Abstract State Machines

Angelo Gargantini

2023

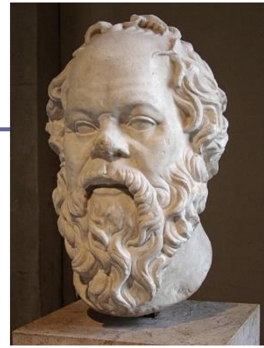
Testing e verifica del sw

Scenario-based validation

Motivations

- **Validation:** investigating a model with respect to its user perceptions, in order to ensure that the it really satisfies the user needs
  - detect faults as early as possible
  - possible techniques: scenarios generation, development of prototypes, animation, simulation, and also testing
- **Scenario**: description of a possible behavior of the system
  - observable interactions between the system and its environment in specific situations

A philosophical view



MODEL — *All men are mortal.*

VERIFICATION — *Socrates is a man **implies** Socrates is mortal.*

Model checking and similar techniques

Scenario-based VALIDATION — *Find Socrates, check he is a man, and he is dead*

testing

J Gen Philos Sci (2008) 39:85–113
DOI 10.1007/s10838-008-9068-7

DISCUSSION

Towards a Philosophy of Software Development: 40 Years after the Birth of Software Engineering

Mandy Northover · Derrick G. Kourie · Andrew Boake · Stefan Gruner · Alan Northover

"the susceptibility of a formal specification to scenario-based validation demonstrates its **falsifiability**, and thus the scientific nature of software development."

Related works 1

- From telecommunication systems
  - Message Sequence Charts (MSCs) (graphical)
  - Life Sequence Charts (LSCs)
    - W. Damm and D. Harel. *LCSs*: *Breathing life into message sequence charts*: extends the MSCs by providing the "clear and usable syntax and a formal semantics" MSCs lack of.
- UML Use cases
  - black box view – graphical notation
- Temporal Logical/ formal methods
    - Albert II formal language and scenarios are represented by MSC

Related works 2

- For ASM
  - W. Grieskamp, N. Tillmann, and M. Veanes. *Instrumenting scenarios in a model-driven development environment*, 2004.
  - Spec# specifications are instrumented to allow validation
    - E.g. "to describe observations in scenarios, we extend Spec# by the so-called **`expect`** statement"
- **For a survey:**
  DANIEL AMYOT
  An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development

Our proposal in ASMETA – ABZ08

International Conference on Abstract State Machines, B and Z

↳ ABZ 2008: **Abstract State Machines, B and Z** pp 71–84 | Cite as

A Scenario-Based Validation Language for ASMs

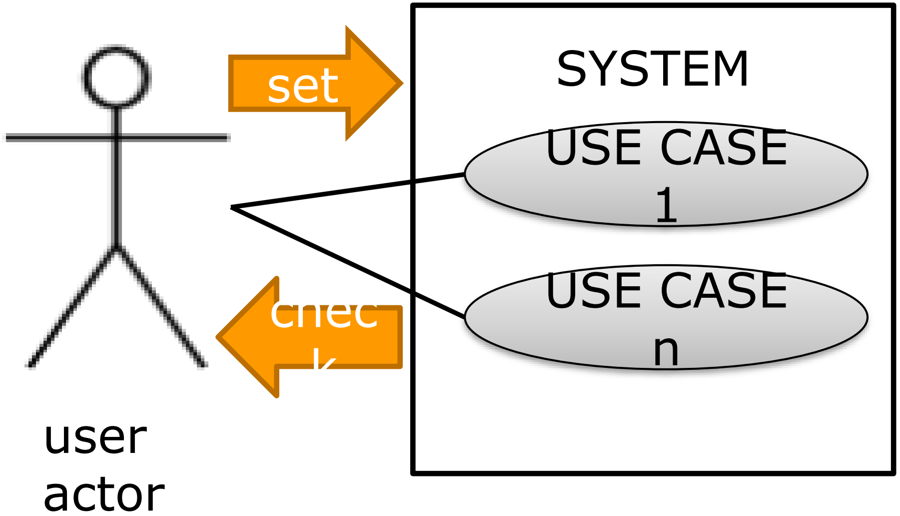
Alessandro Carioni, Angelo Gargantini, Elvinia Riccobene & Patrizia Scandurra

Goals

- **Textual** notation
  - Similar to programs (as Spec#)
- Clear **semantics**
  - As LSC (e.g. clear definition of necessary and possible)
  - defined by ASMs ?
- Able to describe **internal** details
  - Not only black box as UML use cases
- Similar to testing notations?
  - Like Use case maps similar to TTCN
- To validate ASM written in AsmetaL
  - To be integrated within the ASMETA framework

From UML Actor to ASM Actor

- UML USE CASE: actor interacts with the system.
- One or more scenarios may be generated from each use case
- <u>BLACK BOX VIEW</u>

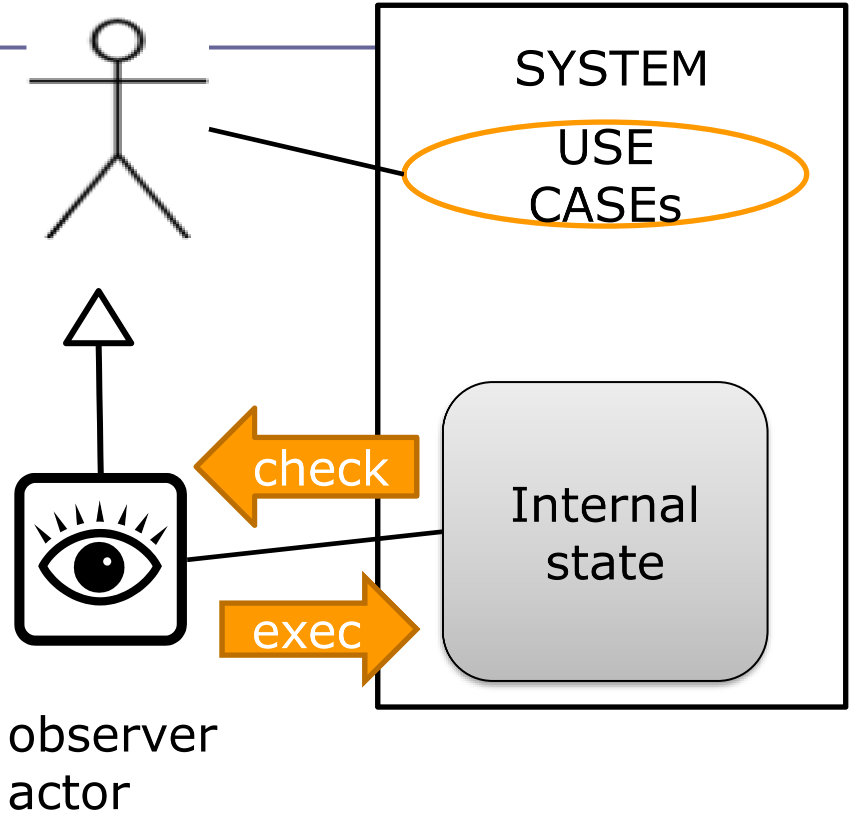


ASM Actor

- **set**s monitored and shared functions (environment)
- **check**s out functions (machine reaction)

ASM observer

- ASM observer
  - **check**s machine internal state and **invariants**
  - requires the **exec**ution of arbitrary rules
  - GRAY BOX VIEW



Twofold use of scenarios

- Two kinds of external actors:
  - user, who has a **black** box view of the system
  - observer, who has a **gray** box view
- Two goals for scenarios
  - classical validation
    - user actions and machine reactions
  - testing activity
    - observer inspection of the internal state of the machine

ASM scenario

- interaction sequence consisting of actions:
- by observer
  1. set the environment (i.e. the values of monitored/shared functions)
  2. check for the machine outputs (i.e. the values of out functions),
  3. check the machine state and invariants
  4. ask for the execution of given transition rules
- by machine
  - makes one step as reaction of the actor actions
- written in

Asm **Val**idation **La**nguage → AValLa

AValLa primitives

set	A command to set the location of a (monitored) function to a specific value: it simulates the **environment**
check	To inspect external values and (only for the observer) to inspect internal values in the current state
step	To signal that the environment has finished to update the monitored locations, hence the machine can perform a step
step until	To signal that the machine can perform a step iteratively until a specified condition becomes true
invariant	To state critical specification properties that should always hold for a scenario
exec	To execute transition rule when required by the observer

USING AVALLA

Advanced Clock

- Advanced clock:
  - A clock with seconds, minutes, and hours
  - At every step the second is incremented by 1

Scenario 1

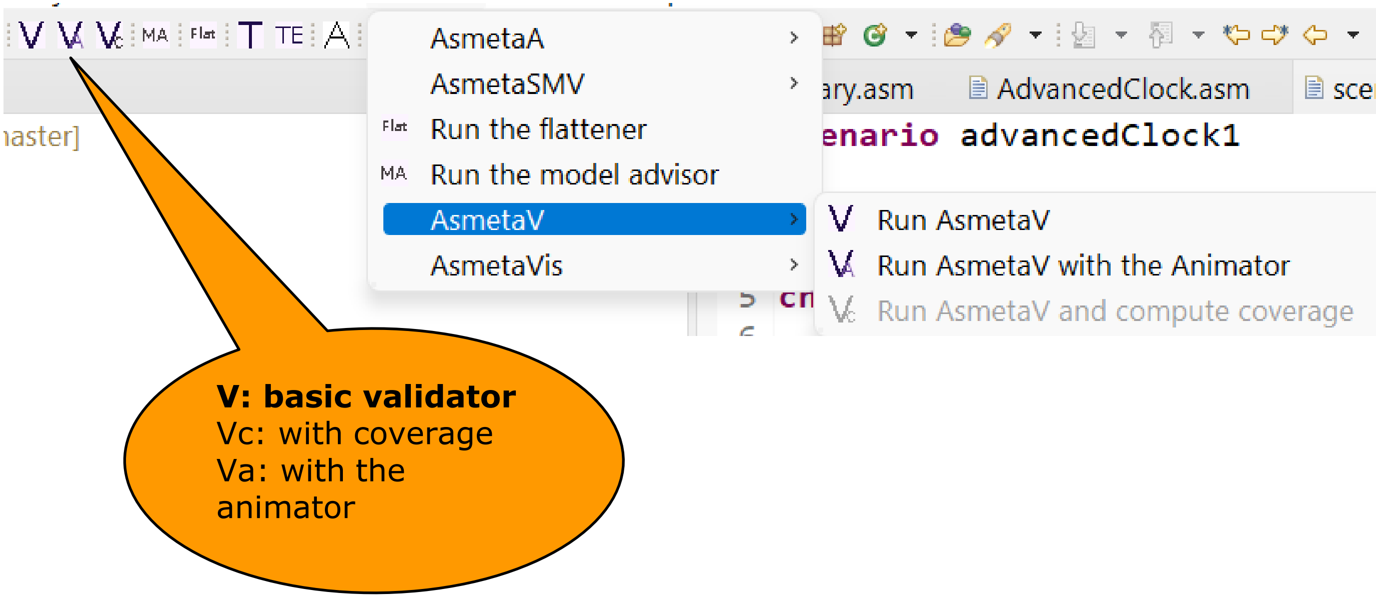
- Check that at the beginning the clock is at midnight (00:00:00);
- Perform a step of the machine
- Check now that the time is 00:00:01;
- Another step
- Check now that the time is 00:00:02;

Scenario 1

asm to validate	`scenario advancedClock1` `load AdvancedClock.asm`
Check initially is 00:00:00	`check hours = 0 and minutes = 0 and seconds = 0;`
Perform a step	`step`
Check that now is 00:00:01	`check hours = 0 and minutes = 0 and seconds = 1;`
Perform a step	`step`
check again	`check hours = 0 and minutes = 0 and seconds = 2;`

```
scenario advancedClock1

load AdvancedClock.asm

check hours = 0 and minutes = 0 and
seconds = 0;

step

check hours = 0 and minutes = 0 and
seconds = 1;

step

check hours = 0 and minutes = 0 and
seconds = 2;
```

How to execute the scenario



Advanced Clock

- Scenario 1 can be very long
- Scenario 2:
  - After one hour
  - exec step by step until the hour is 1
  - Check that now it is 01:00:00
  - Step
  - Check that now it is 01:00:01

Scenario 2

```
// using step until
scenario advancedClock2

load AdvancedClock.asm

check hours = 0 and minutes = 0 and seconds = 0;

step until hours = 1;

check hours = 1 and minutes = 0 and seconds = 0;
```

Scenario 3 - invariants

- Invariants can be used to check
  - Properties generic to the machine are always guaranteed
  - Properties for the scenario are true:

- Scenario 2:
  - Seconds are always lower than 60
    - For every execution
  - Hours are lower of equal 1
    - Specific of the scenario
- Let’s execute the scenario
- Let’s check that if both invariants are actually checked

Scenario 4 – using exec

- Sometimes we want to change the state of the machine directly
  - against information hiding
  - improve the testability without changing the visibility
- Scenario 4
  - Set with a par rule 02:01:59;
  - Perform a step of the clock
  - Check now that now is 02:02:00;

Introducing monitored functions

- Till now Advanced Clock is a closed system
  - Its behavior does not depend on the environment
  - **Closed system**

- **Second version:**
  - The clock is connected to signal and the seconds are incremented only if signal is true

Scenarios

- Scenario 5:
  - **Set** signal true and check that seconds is incremented
  - **Set** signal false and check that seconds is NOT incremented

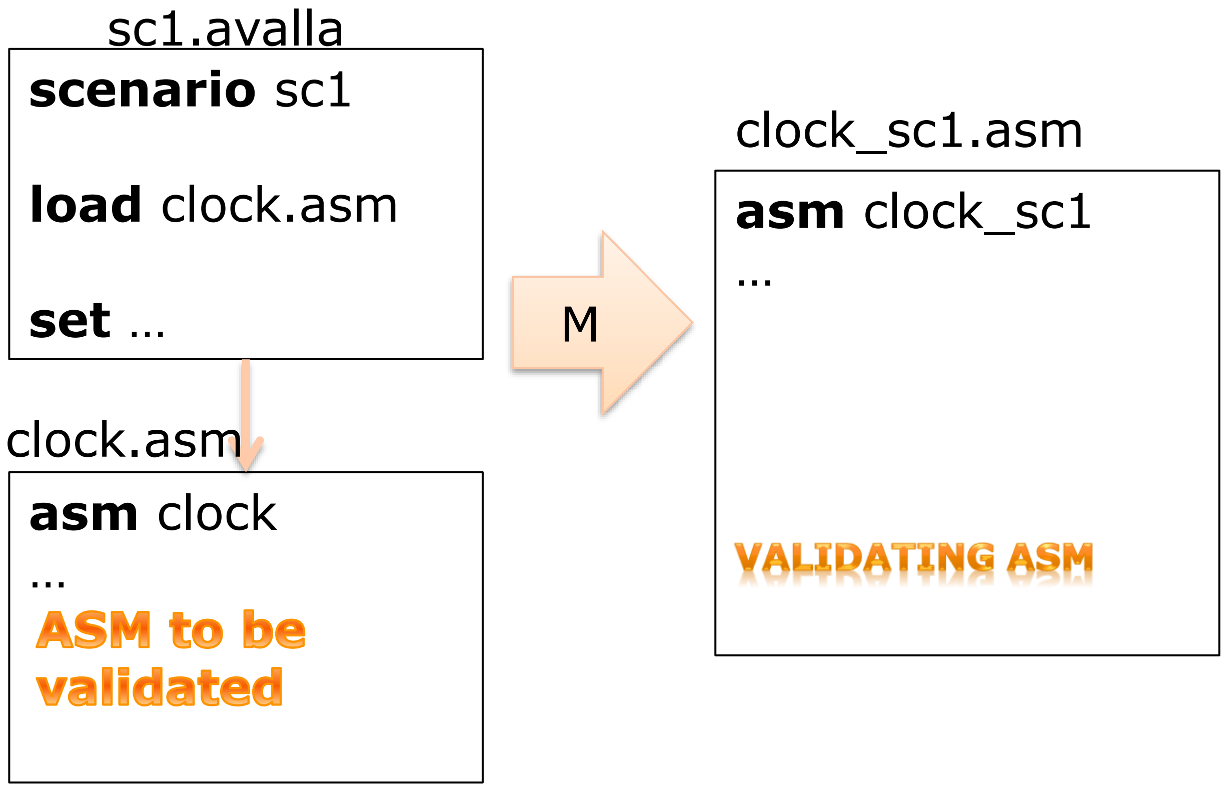
- Scenario 6:
  - Using step until

SEMANTICS

AValLa sematics

- Semantics of a metamodel-based language *L* can be given by an ASM-based semantic framework
  - some details in the paper
- Intuition
  - every program written in *L* becomes an ASM
  - definition of a mapping M from the elements of *L* to elements of ASM
  - M can be defined at the metamodel (abstract syntax) level
- For AValLa
  - Given a scenario, obtain an ASM (validating ASM)

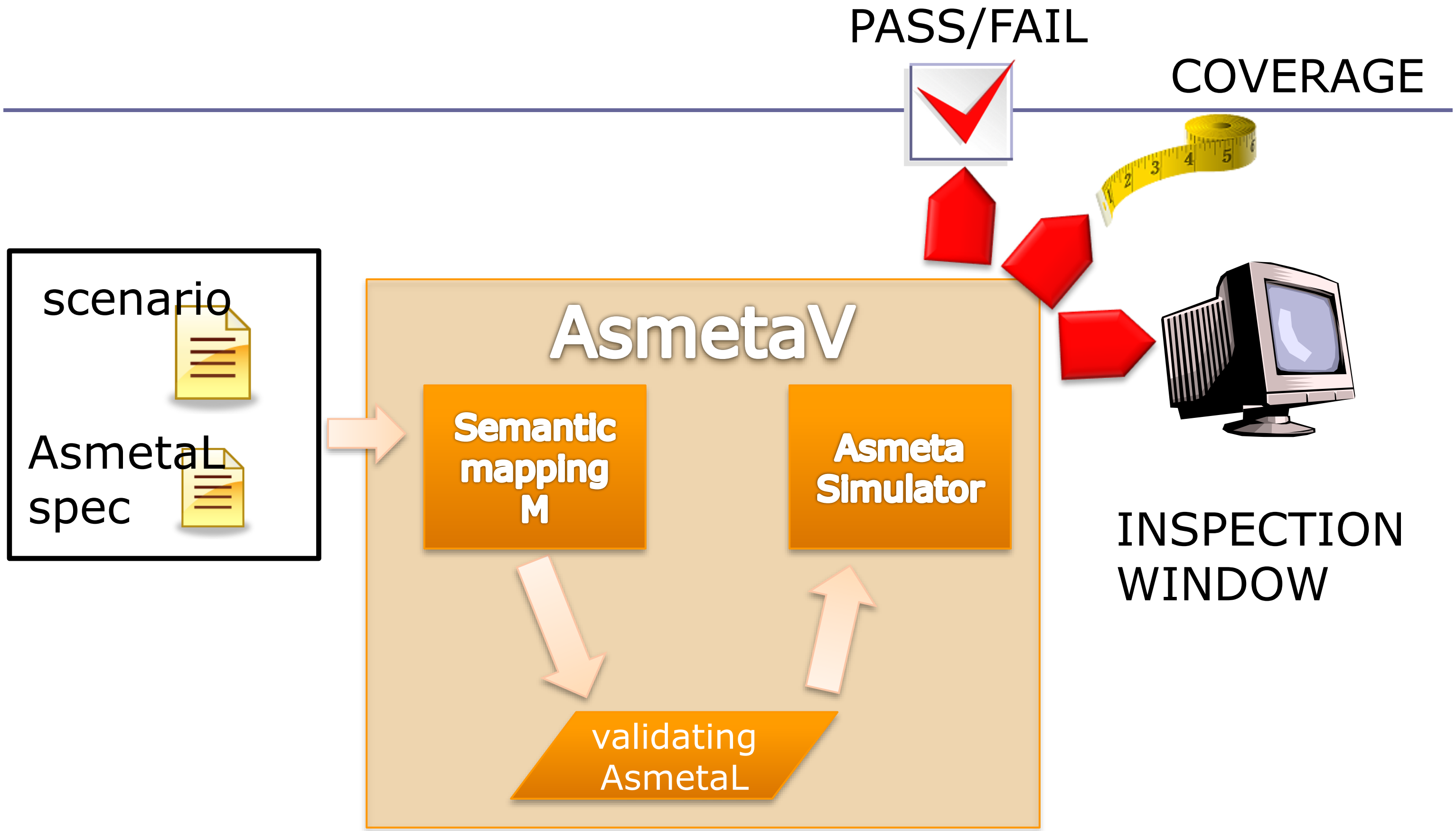
In practice …


sc1.avalla
scenario sc1
load clock.asm
set …
clock.asm
asm clock
…
ASM to be validated
M
clock_sc1.asm
asm clock_sc1
…
VALIDATING ASM

M for AValLa elements

AValLa	AsmM
Set `l:=v`	UpdateRule `l:=v`
Check `expr`	ConditionalRule with guard `expr` and body `allChecksOk := false`
Invariant `expr`	Axiom `expr`
Exec	Rule
Step	MacroDeclaration *r_step_i*
StepUntil	Two macroDeclations *r_step_i* and *r_step_i_until*

AsmetaValidator



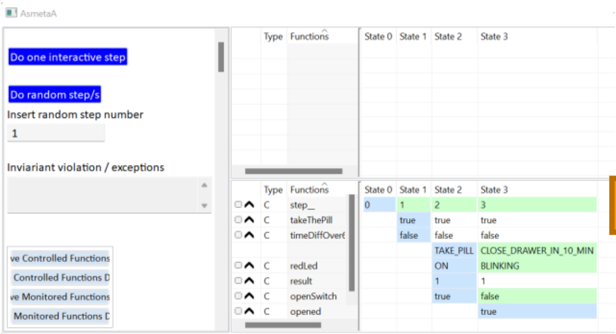
ADVANCED USE

Two extensions

- Coverage
- Animating the scenarios

- demo

Simulation <-> scenarios

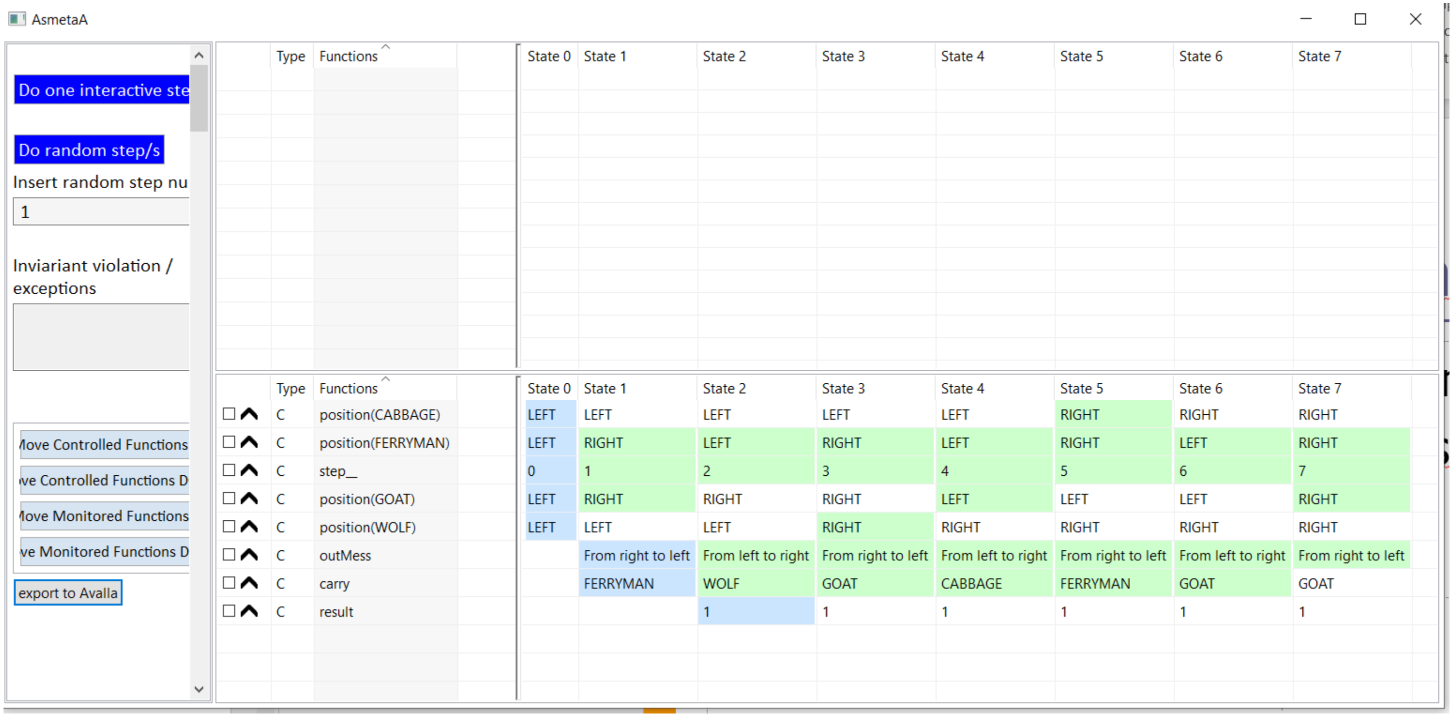


From the animator to avalla

```
check outMe... CLOSE_DRAWER_IN_10_MIN;
check logMess... NONE;

// *****************************************
// Close the drawer within 10 minutes
// *****************************************
set timeDiffOver600 := false;
set openSwitch := false;
step
check redLed = OFF;
check outMess = NONE;
check logMess = NONE;
```

Animate the avalla

Building the scenarios from animations



Type	Functions	State 0	State 1	State 2	State 3	State 4	State 5	State 6	State 7
C	position(CABBAGE)	LEFT	LEFT	LEFT	LEFT	LEFT	RIGHT	RIGHT	RIGHT
C	position(FERRYMAN)	LEFT	RIGHT	LEFT	RIGHT	LEFT	RIGHT	LEFT	RIGHT
C	step__	0	1	2	3	4	5	6	7
C	position(GOAT)	LEFT	RIGHT	RIGHT	RIGHT	LEFT	LEFT	LEFT	RIGHT
C	position(WOLF)	LEFT	LEFT	LEFT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT
C	outMess		From right to left	From left to right	From right to left	From left to right	From right to left	From left to right	From right to left
C	carry		FERRYMAN	WOLF	GOAT	CABBAGE	FERRYMAN	GOAT	GOAT
C	result			1	1	1	1	1	1

- The scenario is printed in the console

Asmeta console

```
Running interactively ferrymanSimulator.asm
INITIAL STATE:
Insert a symbol of Actors in [FERRYMAN, GOAT, CABBAGE, WOLF] for carry:
//// starting scenario
scenario SCENARIO_NAME
load __tempAsmetaV14229261553513561718.asm
check position(CABBAGE) = LEFT;
check position(FERRYMAN) = LEFT;
check step__ = 0;
check position(GOAT) = LEFT;
check position(WOLF) = LEFT;
step
check position(CABBAGE) = LEFT;
check position(FERRYMAN) = RIGHT;
check step__ = 1;
check position(GOAT) = RIGHT;
check position(WOLF) = LEFT;
check outMess = From right to left;
check carry = FERRYMAN;
step
check position(CABBAGE) = LEFT;
check position(FERRYMAN) = LEFT;
check step__ = 2;
check position(GOAT) = RIGHT;
check position(WOLF) = LEFT;
check outMess = From left to right;
check carry = WOLF;
check result = 1;
step
check position(CABBAGE) = LEFT;
check position(FERRYMAN) = RIGHT;
check step__ = 3;
check position(GOAT) = RIGHT;
check position(WOLF) = RIGHT;
check outMess = From right to left;
check carry = GOAT;
check result = 1;
step
check position(CABBAGE) = LEFT;
check position(FERRYMAN) = LEFT;
check step__ = 4;
check position(GOAT) = LEFT;
check position(WOLF) = RIGHT;
check outMess = From left to right;
check carry = CABBAGE;
```

Using blocks

- It is possible to define a scenario **block:**
  - Sequence of commands to be reused
- Definition of a block **primo_scenario.avalla**:

```
scenario first_scenario
load ./mioModello.asm
begin blockname
…
end
```

  the block will be executed

- Calling a block

```
scenario first_scenario
load ./mioModello.asm
execblock primo_scenario:nomeblocco
```

Complex example

```
scenario scenario1
load pillbox_0.asm

begin takePill

    //**********************************************************
    // Setting-up the initial state, where everything is OFF
    //**********************************************************
        set openSwitch := false;
        step
        check redLed = OFF;
        check outMess = NONE;
        check logMess = NONE;
    //*******************************************
    // Time to take the pill
    //*******************************************
        set takeThePill := true;
        set timeDiffOver600 := false;
        step
        check redLed = ON;
        check outMess = TAKE_PILL;
        check logMess = NONE;

end

begin openDrawerIn10Min

    //*******************************************
    // Open the drawer within 10 minutes
    //*******************************************
    set openSwitch := true;
    set timeDiffOver600 := false;
    step
    check redLed = BLINKING;
    check outMess = CLOSE_DRAWER_IN_10_MIN;
    check logMess = NONE;

end
```

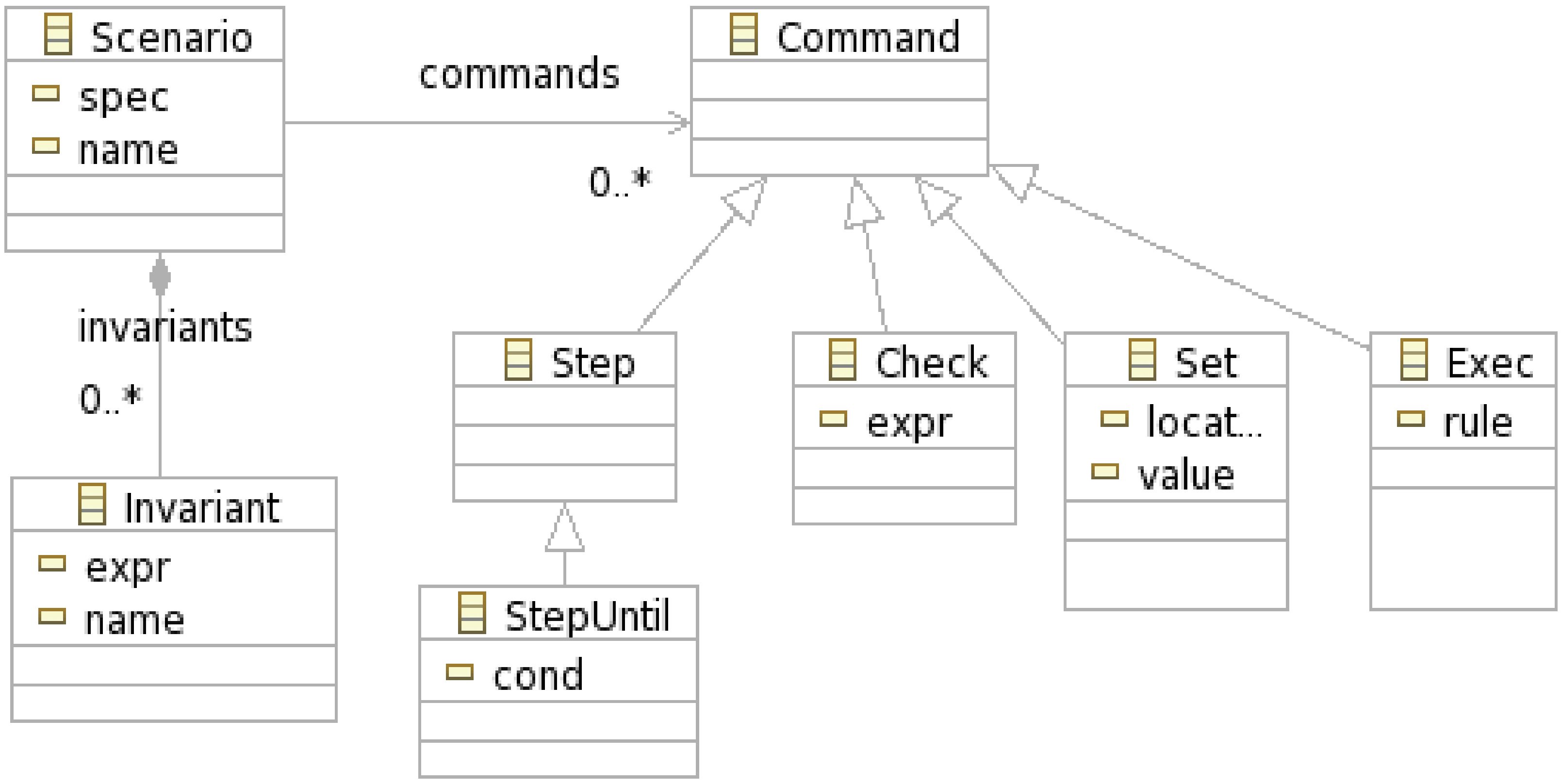
```
scenario scenario2
load pillbox_0.asm

//**********************************************************
// Initialization and need to take the pill
//**********************************************************
execblock pillbox_0_scenario1:takePill;

//**********************************************************
// Open the drawer within 10 minutes
//**********************************************************
execblock pillbox_0_scenario1:openDrawerIn10Min;

//**********************************************************
// Do not close the drawer within 10 minutes, and overpass
//**********************************************************
set timeDiffOver600 := true;
set openSwitch := true;

step

check redLed = OFF;
check opened = true;
check outMess = NONE;
check logMess = DRAWER_NOT_CLOSED;
```

HOW AVALLA IS DEFINED

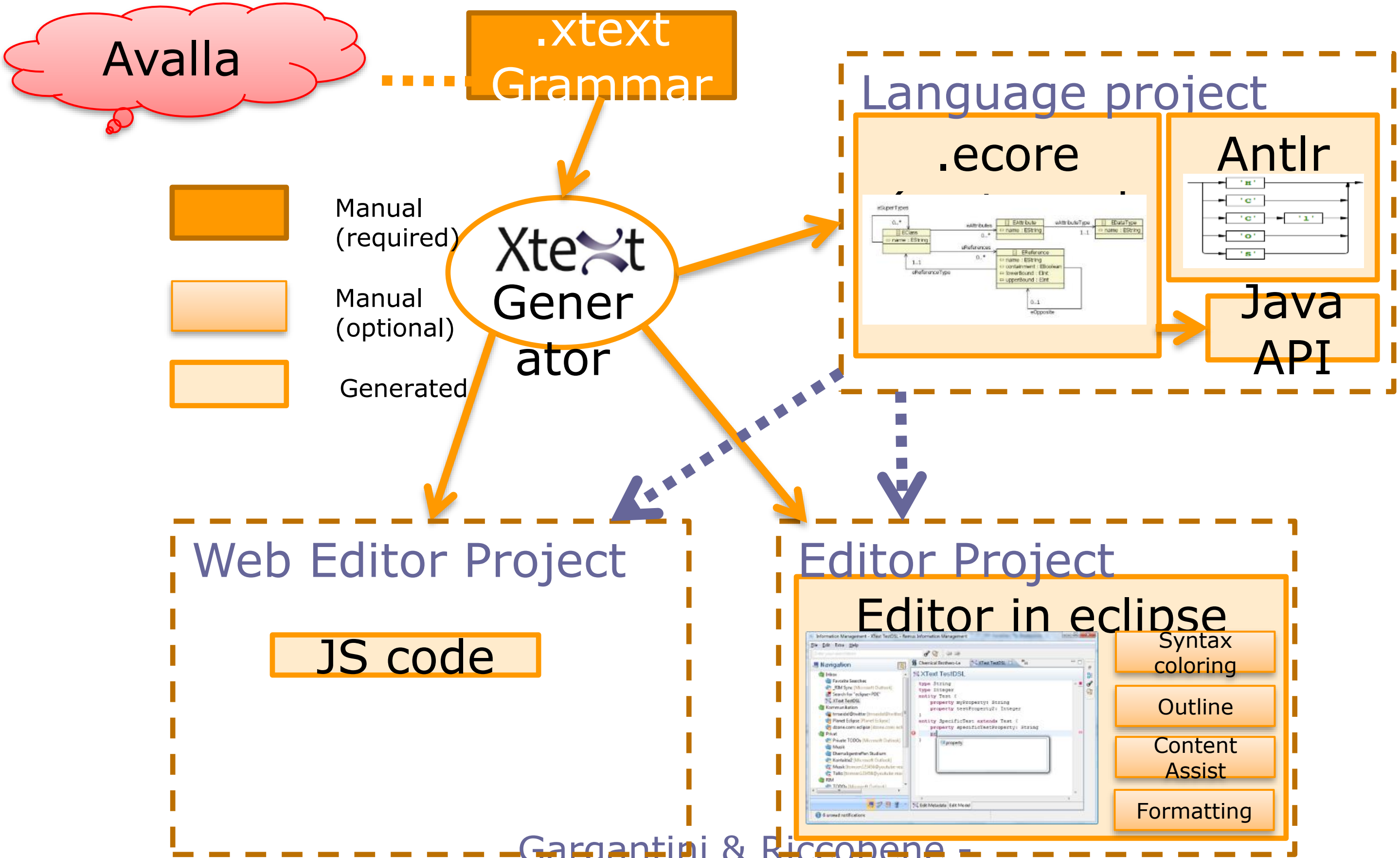
Model-driven language engineering

- AValLa is defined following the Model Driven Engineering for language definition
- MDE for languages:
  - Definition of the abstract syntax by an object-oriented model (metamodel)
  - Derivation of concrete syntaxes from the metamodel
  - Supporting tools and technologies
    - EMF: eclipse modelling framework
    - MOF: OMG Meta Object Facility
    - …
  - …
- As done for the Asmeta Abstract State Machines

AValLa Metamodel

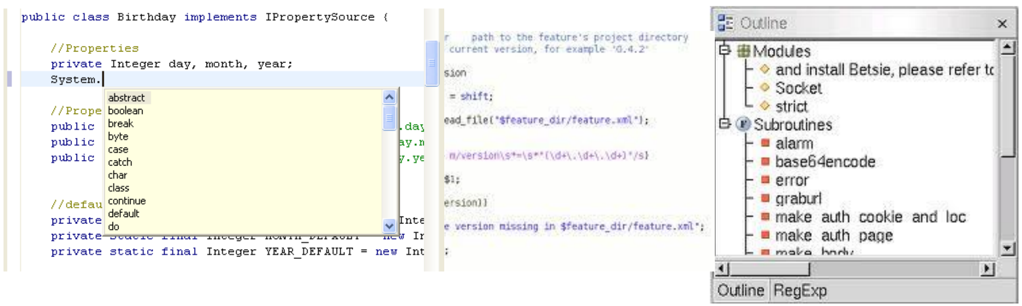


Avalla in XTEXT



Editor features

- Syntax Coloring
- Content Assist
- Template Proposals
- Rich Hover
- Rename Refactoring
- Quick Fixes
- Outline
- Folding
- Hyperlinks for all Cross References
- Find References
- Toggle Comment
- Mark Occurrences
- Formatting



Lift case study

Lift Control : The Problem

Design the logic to move n lifts bw m floors, and prove it to be well functioning, where

- Each lift has for each floor one button which, if pressed, causes the lift to visit (i.e. move to and stop at) that floor.
- Each floor (except ground and top) has two buttons to request an up-lift and a down-lift. They are cancelled when a lift visits the floor and is either travelling in the desired direction, or visits the floor with no requests outstanding. In the latter case, if both floor request buttons are illuminated, only one should be cancelled.
- A lift without requests should remain in its final destination and await further requests.
- Each lift has an emergency button which, if pressed, causes a warning to be sent to the site manager. The lift is then deemed ‘out of service’. Each lift has a mechanism to cancel its ‘out of service’ status. *(sikip this part for now)*

Lift Example

validate lift2.asm

Set the button at ground floor to off
Switch off all the buttons

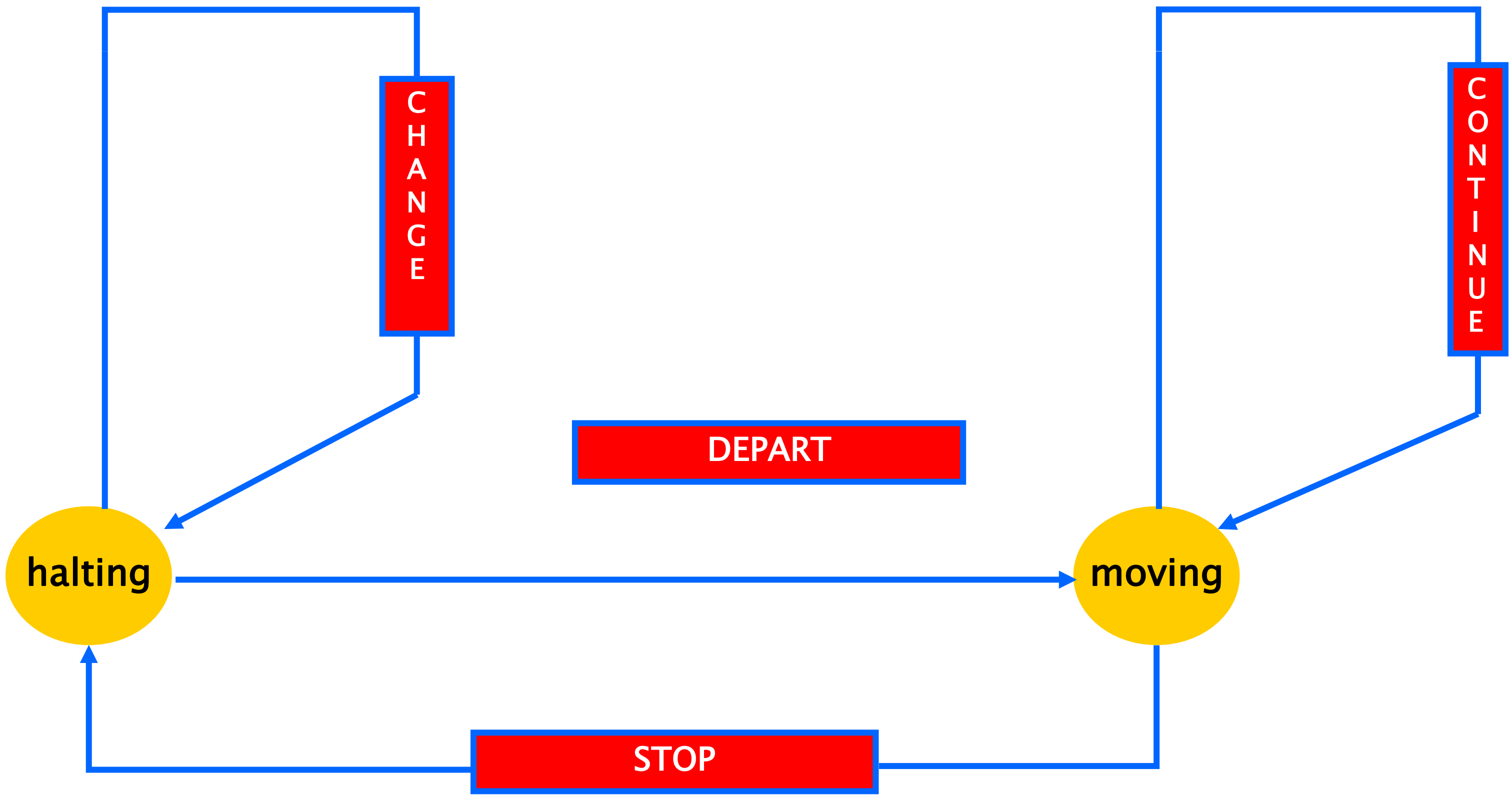
Check that the lift is halted at ground floor, direction UP

Make a step

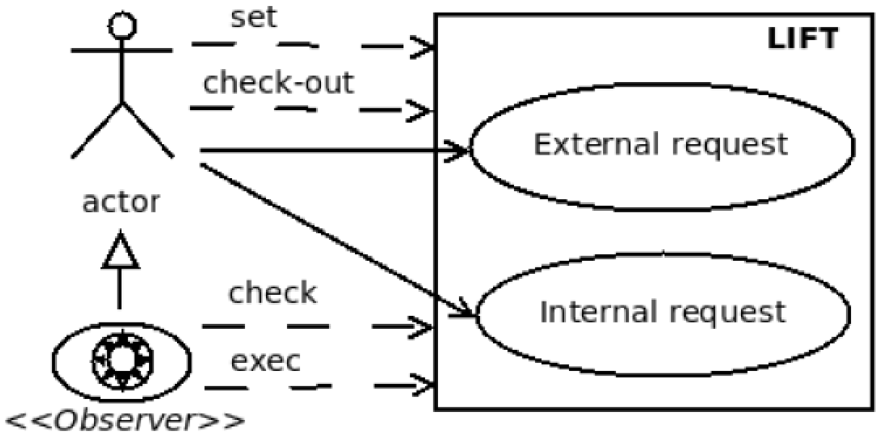
Check again

```
scenario lift2_s0
load ./lift2.asm

// init monitored functions
set hasToDeliverAt(lift1, 0)  := false;
set existsCallFromTo(0, UP)    := false;
set existsCallFromTo(0, DOWN)  := false;
…


check floor(lift1) = 0;
check ctlState(lift1) = HALTING;
check dir(lift1) = UP;

step

check floor(lift1) = 0;
check ctlState(lift1) = HALTING;
check dir(lift1) = UP;
```

Lift Control : control state ASM



Esempio del Lift



(A)

- **existsCallFromTo(floor,dir):** richiesta esterna di selezione **dir** (=**UP/DOWN**) da piano **floor**
- **hasToDeliverAt(lift,floor)**: richiesta interna al piano **floor**
  - se consumati, gli eventi diventano **false**

Lift model

```
asm lift3

import ../LIB/StandardLibrary

signature:
   abstract domain Lift
   domain Floor subsetof Integer
   enum domain Dir = {UP | DOWN}
   enum domain State = {HALTING | MOVING}
```

Lift functions

```
// lift direction of travel
dynamic controlled dir:  Lift -> Dir
// lift control state
dynamic controlled ctlState: Lift -> State
//lift current floor
dynamic controlled floor:  Lift -> Floor
// internal request
dynamic monitored hasToDeliverAt: Prod(Lift, Floor) -> Boolean
// external request
dynamic monitored existsCallFromTo: Prod(Floor, Dir) -> Boolean

derived hasToVisit: Prod(Lift, Floor) -> Boolean
derived attracted: Prod(Dir, Lift) -> Boolean
derived canContinue: Lift -> Boolean
static opposite: Dir -> Dir
```

Lift

```
// consts
static ground: Integer
static top: Integer
static lift1: Lift

definitions:

domain Floor = {0..4}
function ground = 0
function top = 4

function opposite ($d in Dir) =
  if ($d = UP) then DOWN else UP endif
```

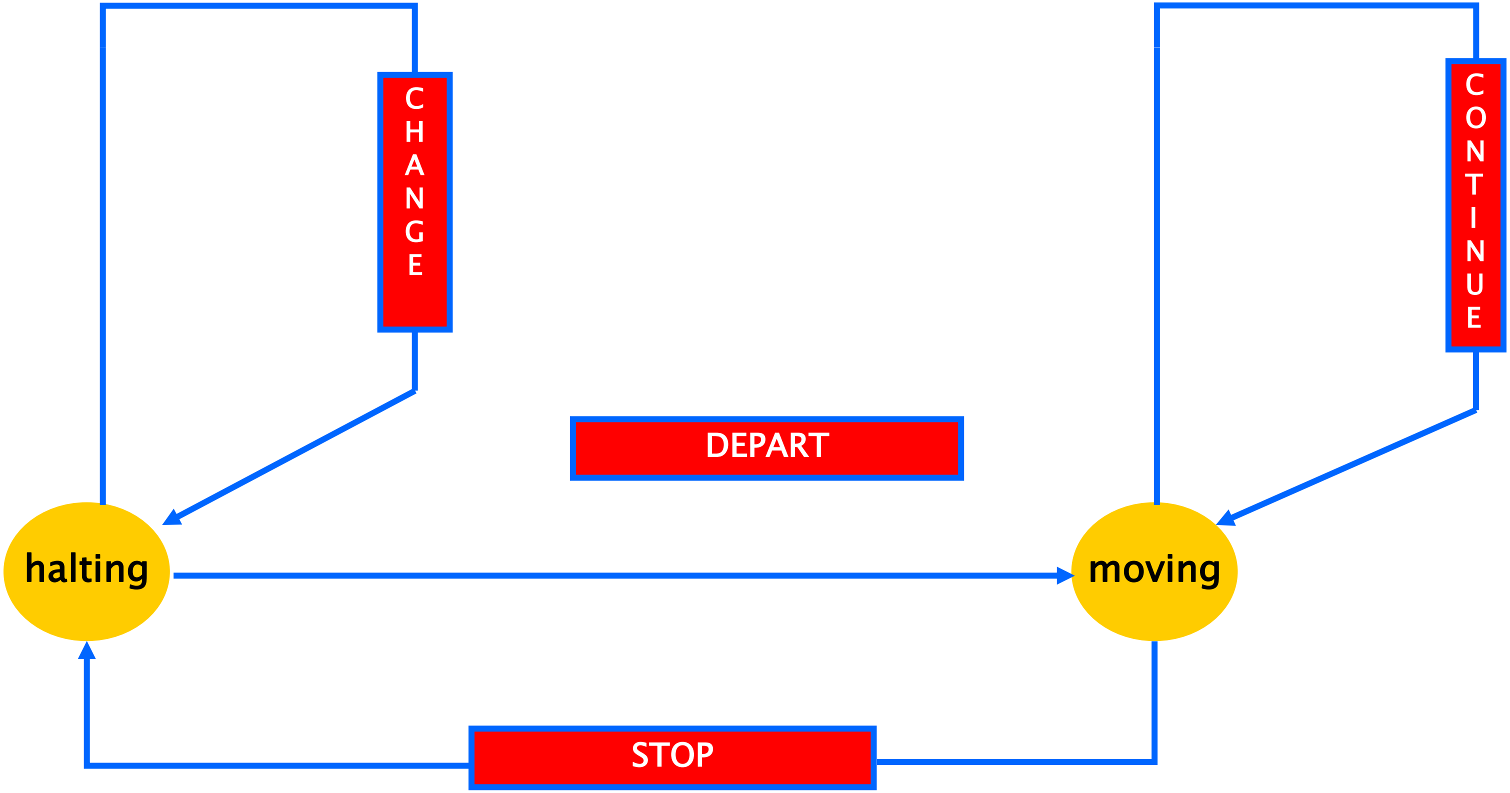
Lift

```
function hasToVisit($l in Lift, $floor in Floor) =
   hasToDeliverAt($l, $floor)
   or existsCallFromTo($floor, UP)
   or existsCallFromTo($floor, DOWN)

function attracted($dir in Dir, $l in Lift) =
   $dir = UP and (exist $floor in Floor with $floor > floor($l)
   and hasToVisit($l, $floor))
   or
   $dir = DOWN and (exist $floor2 in Floor with $floor2 <
   floor($l) and hasToVisit($l, $floor2))

function canContinue($l in Lift) =
   attracted(dir($l), $l)
   and not hasToDeliverAt($l, floor($l))
   and not existsCallFromTo(floor($l), dir($l))
```

Lift

```
macro rule r_cancelRequest($dir in Dir, $l in Lift) =
   par
      hasToDeliverAt($l, floor($l)) := false
      existsCallFromTo(floor($l), $dir) := false
   endpar

macro rule r_moveLift($l in Lift) =
   par
      if dir($l) = UP then floor($l) := floor($l) + 1
      endif
      if dir($l) = DOWN then floor($l) := floor($l) - 1
      endif
   endpar
```

Lift Control : control state ASM



Lift

```
macro rule r_depart($l in Lift) =
   if ctlState($l) = HALTING and attracted(dir($l), $l) then
      par
         r_moveLift[$l]
         r_cancelRequest[dir($l), $l]
         ctlState($l) := MOVING
      endpar
   endif


macro rule r_continue($l in Lift) =
   if ctlState($l) = MOVING and canContinue($l) then
        r_moveLift[$l]
   endif
```

Modello per Lift

```
macro rule r_stop($l in Lift) =
   if ctlState($l) = MOVING and not canContinue($l) then
      par
         r_cancelRequest[dir($l), $l]
         ctlState($l) := HALTING
      endpar
   endif

macro rule r_change($l in Lift) =
   let ($d = dir($l), $d2 = opposite($d)) in
      if ctlState($l) = HALTING and not attracted($d, $l) and
      attracted($d2, $l) then
         par
            dir($l) := $d2
            r_cancelRequest[$d2, $l]
         endpar
      endif
   endlet
```

Lift

```
macro rule r_lift($l in Lift) =
   par
      r_depart[$l]
      r_continue[$l]
      r_stop[$l]
      r_change[$l]
   endpar

invariant over existsCallFromTo:
not existsCallFromTo(ground, DOWN) and not existsCallFromTo(top,
UP)

main rule r_main = r_lift[lift1]

default init s0:

function floor($l in Lift) = ground
function dir($l in Lift) = UP
function ctlState($l in Lift)= HALTING
```

First scenario

- Description:
  - The lift is at ground and there are no requests (internal no external)
  - The lift remains in the position

Primo scenario per il Lift

valida lift2.asm

All the requests are off

Check that the elevator is halting

Perform a step

Check again

```
scenario lift2_s1
load ./lift2.asm

// init monitored functions
set hasToDeliverAt(lift1, 0)  := false;
set existsCallFromTo(0, UP)    := false;
set existsCallFromTo(0, DOWN)  := false;
..............


check floor(lift1) = 0;
check ctlState(lift1) = HALTING;
check dir(lift1) = UP;

step

check floor(lift1) = 0;
check ctlState(lift1) = HALTING;
check dir(lift1) = UP;
```

Scenario 2

- Description:
  - The list is at ground floor (0). An user calls the elevator from floor 4 and wants to go to floor 2. She enters the elevator and presses floor 2.
  - Check that the elevator performs all the required action

Scenario 2 in Avalla

```
scenario lift2_s2
load ./lift2.asm

// ....setting inizial state
// an external request to floor 4
set existsCallFromTo(4, DOWN) := true;
// lift goes to floor 4
step until ctlState(lift1) = HALTING and floor(lift1) = 4;
// request to floor 2
set hasToDeliverAt(lift1, 2) := true;
step
// must go down to floor 2, down dir
check dir(lift1) = DOWN;
// the request at floor 4 is cancelled
check not existsCallFromTo(4, DOWN);
// goes to floor 2
step until ctlState(lift1) = HALTING and floor(lift1) = 2;
// request to floor 2 is cancelled
check not hasToDeliverAt(lift1, 2);
```

Scenario 3

- Description:
  - Lift at ground and all the external requests are ON (up and down).
  - The lift goes UP from floor 0 to the last one (4). All the requests to go UP are cancelled.
  - All the requests to go down are not cancelled
  - Richiediamo l'invariante: l'ascensore non cambia direzione mentre sale: **`dir(lift1) != DOWN`**

Scenario 3 in Avalla

```
scenario lift2_s3
load lift.asm
invariant neverDown: dir(lift1) != DOWN;
exec //set floor requests (all ext. buttons UP and DOWN pushed)
   forall $i in {0..4} do
     par
       hasToDeliverAt(lift1, $i) := false
       if $i != top then existsCallFromTo($i, UP) := true endif
       if $i != ground then existsCallFromTo($i, DOWN) := true endif
     endpar;
//the lift goes up to floor 4, then goes down
step until ctlState(lift1) = HALTING and floor(lift1) = 4;
// check that the UP-external requests have been satisfied, while
the DOWN-requests are still pending
check (forall $i in {0..4} with existsCallFromTo($i, DOWN) = true);
check (forall $i in {0..4} with existsCallFromTo($i, UP) = false);
```

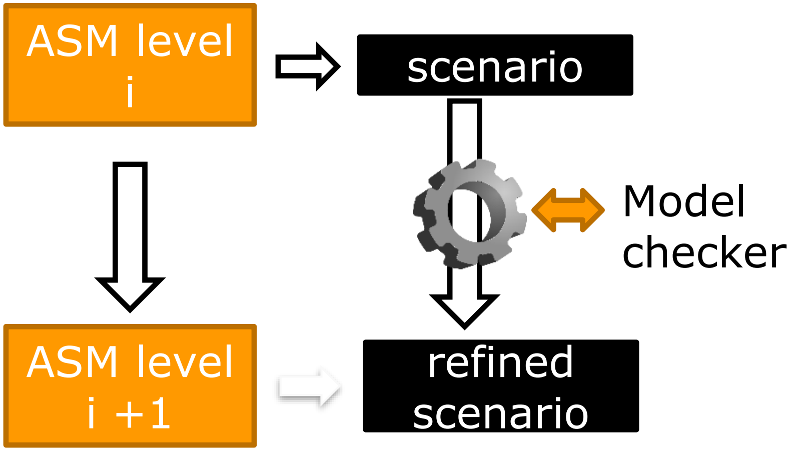
Lift scenarios

		# Commands	Coverage
S0	The lift is halted at ground floor, no request, it should stay	18	6/8 
S1	External request at the same floor (ground) and direction, internal request for floor 2. The lift should reach floor 2	24	7/8 
S2	External request at floor 4, enters and ask for floor 2	22	8/8 
s3	All external (UP and DOWN) buttons have been pushed. The lift reaches the top floor and UP requests are canceled	4 – 1 invariant	8/8 

NEW IDEAS

Scenarios and refinement

- Scenarios can be automatically refined when a specification is refined

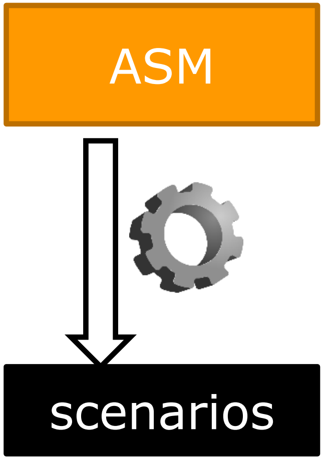


Paolo Arcaini, Elvinia Riccobene,
Automatic Refinement of ASM Abstract Test Cases
A-MOST workshop 2019 IEEE International Conference on Software Testing, Verification and Validation

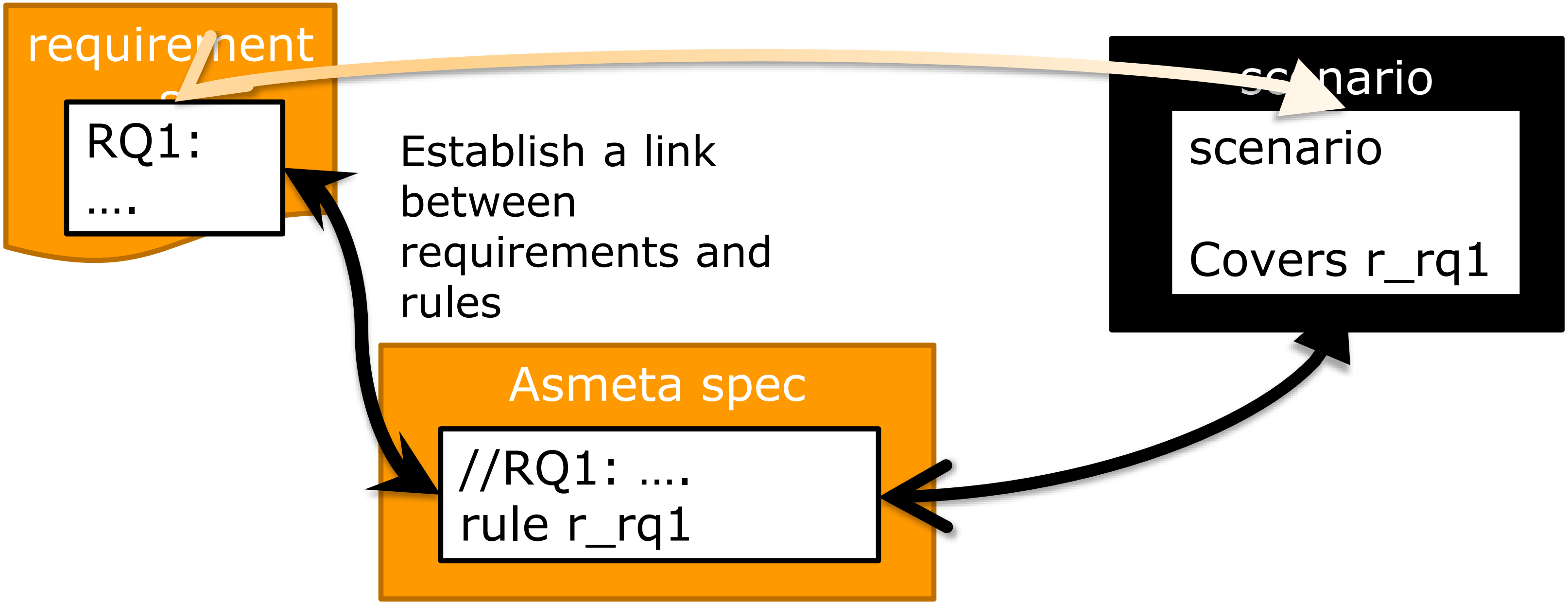
- A way to check if:
- The refinement is correct
- It captures the desired behaviors
  - Manual checking of scenarios

Automatic generation of scenarios

- Using the model checker and its capability to generate counter examples,
- Several coverage criteria
- The user can inspect and validate the scenarios



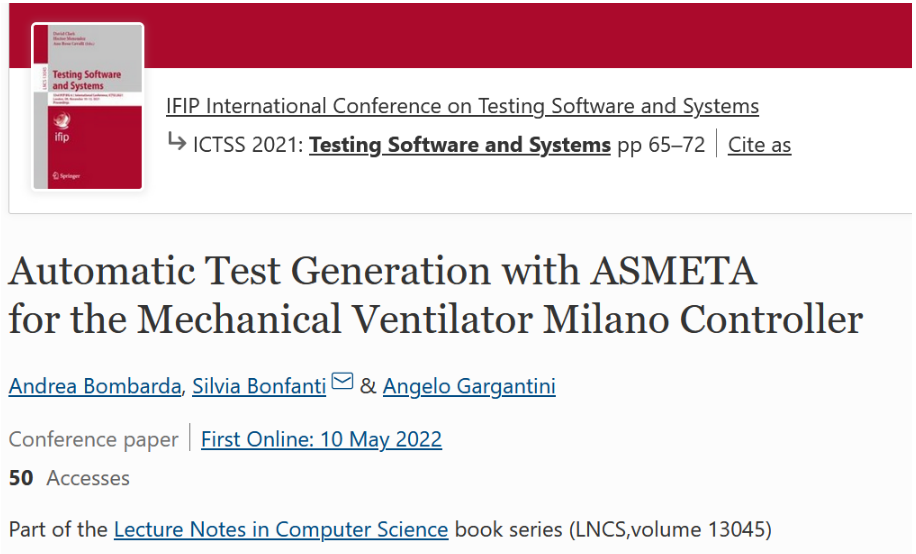
Scenarios and traceability (ABZ 2021)



The validator links scenarios and rules

From scenarios to unit tests for code

- Scenarios can be translated to executable Unit test code



IFIP International Conference on Testing Software and Systems

↳ ICTSS 2021: **Testing Software and Systems** pp 65–72 | Cite as

Automatic Test Generation with ASMETA for the Mechanical Ventilator Milano Controller

Andrea Bombarda, Silvia Bonfanti & Angelo Gargantini

Conference paper | First Online: 10 May 2022

50 Accesses

Part of the Lecture Notes in Computer Science book series (LNCS,volume 13045)

Use of scenarios

- Regression testing
  - Scenario can be executed to check the modifications do not introduce unintended behaviors
    - Scenario-driven development?
    - What about refinement?
- Coverage
  - Scenario can give a measure of which rules are covered
    - When stopping writing scenarios?
    - Or better use mutation testing?
  - Traceability (later)
    - Link between scenarios and rules is useful trace requirements to scenarios

Test generation and execution process