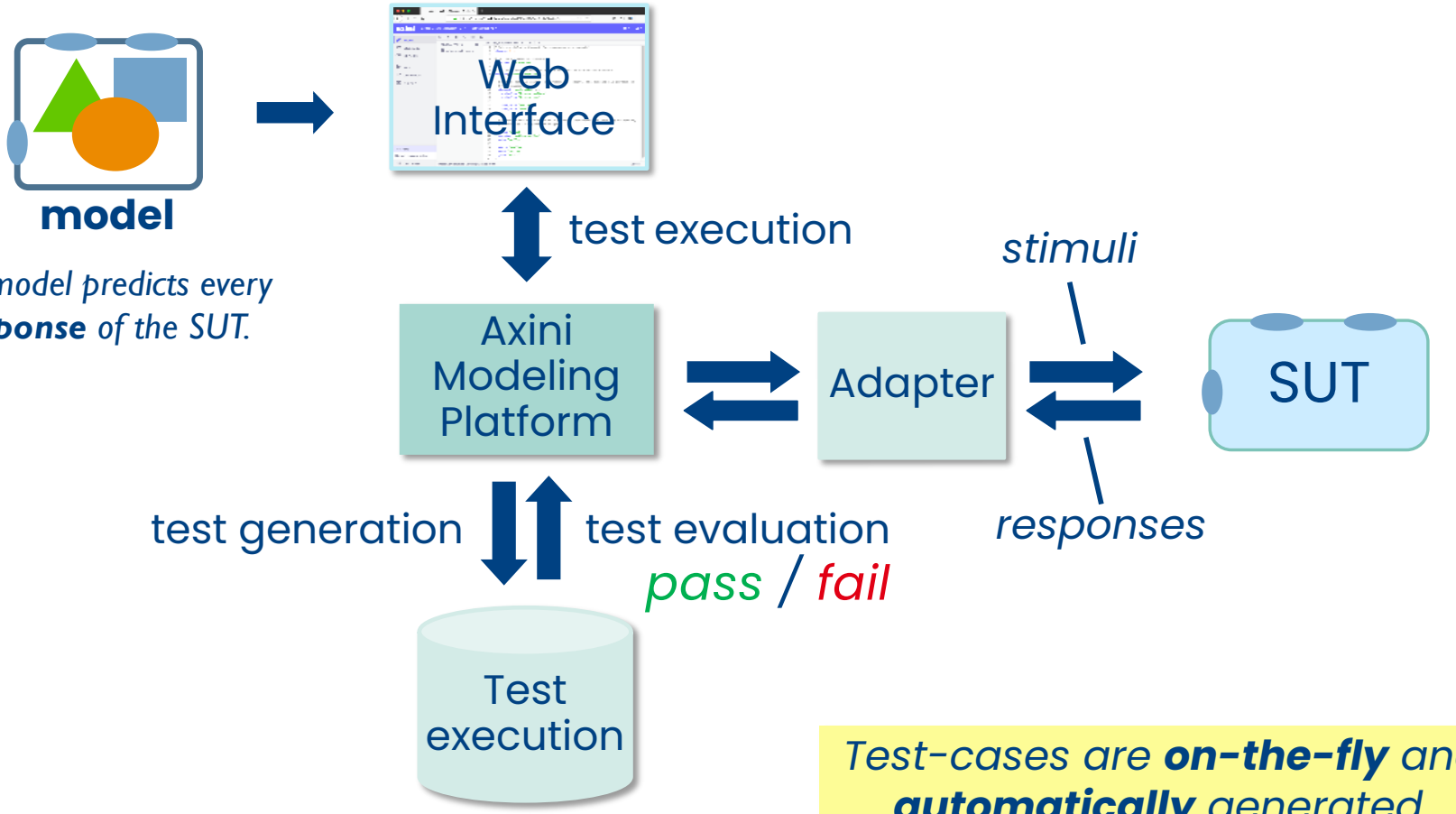
MBT of NewsFeed with AMP

- 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

AMP architecture

*You 'only' need to write the **model** and develop an **adapter**. ;-)*



*Test-cases are **on-the-fly** and **automatically** generated.*

Inside AMP

- basis: on-the-fly, **ioco**-based conformance algorithm
- **solving** constraints of stimuli: **GProlog** or **Z3**.
- **choices** while traversing the model:
  - **direct** choices (`choice` or `repeat` statement)
  - **indirect** choices: parallel processes
- When there are choices, **AMP** will decide what do: **inject** a stimulus to the SUT or **wait** for a response.
  - AMP's **strategy** decides what **option** to choose.

*Except the "Random" strategy, all other strategies try to **cover the transitions of the model as soon as possible**.*

Built-in strategies of AMP:

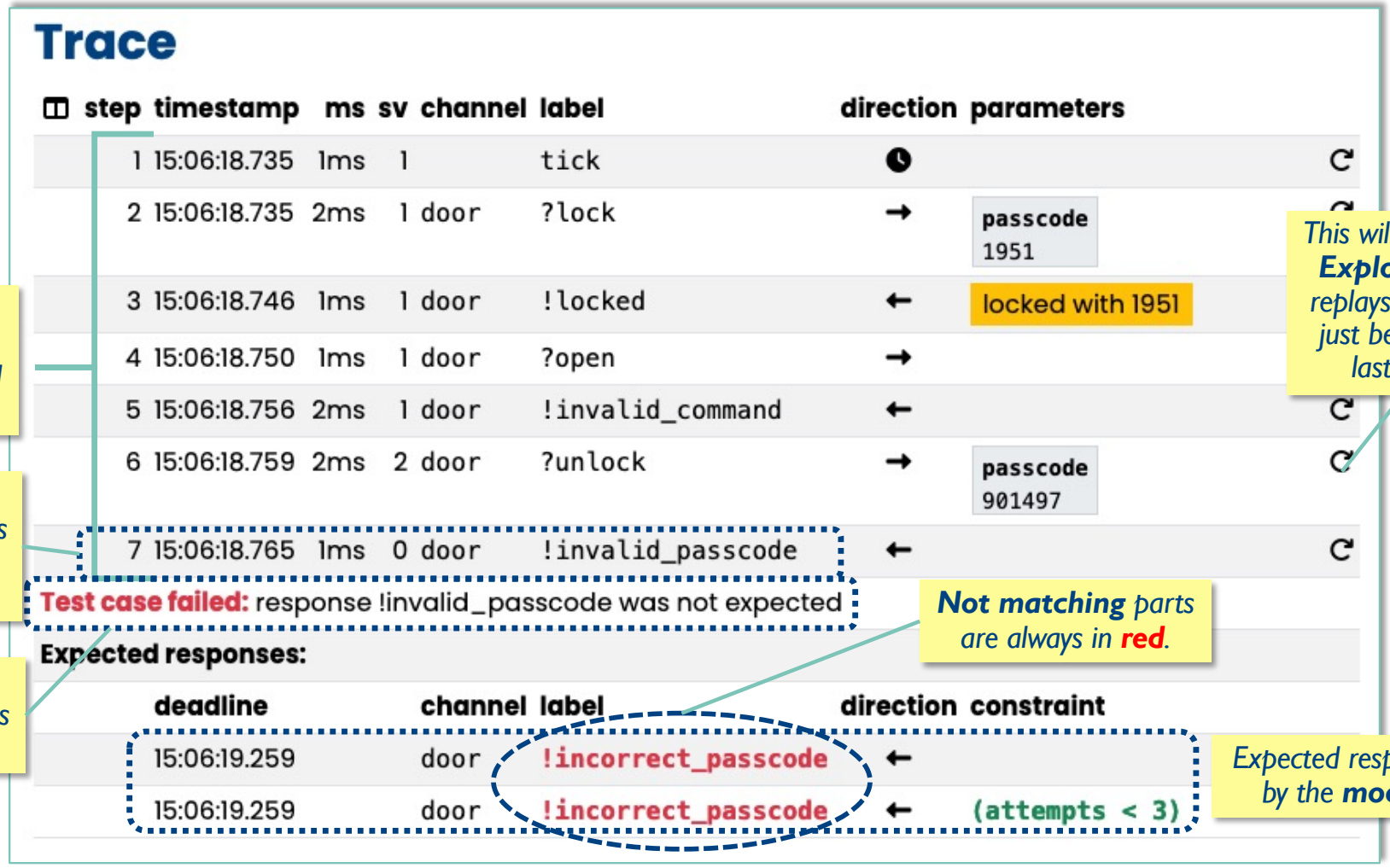
Strategy::Random
Strategy::WeightedLabelRandom
Strategy::GlobalTransitionCoverage
Strategy::LocalTransitionCoverage
Strategy::GlobalTransitionPairCoverage Beta
Strategy::LocalTransitionPairCoverage Beta

axini

Understanding Fails

Theo Ruys

Understanding the Trace



All *numbered* *steps have really happened at the* ***SUT***.

The last step is the one which was ***not*** *expected by the* ***model***.

AMP will give a ***hint*** *on what has gone* ***wrong***.

This will start the ***Explorer*** *and replays the case just before the last step.*

Not matching *parts are always in* ***red***.

Expected responses by the ***model***.

axini

Possible Fail (1)

Failed *test cases are caused by* ***unexpected responses*** *or* ***missed responses****.*

Not matching *parts are always in* ***red****.*

① ***Unexpected*** *response*

4	15:46:58.741	1ms	1 door	?lock	→	passcode 0
5	15:46:58.747	1ms	1 door	!invalid_command	←	
6	15:46:58.750	1ms	2 door	?unlock	→	passcode 268435455
7	15:46:58.755	1ms	0 door	!invalid_passcode	←	

Test case failed: response !invalid_passcode was not expected

Expected responses:

deadline	channel	label	direction	constraint
15:46:59.250	door	!incorrect_passcode	←	
15:46:59.250	door	!incorrect_passcode	←	(attempts < 3)

Read AMP's analysis.

The ***response*** *observed was* ***not expected****!*

Possible Fail (2)

*Not matching parts are always in **red**.*

② *Response was expected, but **constraint** evaluates to **false**.*

Read AMP's analysis.

```
4 16:08:30.003 newsfeed ?ProtocolRequest  →   ProtocolVersions ["2.1"]   CorrelationId 3
5 16:08:30.021 newsfeed !ProtocolResponse ←   ProtocolVersion "2.1"      CorrelationId 3
Test case failed: response !ProtocolResponse did not satisfy all constraints
6 16:08:30.027 newsfeed !AvailableTopics  ←   Topics ["general", "breaking", "sport", "weather", "culture", "europe", "funny"]
Expected responses:
    deadline     channel  label                direction  constraint
    16:08:34.003 newsfeed !ProtocolResponse    ←   (ProtocolVersion == "2.2") && (CorrelationId == 3)
```

Response was expected…

*… but **label parameter** is different from the model.*

parts in green are correct

*This is a **trailing response** which was observed **after** the test had already failed.*

Possible Fail (3)

*Not matching parts are always in **red**.*

③ *Expected response was **too late**.*

12	17:17:23.773	door	?close	→	
13	17:17:24.041	door	!invalid_command	←	
14	17:17:24.046	door	?unlock	→	passcode 1
15	17:17:24.602	door	!incorrect_passcode	←	

Test case failed: response !incorrect_passcode was observed too late

Expected responses:

deadline	channel	label	direction	constraint
17:17:24.546	door	!unlocked	←	
17:17:24.546	door	!incorrect_passcode	←	

Read AMP's analysis.

Response was expected …

*… but its **deadline** has passed.*

One of the erroneous SUTs.

Possible Fail (4)

***Not matching** parts are always in **red**.*

④ *A response was expected, but **'quiescence'** was **observed**.*

Quiescence means 'silence': no response has been observed.

56	17:07:29.664	door	?open	→
57	17:07:29.667	door	!invalid_command	←
58	17:07:29.668	door	?unlock	→ passcode 5903
59	17:07:29.674	door	!incorrect_passcode	← incorrect 2
60	17:07:30.174		tick	

Test case failed: quiescence was observed, but a response was expected

Expected responses:

deadline	channel	label	direction	constraint
17:07:30.174	door	!shut_off	←	

Read AMP's analysis.

*Label **tick** means that AMP has **waited** for some time.*

***!shut_off** was expected …*

*… but did **not arrive** at all.*

*In this case the **model** was **incorrect**: it expected !shut_off already after two incorrect passwords.*

axini

Possible Fail (5)

***Not matching** parts are always in **red**.*

⑤ ***Quiescence** was **expected**, but a response was **observed**.*

12	17:36:42.291	door	?open	→
13	17:36:42.560	door	!opened	← opened
14	17:36:42.562	door	?close	→
15	17:36:43.102	door	!closed	←

Test case failed: response !closed was observed, but quiescence was expected

Expected labels:

deadline	channel	label	direction	constraint
		tick		

Read AMP's analysis.

*In this case the **model** seems **incorrect**: in the model the response 'closed' is not sent.*

*The **tick** now indicates that the model does **not expect** any **response**.*

***!closed** is observed, but tick (**quiescence**) was expected.*

axini

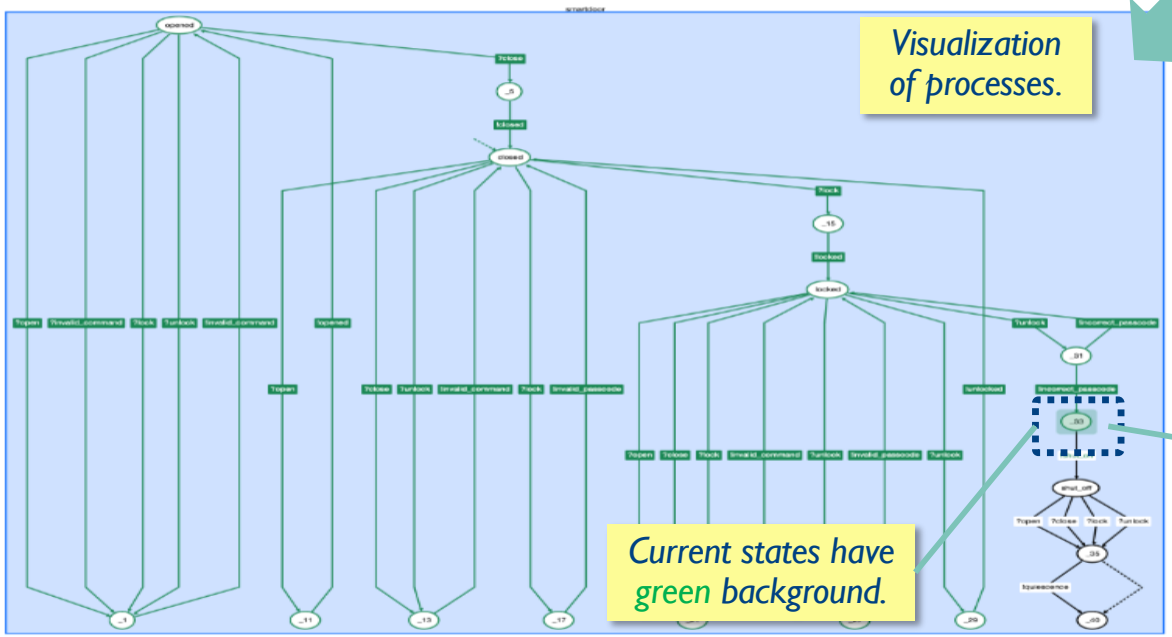
Analysing Fails

42	19:55:35.980	door	?unlock	→	passcode 5897
43	19:55:35.984	door	!incorrect_passcode	←	incorrect 2
44	19:55:36.491		tick	🕒	

Test case failed: quiescence was observed, but a response was expected

Expected responses:

deadline	channel	label	direction	constraint
19:55:36.484	door	!shut_off	←	



Visualization of processes.

Current states have green background.

*Press 'rounded-arrow' to start **Explorer**.*

Restart | Visualize

Press 'Visualize' to see where we are.

Next label

Advance	![door]shut_off
Advance	tick

*Menu of **possible actions** which can be chosen after step 43.*

State vectors (1)

Shared valuations

smartdoor

entered_passcode	unlock_passcode	attempts
9999	:void	2

clock
time(2024-01-23 18:55:35.984368061 +0000)

last_external_time
time(2024-01-23 18:55:35.984368061 +0000)

*Current values of the **variables**.*

State vectors

smartdoor

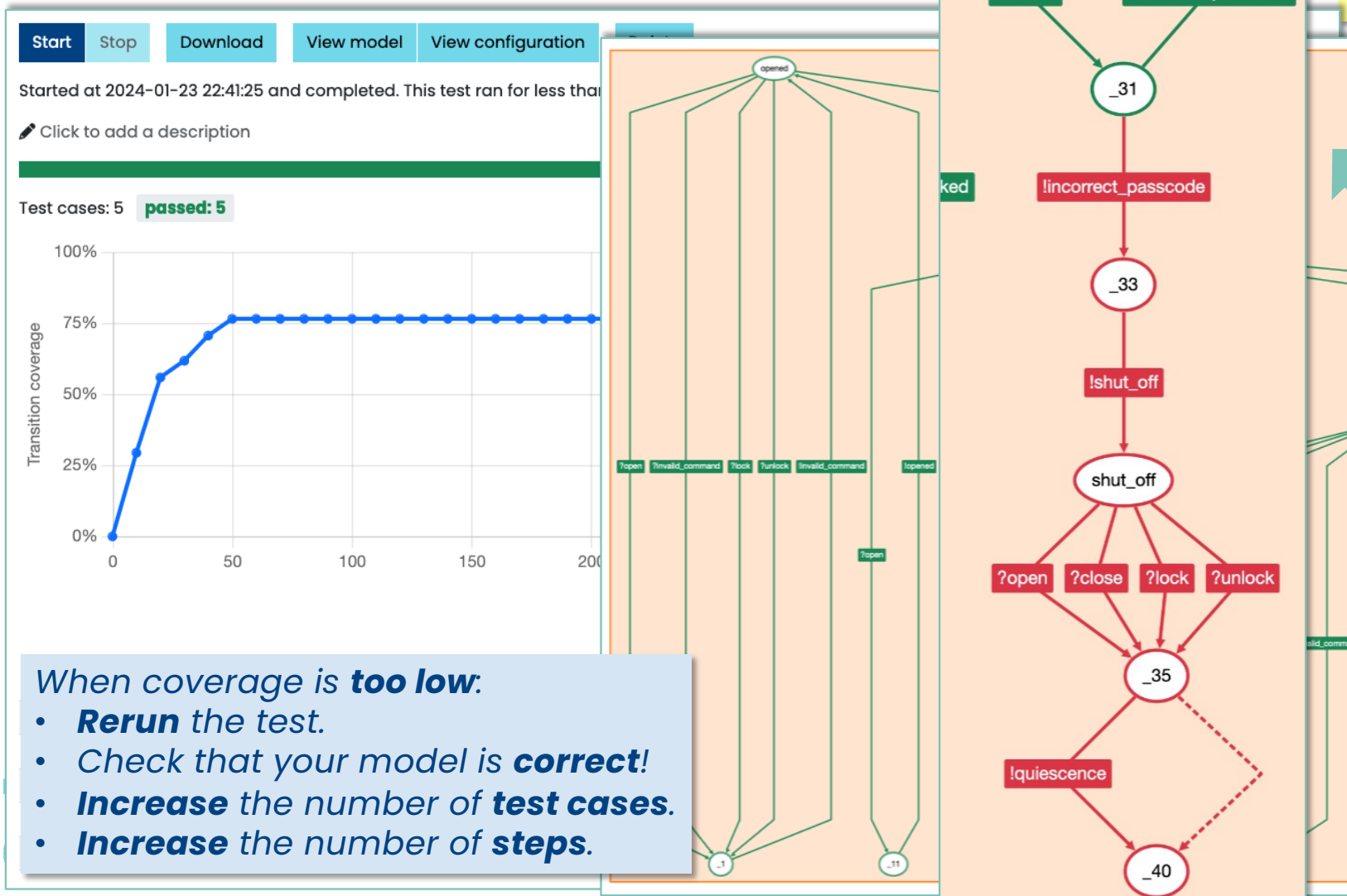
smartdoor__33

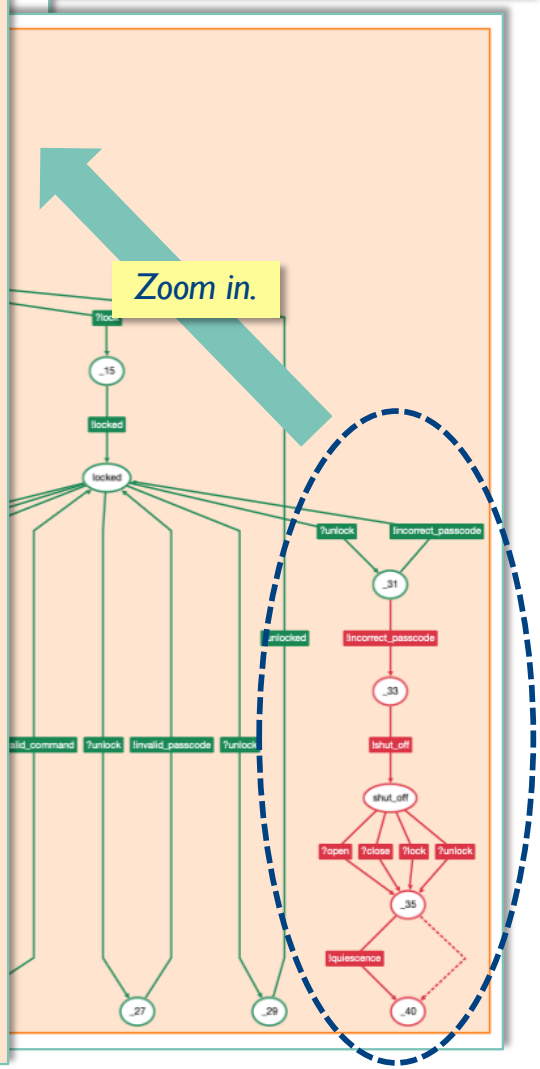
*Current **state(s)**.*

Explored trace

step	timestamp	label	direction	parameters
1	19:55:35.742	tick	🕒	
2	19:55:35.742	?close	→	

Check the coverage





*Always check the **coverage** when it is **less** than **100%**.*

*When coverage is **too low**:*

- ***Rerun** the test.*
- *Check that your model is **correct**!*
- ***Increase** the number of **test cases**.*
- ***Increase** the number of **steps**.*

axini

Discussion and Evaluation of the

SmartDoor Exercise

Theo Ruys

SmartDoor – some questions

- How to **test** SmartDoor **without MBT**?
  - assuming that you have adapters for the messages
- Are the **test cases** (trails) **always the same**?
  - probably not, due to: non-determinism of SUT, non-determinism of (stimuli) solver, random generator
- How to **cover** the following requirements?
  - *DIALOG-10*: when an **unknown command** is received…
  - *SECLOC-07*: the door does not response to any commands until it is restarted.
- How to find **other errors** after observing a **fail**?

SmartDoor – evaluation

UT STAR 2025

Nice Work!

- **Many bugs** have been found.
  - Best model found all 11 bugs! Second time ever.
  - One bug (in Universolutions) was very hard: "shuts off when receiving unknown message" (DIALOG-10).
- Some AML models are **hard to read** and **understand**.
  - Many statements on the same line.
  - Options of the labels are not structured.
  - Long state names, confusing variable names.
- **Masking** of bugs
  - Several groups have implemented this. Nice!
- All **reports** on a **single page**. Thank you!

*But next time, I will also prescribe a **minimal font size**. ;-)*

SmartDoor – some observations (1)

- Avoid **Hardcoded** passcodes (mimics unit tests).
  - `correct_passcode = 50`, `incorrect_passcode = 51`
  - idea: use AMP's solver instead.
  - Alternatively: min, max, some random values, and some special values, for example: 999, 1000, 9999.
- Do **not** use **long** (redundant!) **state names**.
  Instead of `'door_closed_unlocked'`, use: `'closed'`.
- **Notes** could/should (only!) be used to summarize/highlight labels. `send 'opened', note: 'opened'`
- Some reports mention the **number** of **failed test cases**.
  - This is **not useful** information; it is usually the same bug.

SmartDoor – some observations (2)

- Interesting idea: use **variable** to keep the **state of the door** (instead of hardcode states):
  - *con*: long and **complex constraints**
  - *con*: model **harder to maintain** and **understand**
  - *con*: **visualization harder** to **understand**
  - *pro*: model can be **shortened** (slightly) due to the same behavior for different states

cons outweight the pro!

Tip: try to **limit** the number of **variables**.

- **SECLOC-07** is **not** modelled, after `!shut_off`, the system should not **respond** to any stimulus.
- Some groups have created **Test Sets** per **person**, and used the Configuration page to set the manufacturer.
  - This is not ideal: you now should use **tags** on your **Test Runs** to identify the **manufacturer** that has been used for each test.
  - It is better to keep using separate Test Runs for each manufacturer.

axini

SmartDoor: some AML fragments (1/5)

```
state 'opened'
  choice {
    o {
      receive 'open'
      send 'invalid_command'
      goto 'opened'
    }
    o {
      receive 'close'
      send 'closed', note: 'closed'
      goto 'closed'
    }
    o {
      receive 'lock', constraint: 'passcode != :void'
      send 'invalid_command'
      goto 'opened'
    }
    o {
      receive 'unlock', constraint: 'passcode != :void'
      send 'invalid_command'
      goto 'opened'
    }
    o { # DIALOG-10
      receive 'invalid_command', note: 'unknown command #red'
      send 'invalid_command'
      goto 'opened'
    }
  }
```

*Approach is straightforward: in **all states**, try **all stimuli** and add the **corresponding responses**.*

*To make trails more **readable**. Only when the state changes.*

Add stimulus to channel:

```
channel('door') { stimulus 'invalid_command' }
```

SmartDoor: some AML fragments (2/5)

State variables

```
var 'entered_passcode', :integer
var 'attempts', :integer, 0
```

```
state 'closed'
  choice {
    o {
      receive 'lock',
          constraint: 'passcode >=0 && passcode <= 9999',
              update: 'entered_passcode = passcode;
                       attempts = 0'
      send 'locked', note: 'locked with $entered_passcode
      goto 'locked'
    }
    o {
      receive 'lock',
          constraint: 'passcode < 0 || passcode > 9999'
      send 'invalid_passcode'
      goto 'closed'
    }
    o {
      receive 'unlock', constraint: 'passcode != :void'
      send 'invalid_command'
      goto 'closed'
    }
    # ...
    # ... options for ?open en ?close omitted
  }
```

***Save** the 'entered' passcode and **reset** the number of 'unlock' attempts.*

Older idiom:

`'passcode == passcode'`

Only works for GProlog solver

*Common pattern for a random, **unconstrained** parameter: `'par != :void'`*

axini

SmartDoor: some AML fragments (3/5)

Can be modelled in many alternative ways (see later).

```
state 'locked'
  choice {
    o {
      receive 'unlock', constraint: 'passcode == entered_passcode'
      send 'unlocked', note: 'unlocked, back to closed'
      goto 'closed'
    }
    o {
      receive 'unlock', constraint: '(passcode >= 0 || passcode <= 9999) &&
                                      passcode != entered_passcode',
                              update: 'attempts = attempts + 1'
      choice {
        o {
          send 'incorrect_passcode',
              constraint: 'attempts < 3',  note: ['incorrect', '$attempts #red']
          goto 'locked'
        }
        o {
          send 'incorrect_passcode',
              constraint: 'attempts == 3', note: ['incorrect', '$attempts #red']
          send 'shut_off', note: 'shut_off#red'
          goto 'shut_off'
        }
      }
    }
    # ...
    # ... options for ?open, ?close, ?lock and invalid ?unlock omitted
  }
```

axini

SmartDoor: some AML fragments (4/5)

Add response to channel:

```
channel('door') { response 'quiescence' }
```

SECLOC-07

```
state 'shut_off'
  choice {
    o { receive 'open' }
    o { receive 'close' }
    o { receive 'lock',   constraint: 'passcode != :void' }
    o { receive 'unlock', constraint: 'passcode != :void' }
  }
  optionally { send 'quiescence', timeout: 1.0 }
```

Added as macro to AML

```
optionally {
  send 'quiescence',
     timeout: 1.0
}
```

=

```
choice {
  o { send 'quiescence',
         timeout: 1.0 }
  o {}
}
```

This is not really needed anymore as AMP will also check for quiescence after a test case.

But can be useful in the middle of a test case.

*And more importantly: the **model** should contain the **complete behavior** of the SUT.*

SmartDoor: some AML fragments (5/5)

Ruby macros

```
def bugs_configuration
  {
    besto1: false,
    besto2: false,
    logica1: false,
    ontarget1: false,
    quickerr1: false,
    quickerr2: false,
    smartsoft1: false,
    trusted1: false,
    univer1: false,
    xtrasafe1: fals
  }
end

def bug(key)
  bugs_configuration.fetch(key)
end
```

Using Ruby as a preprocessor for conditional compilation.

Marking and masking the bugs found in the SUT.

```
state 'closed'
  choice {
    o {
      receive 'open'
      send 'opened', note: 'opened'
      goto 'opened'
    }
if bug(:quickerr1)
    o {
      receive 'close'
      send 'closed', note: 'quickerr1#red'
      goto 'closed'
    }
else # specified behavior
    o {
      receive 'close'
      send 'invalid_command'
      goto 'closed'
    }
end
```

AML model conventions

Typical fragment

```
choice {
  o {receive 'unlock', constraint: 'locked_code != passcode &&
(passcode<pass_min || passcode>pass_max)'; send 'invalid_passcode'}
  o {receive 'unlock', constraint: 'passcode == locked_code', update:
'failed_attempts = 0; last_passcode = locked_code'; send 'unlocked',
note: 'state_closed_and_unlocked'; goto 'state_closed_and_un
}
```

*Modeling is like programming. Strictly adhere to **conventions**: models should be readable and understandable to others.*

The model often becomes "the requirements"!

*right-justification on ':' of label **options***

No statement after {

```
choice {
  o {
    receive 'unlock',
        constraint: 'locked_code != passcode &&
                     (passcode<pass_min || passcode>pass_max)'
    send 'invalid_passcode'
  }
  o {
    receive 'unlock',
        constraint: 'passcode == locked_code',
            update: 'failed_attempts = 0;
                     last_passcode = locked_code'
      send 'unlocked',
              note: 'unlocked'
      goto 'closed'
    }
  }
}
```

*Each logical subexpression of a **constraint** on a separate line.*

*Each assignment of an **update** on a separate line.*

*Each **statement** on a **separate line**. Unless the statements are short, without any options (e.g., coffee-machine).*

***Notes** should be short and snappy.*

***State names** should be short and snappy.*

axini

SmartDoor – a (very) a-typical system

- Communication: pairs of **request/response** only.
  - Industrial protocols have more concurrency.
- Only a **few label parameters** (only for lock & unlock).
  - Solver can be used effectively on the passcode.
- **Same response** (invalid_command) after different stimuli.
  Is there a need for a **wildcard** (*) stimulus?
  - Seldomly seen in practice.
  - If needed, can be solved elegantly with Ruby macros.
- Final model is **hard to read** with the many **if-else-end** fragments to include/exclude bugs.
  - smartdoor: small model + many bugs
  - In practice, the ratio bugs/loc is much lower.

```
if bug[:logica_01]
  ...
else
  ...
end
```

stimulus: solving label parameters

*State **'closed'**, the lock and unlock options:*

```
o {
  receive 'lock',
      constraint: 'passcode >=0 && passcode <= 9999',
          update: 'entered_passcode = passcode;
                   attempts = 0'
  send 'locked', note: 'locked with $entered_passcode'
  goto 'locked'
}
o {
  receive 'lock',
      constraint: 'passcode < 0 || passcode > 9999'
  send 'invalid_passcode'
  goto 'closed'
}
o {
  receive 'unlock', constraint: 'passcode != :void'
  send 'invalid_command'
  goto 'closed'
}
```

*AMP's **solver** will **generate** label parameter which makes the **constraint** true.*
- ***boundaries***
- ***random***

*For real systems, the **freedom** for label parameters is **usually limited**. Labels with incorrect parameters will not be accepted by the SUT.*

*Note: **coverage** of a **transition** does **not say** anything of the **data coverage** of the label parameters.*

Hardcoded parameters

*Using AMP's **solver***

```
o {
  receive 'lock',
      constraint: 'passcode >=0 && passcode <= 9999',
          update: 'entered_passcode = passcode;
                   attempts = 0'
  send 'locked', note: 'locked with $entered_passcode'
  goto 'locked'
}
```

versus

***Explicit values** for the label parameter*

Updates truncated.

```
o {
  choice {
    o { receive 'lock', constraint: 'passcode == 0',    update: 'entered...
    o { receive 'lock', constraint: 'passcode == 1',    update: 'entered...
    o { receive 'lock', constraint: 'passcode == 999',  update: 'entered...
    o { receive 'lock', constraint: 'passcode == 1234', update: 'entered...
    o { receive 'lock', constraint: 'passcode == 9999', update: 'entered...
  }
  update 'attempts = 0'
  send 'locked', note: 'locked with $entered_passcode'
  goto 'locked'
}
```

*AMP will report on the **coverage** of the **individual transitions**.*

axini

Using Ruby

Explicit values *for the label parameter*

```
o {
  choice {
    o { receive 'lock', constraint: 'passcode == 0',    update: 'entered...
    o { receive 'lock', constraint: 'passcode == 1',    update: 'entered...
    o { receive 'lock', constraint: 'passcode == 999',  update: 'entered...
    o { receive 'lock', constraint: 'passcode == 1234', update: 'entered...
    o { receive 'lock', constraint: 'passcode == 9999', update: 'entered...
  }
  update 'attempts = 0'
  send 'locked', note: 'locked with $entered_passcode'
  goto 'locked'
}
```

Updates truncated.

or the equivalent

Using Ruby's ***each*** *to* ***generate*** *the transitions.*

```
o {
  choice {
    [0, 1, 999, 1234, 9999].each do |code|
      o { receive 'lock', constraint: "passcode == #{code}",
                          update: 'entered_passcode = passcode' }
    end
  }
  update 'attempts = 0'
  send 'locked', note: 'locked with $entered_pa
  goto 'locked'
}
```

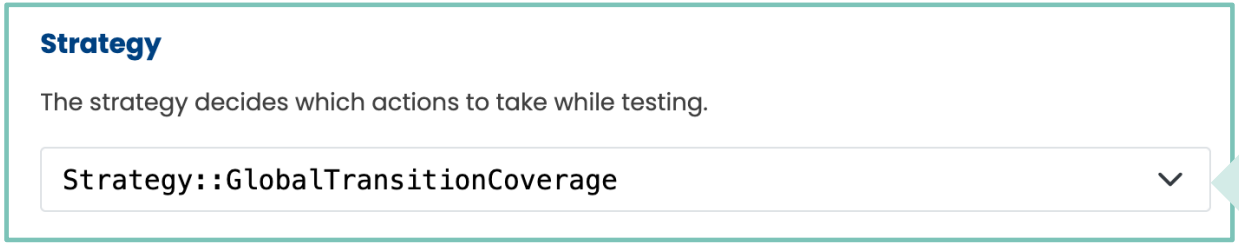
#{...} *= Ruby's* ***string interpolation****. The string value of code is inserted into the string.*

axini

SmartDoor – strategy matters

Strategy: **GlobalTransitionCoverage** leads faster to 100% coverage than **Random**.

Configure page:



Built-in strategies of AMP:

Strategy::Random
Strategy::WeightedLabelRandom
Strategy::GlobalTransitionCoverage
Strategy::LocalTransitionCoverage
Beta *Strategy::GlobalTransitionPairCoverage*
Beta *Strategy::LocalTransitionPairCoverage*

Finding shut_off (1)

Suppose we want AMP to 'find' the ***!shut_off*** *after 3 failed attempts (but without mentioning this response).*

State ***'locked'****, the receive 'unlock' option for incorrect passwords:*

```
receive 'unlock',
    constraint: '(passcode >= 0 || passcode <= 9999) &&
                 passcode != entered_passcode',
        update: 'attempts = attempts + 1'

send 'incorrect_passcode', note: 'incorrect nr $attempts'
goto 'locked'
```

After ***three*** *'incorrect_passcode' responses, the SUT should send a* ***!shut_off****.*

AMP does ***not know*** *that a possible* ***'shut_off'*** *could be generated* ***after*** *'incorrect passcode' and will* ***not treat*** *this option other than the rest.*

As you can ***only escape*** *from 'locked' with a correct passcode, the* ***!shut_off*** *might eventually be observed, though.*

Because 'incorrect_passcode' has been covered, the transition coverage will be 100%.

axini

Finding shut_off (2)

```
optionally {
  send 'quiescence',
     timeout: 1.0
}
```

=

```
choice {
  o { send 'quiescence',
         timeout: 1.0 }
  o {}
}
```

*State **'locked'**, the receive 'unlock' option for incorrect passwords:*

```
o {
  receive 'unlock',
      constraint: '(passcode >= 0 || passcode <= 9999) &&
                   passcode != entered_passcode',
          update: 'attempts = attempts + 1'


  send 'incorrect_passcode', note: 'incorrect nr $attempts'
  optionally { send 'quiescence', timeout: 1.0 }
  goto 'locked'
}
```

This response will never be sent by the SUT.

*Still, AMP's strategy will try to cover this transition: it will give **priority** to this 'receive' option over the others.*

*Consequently, after covering all other transitions, AMP will try to cover the "send 'quiescence'". After **three** attempts, a **!shut_off** will be generated, leading to a fail.*

Finding shut_off (3)

A possible solution, counting the attempts.

*State **'locked'**, the receive 'unlock' option for incorrect passwords:*

```
o {
  receive 'unlock',
      constraint: '(passcode >= 0 || passcode <= 9999) &&
                   passcode != entered_passcode',
          update: 'attempts = attempts + 1'
  choice {
    o {
      send 'incorrect_passcode',
          constraint: 'attempts < 3',  note: ['incorrect', '$attempts #red']
      goto 'locked'
    }
    o {
      send 'incorrect_passcode',
          constraint: 'attempts == 3', note: ['incorrect', '$attempts #red']
      send 'shut_off', note: 'shut_off#red'
      goto 'shut_off'
    }
  }
}
```

Two sends, with non-overlapping constraints.

*The `!incorrect_passcode` always happens, and can be lifted **outside the choice**…*

Finding shut_off (4)

*Single **send**, handling the number of attempts afterwards.*

*State '**locked**', the receive 'unlock' option for incorrect passwords:*

```
o {
  receive 'unlock',
    constraint: '(passcode >= 0 || passcode <= 9999) &&
                 passcode != entered_passcode',
        update: 'attempts = attempts + 1'

 send 'incorrect_passcode', note: 'incorrect nr $attempts'
  choice {
    o {
      constraint 'attempts < 3'
      goto 'locked'
    }
    o {
      constraint 'attempts == 3'
      send 'shut_off', note: 'shut_off#red'
      goto 'shut_off'
    }
  }
}
```

Or with a deterministic `_if_then_else`:

```
_if 'attempts < 3',
_then { goto 'locked' },
_else {
  send 'shut_off',
       note: 'shut_off#red'
  goto 'shut_off'
}
```

single receive 'unlock'

*Single **receive** 'unlock', handling the responses afterwards.*

*Trying to **limit** the number of **receives**.*

Save the 'entered' passcode, to compare it afterwards.

```
o {
  receive 'unlock',
      constraint: 'passcode != :void',
          update: 'unlock_passcode = passcode'

  choice {
    o { send 'unlocked',
              constraint: 'unlock_passcode == entered_passcode'
        goto 'closed'
    }
    o { send 'invalid_passcode',
              constraint: 'unlock_passcode < 0 || unlock_passcode >
        goto 'locked'
    }
    o { send 'incorrect_passcode',
              constraint: 'unlock_passcode != entered_passcode &&
                           (unlock_passcode >= 0 && unlock_passcode
                  update: 'attempts += 1'

      _if 'attempts < 3',
      _then { goto 'locked' },
      _else {
        send 'shut_off'
        goto 'shut_off'
} } } }
```

*But AMP has **no means** to **steer** the coverage of the **responses**: the parameter 'unlock_passcode' has already be chosen.*

Result: (possible) uncovered transitions.

axini

axini

Some more of AML

choices

Theo Ruys

choice – between labels

- between **stimuli**

  AMP will select one:
  - depends on the **strategy**: random, local-state, global-state, expedited, …

- between **responses**

  specifies possible behavior of the SUT

- between **stimulus** and **response**?

due to (our implementation of) **ioco**

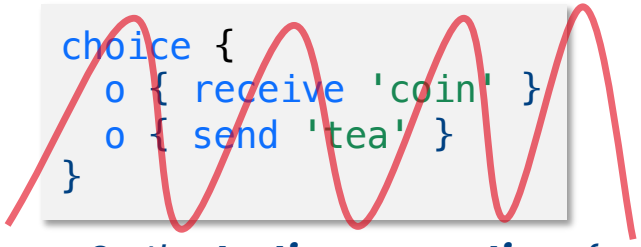
*AMP will **only wait to observe the response**, and will never offer the stimulus.*

Note that AMP' Explorer allows to select the stimulus (fortunately in red).

```
choice {
  o { receive 'coin' }
  o { receive 'kick' }
}
```

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

```
choice {
  o { receive 'coin' }
  o { send 'tea' }
}
```



*So the **testing semantics** of the model is sometimes different from the **intuitive** and intended **behavior**.*

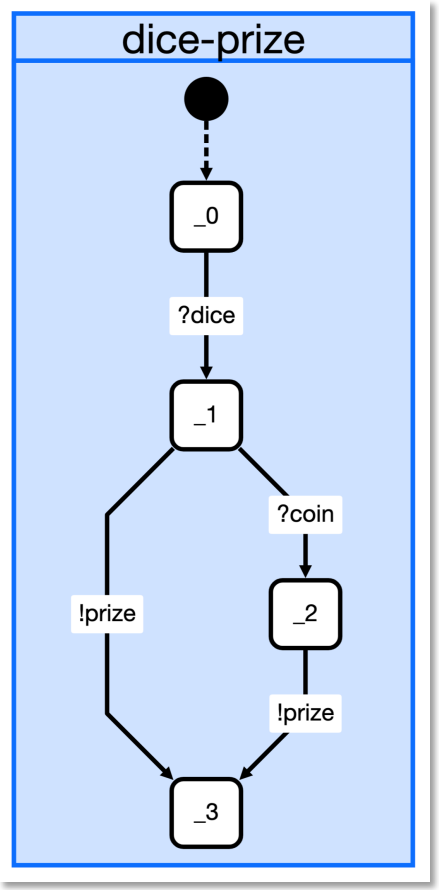
axini

stimulus and response together?

- Another example of a **choice** with *stimulus* versus *response*.

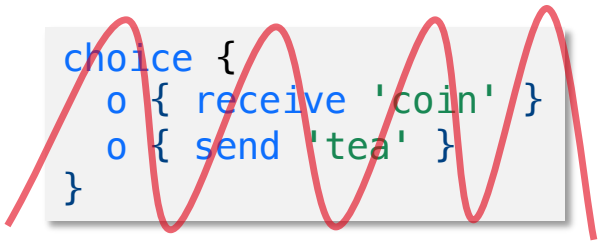
```
receive 'dice'
choice {
  o { receive 'coin'; send 'prize' }
  o { send 'prize' }
}
```

*We can **never be certain** whether the prize was the result of the dice, or the dice followed by the coin.*



stimulus and response together?

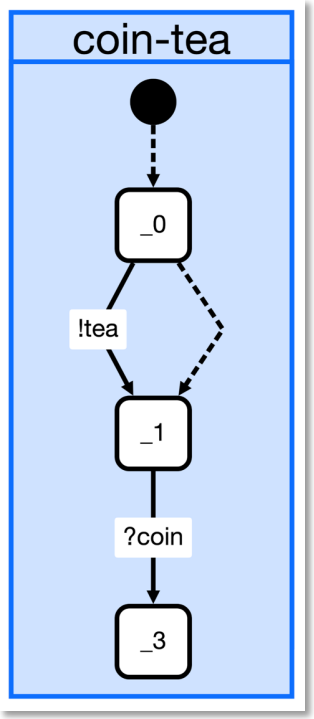
- How to allow **stimuli** and **responses** 'together'?



```
choice {
  o { receive 'coin' }
  o { send 'tea' }
}
```

```
choice {
  o { send 'tea', before: 10.0 }
  o { } # nothing
}
receive 'coin'
```

After waiting for 10 seconds, AMP concludes that 'tea' will not come, and offers the coin.



- This happens regularly, so there is an AML construct for it: **optionally**.

```
choice {
  o { <fragment> }
  o { } # nothing
}
```

```
optionally { <fragment> }
```

axini

optionally and stimulus

- `optionally { stimulus }` before **stimulus**.

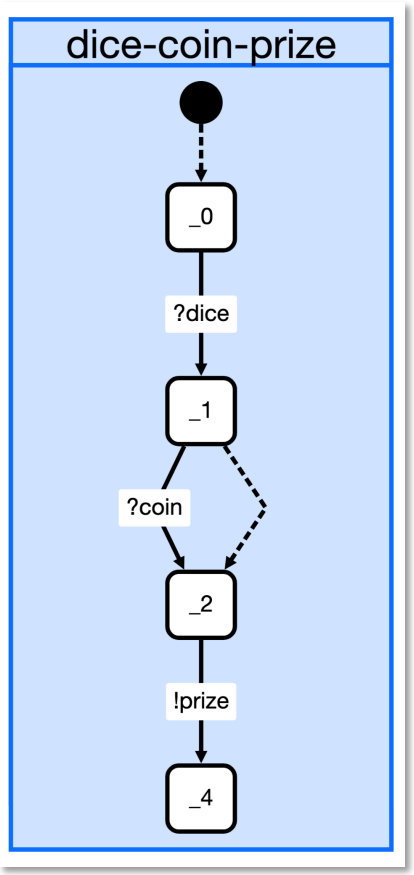
```
optionally { receive 'button_sugar' }
receive 'button_coffee'
send 'coffee'
```

✅

- `optionally { stimulus }` before **response**.

```
receive 'dice'
optionally { receive 'coin' }
send 'prize'
```

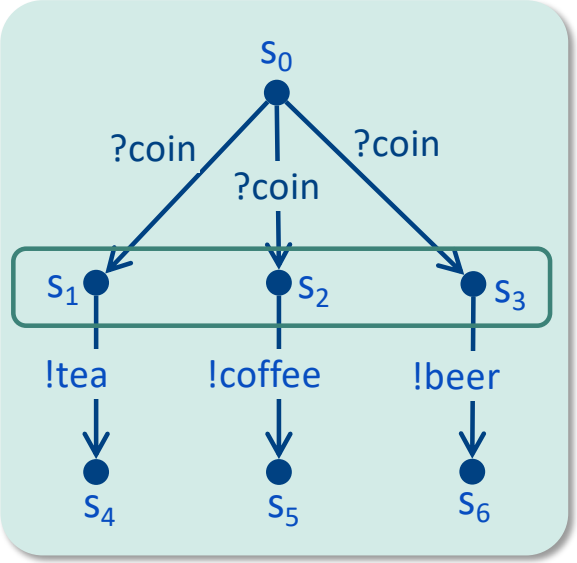
*Again a **choice** between a stimulus and a response.*
*AMS will **never try** to receive the stimulus.*



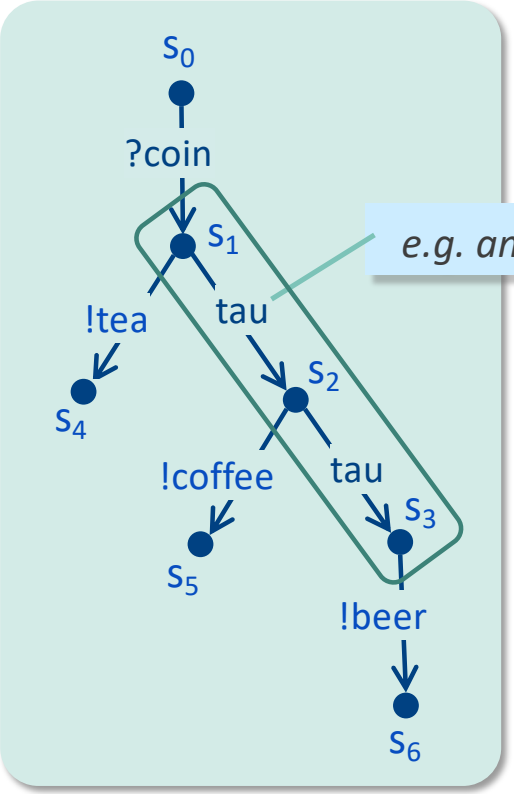
axini

Postponing the choice

only in case of non-determinism



*After the ?coin, the system can be in **three states**: s_1, s_2 and s_3. When the beverage is observed, the actual choice is resolved.*



*AMP's engine will **advance over tau steps** until a state where it expects an **external label**.*

AML features to prioritize stimuli

*Models typically define **infinite behavior** which could run forever.*

- **parallel processes**
  - labels in different processes are considered **independent**: they can not influence each other

The SUT has to be kept alive.

```
process('stimuli') {
  channel('external') { stimulus 'button' }
  repeat {
    o { receive 'button' }
  }
}

process('responses') {
  channel('external') { response 'light' }
  repeat {
    o { send 'light' }
  }
}
```

*AMP's **strategies** will consider both labels when choosing the next action.*

axini

AML features to prioritize stimuli (2/4)

- **response filter**
  - responses can be marked as **non-important**:
    AMP will **never wait** for these responses
    (e.g., heartbeats from the SUT)

```
response_filter ['heartbeat']
```

Can also be configured on the "Configure" page.
*But in the **model** is **preferred**!*

```
process('stimuli') {
  channel('external') {
    stimulus 'coin'
    response 'heartbeat'
  }
  repeat {
    o { receive 'coin' }
    o { send 'heartbeat' }
  }
}
```

*AMP's strategies will **never choose to wait** for the 'heartbeat' (but would accept it when it is observed).*

Without the response filter, AMP would never consider the 'coin' (due to ioco).

axini

AML features to prioritize stimuli (3/4)

- **duplex channel**
  - to model **full-duplex** connections
  - **no difference** between stimuli and responses

```
external 'duplex', type: :duplex

process('duplex-process') {
  channel('duplex') {
    stimulus 'button'
    response 'light'
  }

  repeat {
    o { receive 'button' }
    o { send 'light'     }
  }
}
```

*We are **not longer ioco**: test cases are sometimes harder to analyse.*

***Be careful**: models are harder to understand if one is used to **ioco semantics**.*

axini

AML features to prioritize stimuli (4/4)

- **expedited stimuli**
  - stimuli that **must happen** before a certain time
  - needed to keep the SUT alive (heartbeats **to** SUT)

```
external 'external'

process('expedited') {
  channel('external') {
    stimulus 'still_alive'
    response 'response'
  }

  repeat {
    o { receive 'still_alive',
        expedited: true, before: 5.0 }
    o { send 'response' }
  }
}
```

*Having all these language features in AML is **powerful** (and **needed**!), but **confuses** our users.*

*AMP will never choose `?still_alive`, until its deadline is reached: then AMP will **immediately** inject the stimulus.*

*See also documentation on "**expedited cutoff**".*

axini

axini

See also AML Tutorial

Some more of AML

behaviors

Theo Ruys

Behaviors

*Due to the **Ruby DSL limitations**, the syntax of parameters and return value is awkward.*

- Behaviors are the '**methods**' of AML: the building blocks with 'behavior' of AML. They define a **STS**.
  - **terminating**: when the terminating behavior ends, control is switched back to the caller.
  - **non-terminating**: fancy 'goto'.

*Very useful for **vizialisation**.*

- **Parameters** of behaviors: **pass by value**
  - only terminating behaviors can return values

*Behaviors can use **variables** and **labels** of the **process**.*

```
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>]
behave_as non_terminating_name [, [<expr>, ...]]
```

Behaviors – example

See AML Tutorial for an in-depth explanation.

```
process('behaviors') {
  channel('machine') {
    stimuli 'coin', 'btea', 'bcc
    responses 'tea', 'coffee'
  }

  # ... definition of behaviors

  behave_as 'main'
}
```

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

```
behavior('press_button', :terminating, [], :string) {
  choice {
    o { receive 'btea';    exit_with "'tea'"    }
    o { receive 'bcoffee'; exit_with "'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'
}
```

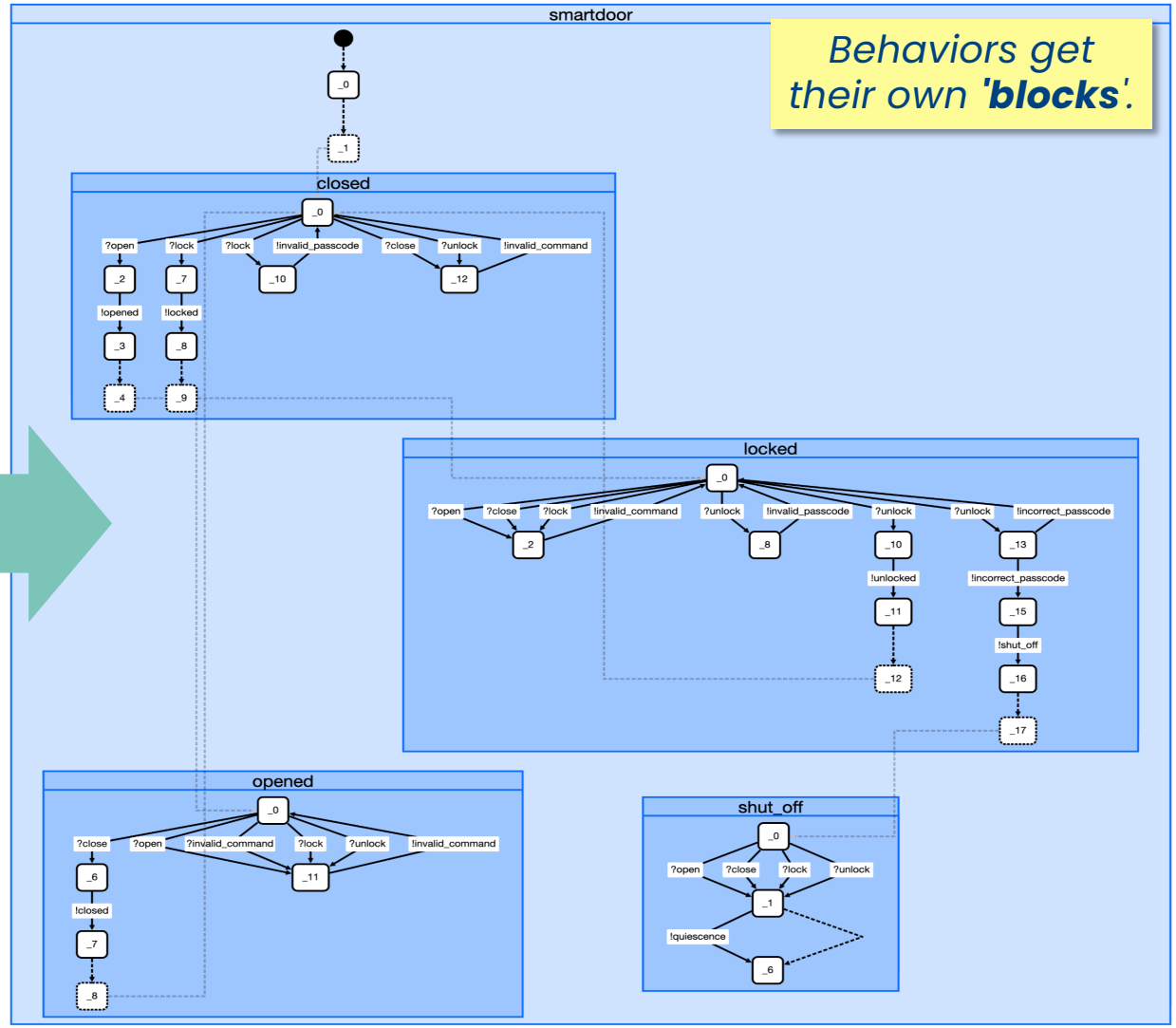
endless loop

```
behavior('insert_coin') {
  receive 'coin'
}
```

Terminating, no parameters, no return type.

Behaviors improve visualisation

```
external 'door'
process('smartdoor') {
  timeout 0.5
  include 'channel.aml'

  var 'entered_passcode', :integer
  var 'unlock_passcode', :integer
  var 'attempts', :integer, 0

  behavior('opened', :non_terminating) {
    # as state 'opened'
  }

  behavior('closed', :non_terminating) {
    # as state 'closed'
  }

  behavior('locked', :non_terminating) {
    # as state 'locked'
  }

  behavior('shut_off', :non_terminating) {
    # as state 'shut_off'
  }

  behave_as 'closed'
}
```



axini

See also AML Tutorial

Some more of AML

internal synchronisation

Theo Ruys

Internal actions – tau

***Internal actions** do **not** appear in **test case**.*

- **observable** actions
  - send, receive on **external** channels

***Only** observable actions end up in the **test cases**.*

- **internal** actions
  - **constraint** statement `contraint 'x > 3'`
  - **update** statement `update 'n = n+1'`
  - **goto** statement `goto 'loop'`
  - **internal communication** between processes

***Standalone** constraint and update **without** ':' !*

- execution of **parallel processes** is **interleaved**
  - AMP *'chooses'* one of the available actions from all defined processes.

Beverage machine with display (1)

single process

```
external 'beverages'
external 'display'

process('main') {
  timeout 2.0

  channel('beverages') {
    stimulus 'coin',
      { 'value' => :integer }
    responses 'tea', 'coffee'
  }

  channel('display') {
    response 'total',
      { 'value' => :integer }
  }

  var 'total', :integer, 0
```

Two external channels

```
repeat {
  o {
    receive 'coin',
        constraint: 'value == 50',
            update: 'total += value'
    choice {
      o { send 'tea'; send 'total', constraint: 'value == total' }
      o { send 'total', constraint: 'value == total'; send 'tea' }
    }
  }

  o {
    receive 'coin',
        constraint: 'value == 100',
            update: 'total += value'
    choice {
      o { send 'coffee'; send 'total', constraint: 'value == total' }
      o { send 'total', constraint: 'value == total'; send 'coffee' }
    }
  }
}
```

*A coin of 50 gives **tea**.*

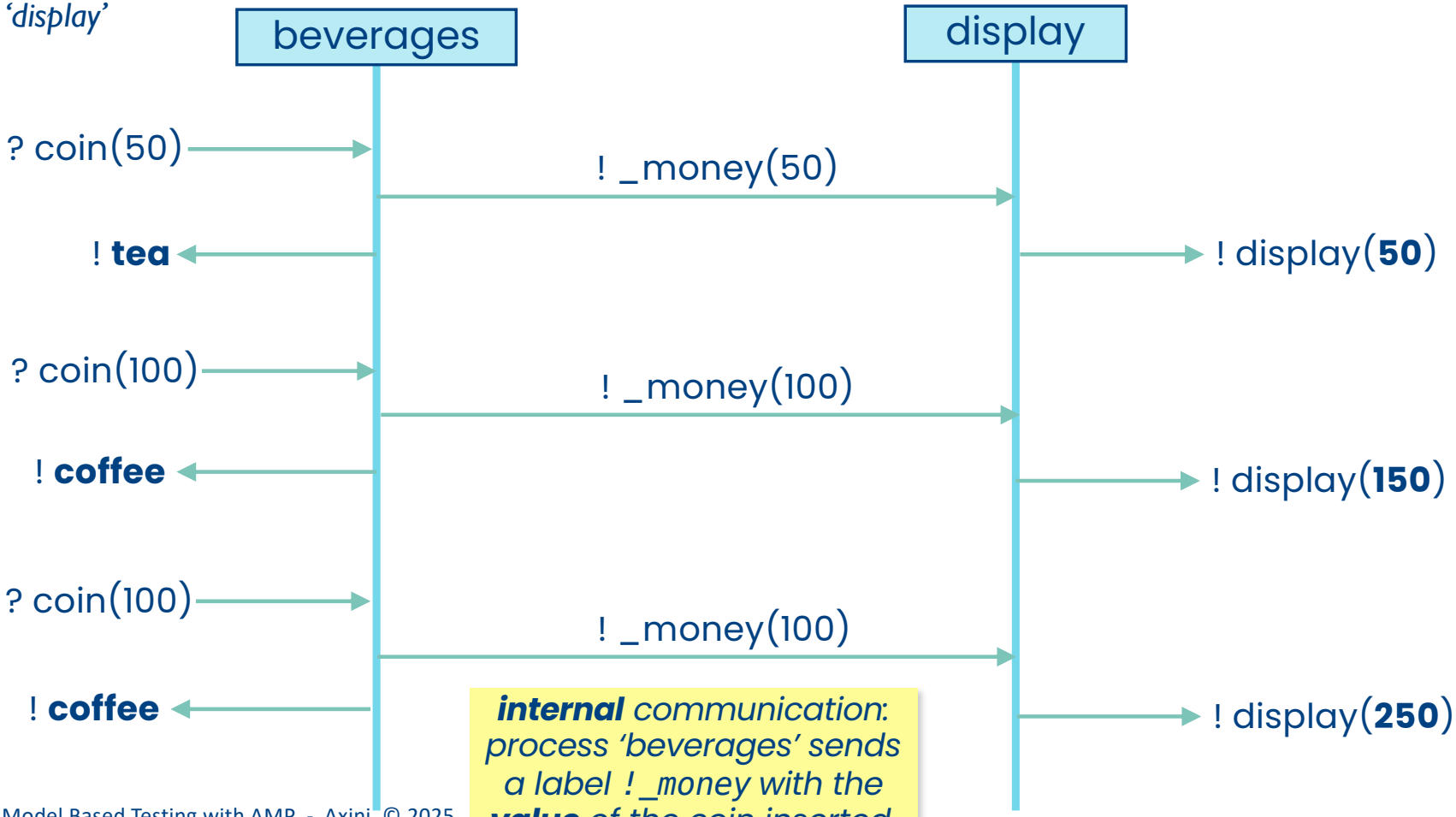
*A coin of 100 gives **coffee**.*

*The order of the responses ('tea' or 'total') is **not fixed**; so we have to be **ready** for **any order**.*

Beverage machine with display (2)

two processes

***Idea**: one process services 'beverages' interface, other process services 'display' interface.*



***internal** communication: process 'beverages' sends a label ! _money with the **value** of the coin inserted.*

Beverage machine with display (3)

two processes

```
process('beverages') {
  channel('beverages') {
    stimulus 'coin',
      { 'value' => :integer }
    responses 'tea', 'coffee'
  }

  channel('internal') {
    response '_money',
      { 'value' => :integer }
  }

  var 'coin_value', :integer, 0
```

*define **labels** of internal channel*

*Convention: we often let internal labels start with an **underscore**.*

Global definitions

```
timeout 2.0

external 'beverages'
external 'display'
internal 'internal'
```

*define **internal** channel*

Process 'beverages' continued.

```
repeat {
  o {
    receive 'coin',
        constraint: 'value in [50, 100]',
            update: 'coin_value = value'

    send '_money', constraint: 'value == coin_value'

    choice {
      o { send 'tea',    constraint: 'coin_value == 50'  }
      o { send 'coffee', constraint: 'coin_value == 100' }
    }
  }
}
```

***sender** must fill the label parameters.*

Beverage machine with display (4)

two processes

Process 'beverages' repeated.

```
repeat {
  o {
    receive 'coin',
        constraint: 'value in [50, 100]',
            update: 'coin_value = value'

    send '_money', constraint: 'value == coin_value'

    choice {
      o { send 'tea',    constraint: 'coin_value == 50'
      o { send 'coffee', constraint: 'coin_value == 100'
    }
  }
}
```

```
process('display') {
  channel('display') {
    response 'total',
      { 'value' => :integer }
  }
  channel('internal') {
    stimulus '_money',
      { 'value' => :integer }
  }

  var 'total_money', :integer, 0

  repeat {
    o {
      receive '_money', update: 'total_money += value'
      send 'total', constraint: 'value == total_money'
    }
  }
}
```

***receiver** process also defines the labels*

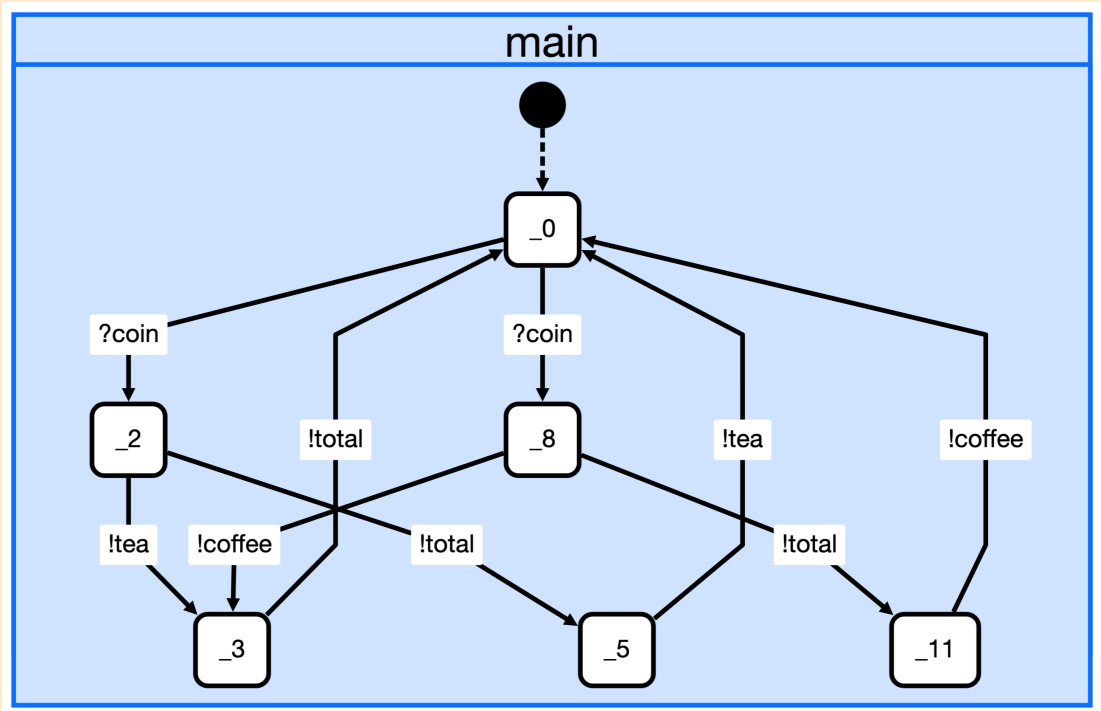
*Communication happens in one **single tau-step**.*

***receiver** gets label with parameters*

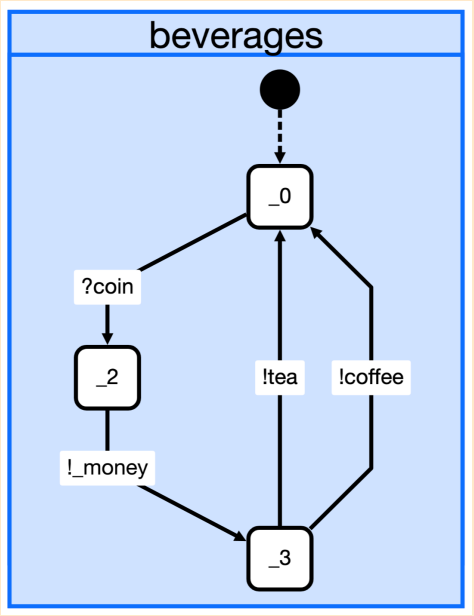
*The direction of **external** communication is from the point of view of the **SUT**. The direction of **internal** communication is from the point of view of the **processes**.*

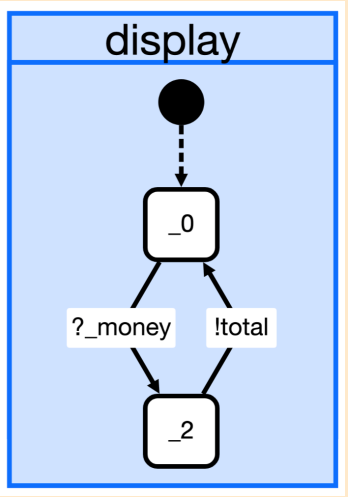
Beverage machine with display (5)

single process



two processes





Internal Communication / Synchronisation

- processes can communicate **pairwise**
- **internal** channel: defined outside processes
- synchronized communication: **handshake**, a single step
- sending and receiving process use **same label parameters**
  - sending process supplies the values for the parameters
  - receiving process can use these values.
- internal communication is an **internal action** (tau)
  - does **not** show up **in test case**
  - may happen; but can also be postponed
- **notes** can be used: will be attached to the **last** observable action

axini

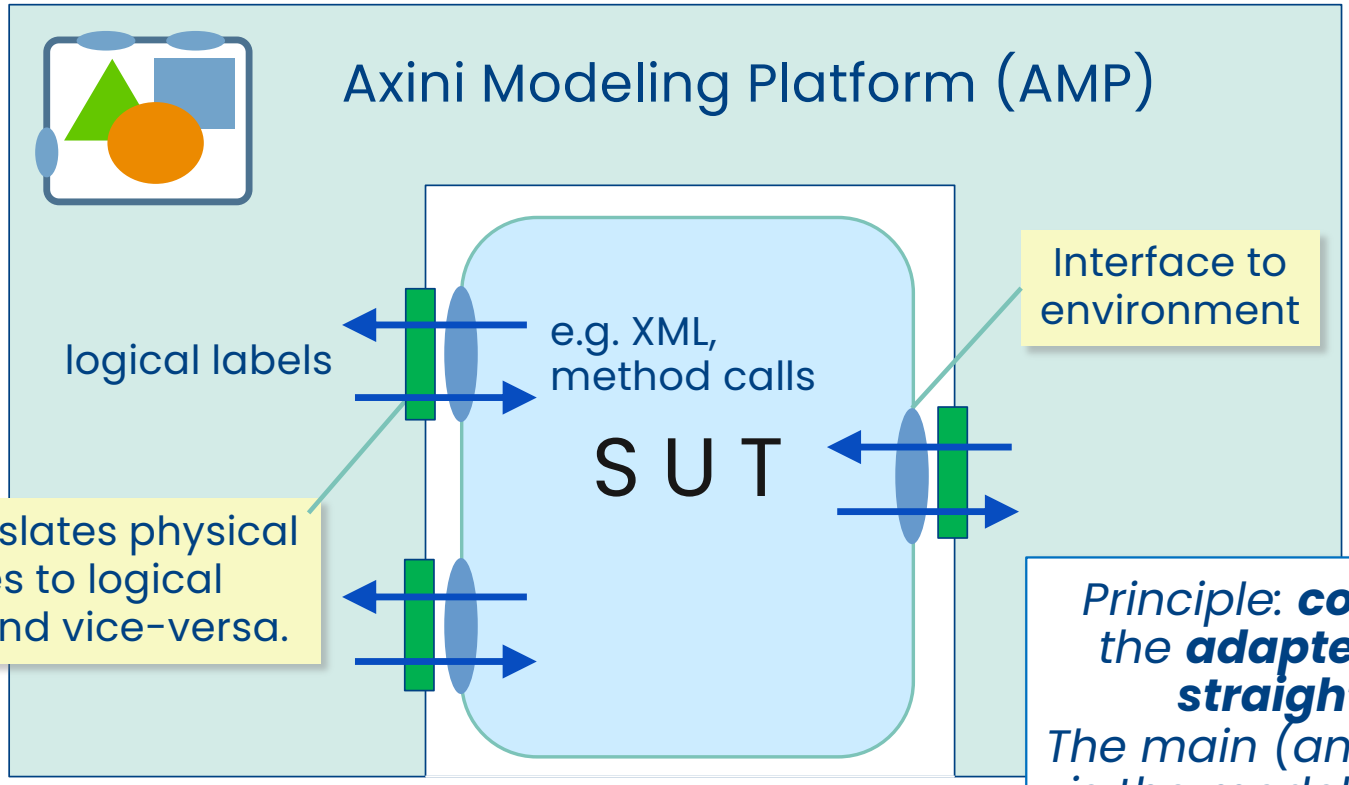
Plugin Adapters

Status on Tue, April 1 10:00h:
- half of the groups have connected with SmartDoor PA adapter
- of these groups, 85% used Python

Theo Ruys

AMP: horseshoe around SUT

REVISIT DAY 1



*Principle: **construction** of the **adapters** should be **straightforward**.*
The main (and difficult) task is the modeling of the SUT.

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

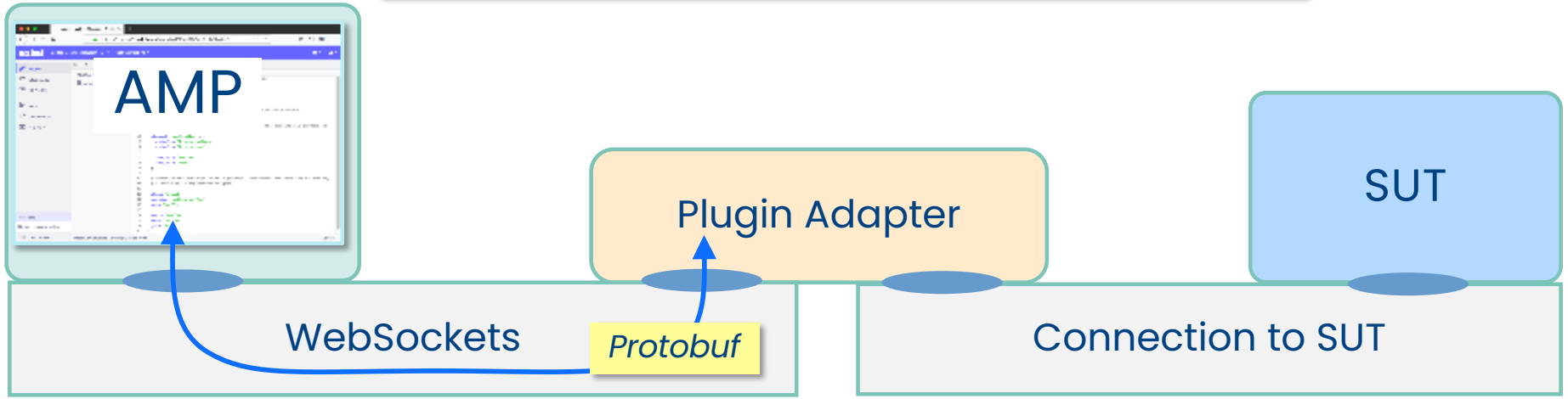
axini

AMP – Plugin Adapters

REVISIT DAY 1

*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**.*

WebSockets? Protobuf?

*For both WebSockets and Protobuf **implementations** are available for **many programming languages**.*

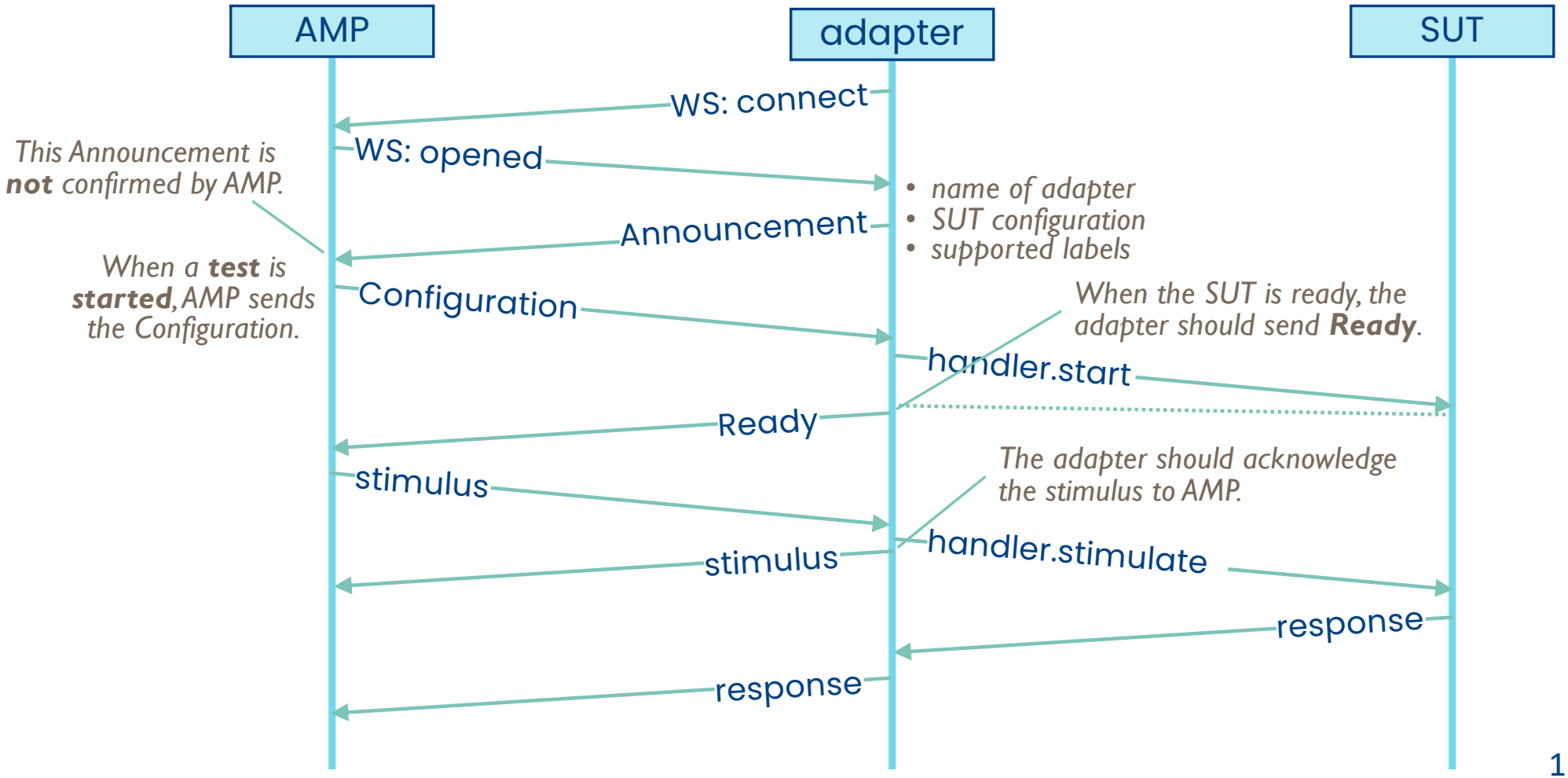
- **WebSockets** https://datatracker.ietf.org/doc/html/rfc6455
  - **full-duplex** communication protocol
  - over a **single** TCP connection, on top of **HTTP(S)**
  - **text** or **binary** transmission
  - programming **libraries** abstract from low-level details
  - uniform resource identifier (URI): ws & wss
    - `ws://localhost:3001/` *Default URI of standalone SmartDoor SUT*
    - `wss://course02.axini.com:443/adapters` *AMP@course02*

- Google **Protocol Buffers** (aka **protobuf**)
  - data format to **serialize structured data**
    - .proto: description language describing the data
  - open-source, cross-platform, language independent

https://developers.google.com/protocol-buffers/

***Axini** has defined these for the **PluginAdapter** protocol.*

AMP – Adapter – SUT



Plugin Adapter Protocol

Only the connection with AMP!

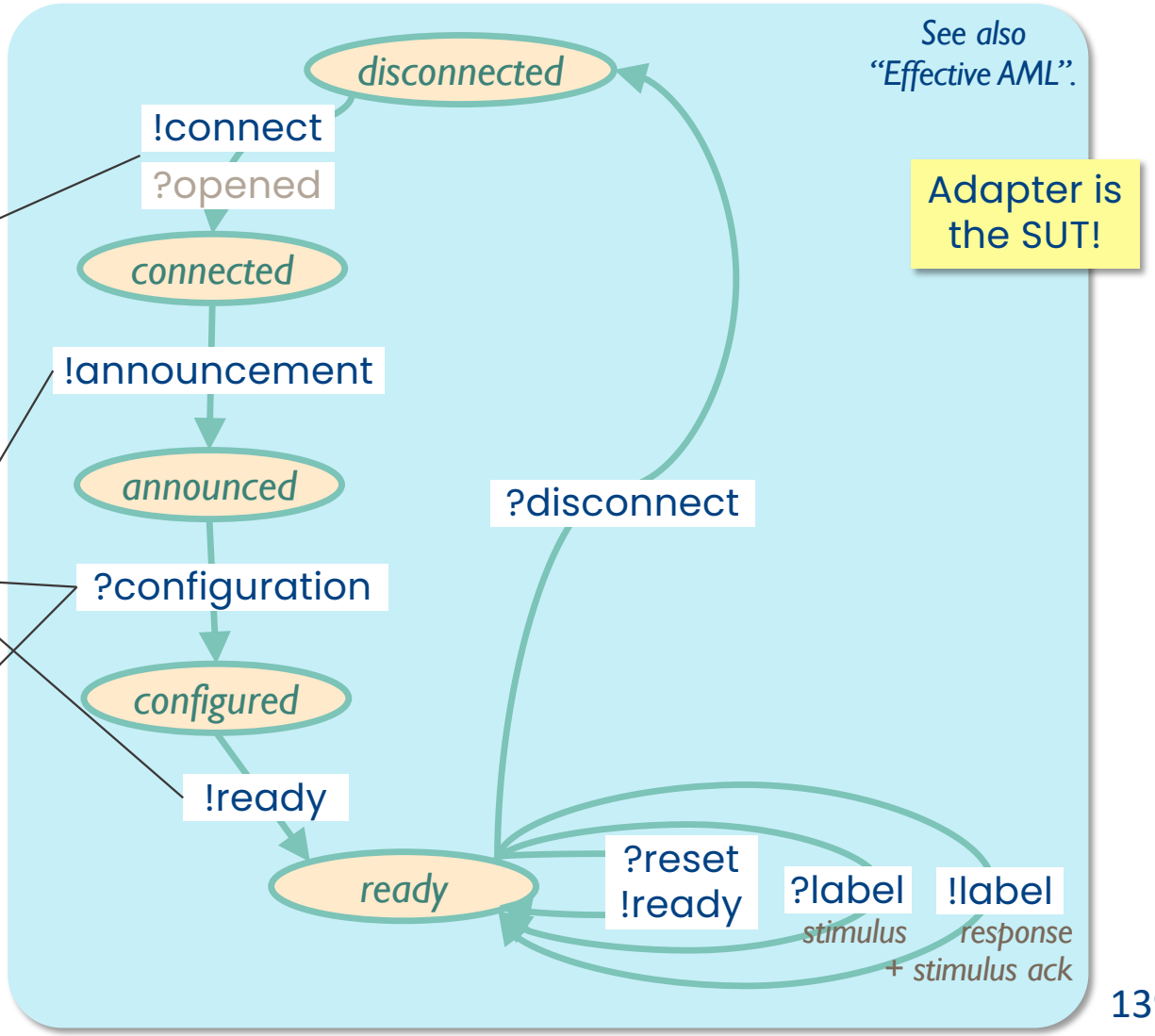
WebSocket

Protobuf messages

*AMP will send the **configuration** when the **test run starts**.*

*There is an **AML model** of the protocol, which has been **verified** with the **SPIN** model checker.*

*State-Transition diagram for the **plugin adapter**.*



Protobuf

https://developers.google.com/protocol-buffers

*Excellent **tutorials** for many programming languages.*

- Protocol Buffers (**Protobuf**): serializing structured data
  - language neutral (C++, C#, Go, Java, Python, Ruby, etc.)
  - platform neutral (Windows, macOS, Linux)
  - free, open-source
  - alternative to JSON, XML
- Forwards **compatible**: never change, only add fields.
- Fast processing, **efficient** binary encoding.
- Message are described by a schema: **.proto** file.
  - Source code in target language is **generated** with **protoc**. E.g.
    - **create** Protobuf objects
    - **unpack** Protobuf objects

*The means to manipulate Protobuf objects is **different** for each **programming language**.*

.proto for Message

This is the message which is sent over the WebSocket connection, in binary form.

```
message Message {
  message Reset {}
  message Ready {}
  message Error {
    string message = 1;
  }

  oneof type {
    Error error = 1;
    Announcement announcement = 2;
    Configuration configuration = 3;
    Label label = 4;
    Reset reset = 5;
    Ready ready = 6;
  }
}
```

```
message Announcement {
  string message = 1;
  Configuration configuration = 2;
  repeated Label labels = 3;
}
```

Name of adapter (unique).

All labels supported by the adapter.

```
message Configuration {
  message Item {
    string key = 1;
    string description = 2;

    oneof type {
      string string = 3;
      int64 integer = 4;
      float float = 5;
      bool boolean = 6;
    }

    repeated Item items = 1;
}
```

.proto for Label

```
message Label {
  enum LabelType {
    STIMULUS = 0;
    RESPONSE = 1;
  }
  LabelType type = 1;

  string label = 2;
  string channel = 3;

  ...

  repeated Parameter parameters = 4;

  uint64 timestamp = 5;
  bytes physical_label = 6;
  uint64 correlation_id = 7;
}
```

Name of the label.

Adapter timestamp.

To correlate stimuli.

```
message Parameter {
  message Value {
    ...

    oneof type {
      string string = 1;
      int64  integer = 2;
      double decimal = 3;
      bool   boolean = 4;
      uint64 date = 5; // seconds
      uint64 time = 9; // nanoseconds

      Array array = 6;
      Hash struct = 7;
      Hash hash_value = 8;
    }

    string name = 1;
    Value value = 2;
```

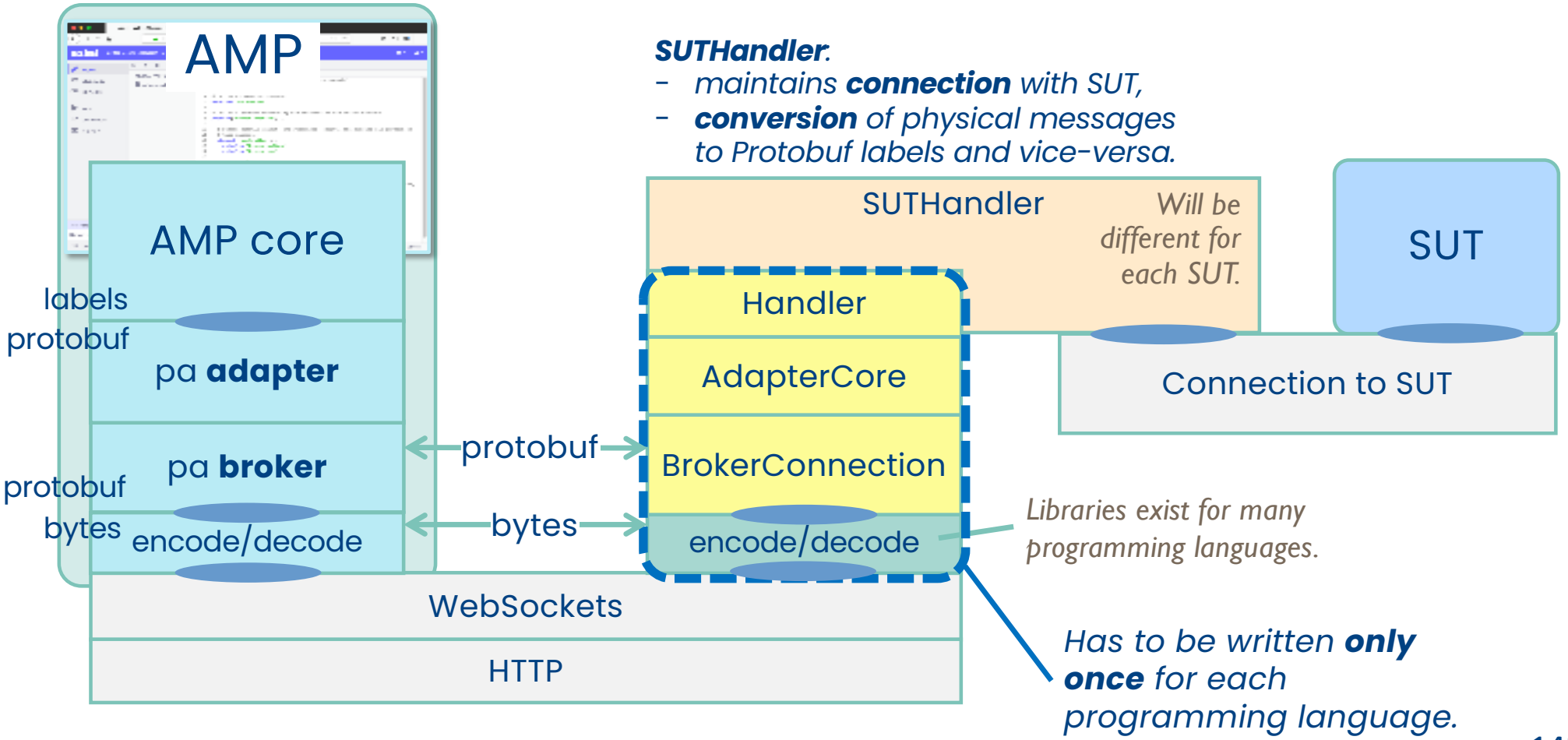
*Added **after** the initial version of the protocol.*

```
message Array {
  repeated Value values = 1;
}

message Hash {
  message Entry {
    Value key = 1;
    Value value = 2;
  }
  repeated Entry entries = 1;
}
```

*The **names** of the attributes are used for the **getters** and **setters** of the objects.*

AMP – Plugin Adapters

***Architecture** for Axini's plugin adapters.*



Axini Plugin Adapters

REVISIT DAY I

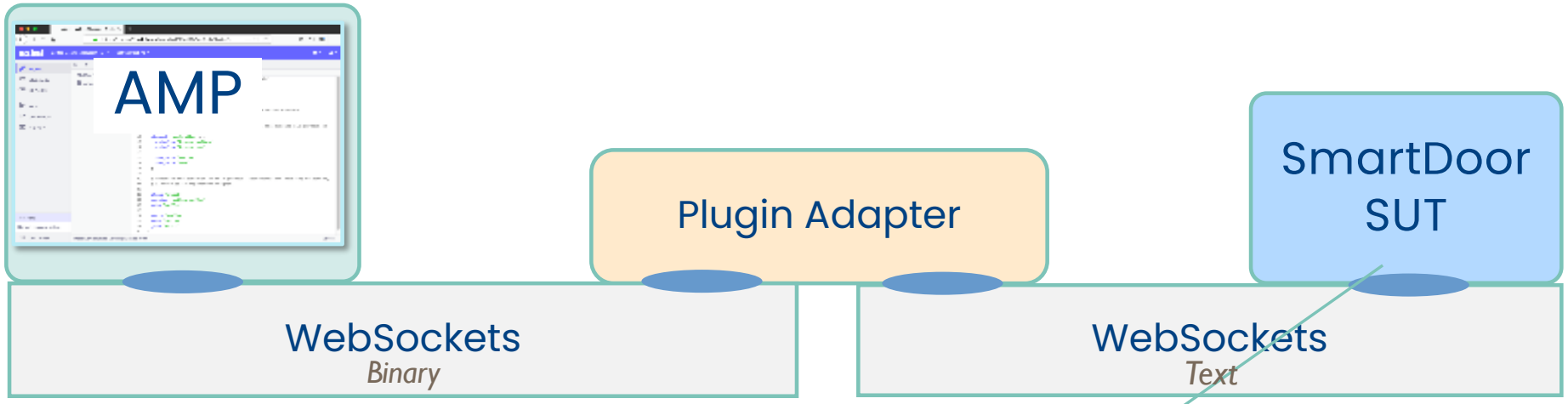
https://github.com/Axini

- **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**

*The **architectures** of these plugin adapters are **very similar.***

Adapter for SmartDoor SUT

REVISIT DAY 1



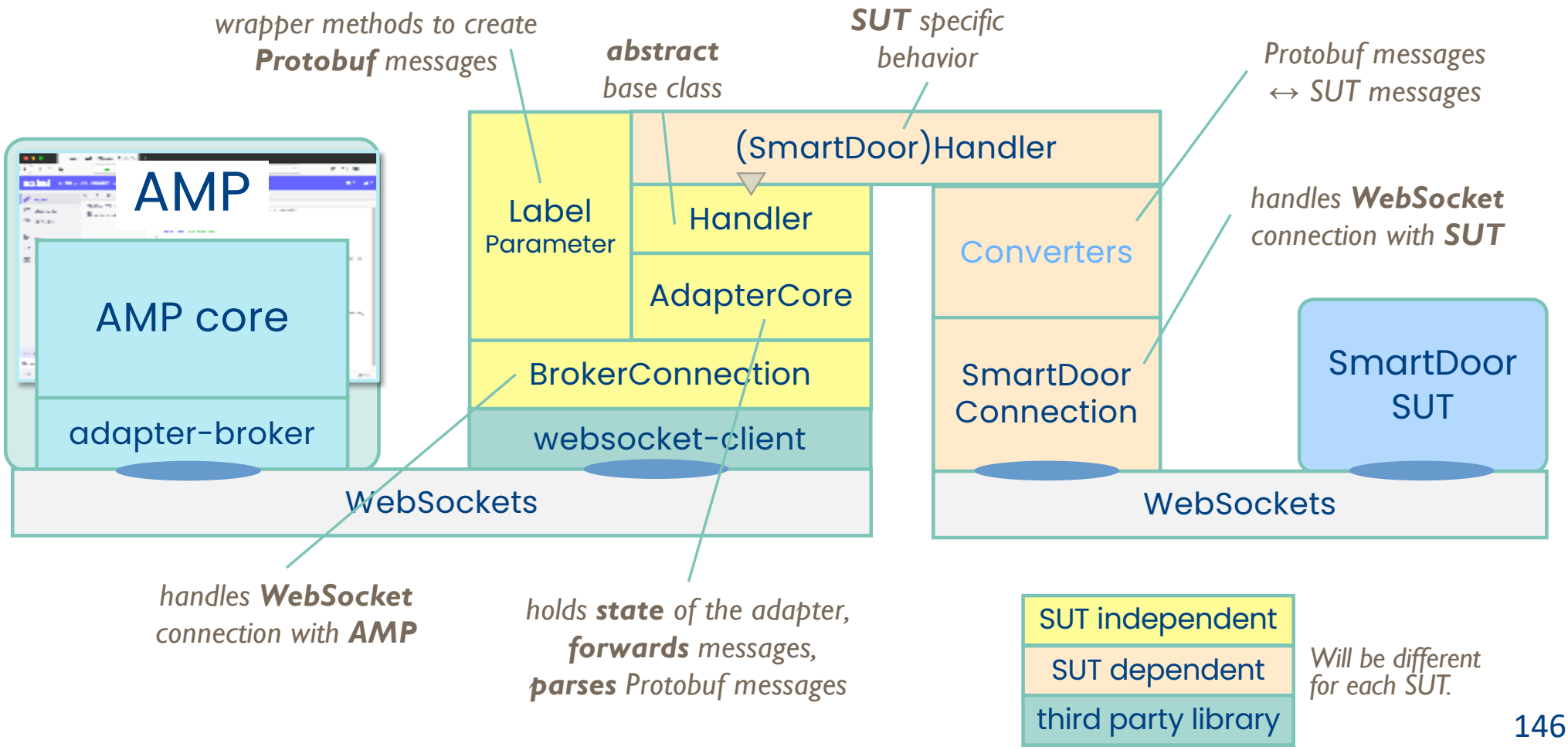
The SmartDoor SUT is a **standalone** SUT from the 'SmartDoor' laboratory exercise.

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

SmartDoor SUT talks in UPPERCASE!

AMP – Plugin Adapter [for SmartDoor SUT]

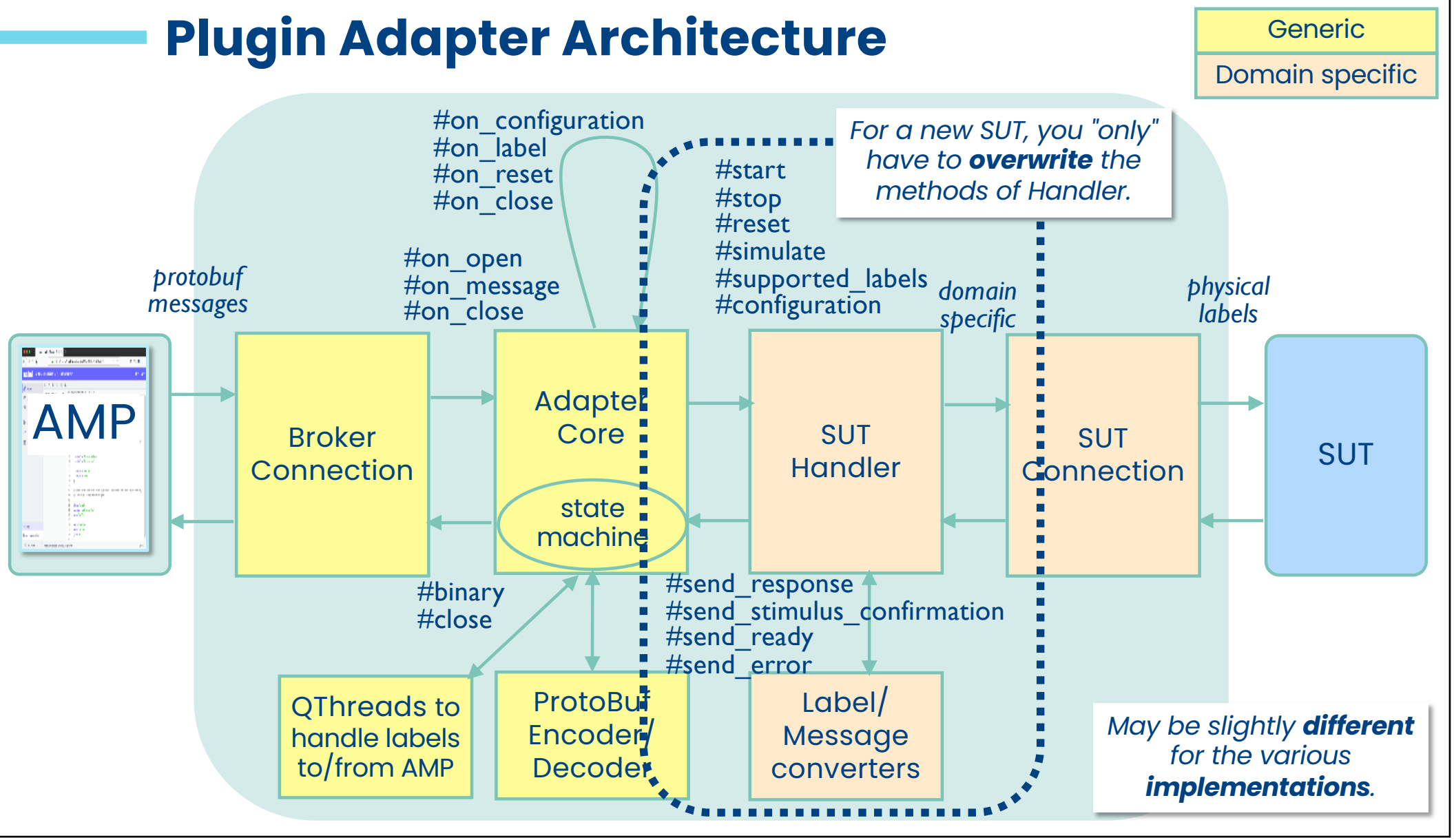
***Python** adapter*



Python adapter - ./src/adapter

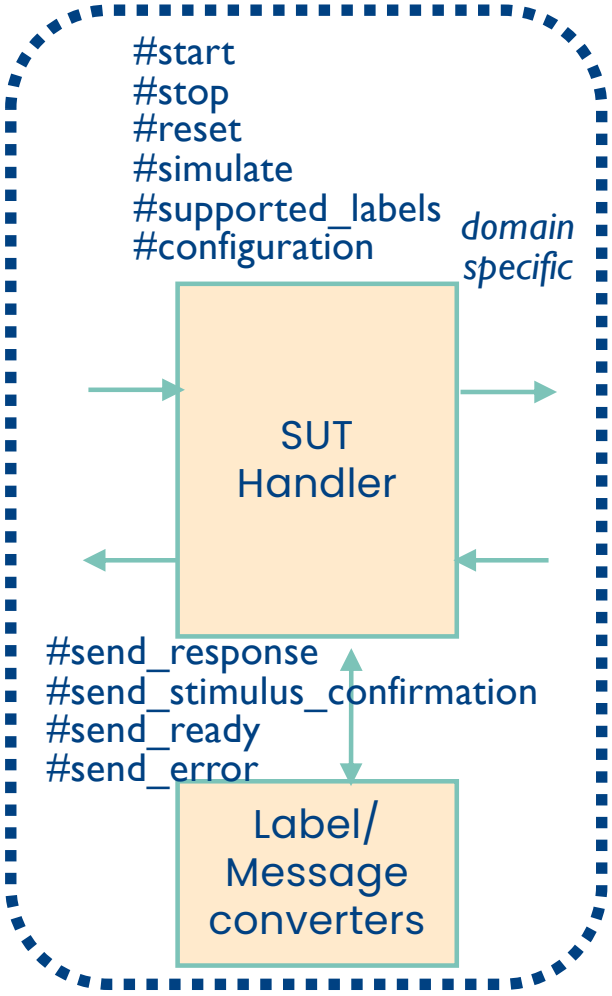
/generic			
	adapter_core.py	AdapterCore	core of the adapter (keeps state diagram)
	broker_connection.py	BrokerConnection	handles WebSocket connection with AMP
	handler.py	Handler	abstract base class for all Handlers
	qthread.py	QThread	thread which process items of a queue
/generic/api			
	*.proto		Protobuf description files (schema's)
	label.py	Label	methods to encode/decode Labels
	parameter.py	Parameter	methods to encode/decode Parameters
/smartdoor			
	handler.py	Handler	implementation of Abstract base class
	smartdoor_connection.py	SmartDoorConnection	handles connection with SmartDoor SUT

Plugin Adapter Architecture



class Handler

abstract *super class in* `./generic`
implementation *in* `./smartdoor`



AdapterCore *will* call **Handler**'s methods:

- `start`: **start** a new test case
- `stop`: **stop** the current test execution
- `reset`: **reset** the connection with the SUT
- `stimulate`: inject the **stimulus** into the SUT
- `supported_labels`: all **labels** supported
- `default_configuration`: the **default configuration** of this adapter (for AMP)
- `configuration`: current configuration (for AMP)

`stop()`
`start()`

needed for Announcement

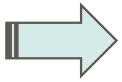
Handler can call **AdapterCore**'s methods:

- `send_response`: send a **response** to AMP
- `send_stimulus_confirmation`: send the **confirmation** of the stimulus to AMP
- `send_ready`: send **Ready** to AMP
- `send_error`: send **Error** to AMP

utility classes for Protobuf objects

***Python** adapter*

***Protobuf** objects are **tedious** to manipulate:*
e.g, label_pb.Label, label_pb.Label.Parameter



*Utility classes **`Label`**, **`Parameter`** and **`Configuration`** in `./generic/api`*

class Label	*./generic/api/Label.py*
+sort: Sort	
+name: str	
+channel: str	
+parameters: List[Parameter]	
+timestamp: datetime	
+physical_label: bytes	
+correlation_id: int	
+ctor()	constructs Label object from name, parameters, etc.
+encode()	returns Protobuf label_pb.Label object
+decode()	unpacks Protobuf label_pb.Label object

*The **attributes** of a Label object can simply be **set** and **read**.*

class method

de- & encoding Protobuf objects

Python adapter

for ***stimulus***

Decoding a Protobuf message

```
# raw_message: str

pb_message = message_pb2.Message() # empty

try:
  pb_message.ParseFromString(raw_message)
except Exception as e:
  logger.error('Could not decode ...')

if pb_message.HasField('label'):
  pb_label = pb_message.label
  label = Label.decode(pb_label)
```

AdapterCore: #_handle_message, #on_label, and (SmartDoor)Handler: #stimulate.

Now you can access the contents of the Label.

for ***response***

Encoding a Protobuf message

```
label = Label(
  sort=Sort.RESPONSE,
  name='opened',
  channel='door',
  physical_label=bytes('OPENED', 'UTF-8'),
  timestamp=datetime.now()
)
pb_label = label.encode()
pb_message = message_pb2.Message(label=pb_label)
```

(SmartDoor)Handler: #_message2label
AdapterCore: #send_response

Developing an adapter for NewsFeed

*Use the (connection) **scripts** that you used to test NewsFeed previously (manually or scripts) as a **basis** for the connection with the NewsFeed SUT.*

- Create `NewsFeedHandler`, a subclass of `Handler`.
- **Implement** the methods of `NewsFeedHandler`
  - `start`, `stop`, `reset`, `stimulate`, etc.
  - capture the responses from the NewsFeed SUT and send them to AMP
- Start with a **very basic AML model** (tracer bullet)
  - try **all stimuli** (but not all at once)
  - accept all responses (without checking the contents)
- When the adapter is working, you can start creating the **real model**, on the basis of the NewsFeed specification.

Similar to the coffee machine model, which just receives button presses, and accepts all beverages.

The laboratory "Roadmap" is written for the NewsFeed app built into the Axini server. Some aspects do not apply to the NewsFeed PA set-up.

axini

Conclusions

Theo Ruys

Day 2 – Tue 1 Apr

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

- **SmartDoor** exercise
  - understanding fails
  - discussions on possible solutions
- Some more **AML**
  - choices
  - behaviors
  - internal communication
- **Plugin Adapter**
  - general design
  - plugin adapter for **SmartDoor** SUT in **Python**

*We are always looking for enthusiastic students who want to their **MSc project** at Axini!*

https://www.axini.com/en/students/