Testing Distributed Systems

Part I: Introduction to Model-Based Testing
2012-08-01

Jan Peleska and Wen-ling Huang
University of Bremen
{jp,huang}@informatik.uni-bremen.de

Acknowledgements. *This seminar has been elaborated in the context of the EU FP7 COMPASS project under grant agreement no.287829.*

Model-Based Testing

- Model-based testing (MBT) as defined in Wikipedia [1]
  - *"**Model-based testing** is the application of Model based design for designing and optimally executing the necessary artifacts to perform software testing. Models can be used to represent the desired behavior of the System Under Test (SUT), or to represent the desired testing strategies and testing environment."*

Model-Based Testing

- Model-based testing (MBT) as defined in Wikipedia [1]
  - "***Model-based testing*** *is the a* *of* *Model based design* *for des* *optimally executing the necessary artifacts to perform* *software testing*. *Models can be used to represent the desired behavior of the System Under Test (SUT), or to represent the desired testing strategies and testing environment."*

We would say:
system or software testing

Model-Based Testing

- Let's analyze this definition
  - *"Apply model-based design"*: use modeling formalism to specify any test-related information
  - *"Models ...represent desired behavior of ... SUT"*: this is the **gold-plated approach to MBT**
    - Just specify the desired capabilities of the SUT
  - . . . or, alternatively . . .

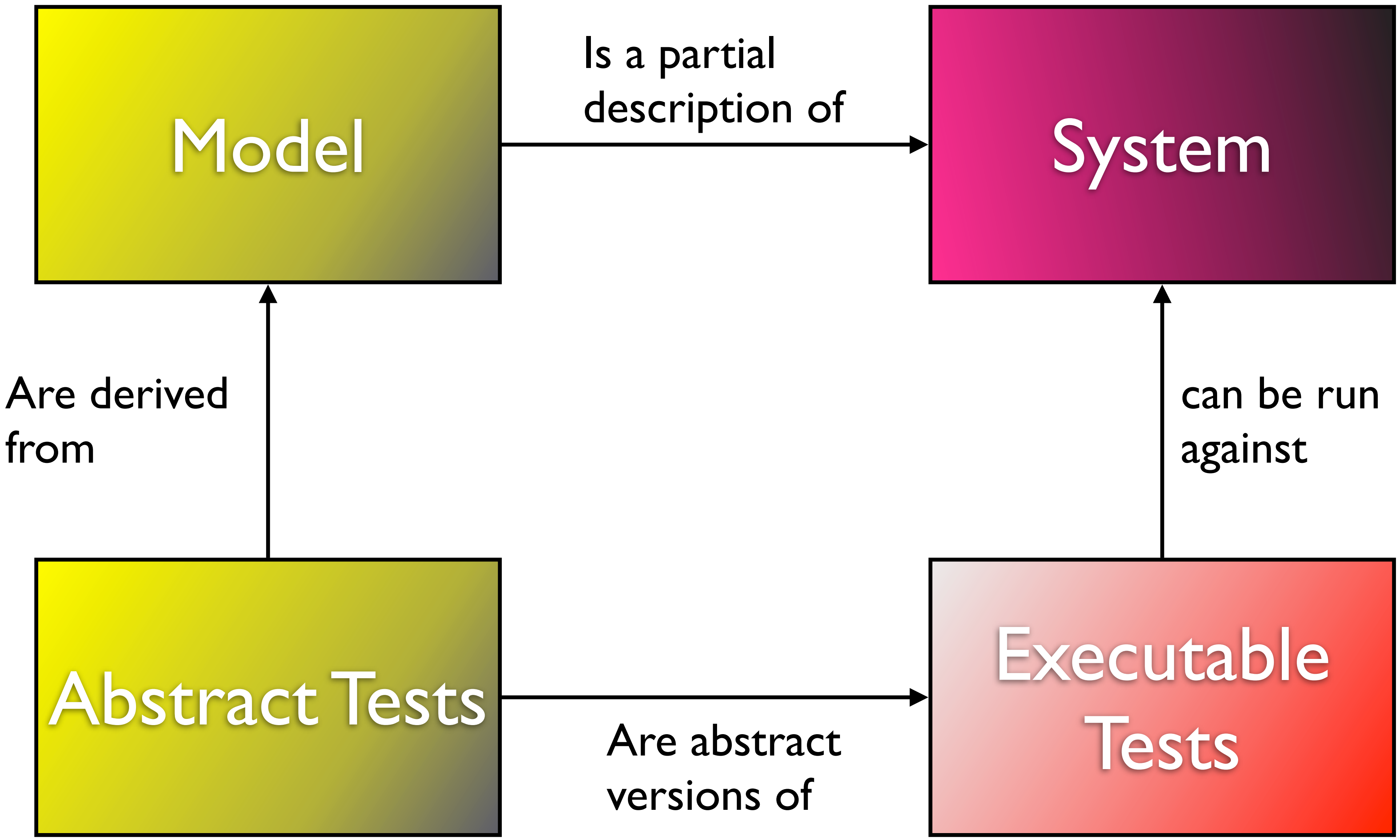
Model-Based Testing

- *"Models ... represent the desired testing strategies and testing environment"*: this is the **pedestrian approach**
  - Test cases and associated test data are modeled in an explicit way – they are identified and calculated by the test engineers
  - MBT only helps by transforming this into executable test procedures

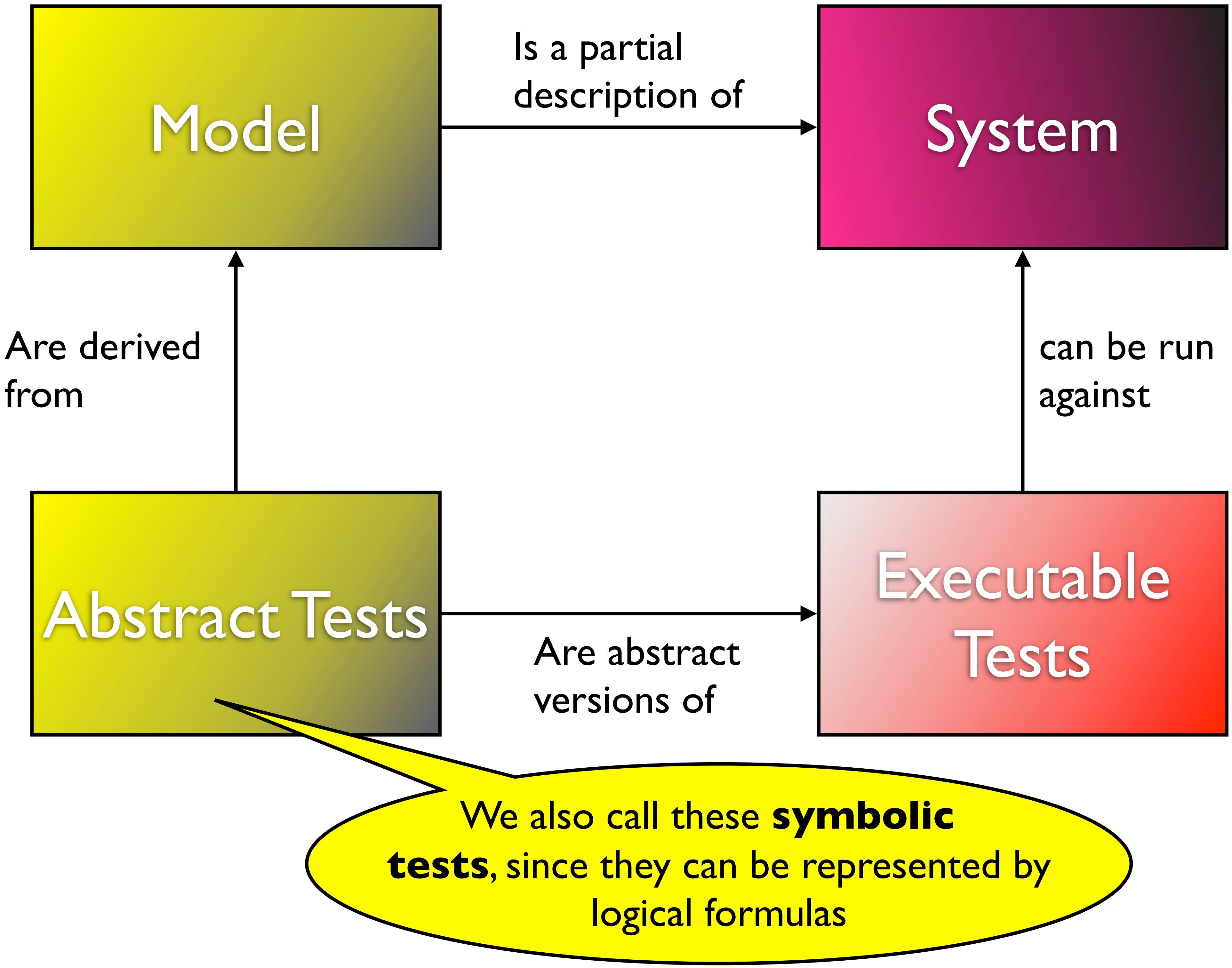
Our MBT Approach

- Instead of writing test procedures,
  - develop a **test model** specifying expected behavior of system under test (SUT) ☞ the gold-plated approach
  - Use **generator** to identify “relevant” test cases from the model and calculate concrete test data
  - Generate **test procedures** fully automatic
  - Perform **tracing** requirements ↔ test cases in a fully automatic way

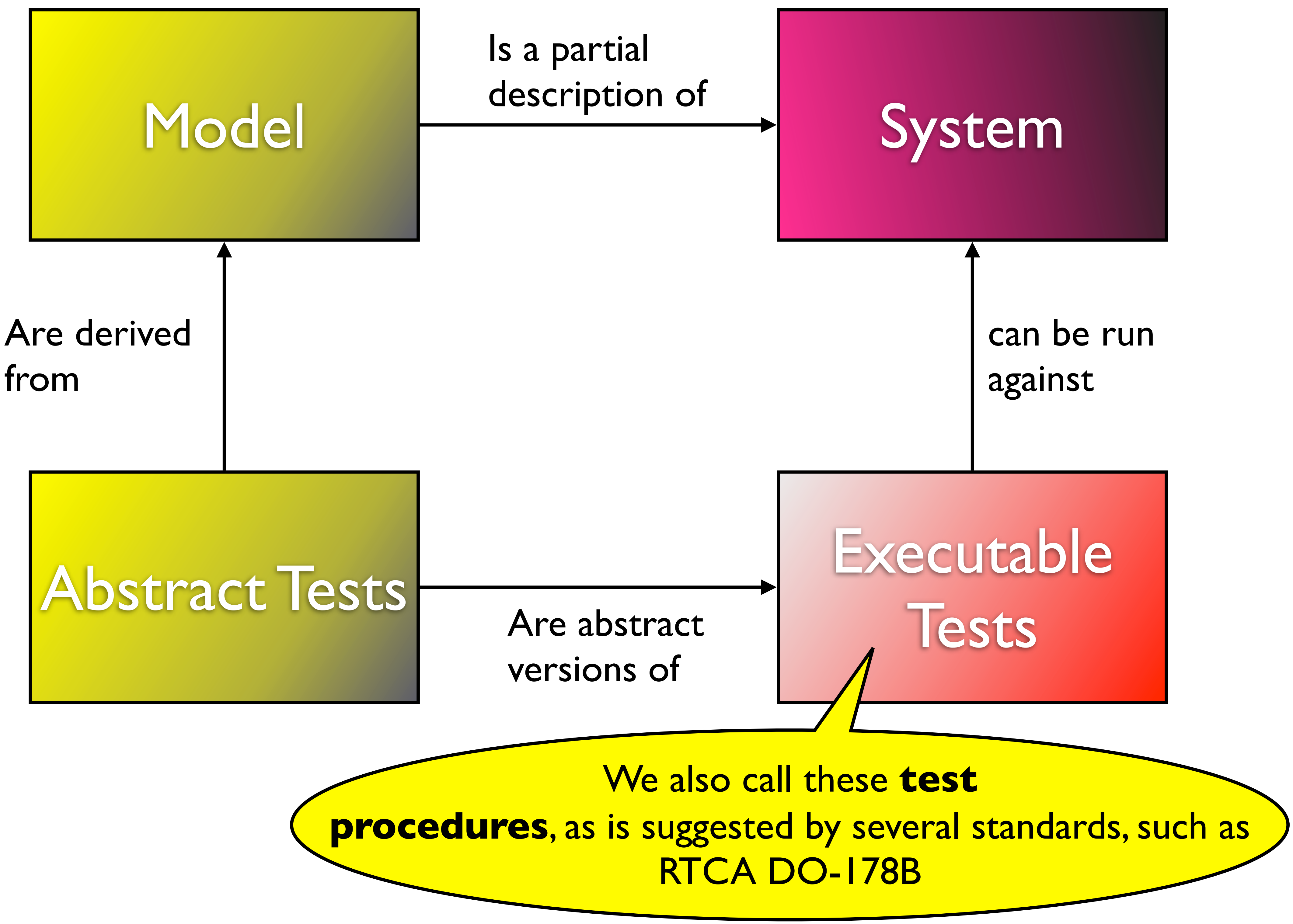
MBT-Paradigm



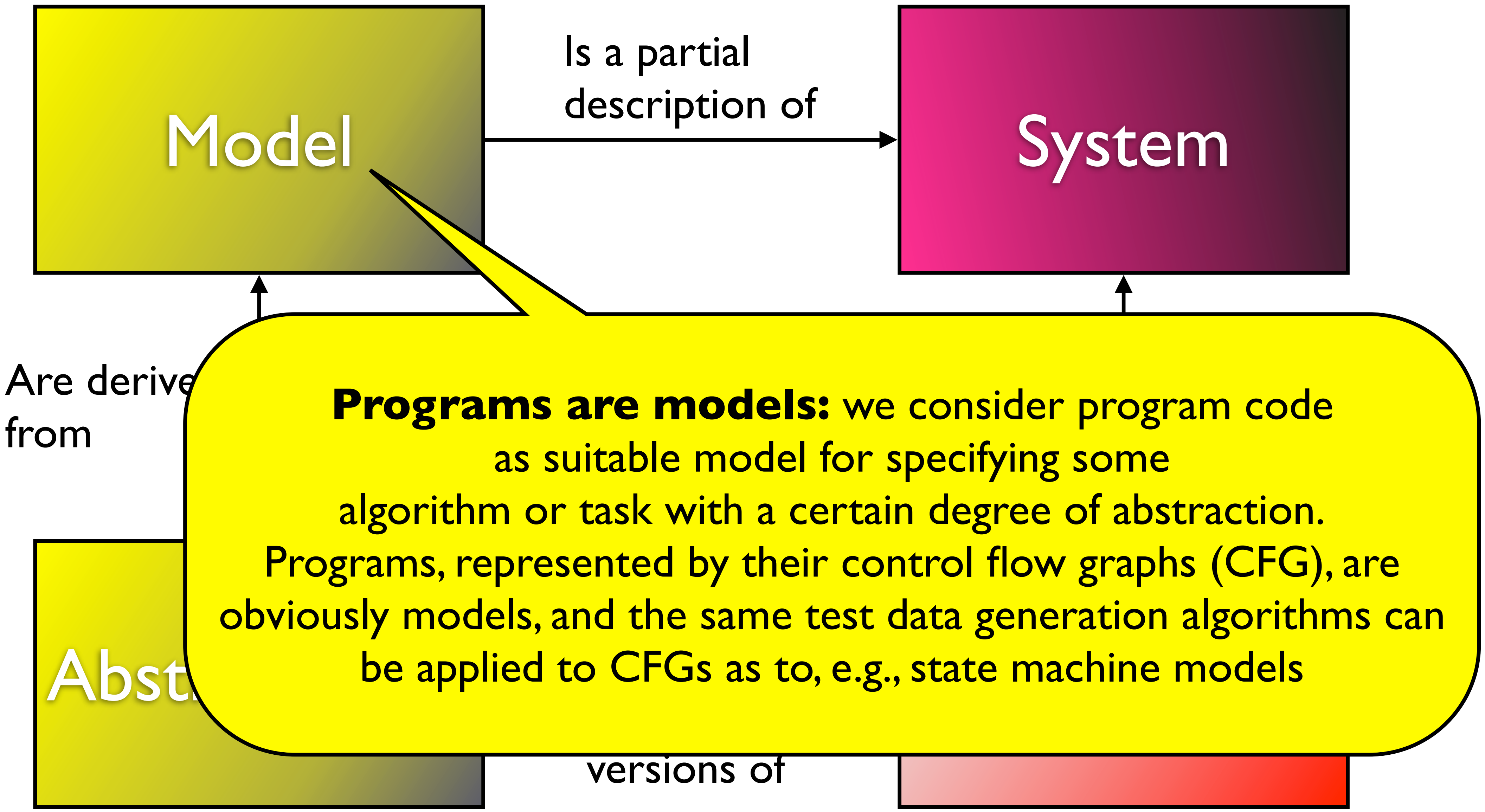
MBT-Paradigm



MBT-Paradigm



MBT-Paradigm



Modeling Formalisms

- Any formalism used to model expected SUT behavior should comply with the **testing hypothesis**
  - The testing hypothesis assumes that the true behavior of the SUT — as far as relevant from the requirements' point of view — can be fully specified by means of this formalism

MBT Modeling Formalisms

- VDM [9]
- Z [3]
- B
- LOTOS [5]
- CSP [4]
- CCS [6]
- Time Automata [11,8]
- TTCN*
- TTCN3* [2]
- SDL
- SCADE [10]
- UML [2,7]
- SysML [7]

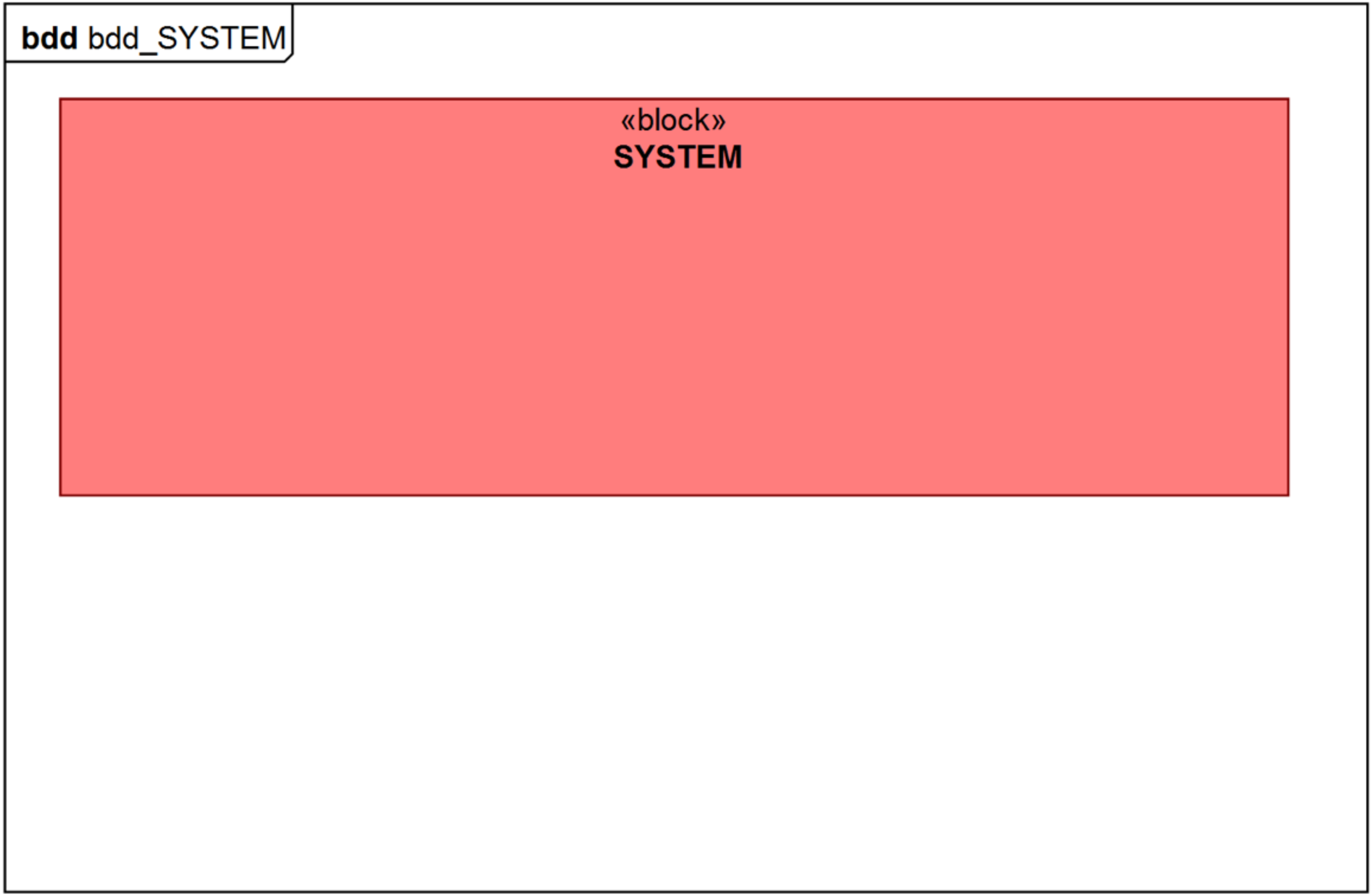
* These formalisms are dedicated test specification languages

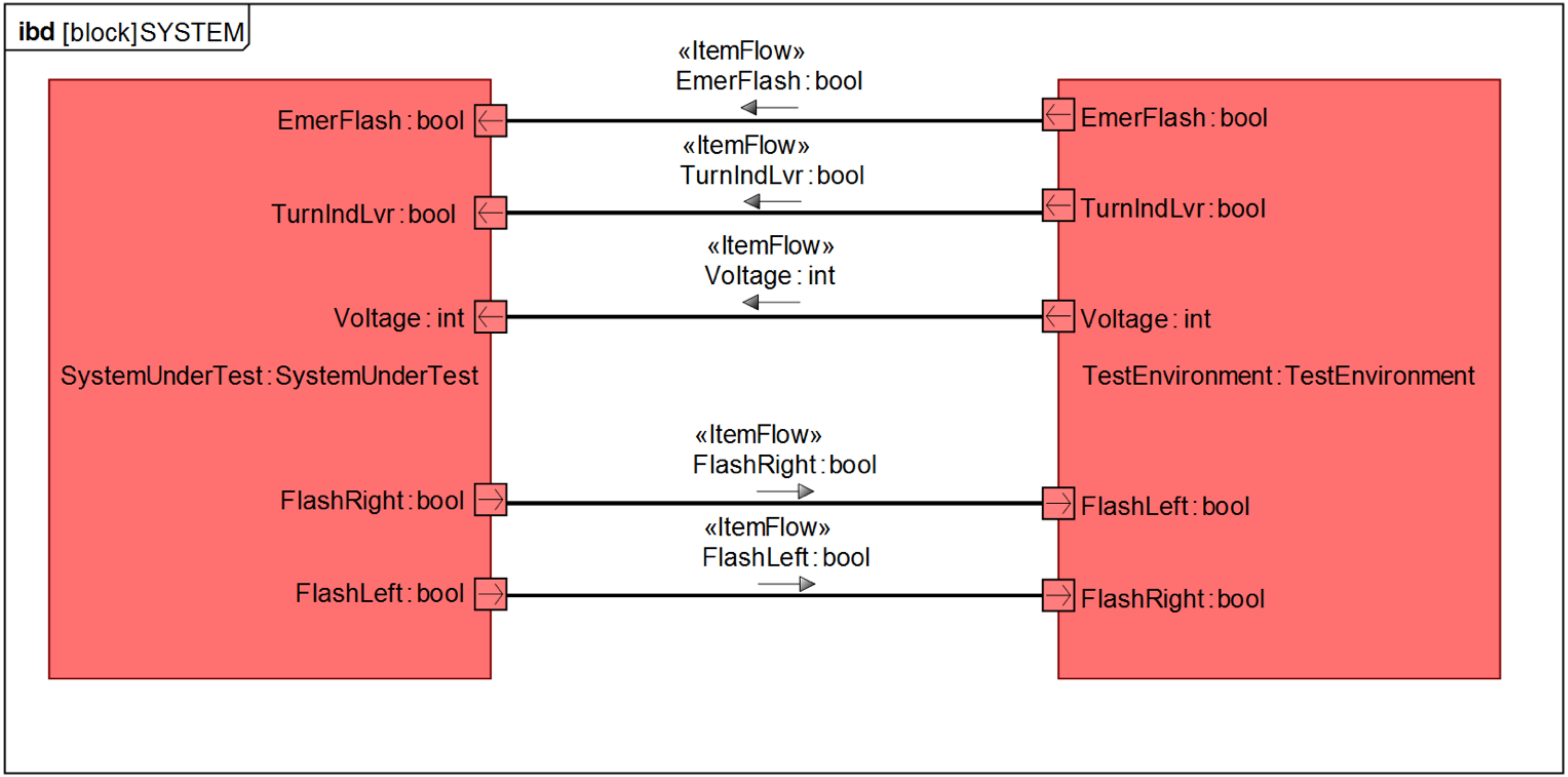
Modeling Formalisms – UML

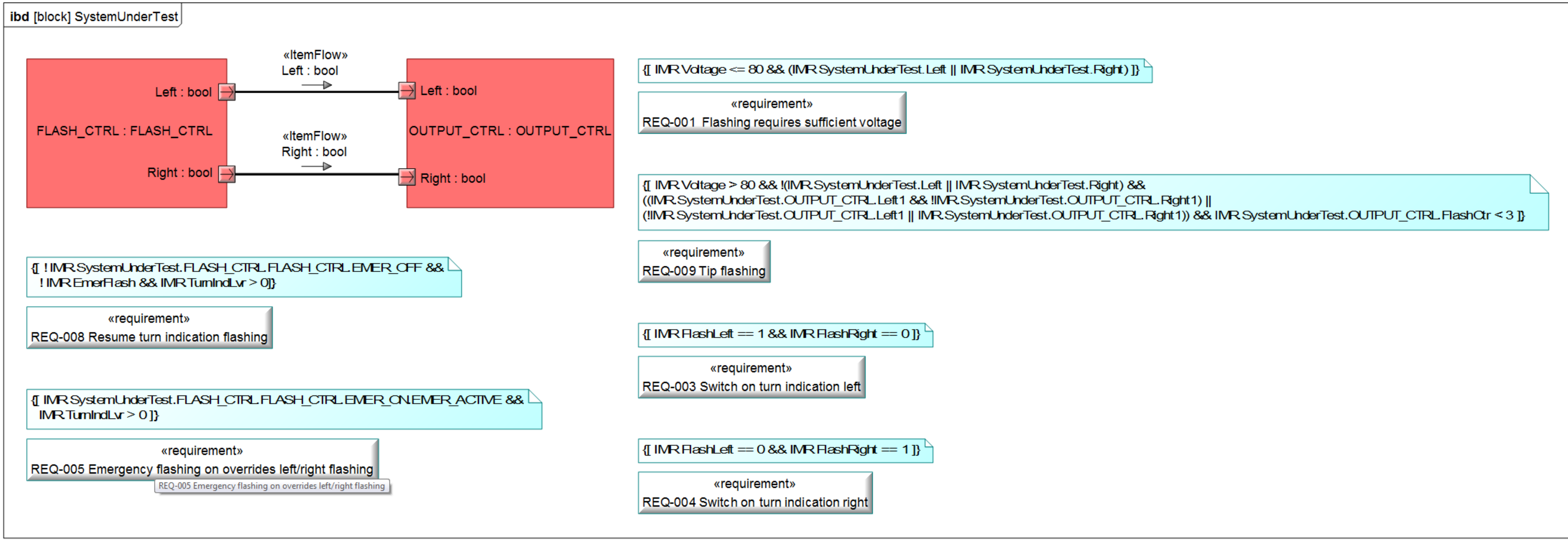
- Unified Modeling Language
  - Wide-spectrum graphical modeling language
  - Combined with OCL – Object Constraint Language for textual specification of algorithms

Modeling Formalisms – SysML

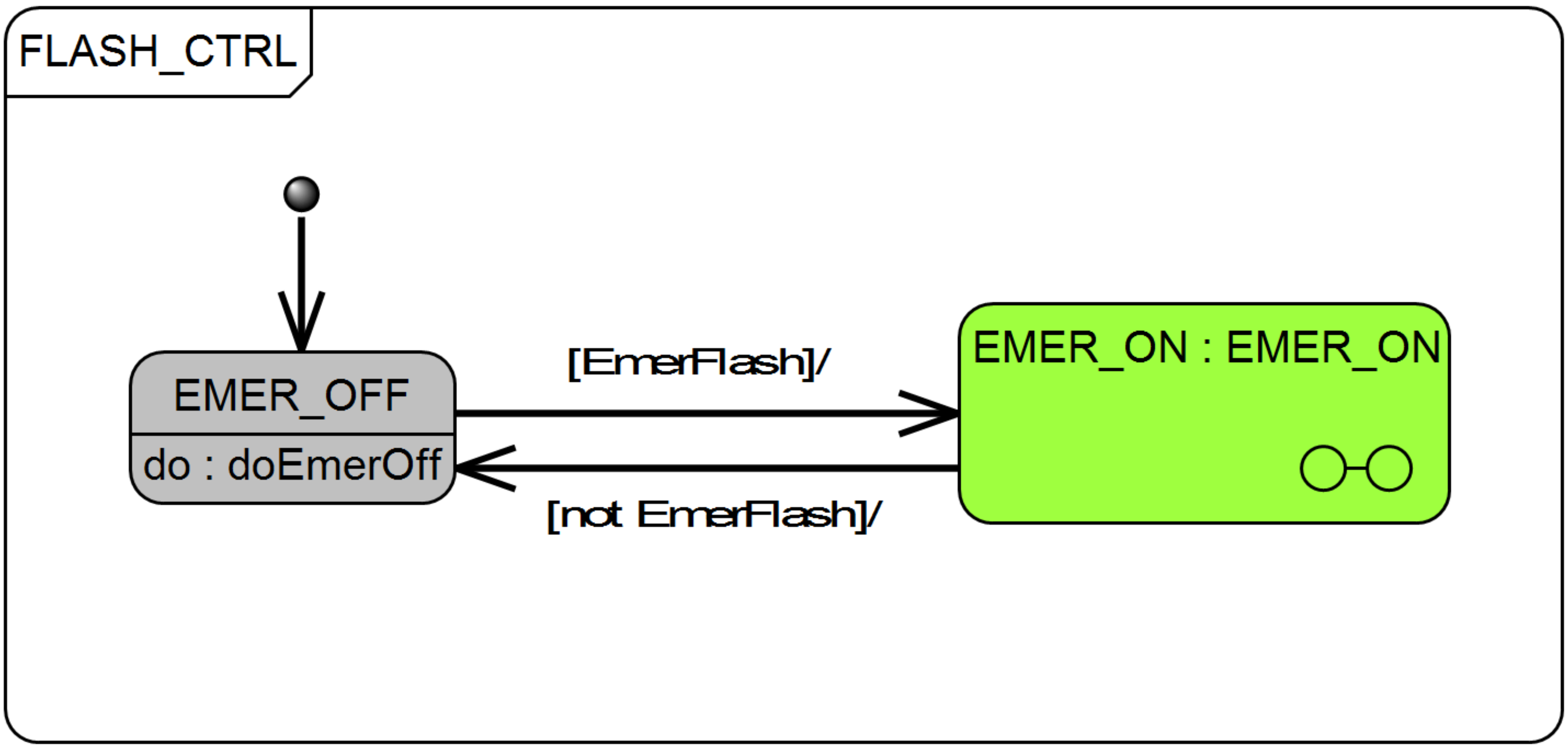
- SysML is a UML profile for modeling systems
  - Extends UML capabilities by
    - requirements engineering support
    - block diagrams
    - parametric constraints

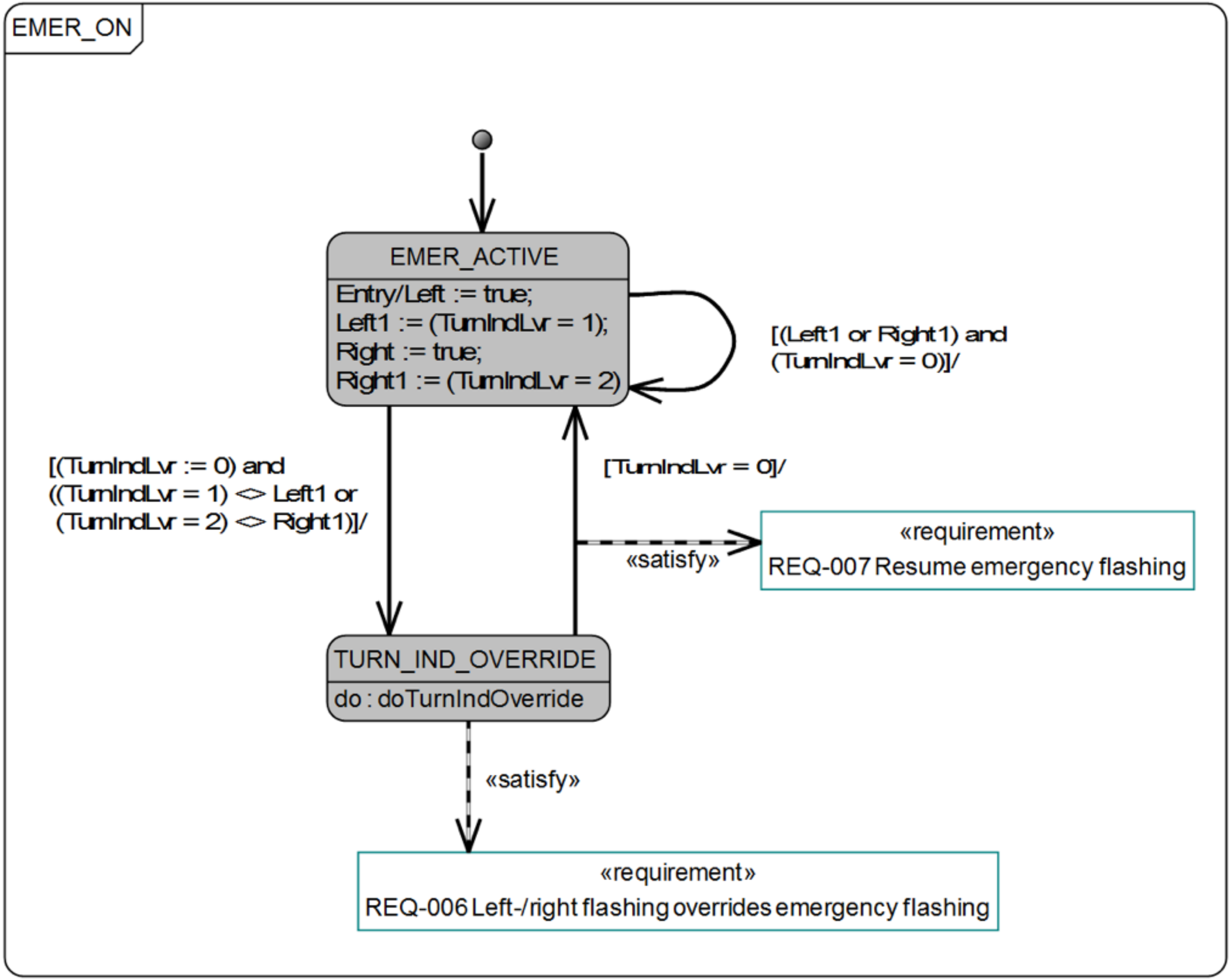

bdd bdd_SYSTEM
«block»
SYSTEM

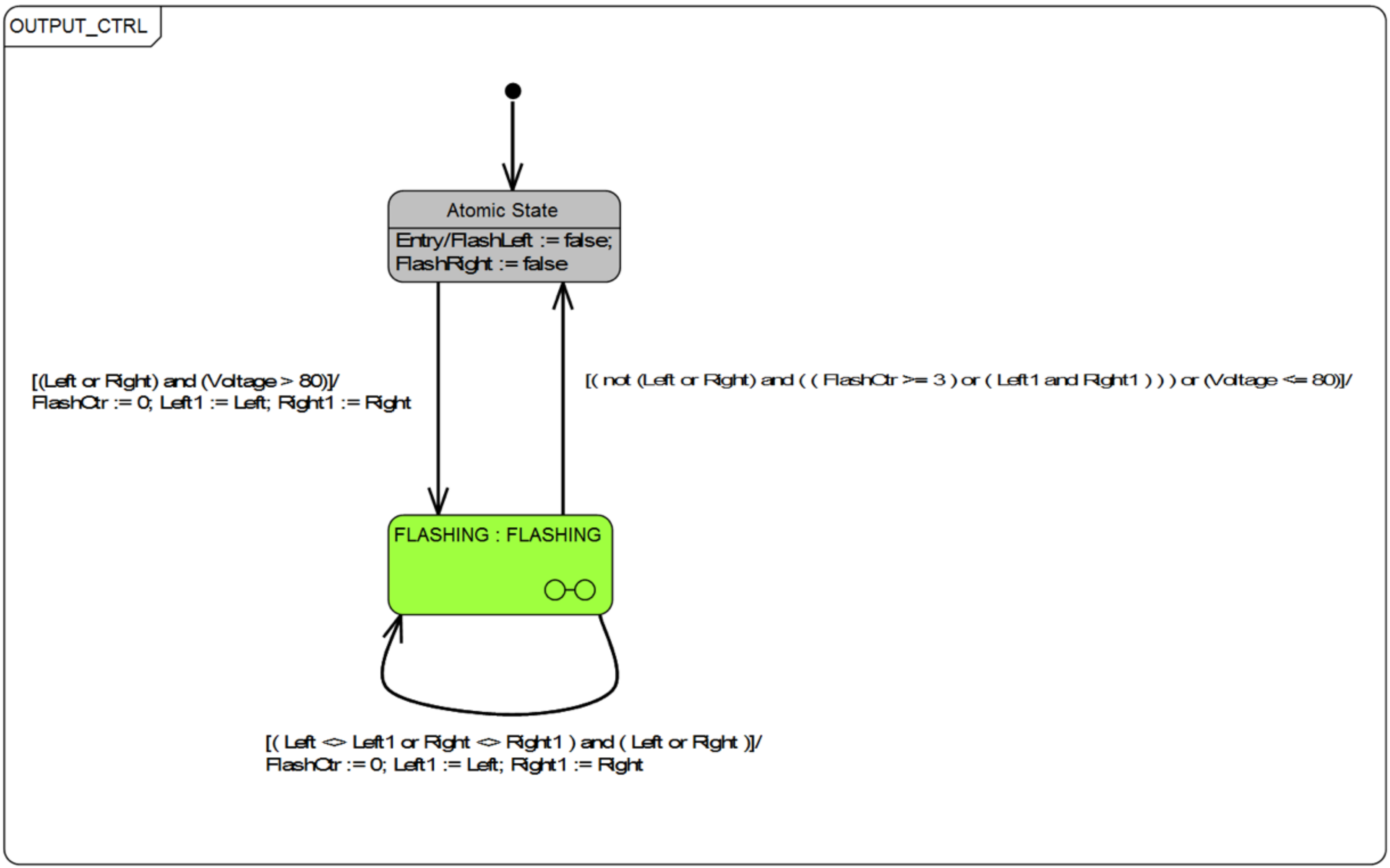

ibd [block]SYSTEM
«ItemFlow»
EmerFlash : bool
EmerFlash : bool
EmerFlash : bool
«ItemFlow»
TurnIndLvr : bool
TurnIndLvr : bool
TurnIndLvr : bool
«ItemFlow»
Voltage : int
Voltage : int
Voltage : int
SystemUnderTest : SystemUnderTest
TestEnvironment : TestEnvironment
«ItemFlow»
FlashRight : bool
FlashRight : bool
FlashLeft : bool
«ItemFlow»
FlashLeft : bool
FlashLeft : bool
FlashRight : bool

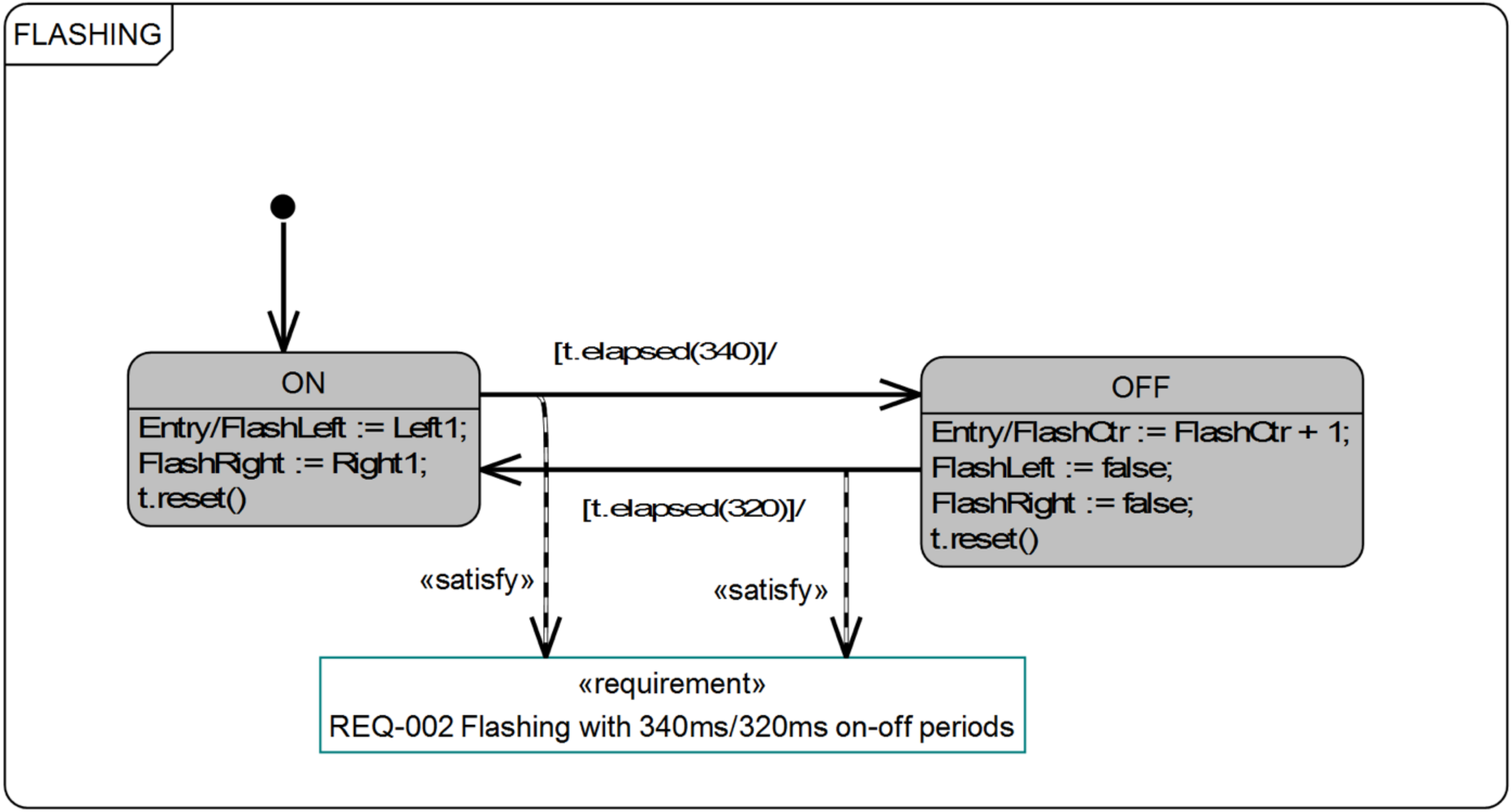


ibd [block] SystemUnderTest
FLASH_CTRL : FLASH_CTRL
Left : bool
Right : bool
«ItemFlow»
Left : bool
«ItemFlow»
Right : bool
OUTPUT_CTRL : OUTPUT_CTRL
Left : bool
Right : bool
{[IMR.Voltage <= 80 && (IMR.SystemUnderTest.Left || IMR.SystemUnderTest.Right)]}
«requirement»
REQ-001 Flashing requires sufficient voltage
{[IMR.Voltage > 80 && !(IMR.SystemUnderTest.Left || IMR.SystemUnderTest.Right) &&
((IMR.SystemUnderTest.OUTPUT_CTRL.Left1 && !IMR.SystemUnderTest.OUTPUT_CTRL.Right1) ||
(!IMR.SystemUnderTest.OUTPUT_CTRL.Left1 || IMR.SystemUnderTest.OUTPUT_CTRL.Right1)) && IMR.SystemUnderTest.OUTPUT_CTRL.FlashCtr < 3]}
«requirement»
REQ-009 Tip flashing
{[!IMR.SystemUnderTest.FLASH_CTRL.FLASH_CTRL.EMER_OFF &&
!IMR.EmerFlash && IMR.TurnIndLvr > 0]}
«requirement»
REQ-008 Resume turn indication flashing
{[IMR.FlashLeft == 1 && IMR.FlashRight == 0]}
«requirement»
REQ-003 Switch on turn indication left
{[IMR.SystemUnderTest.FLASH_CTRL.FLASH_CTRL.EMER_ON.EMER_ACTIVE &&
IMR.TurnIndLvr > 0]}
«requirement»
REQ-005 Emergency flashing on overrides left/right flashing
REQ-005 Emergency flashing on overrides left/right flashing
{[IMR.FlashLeft == 0 && IMR.FlashRight == 1]}
«requirement»
REQ-004 Switch on turn indication right


FLASH_CTRL
EMER_OFF
do : doEmerOff
[EmerFlash]/
[not EmerFlash]/
EMER_ON : EMER_ON


EMER_ON
EMER_ACTIVE
Entry/Left := true;
Left1 := (TurnIndLvr = 1);
Right := true;
Right1 := (TurnIndLvr = 2)
[(Left1 or Right1) and
(TurnIndLvr = 0)]/
[(TurnIndLvr := 0) and
((TurnIndLvr = 1) <> Left1 or
(TurnIndLvr = 2) <> Right1)]/
[TurnIndLvr = 0]/
«satisfy»
«requirement»
REQ-007 Resume emergency flashing
TURN_IND_OVERRIDE
do : doTurnIndOverride
«satisfy»
«requirement»
REQ-006 Left-/right flashing overrides emergency flashing


OUTPUT_CTRL
Atomic State
Entry/FlashLeft := false;
FlashRight := false
[(Left or Right) and (Voltage > 80)]/
FlashCtr := 0; Left1 := Left; Right1 := Right
[(not (Left or Right) and ((FlashCtr >= 3) or (Left1 and Right1))) or (Voltage <= 80)]/
FLASHING : FLASHING
[(Left <> Left1 or Right <> Right1) and (Left or Right)]/
FlashCtr := 0; Left1 := Left; Right1 := Right


FLASHING
ON
Entry/FlashLeft := Left1;
FlashRight := Right1;
t.reset()
[t.elapsed(340)]/
OFF
Entry/FlashCtr := FlashCtr + 1;
FlashLeft := false;
FlashRight := false;
t.reset()
[t.elapsed(320)]/
«satisfy»
«satisfy»
«requirement»
REQ-002 Flashing with 340ms/320ms on-off periods

Test Cases

- **Test cases** are specifications of (subsets of) SUT computations which are suitable to check given requirements, the so-called **test objectives**
- **RTCA DO178B** defines test cases as follows

> *A set of test inputs, execution conditions, and expected results developed for a particular test objective, such as to exercise a particular program path or to verify compliance with a specific requirement*

Test Cases

- Test cases are typically structure into
  - **Test objective.** What should be verified by means of this test case?
  - **Execution conditions.** Which pre-states of the SUT are suitable to test the objective?
  - **Inputs.** Which inputs are suitable for testing the objective from the current SUT pre-state?
  - **Expected results.** What are the expected outputs of the SUT as a reaction to the inputs?

Test Cases

- Test cases are typically structure into
  - **Test objective.** What should be verified by means of this test case
  - **Execution condi**… e SUT are suitable to test the objective?
  - **Inputs.** Which inputs are suitable for testing the objective from the current SUT pre-state?
  - **Expected results.** What are the expected outputs of the SUT as a reaction to the inputs?

For testing reactive systems, the notions of **inputs** and **outputs** require some closer consideration!

Functional requirements and test cases

- **Functional requirements** specify the expected behavior of the SUT (also called *behavioral* requirements)
- Behavior is specified by computations
- **Computations** - in the most general case - are conceptually infinite sequences of
  - (Timestamp,Event,State,Flow)

- General form of **computations**

$$\langle (t_0, e_0, s_0, f_0), (t_1, e_1, s_1, f_1), \ldots \rangle$$

- Time stamps in **dense time** $t_i \in \mathbf{R}$

- Time **monotonicity**: $\forall i : t_i \leq t_{i+1}$

- Time **divergence** applies to infinite computations (so-called *non-Zenoness* condition): $\Sigma_{i \geq 0}(t_{i+1} - t_i) = \infty$

- **Events** abstract significant state changes occurring at specific points in time

$$e_i \in \Sigma \quad \text{(alphabet of SUT)}$$

- **State valuations** are functions from variable symbols x to their domain (= type) D_X

$$s_i : V \to D$$

$$\forall i \in \mathbf{N}_0, x \in V : s_i(x) \in D_x$$

$$D = \bigcup_{x \in V} D_x$$

- **Flows** are time-continuous functions defined between two time stamps

$$f_i : [t_i, t_{i+1}) \times V \to D$$
$$\forall x \in V \, : \, f_i(t_i, x) = s_i(x)$$

- As a consequence the valuation of variables with discrete domain cannot change in time interval

$$[t_i, t_{i+1})$$

Traces

- **Traces** are finite prefixes of computations – these are the objects considered during (dynamic) testing, since every test has to terminate after a finite number of steps
- **Inputs** are finite traces of the form

$$\langle (t_0, e_0^I, s_0|_I, f_0|_I), (t_1, e_1^I, s_1|_I, f_1|_I), \ldots \rangle$$

where events e, valuation functions *s* and flow functions *f* are restricted to input symbols

Traces

- **Outputs** are traces of the form

  $$\langle (t_0, e_0^O, s_0|_O, f_0|_O), (t_1, e_1^O, s_1|_O, f_1|_O), \ldots \rangle$$

  where events, states and flows are restricted to output symbols

- Input and output traces associated with a test case are generally **interleaved**

- **Symbolic test cases** specify subsets of computations which are suitable to test a given requirement
- **Concrete test cases** are specific traces complying with the specification of the associated symbolic test case

Test Procedures

- **Test procedures** are (possibly executable) "recipes" describing how one or more test cases should be executed on the SUT in specified order
- **RTCA DO178B** defines test procedure as follows

  Detailed instructions for the set-up and execution of a given set of test cases, and instructions for the evaluation of results of executing the test cases

Test Suites

- A **test suite** is a collection of test procedures, whose execution is suitable to check a well-defined set of test objectives
- A test suite is **exhaustive** if the SUT **conforms** to the specification model whenever all tests in the suite have been passed

Conformance Relations

- Conformance relations specify the "likeness" between SUT and specification model
- For the context of this seminar, where only non-blocking untimed systems are considered, a very simple conformance relation suffices:

The SUT conforms to the specification model A if and only if all input traces result in the same output traces as for A and the interleaving of inputs and outputs is the same for SUT and A

Manual vs. Automated MBT

- MBT does not necessarily mean that test should be generated in an automated way
  - The UML testing profile [2], for example, explains how to specify test cases and procedures and the testing environment (TE) in UML, but leaves it open whether executable procedures are written by hand or generated automatically
  - Observe that [2] describes the pedestrian approach to MBT

Fundamental System Classification

- **Combinatorial systems.** Output is a function of the input vector — SUT behavior can be specified by mathematical function

$$
\begin{aligned}
f : D_1 \times \ldots \times D_n \to E_1 \times \ldots \times E_m, \\
\vec{x} \mapsto f(\vec{x}) = (f_1(\vec{x}), \ldots, f_m(\vec{x})), \\
\vec{x} = (x_1, \ldots, x_n)
\end{aligned}
$$

Fundamental System Classification

- **Reactive systems.** Output is a function of the *timed trace of input vectors* — SUT behavior can be specified by mathematical function of the sequence input trace, and delivers an output trace

Input trace $\langle (t_0, e_0^I, s_0|_I, f_0|_I), (t_1, e_1^I, s_1|_I, f_1|_I), \ldots \rangle$

- For reactive systems, the output is a function of input and internal state

Impact of System Classification on Testing

- Combinatorial systems can be exhaustively tested by generating all possible input vectors $\vec{x}$ and checking for each vector whether the output complies to the expected result $f(\vec{x})$

Impact of System Classification on Testing

- Reactive systems always have to be stimulated by input traces whose length is generally > 1
- Some objectives for testing reactive systems require to calculate an input trace that "drives" the SUT into an internal state which is suitable to check the test objective

  ☞ This is a constraint solving problem

Impact of System Classification on Testing

- **Note.** If the internal state of a reactive (time-discrete) system can be manipulated by the test system, it may be tested like a combinatorial system, because the next SUT reaction is always a function of the current state and the input vector

$$f : D \times S \to E \times S,$$
$$(\vec{x}, \vec{s}) \mapsto (\vec{y}, \vec{u})$$

Impact of System Classification on Testing

- Examples for reactive systems which can be tested as combinatorial systems
  - Electronic circuits with latches
  - Object-oriented software with getter/setter functions for internal state
  - Database applications

Complexity Considerations

- For a **combinatorial system** with input vector ***x*** and settable state vector ***s*** a test suite is exhaustive if all combinations of ***(x,s)*** are exercised on the SUT

- Let the ***b(x)*** the bit width of the input vector and ***b(s)*** the bit width of the state vector

- The number of possible test inputs is

$$2^{b(\vec{s})+b(\vec{x})}$$

Complexity Considerations

- For a **reactive system** with internal state vector ***s*** (bit width ***b(s)***) and input vector ***x*** (bit width ***b(x)***) the asymptotic complexity (i.e. asymptotic number of test cases required for an exhaustive test suite) is

$$O\left(2^{2 \cdot b(\vec{s})+(1+k) \cdot b(\vec{x})}\right)$$

k is the maximal number of additional states in the implementation, compared to the size of the specification model state space. This will be explained later, when discussing Chow's W-Method.

Further Reading

1. http://en.wikipedia.org/wiki/Model-based_testing, (date: 2012-06-14)

2. Paul Baker et. al.: *Model-driven testing – Using the UML testing profile.* Springer, Berlin, 2008.

3. H.-M. Hörcher and J. Peleska: Using formal specifications to support software testing. Software Quality Journal 4, 309-327. (1995).

4. J. Peleska and M. Siegel: Test Automation of Safety-Critical Reactive Systems.
South African Computer Jounal (1997) 19: 53-77.

5. Ed Brinksma: A theory for the derivation of tests. In P.H.J. van Eijk, C.A.Vissers and M. Diaz (eds.): *The Formal Description Technique LOTOS.* Elsevire Science Publishers B.V. (North-Holland), (1989) 235-247.

6. M.C. Hennessy: *Algebraic Theory of Processes.* IT Press (1988).

7. Jan Peleska, Artur Honisch, Florian Lapschies, Helge Löding, Hermann Schmid, Peer Smuda, Elena Vorobev, and Cornelia Zahlten: A Real-World Benchmark Model for Testing Concurrent Real-Time Systems in the Automotive Domain. In: Burkhart Wolff and Fatiha Zaidi (Eds.): Testing Software and Systems. Proceedings of the 23rd IFIP WG 6.1 International Conference, ICTSS 2011, Paris, France, November 2011, Springer, LNCS 7019, pp. 146-161 (2011).

8. David, A., Larsen, K.G., Lis, S. and Nielsen, B.: Timed testing under partial observability. In: Proc. 2nd International Conference on Software Testing, Verification and Validation (ICST'09), pp.61-70. IEEE Computer Society (2009)

9. Modelling Systems - Practical Tools and Techniques in Software Development, 2nd edition, John Fitzgerald and Peter Gorm Larsen, Cambridge University Press, ISBN 0-521-62348-0, 2009

10. V. Papailiopoulou; Automatic Test Generation for LUSTRE/SCADE Programs. In: ASE '08 Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, 517-520, IEEE Computer Society (2008)

11. J.G. Springintveld and F.W. Vaandrager and P.R. D'Argenio: Testing timed automata. Journal of Theoretical Computer Science 254, 1-2, pp. 225-257, (2001)