WETICE 2025

UNIVERSITÀ DEGLI STUDI DI BERGAMO | Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

Are (formal) models still really useful in software engineering?

Angelo Gargantini - University of Bergamo, Italy

FOSELAB

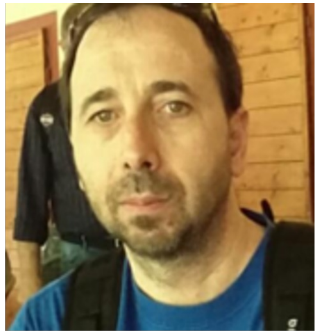
Finanziato dall'Unione europea NextGenerationEU

Ministero dell'Università e della Ricerca

Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA

Credits

Angelo Gargantini



Silvia Bonfanti

Andrea Bombarda

FOSELAB

Formal Methods & Software Engineering

Patrizia Scandurra

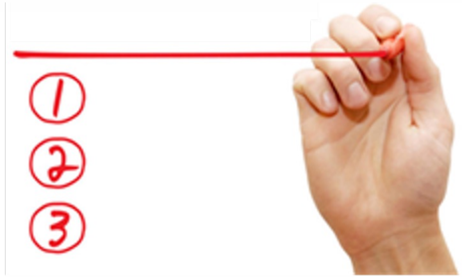
Elvinia Riccobene @ UNIMI

Claudio Menghi

"Better software for a better world"

https://foselab.unibg.it/

Outline



- What are models and why are important (classic view of SE)
- Modeling notation: Abstract State Machines
- Use of models:
  - Code generation
  - Digital twins
  - Models@Runtime
- Problems with models
  - Evolution
- New roles of models

what are models

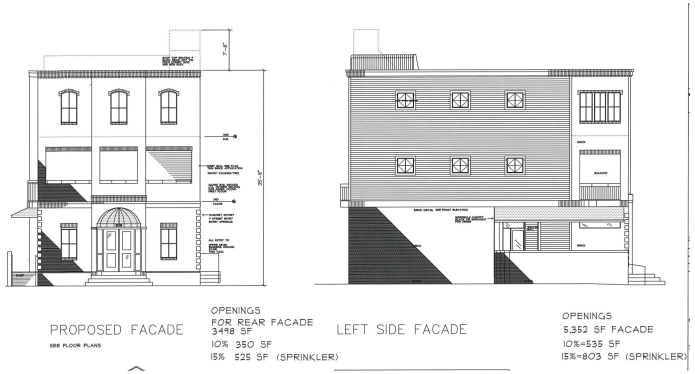
- In software engineering, when we use models, we mean abstract representations of a system.
- Normally, models are a mathematical/algebraic/logic representation of the system or part of it.
- In case the notation is more formally oriented, we use the term formal models.
- Not to be confused with models in ML, or like large language models

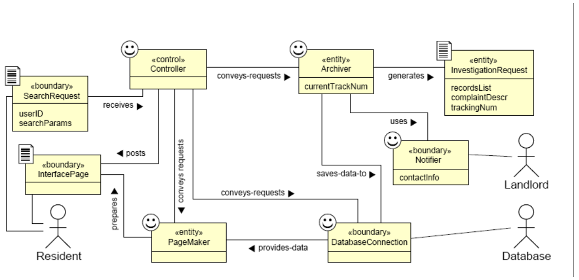
Classical USE of models

- Models are used to help understand, design, analyze, implement, and communicate how a system works or should work.
- This is at least the classical use of models in requirements engineering.

Classical use of models: to build software

MODELS





REALIZATION

Are formal models still useful?

- AGILE
  - “Working software over comprehensive documentation”
  - Agile modeling (AM)
- Machine learning
  - The system is not realized from a representation of it
- Language Models
  - Requirements → implementation
- No Yes/No answer

Abstract State Machines
asmeta

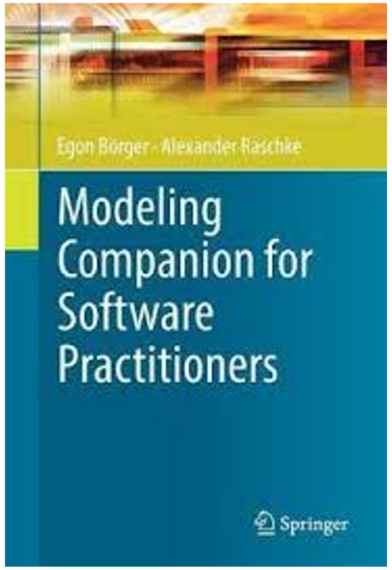
An example of modelling…

Abstract State Machines

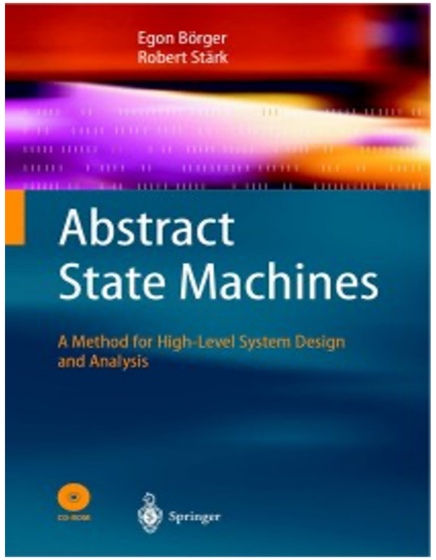
- ASMs are a system engineering method able to guide the development of software systems *seamlessly* from requirements capture to their implementation
- ASMs used in different application domains:
  - definition of industrial standards for programming and modeling languages
  - design and re-engineering of safety-critical systems
  - verification of compilers, security protocols, etc.

Main references: ASM books

E.Boerger and R. Staerk, 2003



E.Boerger and A. Raschke, 2018



Report also on industrial/academic projects where ASMs have been applied

P.Arcaini, A.Bombarda, S.Bonfanti, A.Gargantini, E.Riccobene, P.Scandurra: The ASMETA Approach to Safety Assurance of Software Systems. LNCS (2021). https://doi.org/10.1007/978-3-030-76020-5_13

Why the ASMs

Practitioners are reluctant to model SW REQS by using FMs due to:

- lack of training
- complex formal notations
- lack of easy-to-use tools supporting a developer during the life cycle activities of system development
- lack of a precise development process from REQs to code

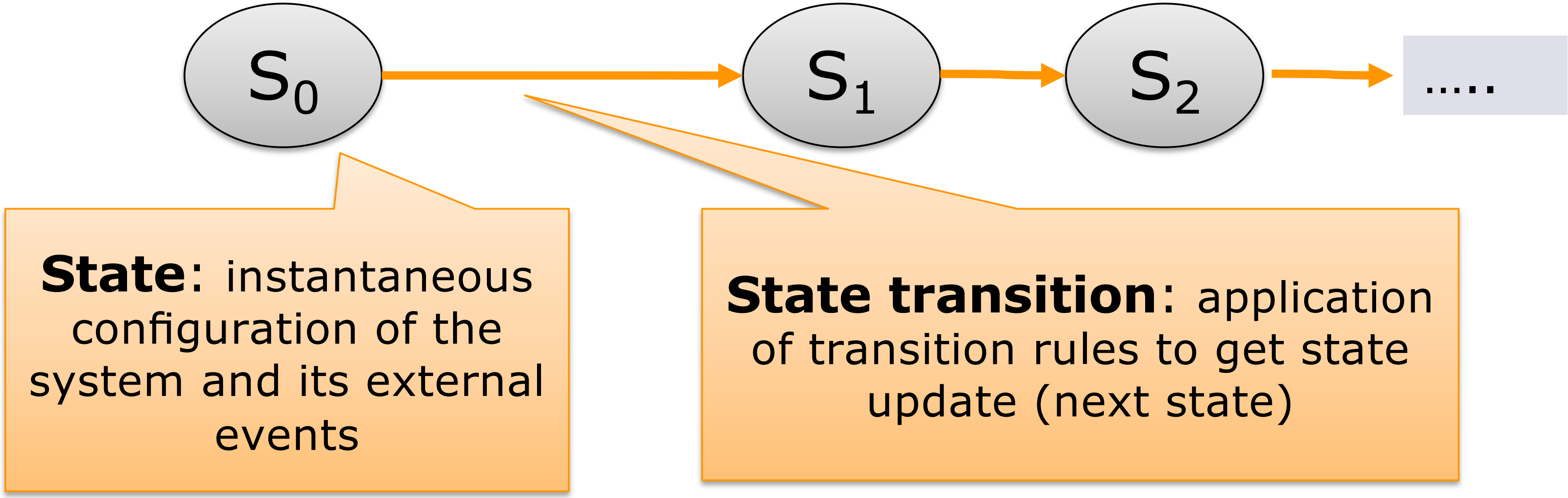
ASM-based Modeling Approach

ASM formal method builds upon three main concepts:

- Abstract State Machines, state-based transition systems that extend Finite State Machines
  - *unstructured control states* are replaced by states with arbitrary complex data
- ground model, an ASM which is a reference model for the design w.r.t. a set of requirements
- model refinement, a general scheme for stepwise instantiations of model abstractions to concrete system elements

Abstract State Machines (ASMs)

ASMs are state-based transition systems:



ASM state

- **state**: multi-sorted first-order structure (*algebra*), i.e. domains of objects with functions defined on them
  - $(f,(v_1,\ldots,v_n))$ are *locations*
    - represent object container (or memory unit)
    - at each state, each location has a value
  - Location updates represent the basic units of state change and they are given as assignments:

  $f(v_1,\ldots,v_n) := v_{next}$

Function classification

- **Dynamic** functions updated by transition rules
    - monitored: read by the machine and written by the env
    - controlled: read and written by the machine
    - out: written by the machine and read by the env
    - shared: read and written by the machine and the env
- **Static** functions not updated by transition rules
- Functions defined in terms of other functions are called **derived**

ASM transitions

- **Transition rules** specify how dynamic functions change from one state to the next
- Basic transition rule (*guarded update*):

  if *Condition* **then** **`Updates`**

  where **`Updates`** is a set of function updates

  f(t1, …,tn):= t

  simultaneously executed when *Condition* is true

ASM transition rules

More complex rule constructors exist:

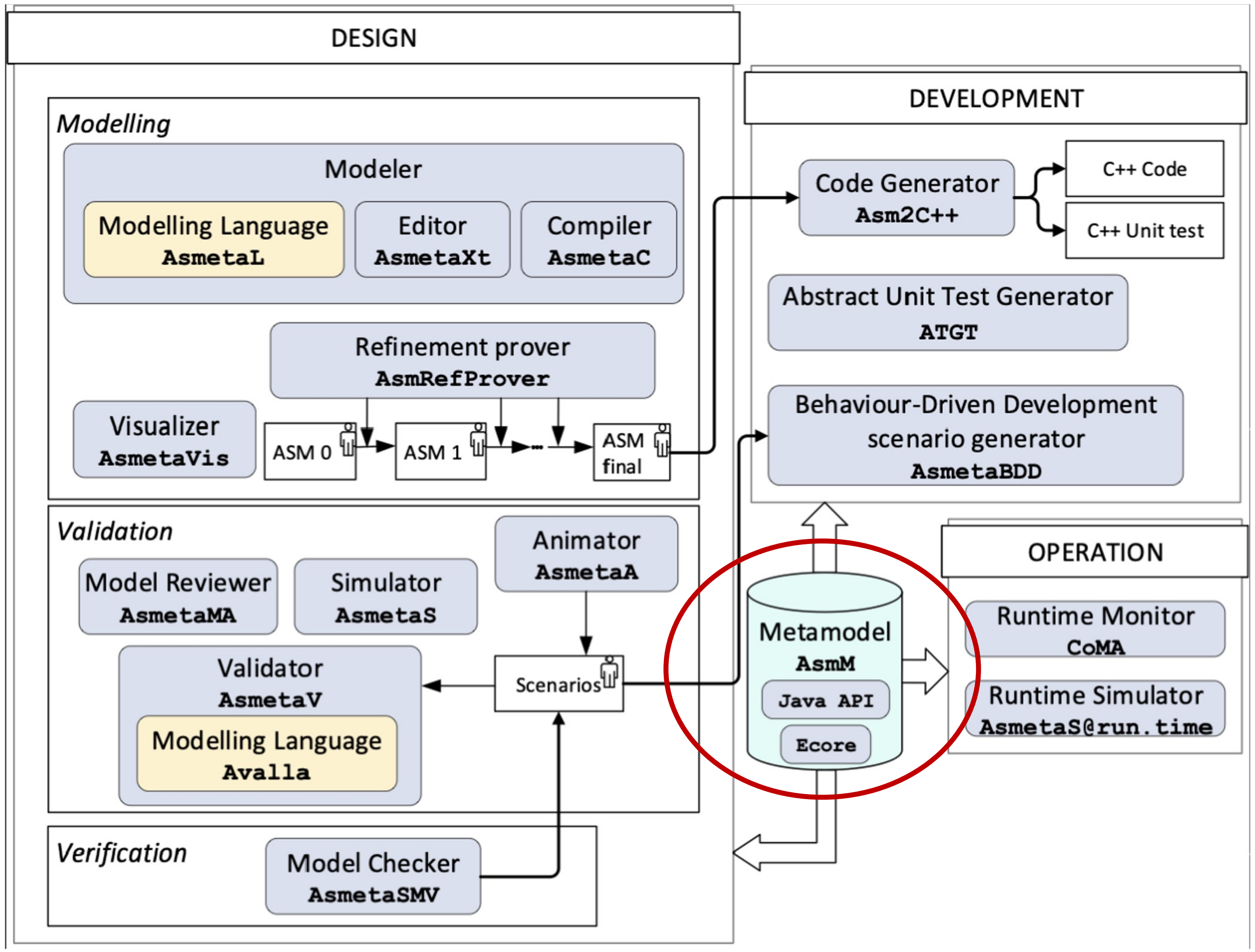
- guarded updates (**if-then-else, switch-case)**
- simultaneous parallel updates (**par**)
- non-determinism (**choose**)
- unrestricted synch. parallelism (**forall**)
- abbreviation on terms of rules (**let**)
- sequential actions (**seq**)
- domain extension (**extend**)

ASMETA Toolset

Tool components and modeling process

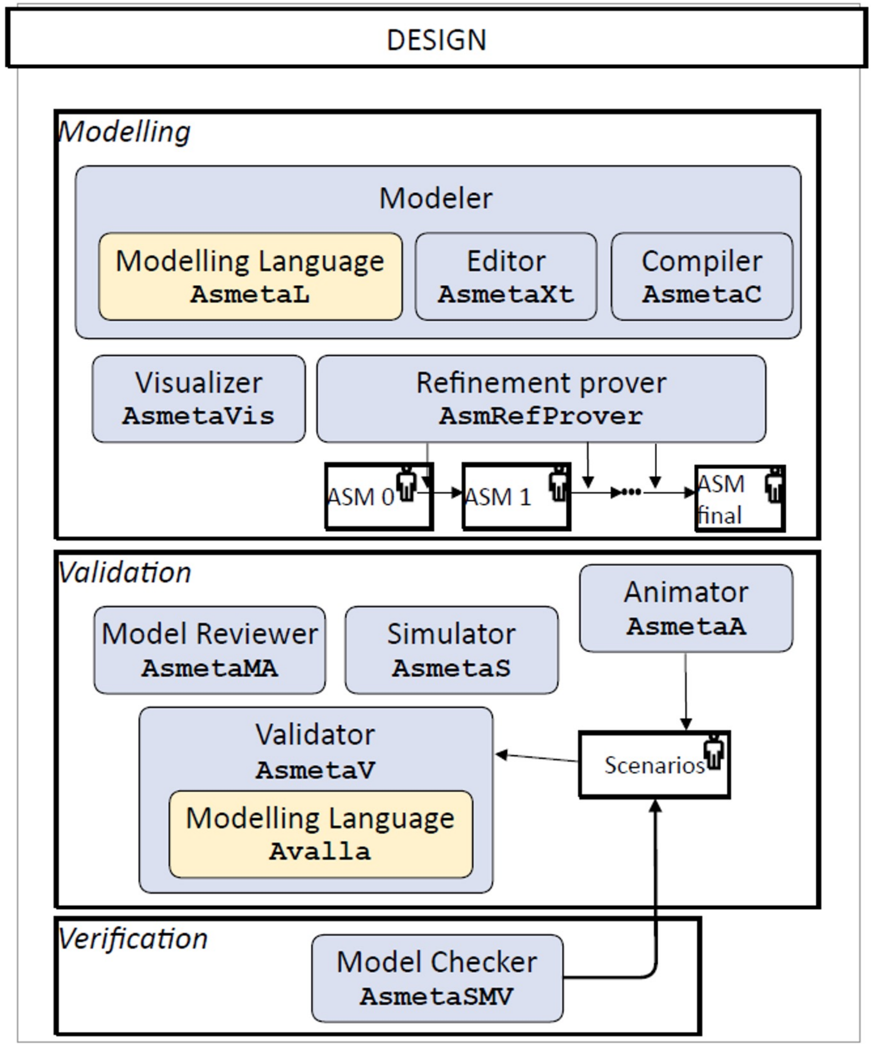
FROM TUTORIAL @ FM – VIDEO AVAILABLE

ASMETA Toolset



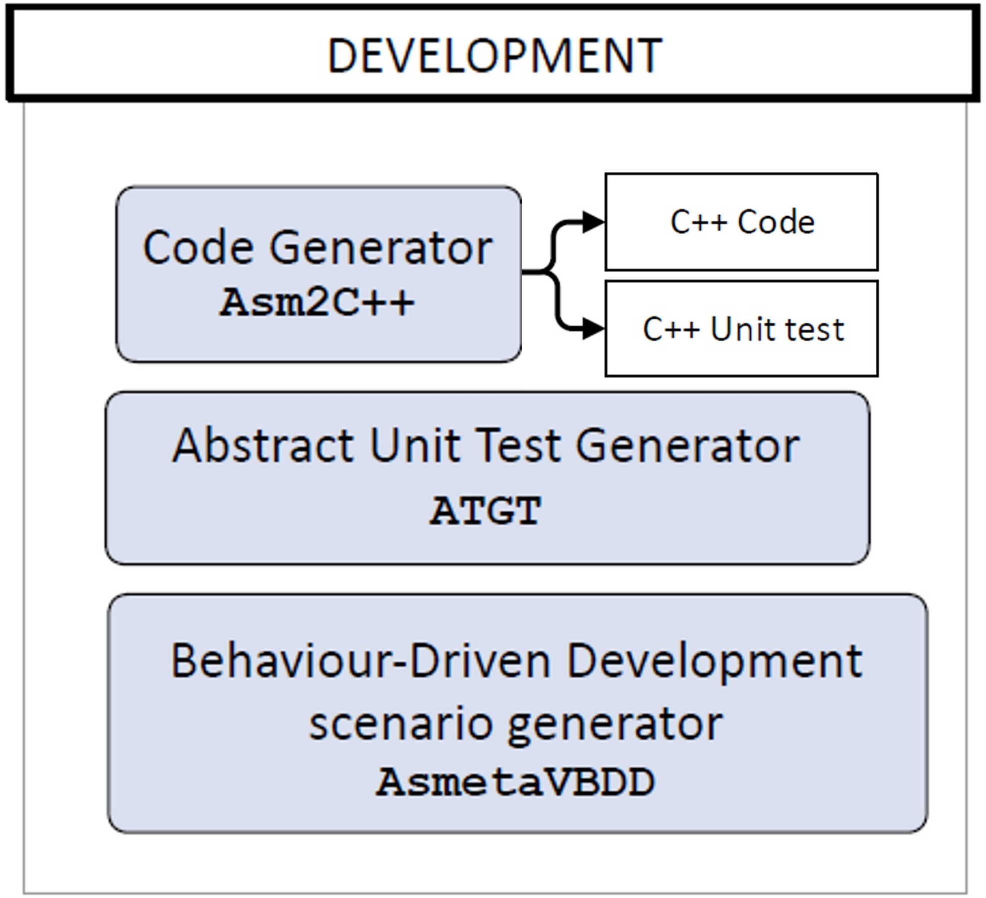
- Developed by exploiting the MDE approach for software development
- Core component: the meta-model AsmM as abstract notation to define an ASM

ASMETA @ design-time



- **Modeling**
  - Modeling Language
  - Refinement
  - Visualization
- **Validation and Verification**
  - Model Simulation
  - Model Animation
  - Scenario-based validation
  - Model reviewing
  - Model checking of temporal properties

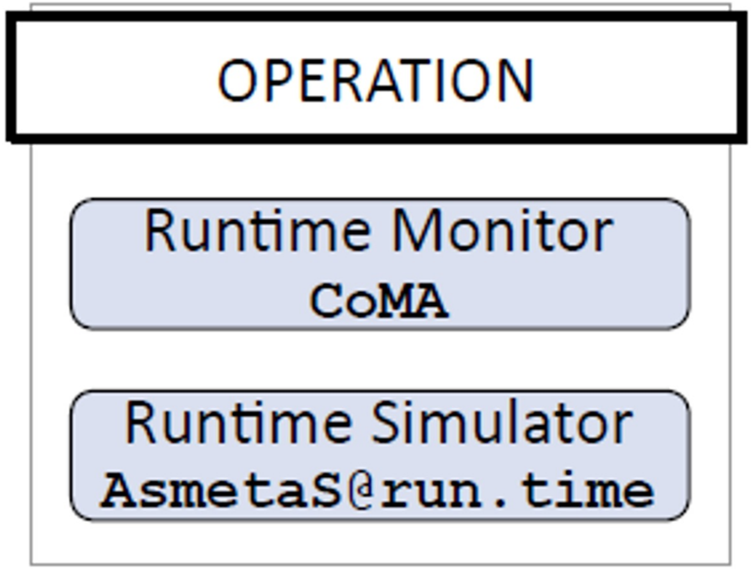
ASMETA @ development time



- Automatic model-based code generation
- Automatic model-based test generation
- Unit test generation
- Behaviour-Driven Development scenarios
  - acceptance test generator for complex scenarios

ASMETA @ operation time

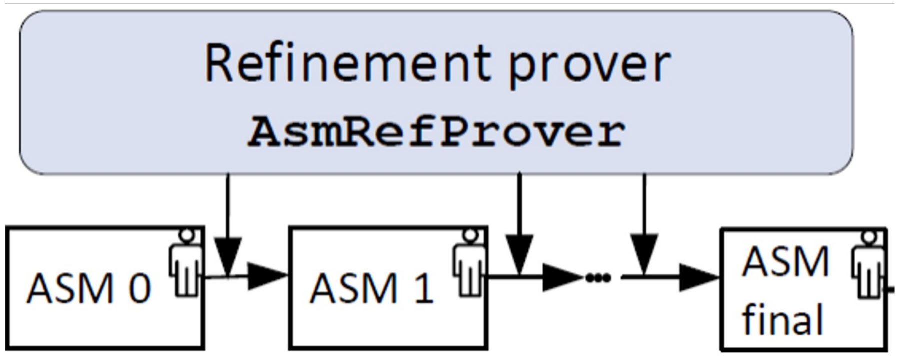
Model as a *twin* of the real system



- **Runtime monitoring**
  - Twin execution used to *check* correctness of the real system behavior w.r.t. model behavior
- Runtime **simulation**
  - Twin execution used to *prevent* misbehavior of the real system in case of unsafe model behavior

ASMETA modeling process

- From ASM_0, through a sequence of refined models ASM_1, ASM_2,. . . , other functional requirements are modeled, till the desired level of completeness
- ASM_{final} captures all intended requirements at the desired level of abstraction
- We support proof of (a form of) correct model refinement step



The textual modeling notation AsmetaL

ASMETA model structure

ASM = (header, body, main rule, initialization)

- header: **import** of modules, **signature** declaration
- body: **defs.** of **domains/functions**, state **invariants**, and **rules**
- main rule: **def.** of the **starting rule** of the machine
- initialization: specifies an **initial state** (an initial value for domains and functions of the signature)

ASMETA model structure in AsmetaL

header

```
asm pillbox_ground

import ../STDL/StandardLibrary
...
signature:
  // DOMAINS
  abstract domain Drawer
  enum domain LedLights = {OFF | ON}
  ...

  // FUNCTIONS
  dynamic monitored isPillTaken: Drawer -> Boolean
  ...
  dynamic controlled drawerLed: Drawer -> LedLights
  ...
  derived isOn: Drawer -> Boolean
  ...
  static drawer1: Drawer
```

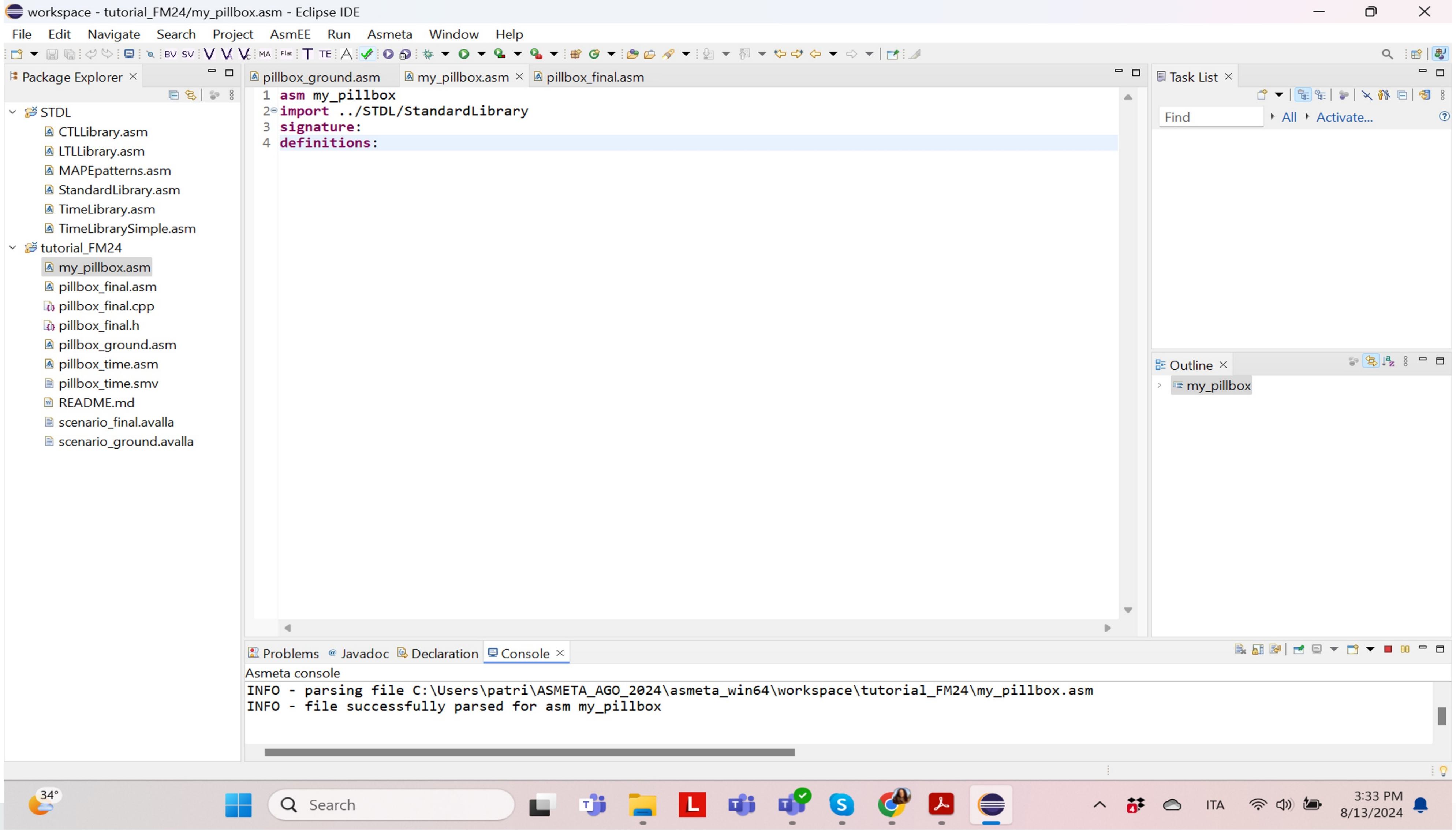
body

```
definitions:
  // FUNCTIONS DEFINITIONS
  function isOn($d in Drawer) =
         (drawerLed($d) = ON)
  ...
  // RULE DEFINITIONS
  rule r_reset($drawer in Drawer) = ...
  ...
  // INVARIANTS AND PROPERTIES
  invariant inv_drawer1 over Drawer = ...
  ...
  // MAIN Rule
  main rule r_Main = ...
// INITIAL STATE
default init s0:
  // Turn-off all the LEDs for the Drawers
  function drawerLed($drawer in Drawer) = OFF
  ...
```

main rule

initialization

ASMETA Eclipse Editor

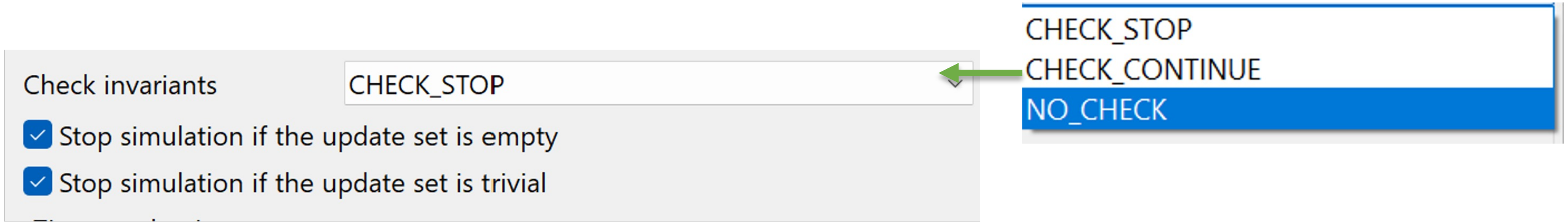


Simulation AsmetaS

AsmetaS

- Before starting the simulation set the preferences:

Window -> Preferences -> Asmeta -> Simulator



AsmetaS

- Axiom checker

  to check model invariants

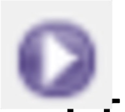
- Consistent Updates checking

  to check for inconsistent updates

AsmetaS

- Random 

Values to monitored functions are automatically assigned

- Interactive 

Values to monitored functions are inserted by the user

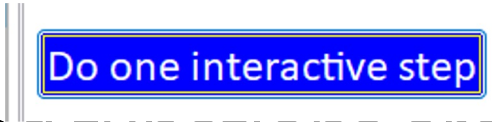
Animation AsmetaA

AsmetaA

- Random

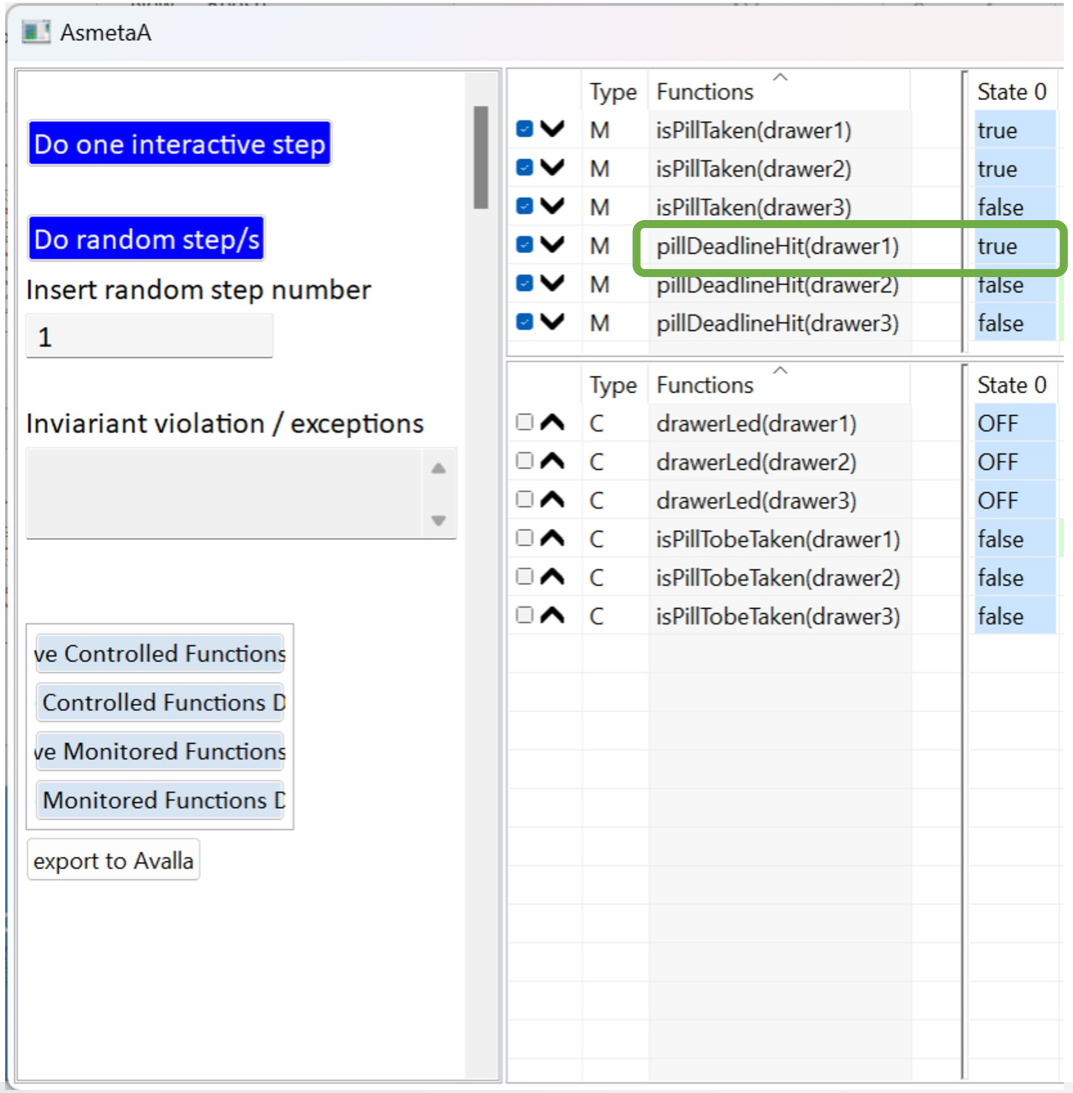
Values to monitored functions are automatically assigned

- Interactive



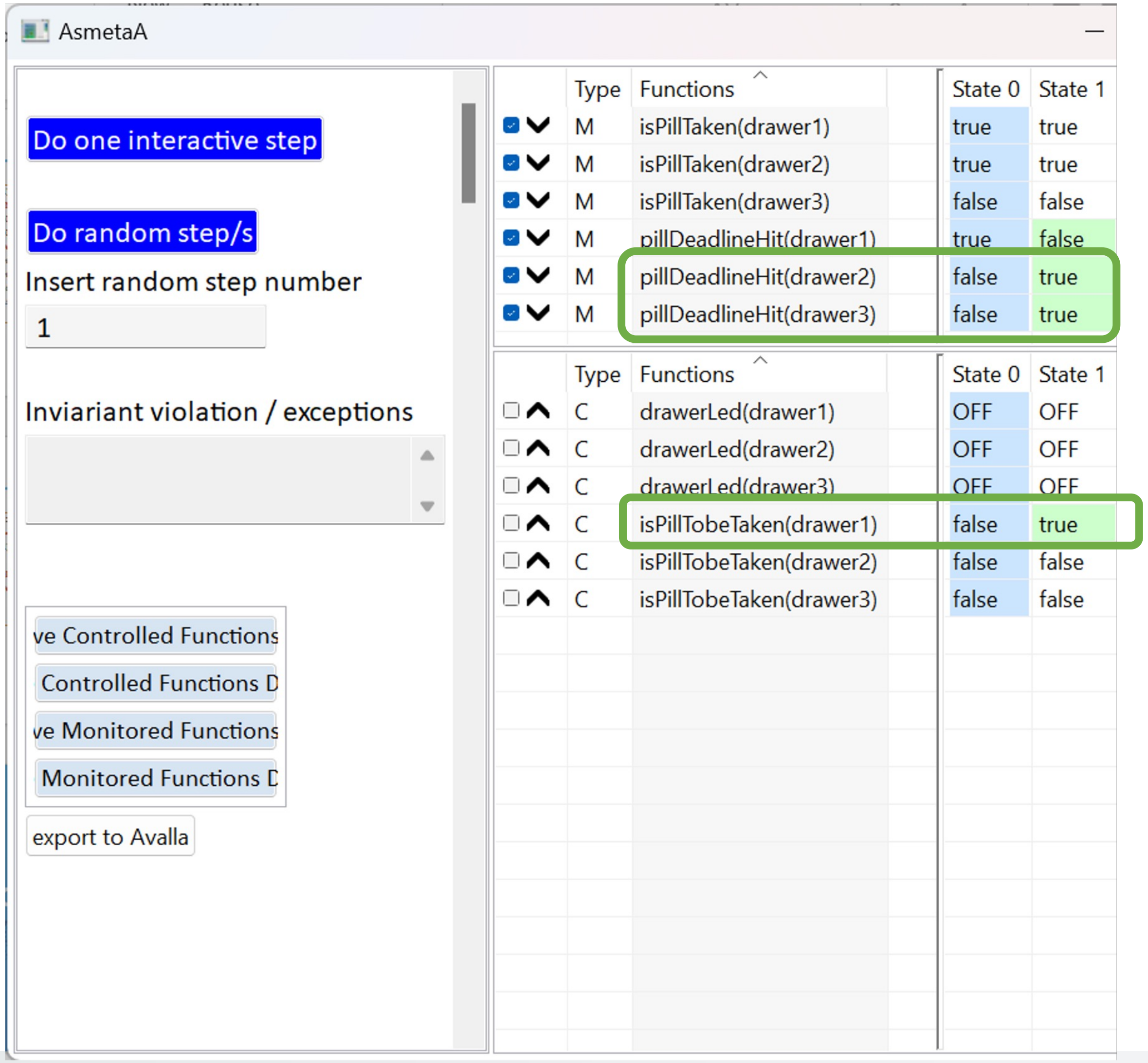
Values to monitored functions are inserted by the user

AsmetaA: random animation



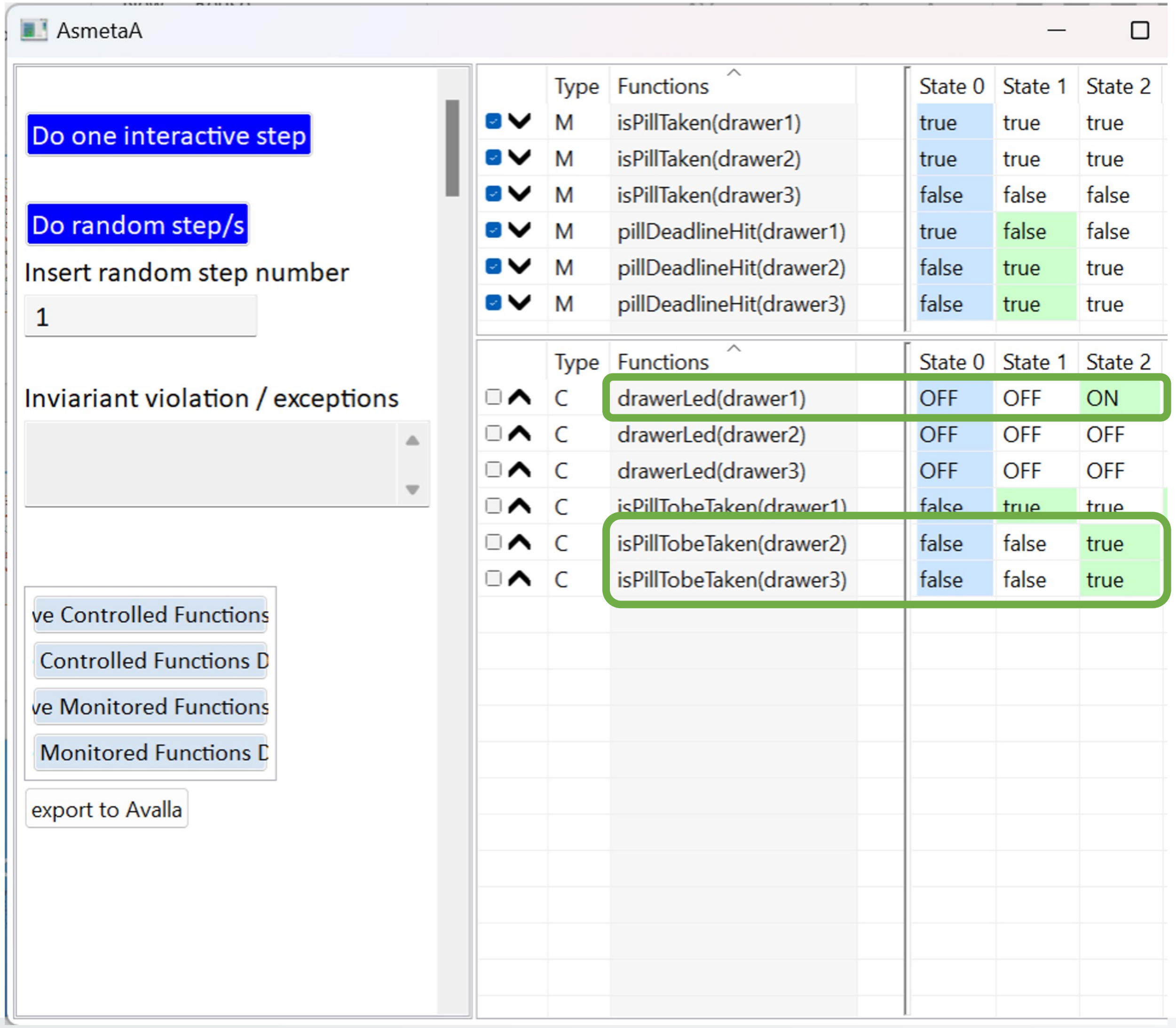
- The pill in drawer 1 hits the deadline

AsmetaA: random animation



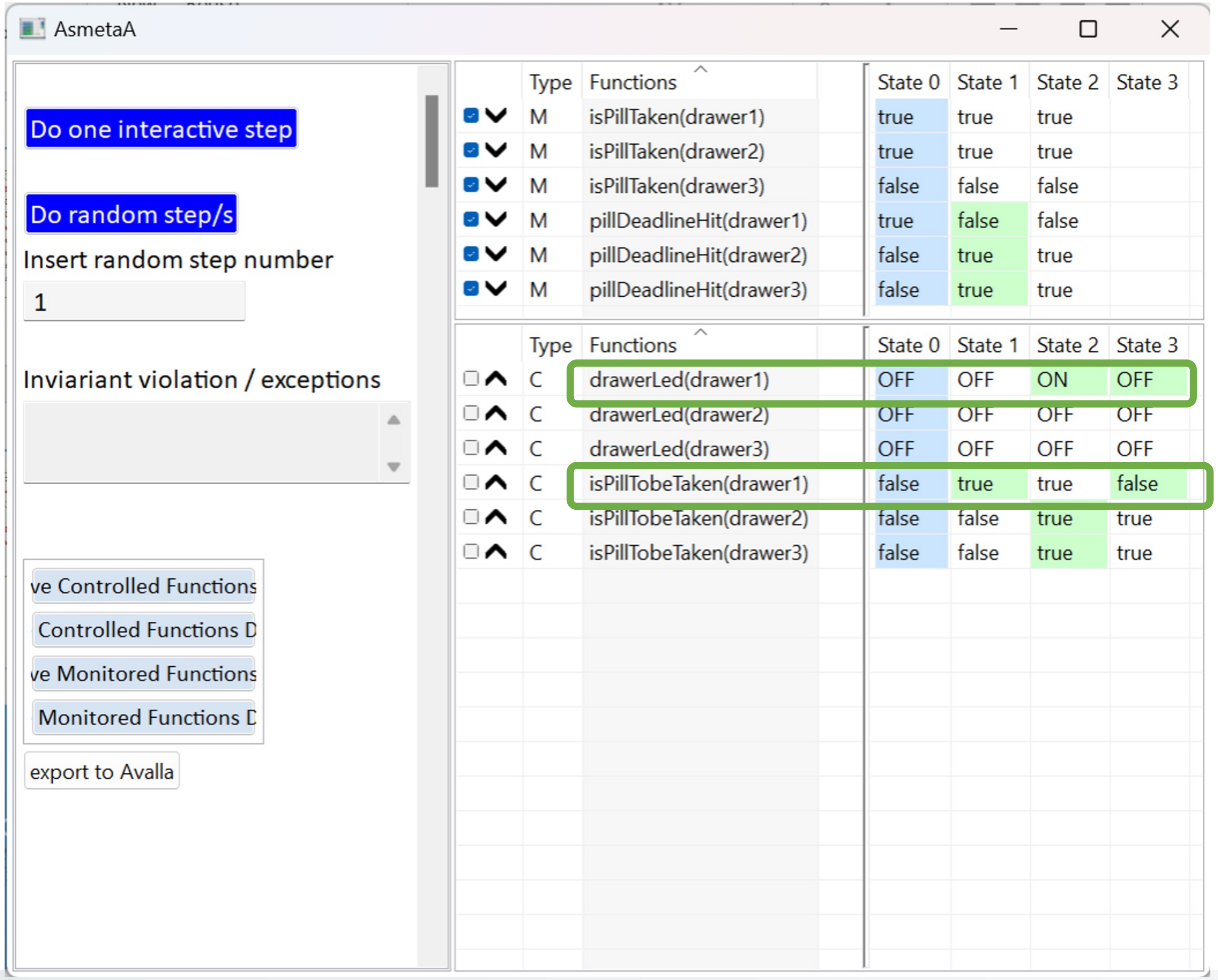
- The pill in drawer 2 and drawer 3 hit the deadline
- The pill in drawer 1 becomes to be taken

AsmetaA: random animation



- The LED in drawer 1 becomes ON
- The pills in drawer 2 and drawer 3 become to be taken

AsmetaA: random animation



- The LED in drawer 1 becomes OFF

Model validation by scenarios construction: AsmetaV

Scenario-based Validation

Scenarios are

- Descriptions of *external actor actions* and *reactions of the system*
- Useful to check the correct behavior of the model
- Written in the **Avalla** language

Detailed look of an Avalla scenario

```
scenario scenario_ground
load pillbox_ground.asm

// Initially all deadlines are not hit
set pillDeadlineHit(drawer1) := false;
set pillDeadlineHit(drawer2) := false;
set pillDeadlineHit(drawer3) := false;
set isPillTaken(drawer3) := false;
set isPillTaken(drawer1) := false;
set isPillTaken(drawer2) := false;
step
// Check that all leds are off
check drawerLed(drawer1) = OFF;
check drawerLed(drawer2) = OFF;
check drawerLed(drawer3) = OFF;
// Now, the time for the pill in the drawer 1 comes
set pillDeadlineHit(drawer1) := true;
step
// Check that pill is ready to be taken
check isPillTobeTaken(drawer1) = true;
```

Scenario name

- Keyword **scenario** followed by the name

Detailed look of an Avalla scenario

```
scenario scenario_ground
load pillbox_ground.asm

// Initially all deadlines are not hit
set pillDeadlineHit(drawer1) := false;
set pillDeadlineHit(drawer2) := false;
set pillDeadlineHit(drawer3) := false;
set isPillTaken(drawer3) := false;
set isPillTaken(drawer1) := false;
set isPillTaken(drawer2) := false;
step
// Check that all leds are off
check drawerLed(drawer1) = OFF;
check drawerLed(drawer2) = OFF;
check drawerLed(drawer3) = OFF;
// Now, the time for the pill in the drawer 1 comes
set pillDeadlineHit(drawer1) := true;
step
// Check that pill is ready to be taken
check isPillTobeTaken(drawer1) = true;
```

Loading Asmeta specification

- Keyword **load** followed by the name (or the path) of a specification
- Each scenario is executed against an Asmeta specification

Detailed look of an Avalla scenario

```
scenario scenario_ground
load pillbox_ground.asm

// Initially all deadlines are not hit
set pillDeadlineHit(drawer1) := false;
set pillDeadlineHit(drawer2) := false;
set pillDeadlineHit(drawer3) := false;
set isPillTaken(drawer3) := false;
set isPillTaken(drawer1) := false;
set isPillTaken(drawer2) := false;
step
// Check that all leds are off
check drawerLed(drawer1) = OFF;
check drawerLed(drawer2) = OFF;
check drawerLed(drawer3) = OFF;
// Now, the time for the pill in the drawer 1 comes
set pillDeadlineHit(drawer1) := true;
step
// Check that pill is ready to be taken
check isPillTobeTaken(drawer1) = true;
```

Setting monitored functions

- Keyword **set** followed by
  - the name of the monitored location (read by the machine from the environment)
  - the value to be assigned

Detailed look of an Avalla scenario

```
scenario scenario_ground
load pillbox_ground.asm

// Initially all deadlines are not hit
set pillDeadlineHit(drawer1) := false;
set pillDeadlineHit(drawer2) := false;
set pillDeadlineHit(drawer3) := false;
set isPillTaken(drawer3) := false;
set isPillTaken(drawer1) := false;
set isPillTaken(drawer2) := false;
step
// Check that all leds are off
check drawerLed(drawer1) = OFF;
check drawerLed(drawer2) = OFF;
check drawerLed(drawer3) = OFF;
// Now, the time for the pill in the drawer 1 comes
set pillDeadlineHit(drawer1) := true;
step
// Check that pill is ready to be taken
check isPillTobeTaken(drawer1) = true;
```

Step execution

- **step** command

… or …

- **stepUntil** command, followed by a Boolean condition

Detailed look of an Avalla scenario

```
scenario scenario_ground
load pillbox_ground.asm

// Initially all deadlines are not hit
set pillDeadlineHit(drawer1) := false;
set pillDeadlineHit(drawer2) := false;
set pillDeadlineHit(drawer3) := false;
set isPillTaken(drawer3) := false;
set isPillTaken(drawer1) := false;
set isPillTaken(drawer2) := false;
step
// Check that all leds are off
check drawerLed(drawer1) = OFF;
check drawerLed(drawer2) = OFF;
check drawerLed(drawer3) = OFF;
// Now, the time for the pill in the drawer 1 comes
set pillDeadlineHit(drawer1) := true;
step
// Check that pill is ready to be taken
check isPillTobeTaken(drawer1) = true;
```

Checking controlled functions

- Keyword **check** followed by
  - the name of the controlled location
  - the value to be checked
- The check can either PASS or FAIL

AsmetaV

After having written a scenario, it can be executed through:

- The simple AsmetaV validator
- The execution of a scenario through animation
- The AsmetaV validator keeping track of covered rules

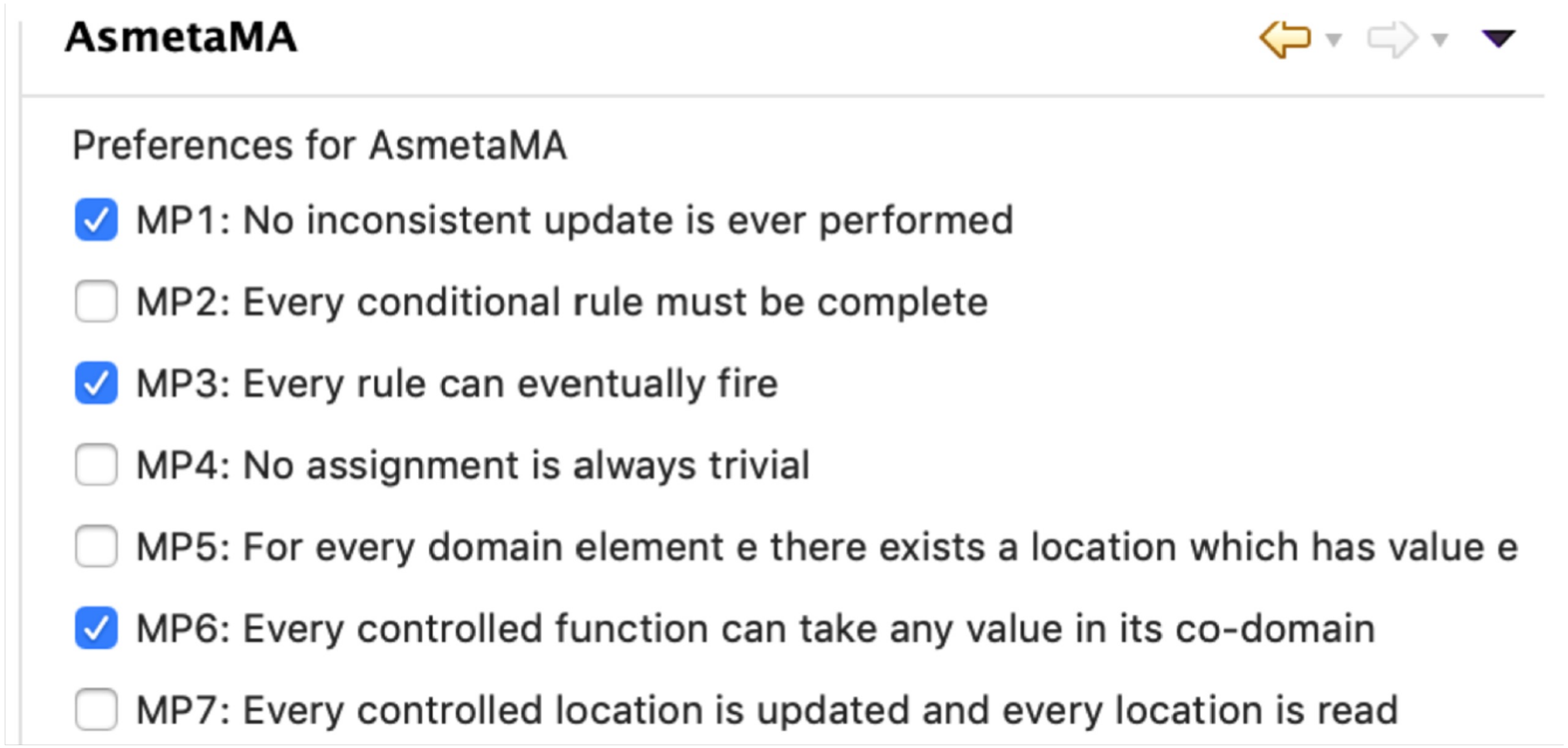
Static analysis of models AsmetaMA

Model Review

- The *automatic* model reviewing activity:
  - Is a form of **static analysis**
  - Automatically captures modeling errors (e.g., inconsistent updates, dead specification parts, …) through 7 meta properties checked by exploiting the model checker

AsmetaMA

- Before starting the model reviewer, set the preferences:



Property Verification AsmetaSMV

Model checking

- Formal verification technique of properties defined in a temporal logic
- A model checker works in three steps:
  1. definition of a model *M* using the *Kripke structures*
  2. definition of a temporal formula φ that describes a property that we want to verify
  3. the model checker verifies that $M \vDash \varphi$
- Exhaustive verification of all the state space
  - With some limitations (finite domains….)

In ASMETA:

```
//
asm pillbox_ground

import StandardLibrary
import CTLLibrary
import LTLLibrary

signature:
    // DOMAINS
    abstract domain Drawer
    // MONITORED AND CONTROLLED FUNCTIONS
    dynamic monitored isPillTaken: Drawer -> Boolean
    // DERIVED FUNCTIONS
    // ...
definitions:
    // STATIC AND DERIVED FUNCTIONS DEFINITIONS
    // RULE DEFINITIONS
    rule r_reset($drawer in Drawer) = skip

    /// rules

    // INVARIANTS AND TEMPORAL PROPERTIES

    CTLSPEC ag((forall $d in Drawer with true))

    // MAIN Rule
    main rule r_Main = skip

default init s0:
```

Domains, functions and rules → Machine M (Kripke structure)

TL properties → Properties to be proven

Temporal logics

- discrete time logics
- Linear Time Logics (LTL) represent time as infinite sequences of instant
  - you can declare properties that must be true over all sequences
- Computational Time Logics (CTL) represent time as a tree, where the root is the initial instant and its children the possible evolutions of the system
  - you can declare properties concerning all the paths or just some of them

Properties for the PillBox

- If the patient takes the pill in drawer1, the light will turn on eventually

```
CTLSPEC ag(pillDeadlineHit(drawer1) implies ef(isOn(drawer1)))
```

- Max one led is on

```
CTLSPEC ag((forall $d in Drawer with isOn($d)
                implies (not areOthersOn($d))))
```

Counter example

- If a property is false, a counter example is shown

```
CTLSPEC
ag(pillDeadlineHit(drawer1)
implies af(isOn(drawer1)))
```

```
-- specification AG (pillDeadlineHit(DRAWER1) ->
AF drawerLed(DRAWER1) = ON) is false
-- as demonstrated by the following execution
sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
pillDeadlineHit(DRAWER1) = false
drawerLed(DRAWER1) = OFF
var_$drawer_0 = DRAWER1
drawerLed(DRAWER2) = OFF
isPillTobeTaken(DRAWER2) = false
drawerLed(DRAWER3) = OFF
isPillTobeTaken(DRAWER3) = false
isPillTobeTaken(DRAWER1) = false
isPillTaken(DRAWER1) = false
isPillTaken(DRAWER2) = false
pillDeadlineHit(DRAWER2) = false
isPillTaken(DRAWER3) = false
pillDeadlineHit(DRAWER3) = false
areOthersOn(DRAWER3) = false
areOthersOn(DRAWER2) = false
areOthersOn(DRAWER1) = false
-> State: 1.2 <-
pillDeadlineHit(DRAWER1) = true
pillDeadlineHit(DRAWER2) = true
-- Loop starts here
-> State: 1.3 <-
…
```

Model Refinement

Pill-Box case study: first refinement step

First model Pillbox ground	**Second model** Pillbox time
• A drawer contains only a single pill (no slots) • Time not modeled: information on the time passed is given by an external event	• A drawer contains only a single pill (no slots) • Time is modeled using a timer

Pillbox time

```
import ../STDL/TimeLibrarySimple
static tenMinutes: Timer
```

Time library features:

- Check if timer is expired
- Reset a timer

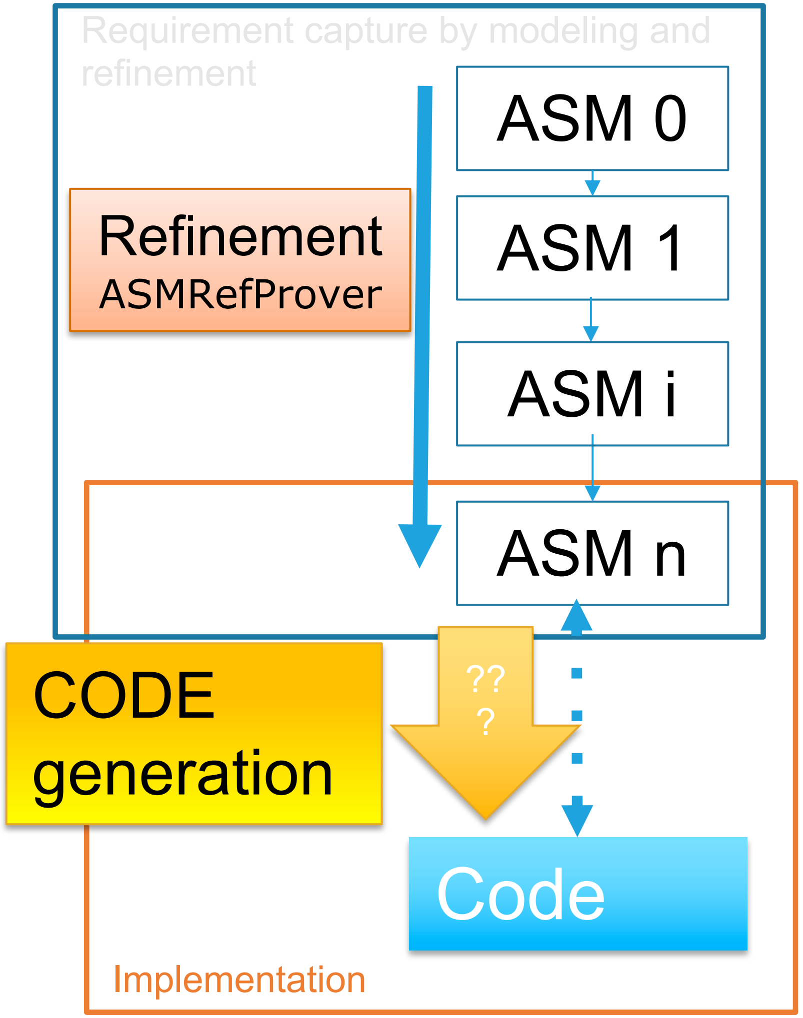
```
function duration($t in Timer) = 600 // Timer initialization
function start($t in Timer) = currentTime($t)
// From the Time library
function currentTime($t in Timer) = mCurrTimeSecs
```

Pill-Box case study: second refinement step

Second model Pillbox time	**Third model** Pillbox final
• A drawer contains only a single pill (no slots) • Time is modeled using a timer	• A drawer contains multiple slots

Code generation

Code generation from Asmeta spec



- model-driven engineering:
  - Code are generated
- the last refinement can be translated to code
  - C++
  - C++ for Arduino
  - Java
- **Asmeta2C++** tool

Asm to C++ : example

```
asm LedSystem
enum LedState = {LOW, HIGH}
monitored dimValue -> Integer
controlled led -> LedState
rule r_setLedLow = led := LOW
rule r_setLedHigh = led := HIGH
rule r_Main =
  if dimValue > 400 then r_setLedHigh[]
  else r_setLedLow[]
  endif
```

.h

LedSystem.h

```
class LedSystem{
 // DOMAIN DEFINITIONS
 enum LedState {LOW,
HIGH};
 // FUNCTIONS
 int dimValue;
 LedState led[2];
public:
 // RULE DEFINITION
 void r_setLedLow();
 void r_setLedHigh();
 void r_Main();
 void getInputs();
 void setOutputs();
 void fireUpdateSet();
};
```

.cpp

LedSystem.cpp

```
void LedSystem:: r_setLedLow(){
 led[1] = LOW
}
void LedSystem::r_Main(){
  if (dimValue > 400){
    r_setLedHigh();
  } else{ .. }
}
```

Answer 1

IS THIS ENOUGH?

- Models are useful if
  1. Specification can be executed/animated
  2. Formal analysis
     - Static
     - Property verification
  3. Tests /scenarios can be introduced
  4. Models can be refined
  5. (part of the) code can be generated

New uses of models

Models @ runtime

Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini,
Nico Pellegrinelli and Patrizia Scandurra

Safety enforcement for autonomous driving on a simulated highway using Asmeta models@run.time

ABZ 2025

What is it?

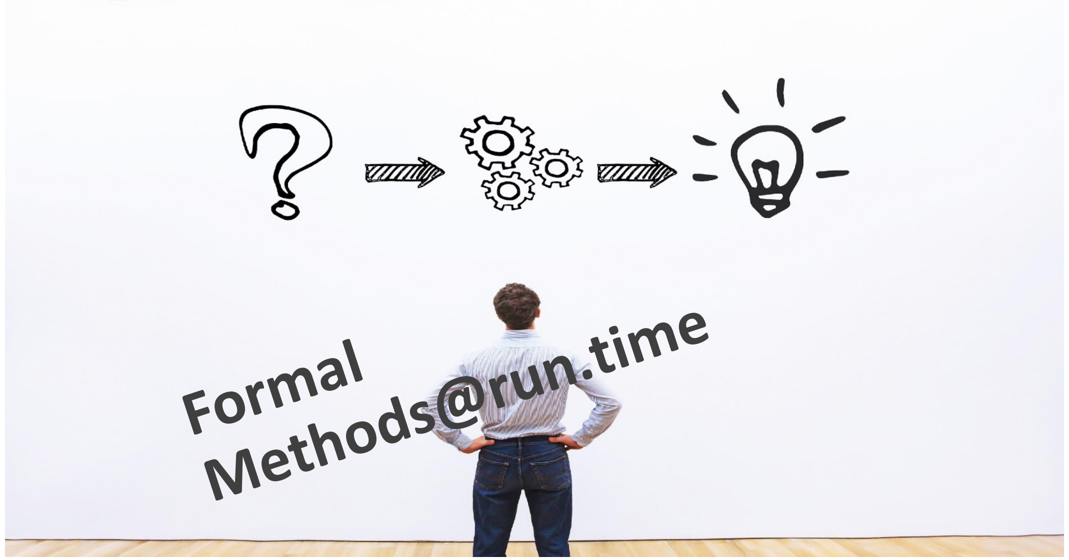
- Safety Assurance in **autonomous software-intensive systems**
- *Formal Methods@run.time*: **Runtime Safety Enforcement** (**RSE**)
- RSE with **Abstract State Machines (ASM) @runtime** and the ASMETA runtime simulator
- **RSE for AVs on a simulated highway**
    - *Offline* V&V of the ASM enforcement model(s)
    - RSE framework architecture
    - *Online* experimental evaluation about effectiveness and efficiency

Software-intensive systems and safety assurance

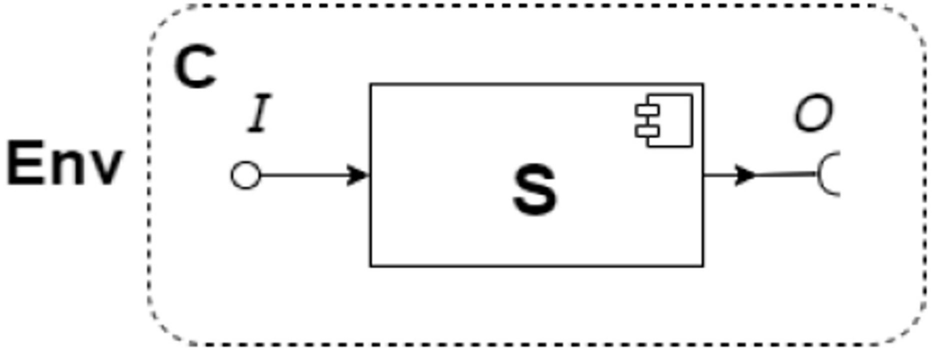
- **Increasingly autonomous, leverage AI/ML** to operate 24/7 and make decisions in real-time, under uncertainty, and with no human intervention
- Since they **integrate "black boxes"**, they are **opaque and less predictable**

Safety Assurance Problem:

How can we ensure that they function safely at *any time* during their lifecycle?



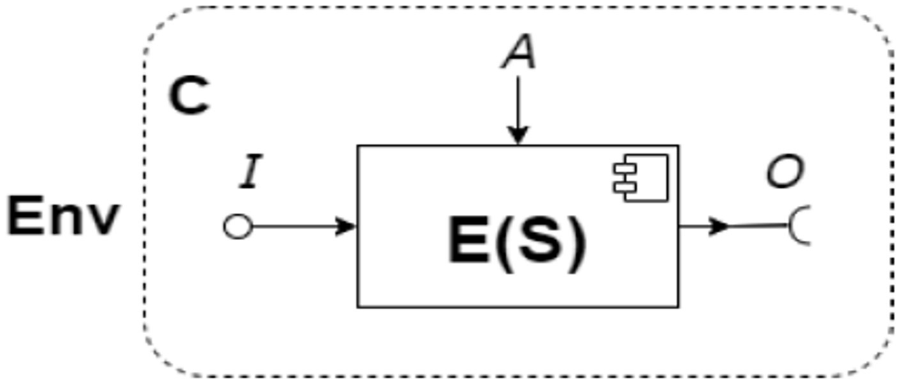
Runtime Safety Enforcement (RSE)*

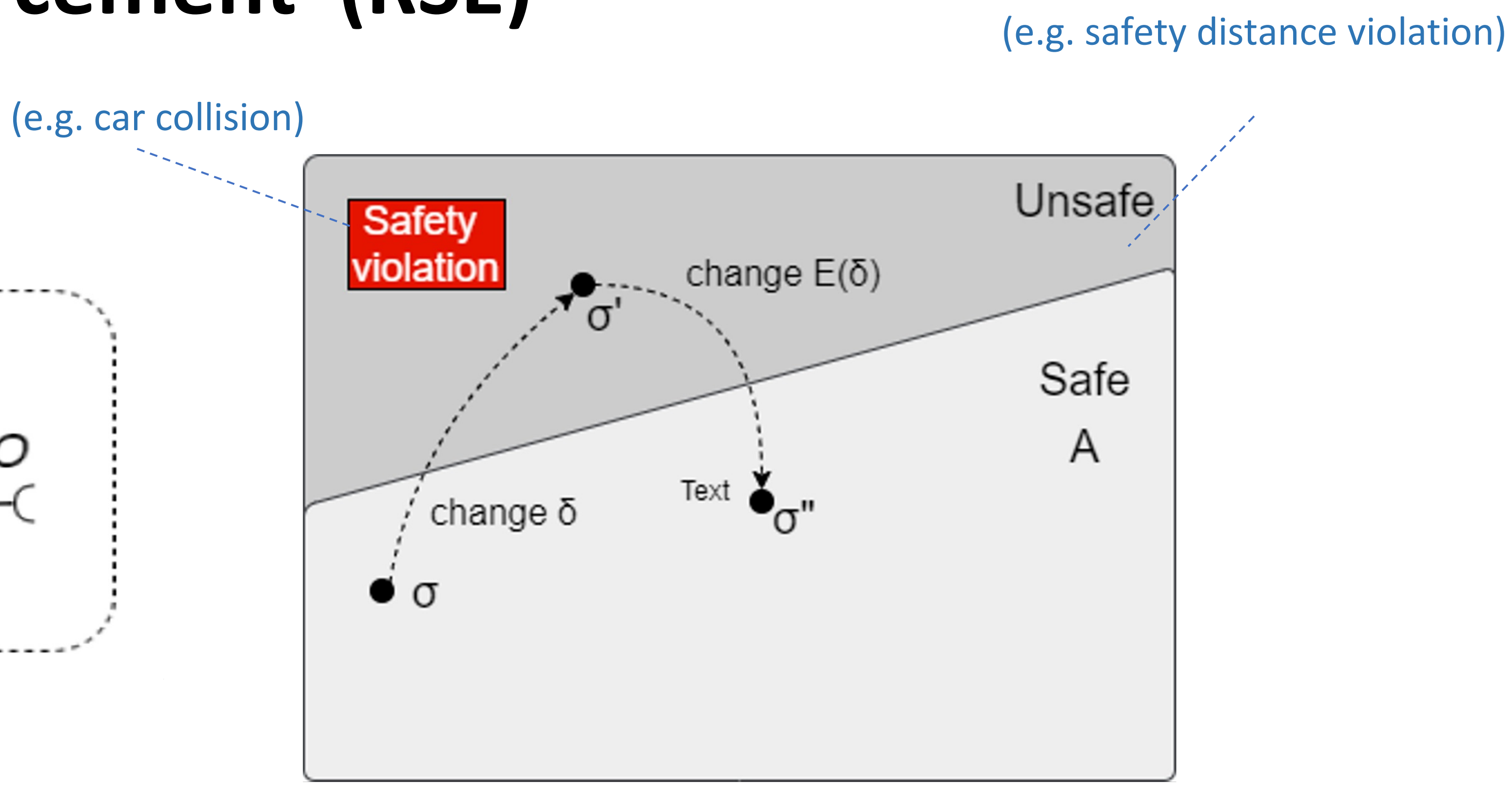


- A **software system *S*** executes in a context C made of environmental entities Env that interact with S trough I/O events

[*] Silvia Bonfanti, Elvinia Riccobene, Patrizia Scandurra: A component framework for the runtime enforcement of safety properties. J. Syst. Softw. 198: 111605 (2023)

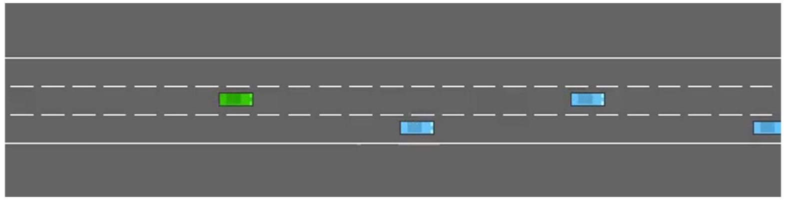
Runtime Safety Enforcement (RSE)





- **Enforcer *E steers S*** to stay in the ***safe region*** (where **safety assertions *A*** hold)
- If S performs an *unsafe step* δ, then *E* makes change E(δ) to bring S back to the safe region
- Ideally: one single change E(δ); in general, more adaptation changes may be necessary

RSE for simulated AVs
(ABZ 2025 case study)



- **RSE co-executes with the pre-trained (unsafe) driving agent** in the simulated highway
- **Output sanitization** of the *ego*'s action {FASTER, SLOWER, IDLE, LANE_LEFT, LANE_RIGHT}
- **Requirements** : **G1: safety** (SAF – no collisions), **G2: high travelled distance**, and **G3:** virtuous behavior (**rightmost lane to prefer** – *multi-lane scenario*)
- **Enforcement strategies** served **by an ASMETA model@run.time**

Goal	Strategy name	Enforcement rule
G1	Go super safe	Brake if the worst case safety distance is violated
G1	Go safe	Brake if the safety distance is violated
G2	Go fast safely	Increase speed if *far away*, i.e. the distance to the front vehicle is $x\%$ (e.g., 70%) greater than the required safety distance
G3	Take the rightmost free lane	Change lane to right if the lane directly right is free

Black-box enforcement: ASMETA models for output sanitization and goal coverage

Enforcement Model	Strategies	Goals
SuperSafe.asm	Go super safe	G1
Slower.asm	Go safe	G1
Faster.asm	Go safe, Go fast safely	G1, G2
KeepRight.asm	Go safe, Go fast safely,	G1, G2
	Take the rightmost free lane	G3

- Different enforcement rules for different goals
- All models available at:

RSE for simulated AVs : offline V&V of ASMETA enforcement model(s)

- Functional correctness of an ASMETA enforcement model must be proved at design-time before its use at run-time (online)
- **Model validation by scenarios** using the validator AsmetaV
- **Invariant verification** using the AsmetaSMV – nuXmv (real numbers!)

```
/*If the ego vehicle is close to the front vehicle, break (go SLOWER)*/
INVARSPEC NAME invar_01 := (actual_distance<=dRSS) -> next(outAction=SLOWER)

/*If the front vehicle is far enough from the ego vehicle, increase the speed (go FAST)*/
INVARSPEC NAME invar_02 := (actual_distance>(dRSS*gofast_perc)) -> next(outAction
      =FASTER)

/*If there is no risk of collision, keep the action decided by the agent*/
INVARSPEC NAME invar_03 := (actual_distance>dRSS and actual_distance<=(dRSS*
      gofast_perc)) -> next(outAction=currentAgentAction)
```

Demo/video

Answer 2

- Systems can be so complex that they CANNOT be modelled
  - Or it is not worthwhile
  - Example a NN

- A model can be defined to check the behaviour of the system

- Can increase the trust and reliability of a AI system

Digital twins

The SAFEST project is funded by the European Union - Next Generation EU, Mission 4, Component 2, Investment 1.1, CUP F53D23004230006, under the National Recovery and Resilience Plan (NRRP) – Grant Assignment Decree No. 959 adopted on 30 June 2023 by the Italian Ministry of University and Research (MUR).

Finanziato dall'Unione europea NextGenerationEU

Ministero dell'Università e della Ricerca

Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA

UNIVERSITÀ DEGLI STUDI DI BERGAMO
Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

SAFEST GOALS – USING DTs for;

- taming the complexity caused by the heterogeneity of the DT components that must be built, operated, coordinated, and evolved together with their **physical and human** counterparts;
- **Increasing the level of trust** in the results and indications coming from a DT, despite modelling approximations and uncertainties caused by incomplete or imprecise data collected in the field.

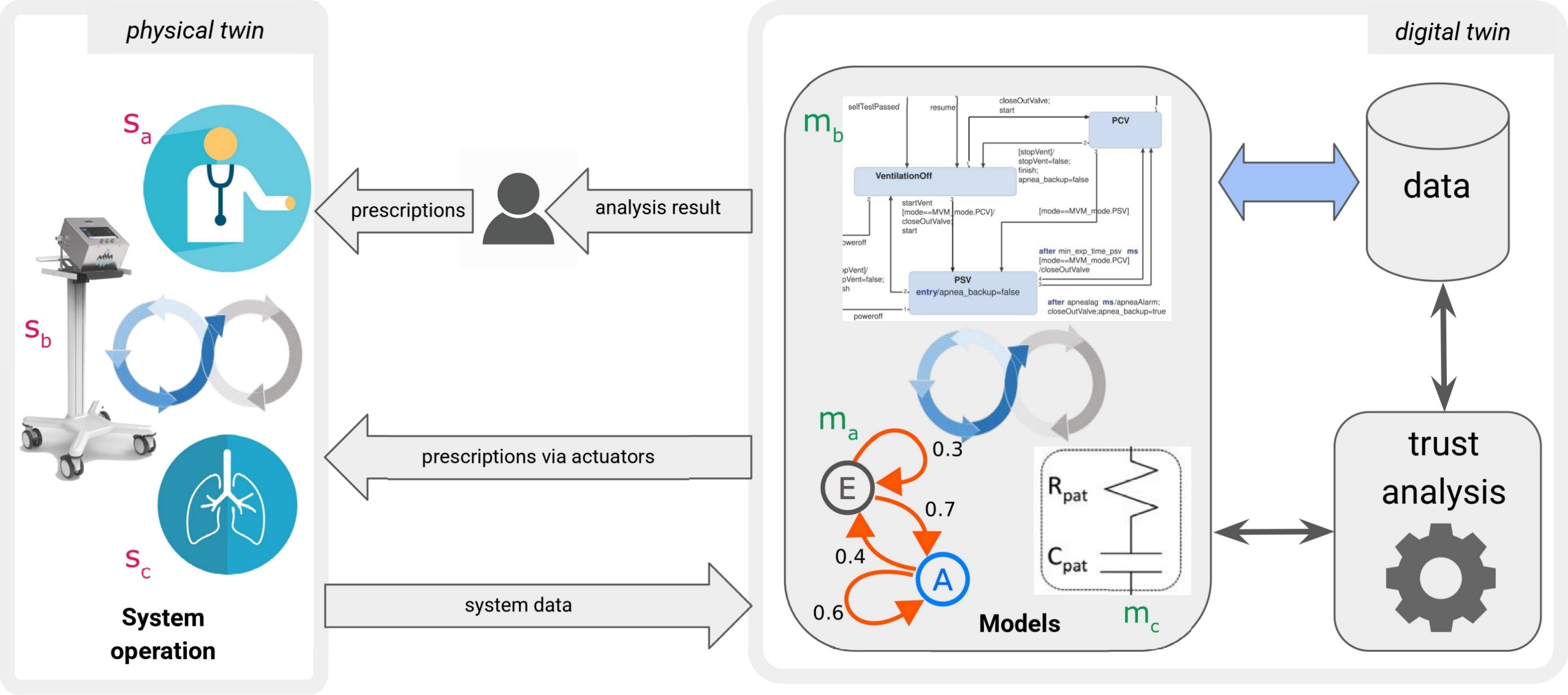
SAFEST:

▶️ developing modeling notations for evolving heterogeneous systems with uncertainty

▶️ providing trust assurances in terms of behavioral conformance, safety, dependability, security, and performance.

- The project's methods and tools will be evaluated through a medical domain case study.

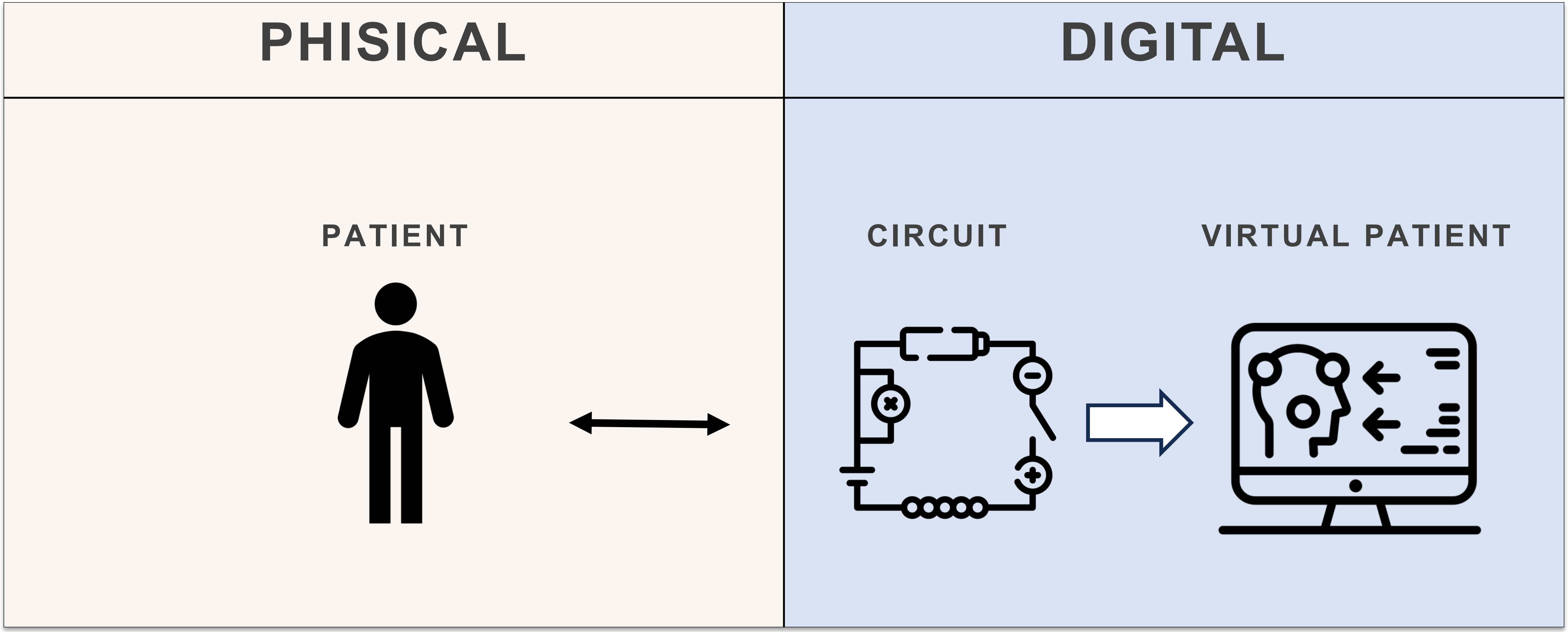
SAFEST - truSt Assurance of digital twins For mEdical cyber-phySical sysTems



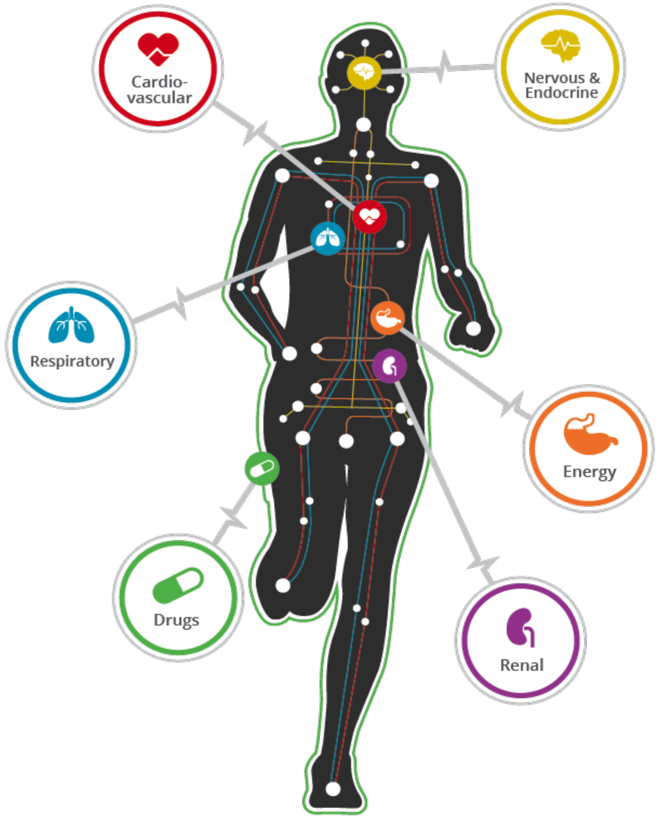
BREATHE: A Digital twin-based Respiratory System Simulator for Mechanical Ventilator Testing and Training

- Medical simulation has become a crucial element in training and furthering clinical skills, particularly in mechanical ventilation.
- Use of digital twins for designing a system that allows the interconnection between a respiratory simulator and a virtual mechanical ventilator, intended for **testing ventilators under development.**

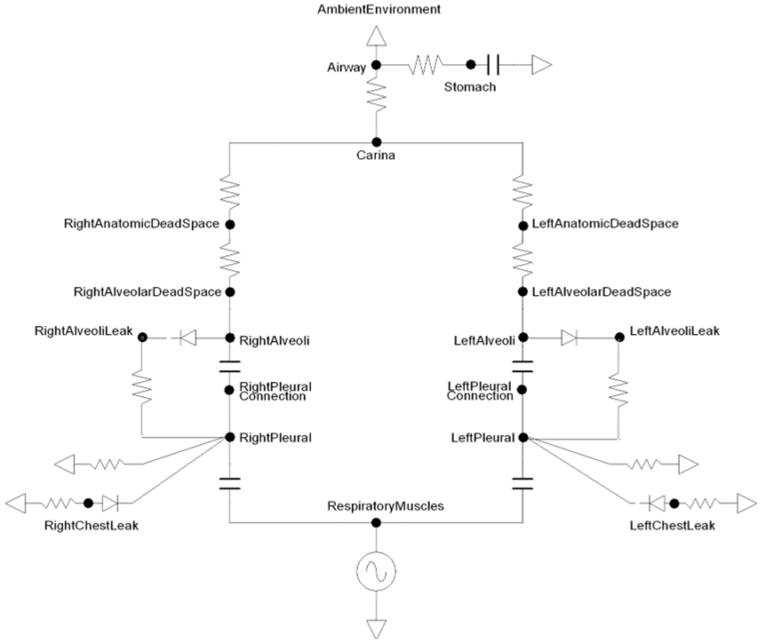
DIGITAL TWIN



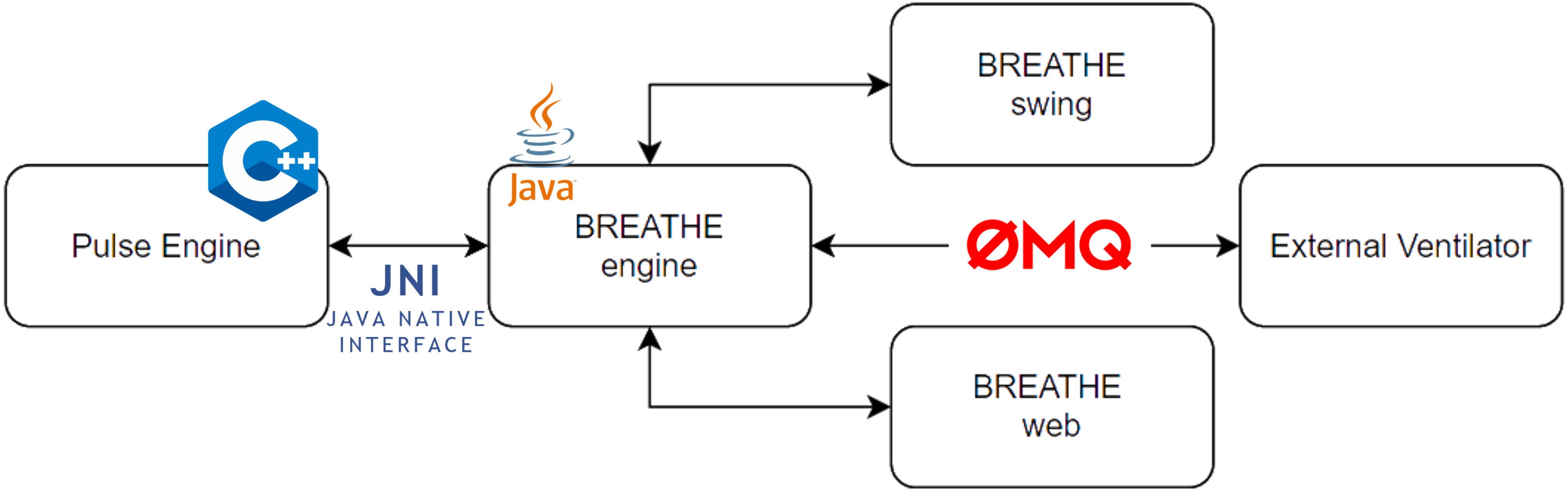
Pulse
Physiology Engine



- OPEN SOURCE
- WHOLE PATIENT PHYSIOLOGY
- M



BREATHE IMPLEMENTATION

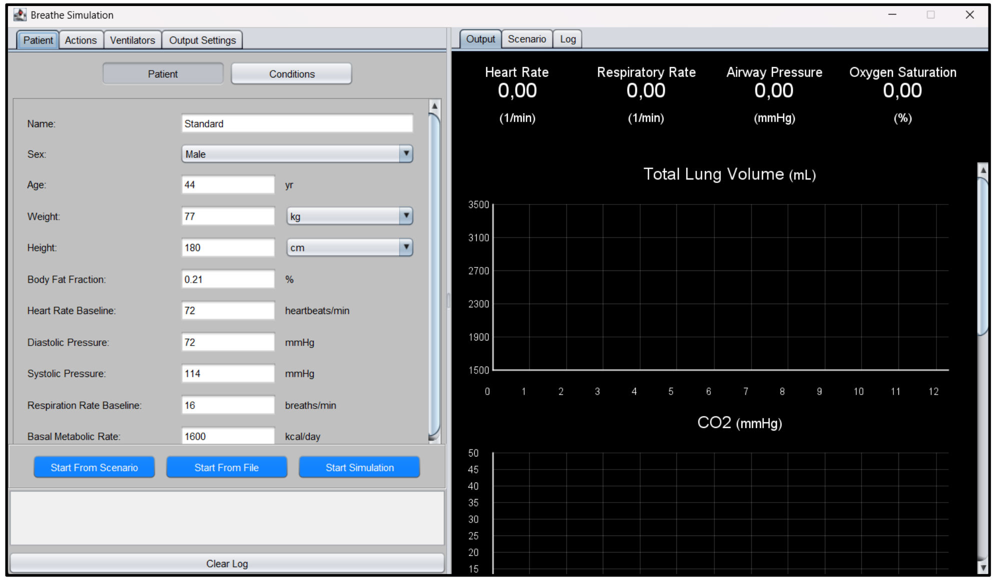


BREATHE INTERFACE

ENGINE

TESTING

TRAINING

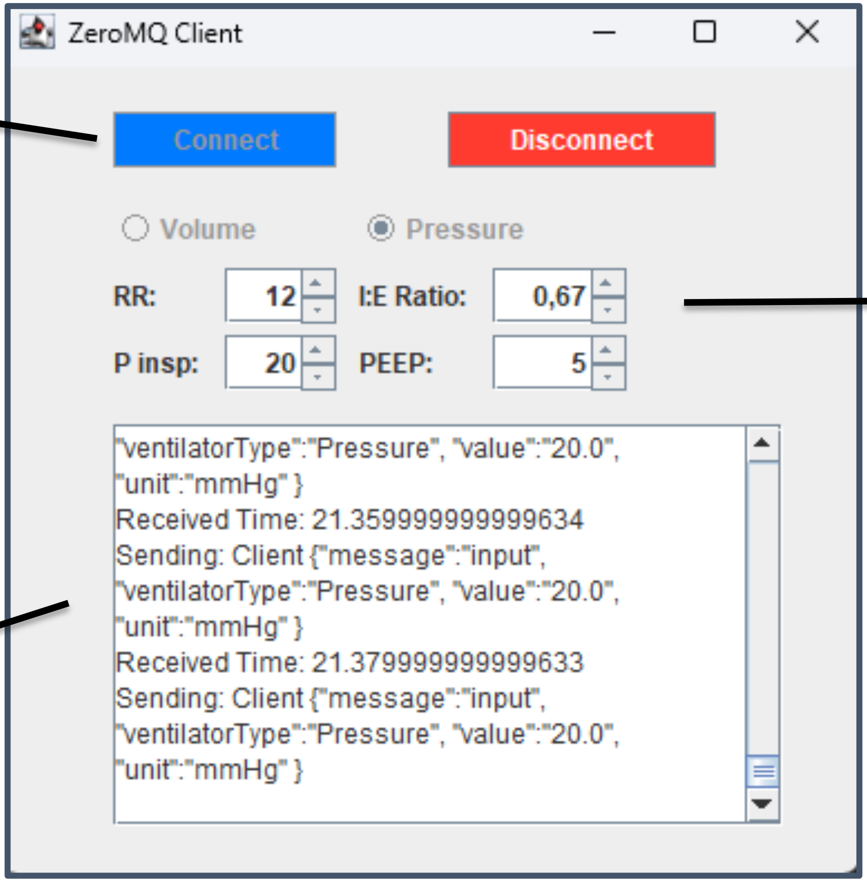


EXTERNAL VENTILATOR

CONNECTION

PARAMETERS

OUTPUT



Demo/video

Answer 3

- Models are useful if
  1. Can be used together with systems that CANNOT be modelled
  2. Can be used instead of real objects

Problems with models

Model/implementation changes

- Models need to be updated / modified
  - **MODEL EVOLUTION** ???
  - Refinement ???
- Models become inconsistent wrt the implementations
  - Dilemma: Spend resources to keep them updated or accept that they are not?
  - WHAT to do with artefacts?

Some approaches - models

- Models are "refined" together with the system
  - Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, Yu Lei, and Feng Duan *RATE: A model-based testing approach that combines model refinement and test execution* in Software Testing, Verification and Reliability, Wiley, vol. 33, n. 2 (2023)

Models and tests evolve together

- AMOST and SPLC

Answer 4

- Models can be a burden when developing software

The future of models

Models for building software?

Classical use of models: to build software

MODELS

If moving objects/elements in a house were easy, would the bricklayer use the models????

REALIZATION

Doubts

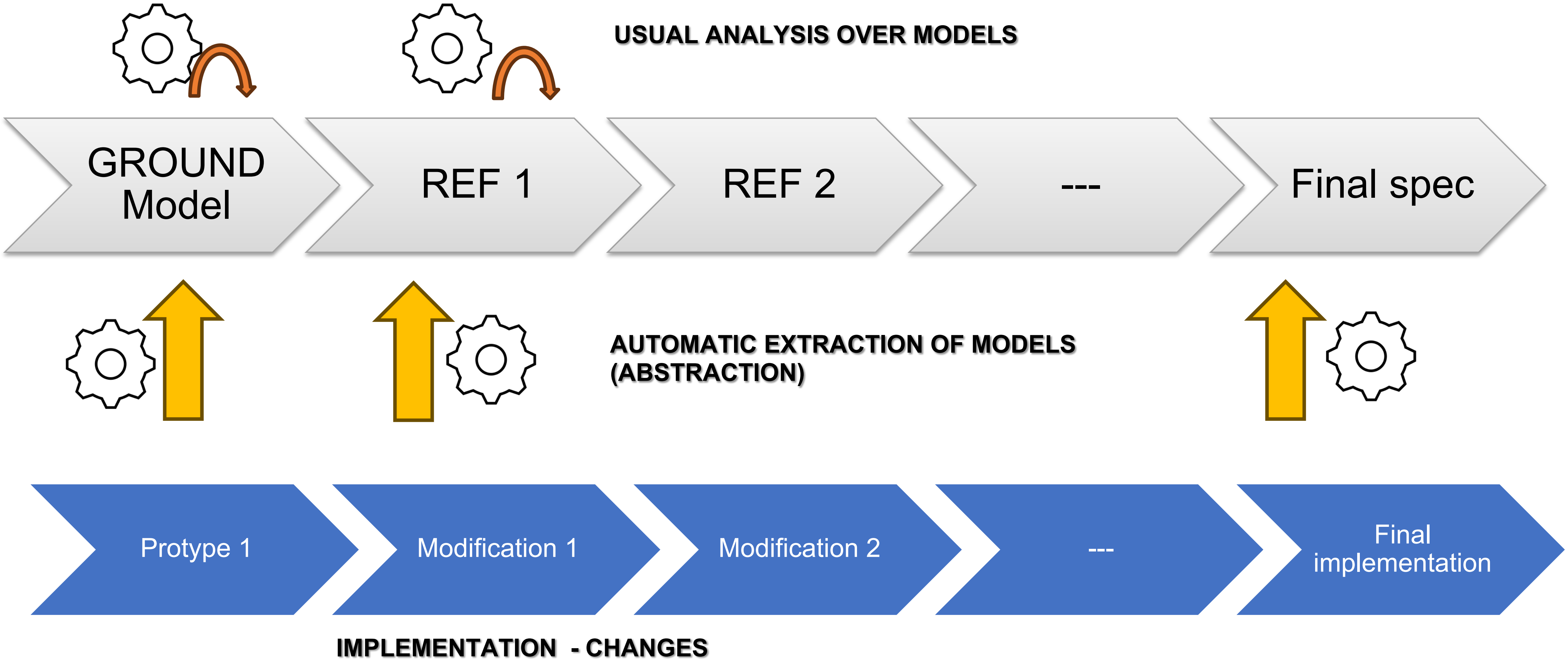
- building software requires models?
- Is it worthwhile?
- What kind of models?

New roles of models

- REVERSE MODELING
  when models are extracted from existing code

- MODELS AND CODE COEVOLUTION
  - when models and code co-evolve together linked in a formal way.

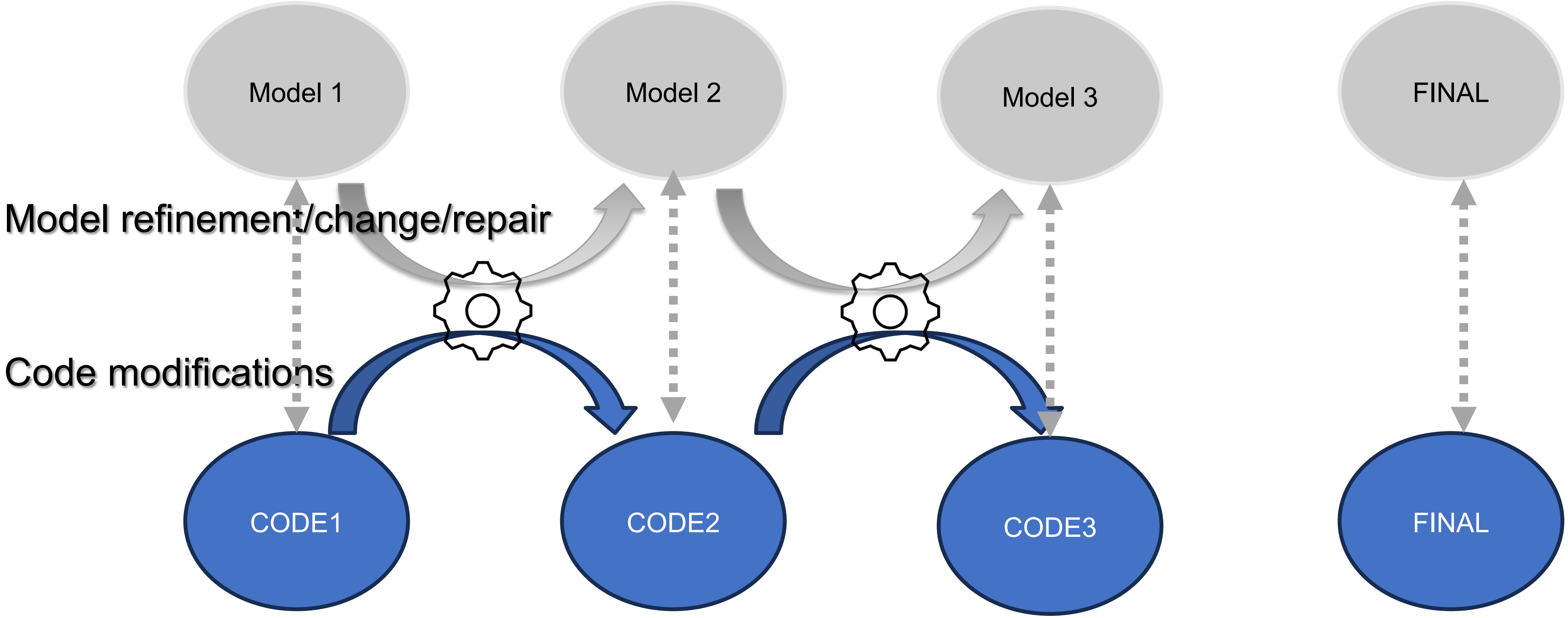
Reverse Modeling



Reverse Modeling - concepts

1. Code is modified (with possible limitations)
2. Models are extracted from code (automatically – abstraction)
3. Analysis activities (verification and validation, refinement checking) are performed at the level of the model (abstract)
4. Only one artefact (code) is maintained

COEVOLUTION



COEVOLUTION - concepts

1. Models and implementations evolve together

2. Every time one is changed, the other is synchronized (automatically?)

3. If a discrepancy between model and code is found, an automatic repair is executed

Conclusions

- Formal Models are great ……
- But or they will revise their role, or they risk to become irrelevant (and the community)
- Ways of use them are promising (digital twins, safety enforcer)
- New ways must be explored