MODPROD

Center for Model-Based Cyber-Physical Product Development

Tutorial 3:

# CRML a Language for Verifying Realistic Dynamic Requirements

Lena Buffoni, Adrian Pop, Linköping University, Sweden
Daniel Bouskela, Audrey Jardin, EDF, France

ITEA 3
EUREKA
innovation across borders

EMBrACE

LINKÖPING UNIVERSITY

eDF

# CRML = Common Requirement Modelling Language

# Agenda

1. Motivation
2. Introduction to CRML
    1. What is a CRML requirement? How to verify it by simulation?
    2. Practical exercises: from textual to formalized requirements in CRML
3. Supporting tools
    1. Overall toolchain and available prototypes → Coffee break
    2. Practical exercises with CRML Modelica library
    3. Practical exercises with CRML compiler
4. CRML methodology by example
5. First demos for industrial use
6. Wrap up and perspectives

# Prerequisite for Practical Exercises

- OpenModelica
- Notepad++

- CRML syntax highlighter for Notepad++: `crml.xml`
- CRML Modelica library: `CRML.mo`
- CRML compiler prototype: https://github.com/lenaRB/crml-compiler

# Motivation

# Motivation for CRML

*Dealing with large cyber-physical systems*

Scope: Cyber-physical systems (CPS), especially energy systems.

Characteristic of CPS projects:
They are long lasting projects involving numerous stakeholders.
Delays induce large over costs, in particular because of financial charges (discount rate).

Challenges:

- How to solve over-constrained problems (due to numerous stakeholders)?
- How to specify the expected behavior of CPS without going into realization details?
- How to propagate changes in assumptions on the system design?
- How to evaluate design alternatives efficiently? How to coordinate stakeholders efficiently?
- How to perform FMECA all along the design lifecycle?
- How to justify and document design choices for the future generations?

# CPS Engineering

*What is the current situation? What should be improved?*

- Today
  - system evaluation is performed mostly with static models
  - most verifications are performed manually and hence not as often as necessary

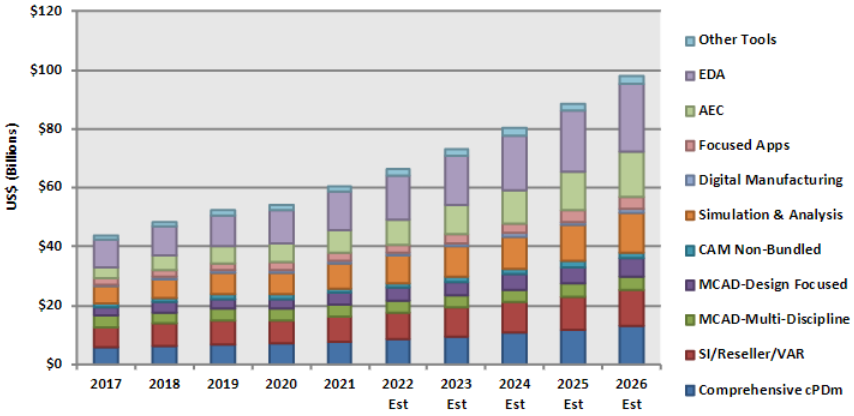
➔ oversizing, late error detections, and eventually delays and cost overruns

- There is a **need for more rigorous engineering method** to
  - **Be more competitive**
    by managing the complexity of CPS (multi-dimensional, numerous stakeholders, uncertain) and by evaluating multiple scenarios to better cover the associated risks
  - **Open the solution space to innovative products or services**
    by providing means to assess new margins and solutions wrt. requirements for non-standard situations

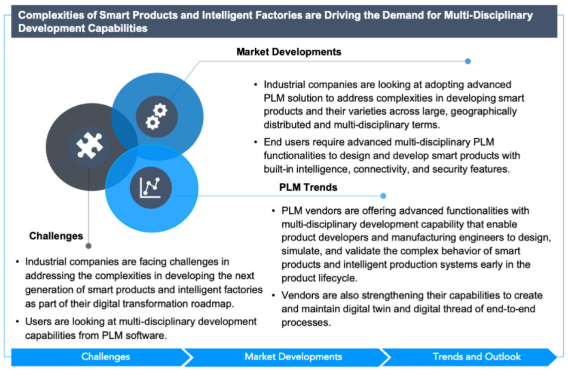
# CPS Engineering

*A trend confirmed by the tool market analysis*

- Market: model-based system engineering
- Part of the CAD/PLM market Simulation & Analysis + cPDm (Collaborative Product Definition Management): approx. $20 billion
- Main SysML based tools
  - Capella (Thalès) (SysML like)
  - Magic Draw (Dassault Systèmes)
  - Modelio (Softeam)
  - Papyrus (CEA, open source)
  - Rhapsody (IBM)
- Main drivers
  - cPDm of increased complexity across an extended enterprise is vital to success in a global market (*)
  - Complexities of Smart Products and Intelligent Factories are **Driving the Demand for Multi-Disciplinary Development Capabilities (**)**
  - Integration of PLM Solution with IoT Platforms (**)
  - **Evolution of PLM Solution towards Product Innovation Platform (**)**




CAD/PLM market (source: CIMdata 2022)




One of PLM market driver (source: Quadrant Knowledge Solutions)

(*) CIMdata (2000)
(**) Quadrant Market Outlook: Product Lifecycle Management (PLM), 2018 –2023, Worldwide

# CPS Engineering

*Solution proposed within CRML*

Use of **realistic dynamic behavioral models** to better handle multi-physics and systems' interactions → **e.g. Modelica**

Use of **formal dynamic requirement models** to evaluate multiple realistic scenarios and automate verifications → **CRML**

**Tutorial's focus**

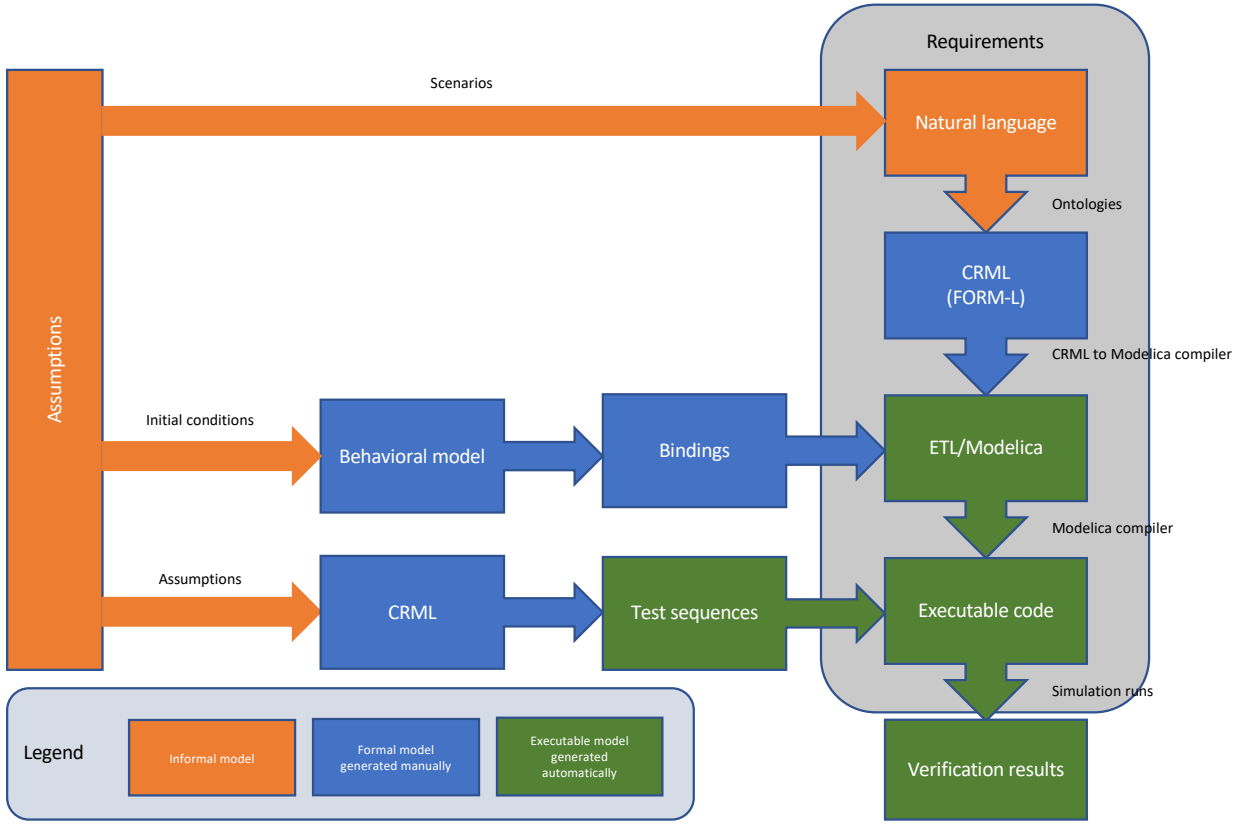
**Rationale**

- System dynamics is considered in models as time may be source of cost reductions and cost overruns
- Formal verifications are performed since safety/availability are important features of CPS and the demonstration that the system operates correctly should follow a rigorous method and is as important as the system itself
- Probabilistic aspects are considered in models in order to be more realistic (e.g. safety/availability targets are defined as a probability threshold since they cannot be achieved at any cost)

# Overall solution offered by CRML

*Towards a more rigorous engineering of complex systems*

- **Requirement models**
  to capture all constraints on the system and define envelopes of acceptable behaviors
- **Behavioral models**
  to capture the behavior of design solutions
- **Verification models**
  to automate tests by using requirement models as observers to check whether design solutions meet requirements or not.



# Overall solution offered by CRML

*A possible answer to two market drivers*

- Market driver 1: Demand for Multi-Disciplinary Development Capabilities
  - CRML can be seen as a Collaborative Product Definition Management
  - Better coordination of stakeholders/engineering teams by sharing a common view on the system over its complete lifecycle (from preliminary design to operation)

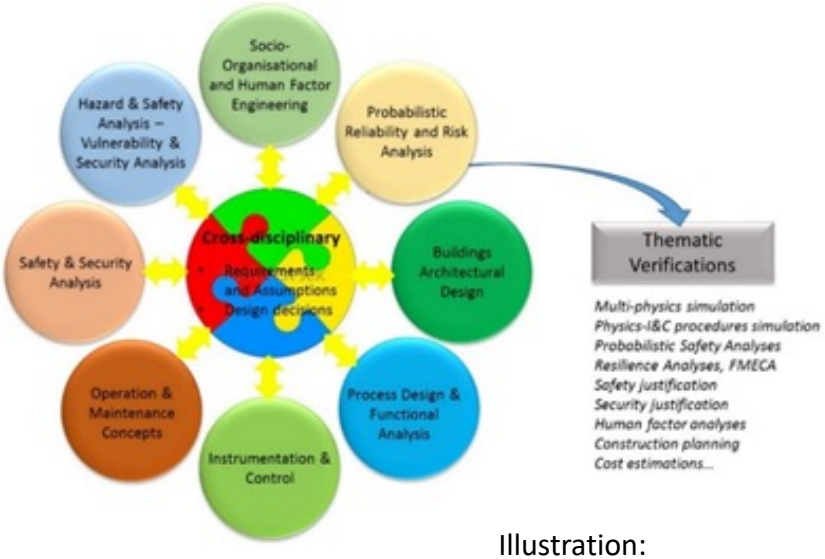
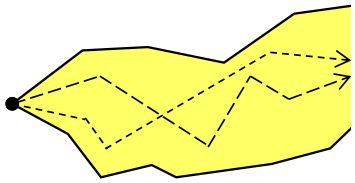



Illustration:
T. Nguyen

- Market driver 2: Evolution of PLM Solution towards Product Innovation Platform
  - CRML opens the solution space by defining the system through its envelopes of acceptable behaviors
  - Avoid to close solution options by going into realization details too early during the design phase

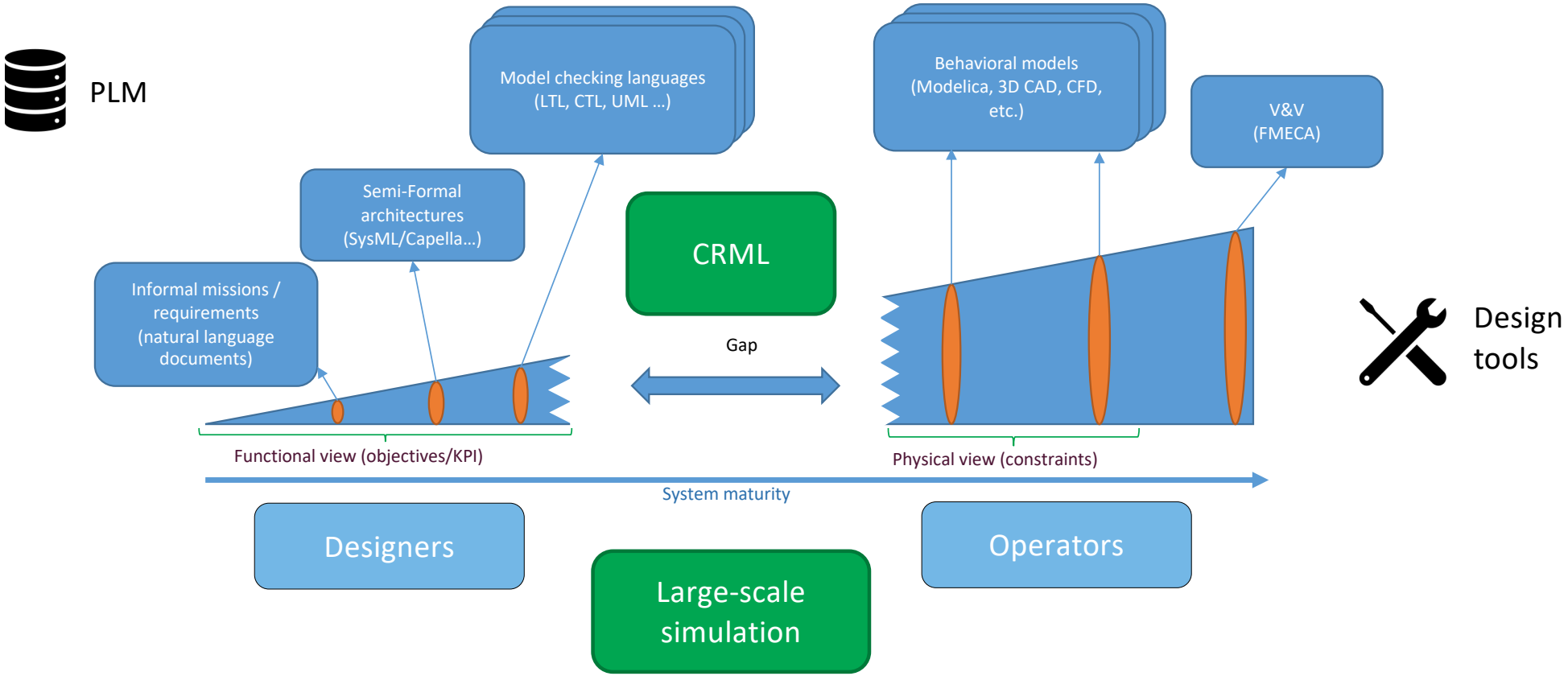


LINKÖPING UNIVERSITY

EDF

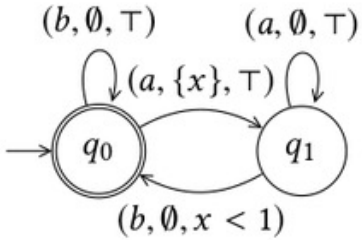
# CRML positioning wrt. state-of-the-art

*A bridge between the functional and the physical views*



# Why a new language?

*Formal requirement tools do not handle physical time*




Existing languages require to model systems as finite-state machines.

They verify the system model,
and not the actual system itself.

They verify only stated requirements:
the validity of the properties cannot be checked.

They only handle real-time aspects for states that are known in advance.

Hence, they do not consider situations where:

- Existing states undergo gradual drift, e.g., due to wear.
- New states appear due to unexpected events.

New language is needed to express realistic physical constraints for any situation.

Deterministic: constraints are scheduled by events occurring in physical time.

Stochastic: because safety and availability cannot be achieved at any cost, failure rate tolerances must be specified.

Verifications must be made against physical behavioral models.

# SHORT History of CRML

*A long-lasting history that starts with the EUROSYSLIB project*


- **EUROSYSLIB project:** start reflections on how to specify systems without describing their detailed behavior
  ➔ need for a formal specification language ➔ investigation of the state-of-the-art.

- **OPENPROD project:** proposition of a link between SysML an Modelica
  ➔ ModelicaML prototype developed by Airbus and tested by EDF.

- **MODRIO project:** proposition by EDF of a new language called FORM-L (Formal Requirement Modelling Language):
  - Specification written by EDF (Thuy Nguyen).
  - Blocks as functions in Modelica.
  - Development of 2 Modelica libraries for the formal capture of requirements: Modelica_Requirement (DLR) and ReqSysPro (EDF).
  - Development of a FORM-L compiler (Inria and Sciworks Technology) on an EDF contract.

- **EMBRACE project:** proposition of CRML as the formal specification of FORM-L.
  - Specification written by EDF.
  - CRML compiler developed by University of Linköping.

# Introduction to CRML

# CRML scope

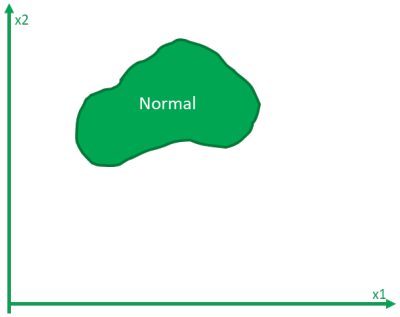
- Definition of a requirement

  *« A statement that identifies a system, product or process characteristic or constraint, which is unambiguous, clear, unique, consistent, stand-alone (not grouped), and verifiable, and is deemed necessary for stakeholder acceptability » (INCOSE 2010)*

- 4 verification methods usually acknowledged: inspection, analysis, demonstration or test
- CRML focuses on **technical requirements**
  - Functional
  - Non-functional (or Quality of service)
  - Constraint
- Which **require simulation over time to be tested** /verified

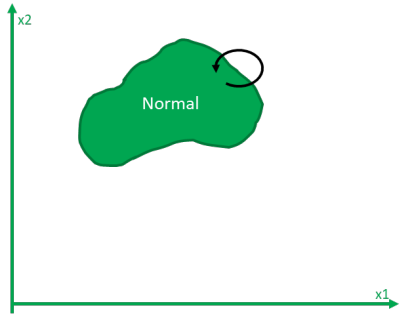

Different Types of Technical Requirements
Source: NASA System Engineering Notebook, 2007

# What is a Typical Technical Requirement on a CPS?



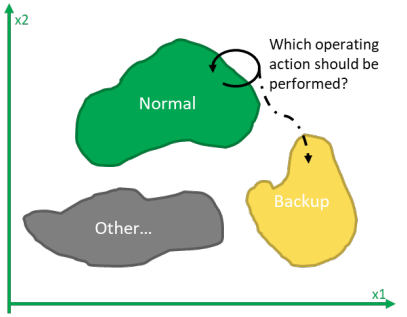

- CRML is aimed at expressing formal realistic requirements on the system behavior.
- Realistic means that requirements must be achievable
  by considering reasonable KPIs such as time delays and safety and dependability constraints.
- The reasonable attribute is quantified by setting confidence levels through probabilistic criteria.
- An example of typical requirement is to ensure that
  1. The system should stay within its normal operating domain.

# What is a Typical (Technical) Requirement on a CPS?



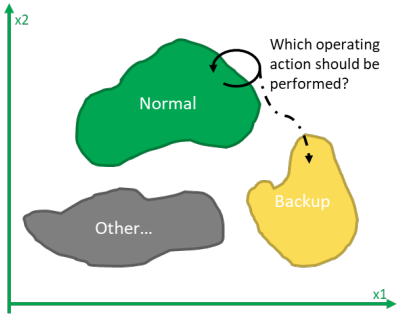

- CRML is aimed at expressing formal realistic requirements on the system behavior.
- Realistic means that requirements must be achievable
  by considering reasonable KPIs such as time delays and safety and dependability constraints.
- The reasonable attribute is quantified by setting confidence levels through probabilistic criteria.
- An example of typical requirement is to ensure that
  1. The system should stay within its normal operating domain.
  2. If partial requirement 1 above fails, then the system should go back to its normal operating domain within a given time delay.

# What is a Typical (Technical) Requirement on a CPS?



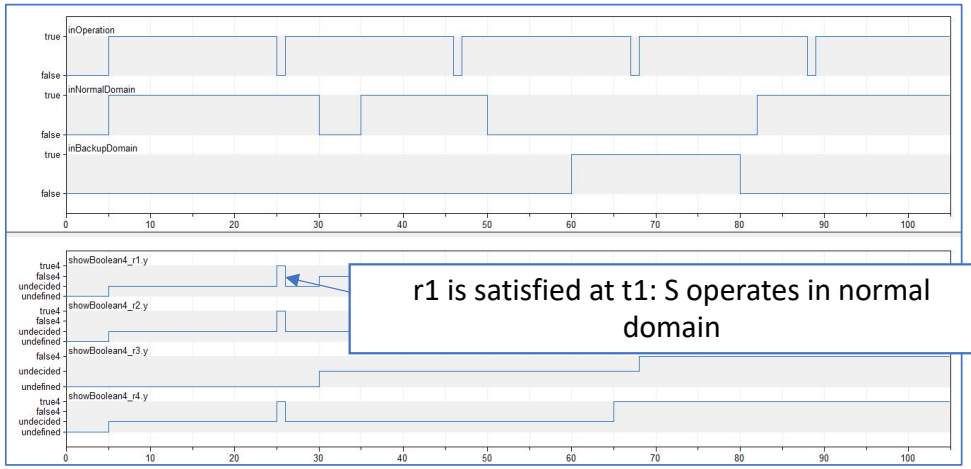

- CRML is aimed at expressing formal realistic requirements on the system behavior.
- Realistic means that requirements must be achievable
  by considering reasonable KPIs such as time delays and safety and dependability constraints.
- The reasonable attribute is quantified by setting confidence levels through probabilistic criteria.
- An example of typical requirement is to ensure that
  1. The system should stay within its normal operating domain.
  2. If partial requirement 1 above fails, then the system should go back to its normal operating domain within a given time delay.
  3. If partial requirement 2 above fails, or if partial requirement 1 fails with a too high failure rate, then the system should go to a safe backup state within a given time delay.

# What is a Typical (Technical) Requirement on a CPS?

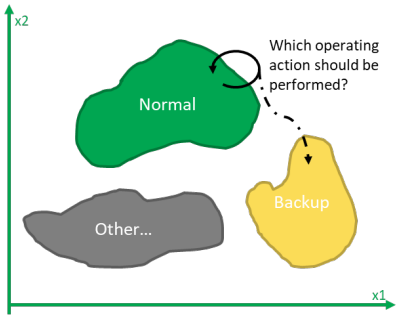



- CRML is aimed at expressing formal realistic requirements on the system behavior.
- Realistic means that requirements must be achievable by considering reasonable KPIs such as time delays and safety and dependability constraints.
- The reasonable attribute is quantified by setting confidence levels through probabilistic criteria.
- An example of typical requirement is to ensure that
  1. The system should stay within its normal operating domain.
  2. If partial requirement 1 above fails, then the system should go back to its normal operating domain within a given time delay.
  3. If partial requirement 2 above fails, or if partial requirement 1 fails with a too high failure rate, then the system should go to a safe backup state within a given time delay.
  4. The complete requirement made of the conjunction of partial requirements 1, 2 and 3 should be satisfied with a given probability (e.g., > 99.99%).
- In addition, requirements can be issued by numerous stakeholders (teams) of different domains and disciplines, that negotiate through contracts to achieve a consistent global set of requirements.

# How CRML Can Deal with the Temporality of Realistic CPS Requirements?




r1 is satisfied at t1: S operates in normal domain




```
// r1 is "During operation, the system should stay within its normal domain."
Requirement r1 is during inOperation ensure inNormalDomain;

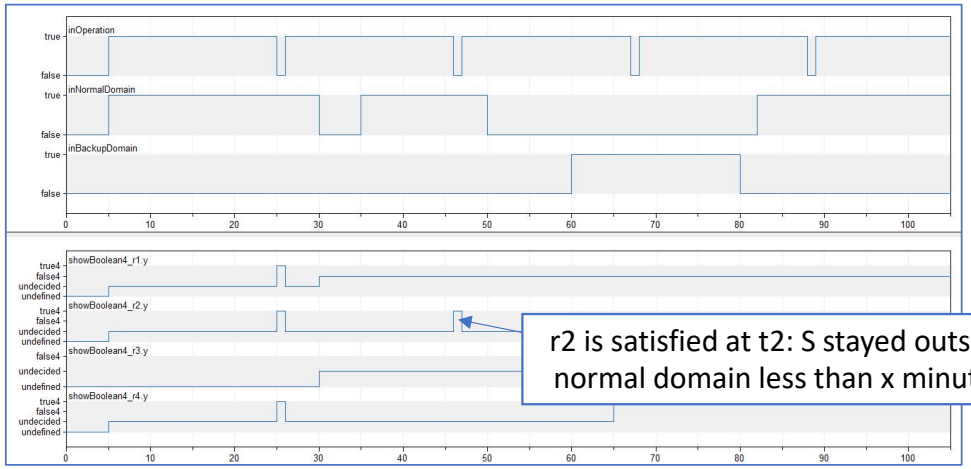
// r2 is "If the system fails to stay within its operating domain, then it should not stay outside of its normal domain for more than x minutes."
Requirement r2 is during inOperation ensure (not inNormalDomain implies r2_outside);
Requirement r2_outside is during [inNormalDomain becomes false, inNormalDomain becomes false + x mn] check at end b;
Boolean b is inOperation implies inNormalDomain;

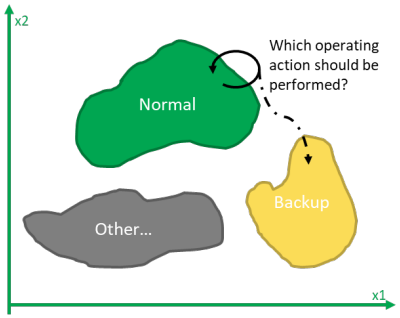
// r3 is "The system should not go outside its normal domain more than n times per year."
Requirement r3 is count ((b becomes false) on [b becomes false, b becomes false + 1 year] <= n;

// r4 is "If (r1 and r2 and r3) fail, then the system should go to its backup domain within y minutes as soon as the failure is detected."
Requirement r4 is not (r1 and r2 and r3) implies
                  during [(r1 and r2 and r3) becomes false, (r1 and r2 and r3) becomes false + y mn] check at end inBackupDomain;

// R is "During system operating life, r1 and r2 and r3 and r4 should be satisfied with a probability of success of p%."
Real prob is estimator Probability (r1 and r2 and r3 and r4) at inSystemOperatingLife becomes false;
Requirement R is during inSystemOperatingLife check at end prob > p;
```

# How CRML Can Deal with the Temporality of Realistic CPS Requirements?




```
// r1 is "During operation, the system should stay within its normal domain."
Requirement r1 is during inOperation ensure inNormalDomain;

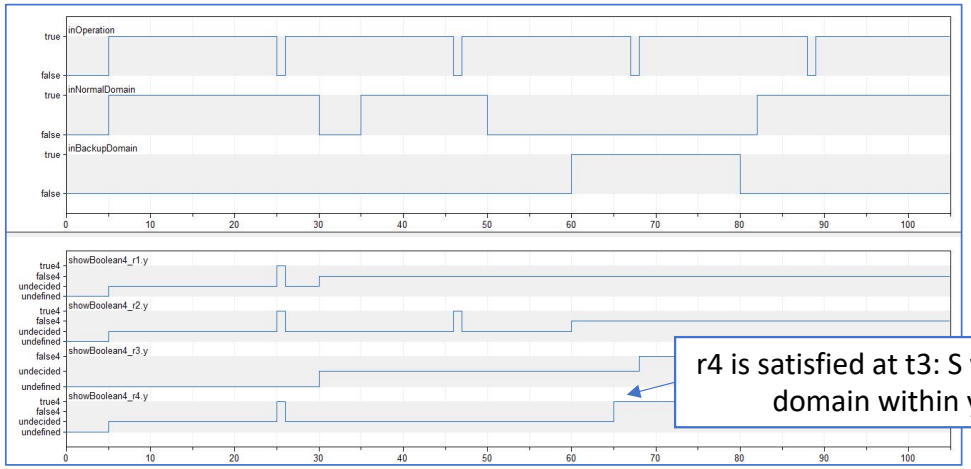
// r2 is "If the system fails to stay within its operating domain, then it should not stay outside of its normal domain for more than x minutes."
Requirement r2 is during inOperation ensure (not inNormalDomain implies r2_outside);
Requirement r2_outside is during [inNormalDomain becomes false, inNormalDomain becomes false + x mn] check at end b;
Boolean b is inOperation implies inNormalDomain;

// r3 is "The system should not go outside its normal domain more than n times per year."
Requirement r3 is count ((b becomes false) on [b becomes false, b becomes false + 1 year] <= n;

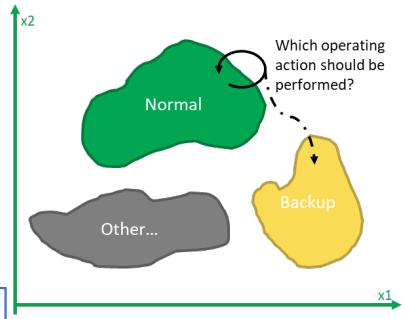
// r4 is "If (r1 and r2 and r3) fail, then the system should go to its backup domain within y minutes as soon as the failure is detected."
Requirement r4 is not (r1 and r2 and r3) implies
                  during [(r1 and r2 and r3) becomes false, (r1 and r2 and r3) becomes false + y mn] check at end inBackupDomain;

// R is "During system operating life, r1 and r2 and r3 and r4 should be satisfied with a probability of success of p%."
Real prob is estimator Probability (r1 and r2 and r3 and r4) at inSystemOperatingLife becomes false;
Requirement R is during inSystemOperatingLife check at end prob > p;
```

# How CRML Can Deal with the Temporality of Realistic CPS Requirements?




r4 is satisfied at t3: S went to backup domain within y minutes




```
// r1 is "During operation, the system should stay within its normal domain."
Requirement r1 is during inOperation ensure inNormalDomain;

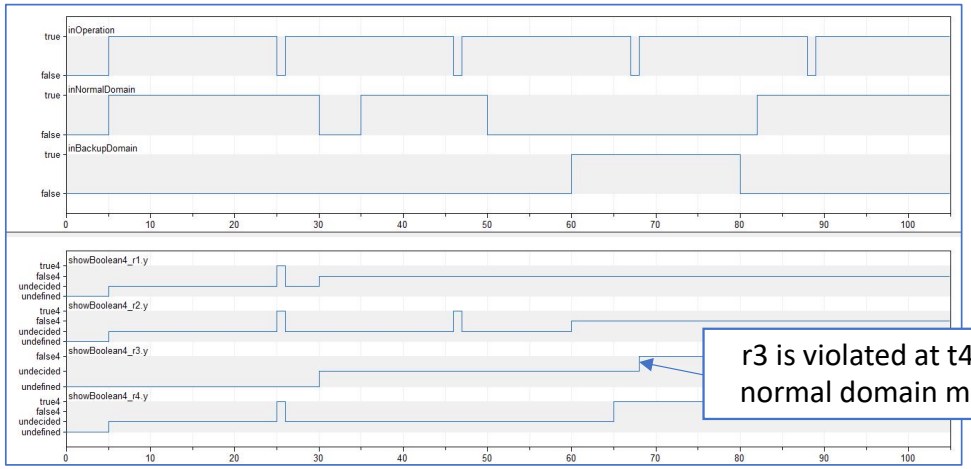
// r2 is "If the system fails to stay within its operating domain, then it should not stay outside of its normal domain for more than x minutes."
Requirement r2 is during inOperation ensure (not inNormalDomain implies r2_outside);
Requirement r2_outside is during [inNormalDomain becomes false, inNormalDomain becomes false + x mn] check at end b;
Boolean b is inOperation implies inNormalDomain;

// r3 is "The system should not go outside its normal domain more than n times per year."
Requirement r3 is count ((b becomes false) on [b becomes false, b becomes false + 1 year] <= n;

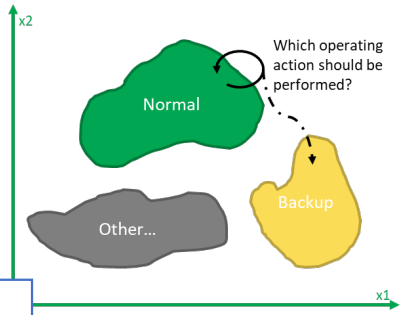
// r4 is "If (r1 and r2 and r3) fail, then the system should go to its backup domain within y minutes as soon as the failure is detected."
Requirement r4 is not (r1 and r2 and r3) implies
                  during [(r1 and r2 and r3) becomes false, (r1 and r2 and r3) becomes false + y mn] check at end inBackupDomain;

// R is "During system operating life, r1 and r2 and r3 and r4 should be satisfied with a probability of success of p%."
Real prob is estimator Probability (r1 and r2 and r3 and r4) at inSystemOperatingLife becomes false;
Requirement R is during inSystemOperatingLife check at end prob > p;
```

# How CRML Can Deal with the Temporality of Realistic CPS Requirements?




r3 is violated at t4: S went outside normal domain more than 2 times




```
// r1 is "During operation, the system should stay within its normal domain."
Requirement r1 is during inOperation ensure inNormalDomain;

// r2 is "If the system fails to stay within its operating domain, then it should not stay outside of its normal domain for more than x minutes."
Requirement r2 is during inOperation ensure (not inNormalDomain implies r2_outside);
Requirement r2_outside is during [inNormalDomain becomes false, inNormalDomain becomes false + x mn] check at end b;
Boolean b is inOperation implies inNormalDomain;

// r3 is "The system should not go outside its normal domain more than n times per year."
Requirement r3 is count ((b becomes false) on [b becomes false, b becomes false + 1 year] <= n;

// r4 is "If (r1 and r2 and r3) fail, then the system should go to its backup domain within y minutes as soon as the failure is detected."
Requirement r4 is not (r1 and r2 and r3) implies
                  during [(r1 and r2 and r3) becomes false, (r1 and r2 and r3) becomes false + y mn] check at end inBackupDomain;

// R is "During system operating life, r1 and r2 and r3 and r4 should be satisfied with a probability of success of p%."
Real prob is estimator Probability (r1 and r2 and r3 and r4) at inSystemOperatingLife becomes false;
Requirement R is during inSystemOperatingLife check at end prob > p;
```

# How to express realistic requirements in CRML?

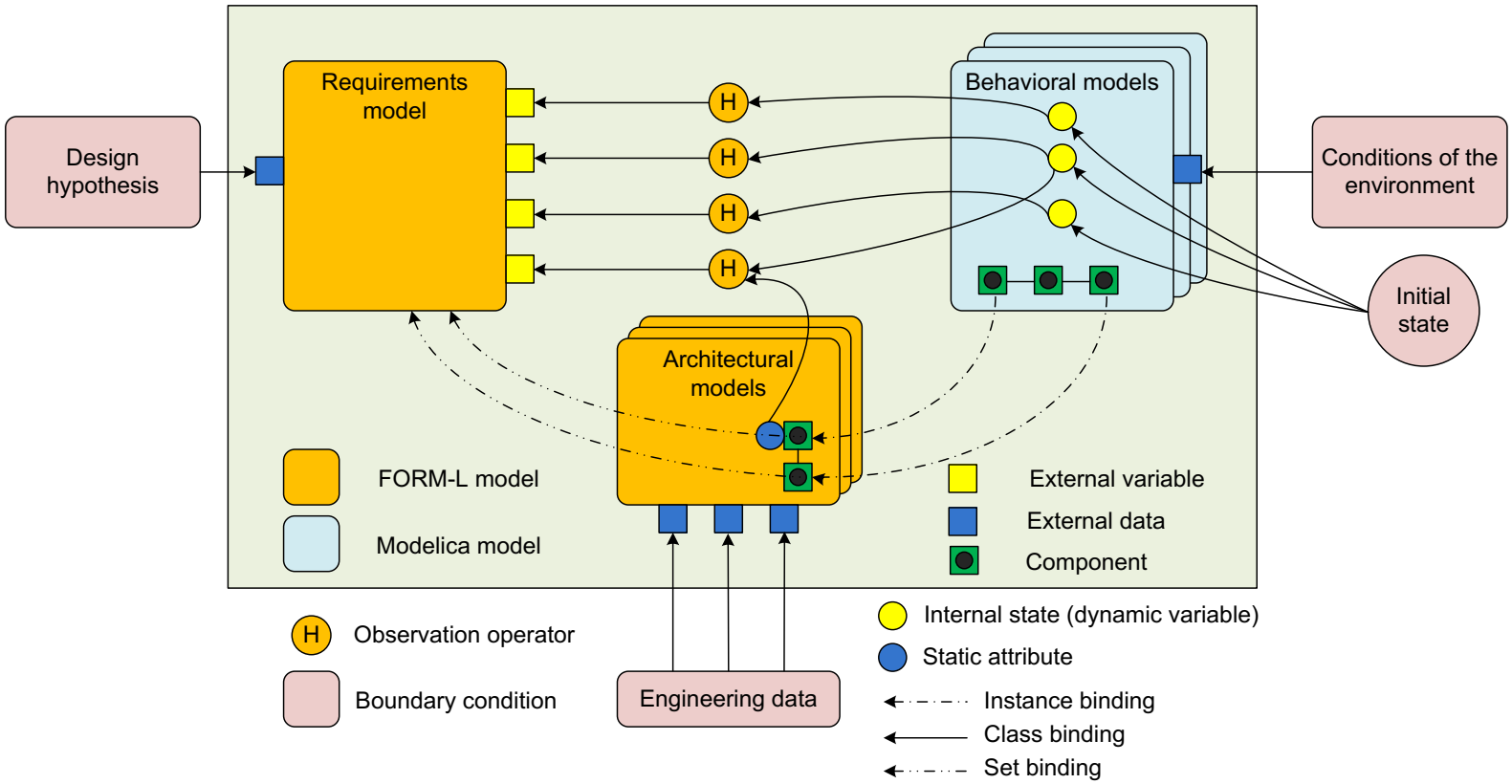
R = [Where or Which] [When] [What] + (optional) [How well]

```
for all' pump 'in' system.pumps 'during' system.inOperation 'check count' (pump.isStarted 'becomes true') '<=' 2;
'during' systemOperatingLife 'check at end' (estimator Probability (noStart at inOperation 'becomes false')) '>' 0.99;
```

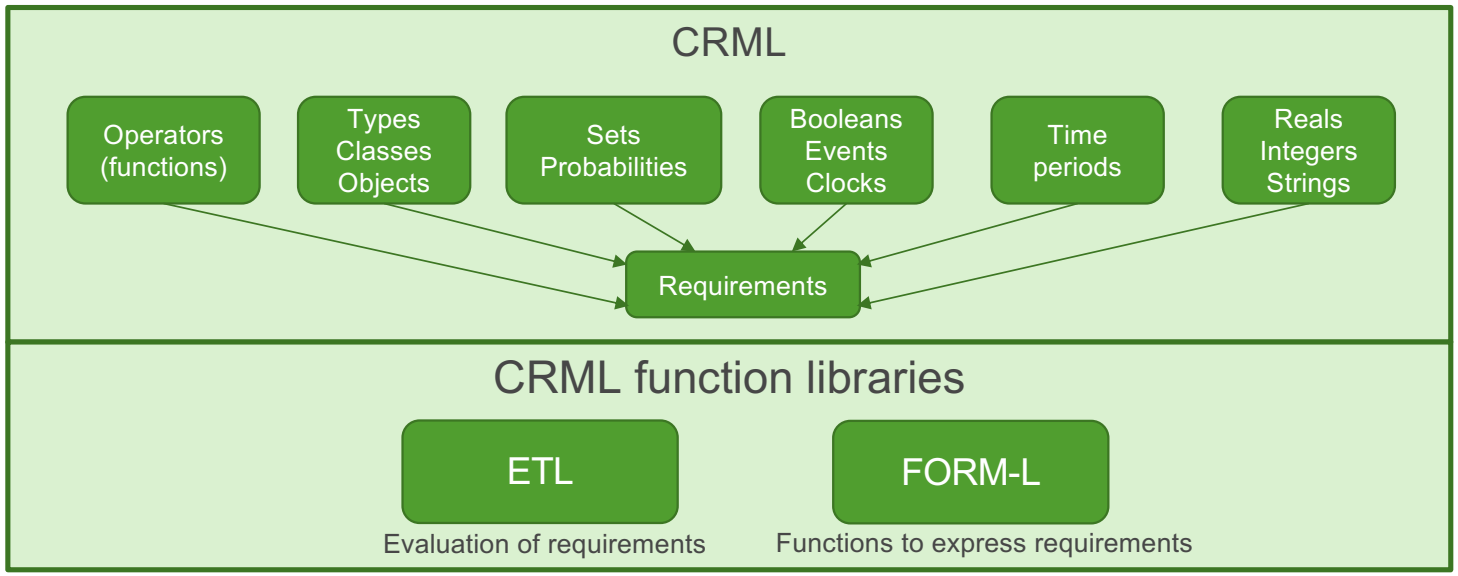
- Requirement R is formalized as a combination of
  - Spatial locators
  - Time locators
  - Condition
  - (optionally) Performance indicator

- It results in a Boolean4 whose value at instant t can be: `true`, `false`, `undefined` or `undecided`

# How to evaluate a CRML requirement?



# Architecture of CRML

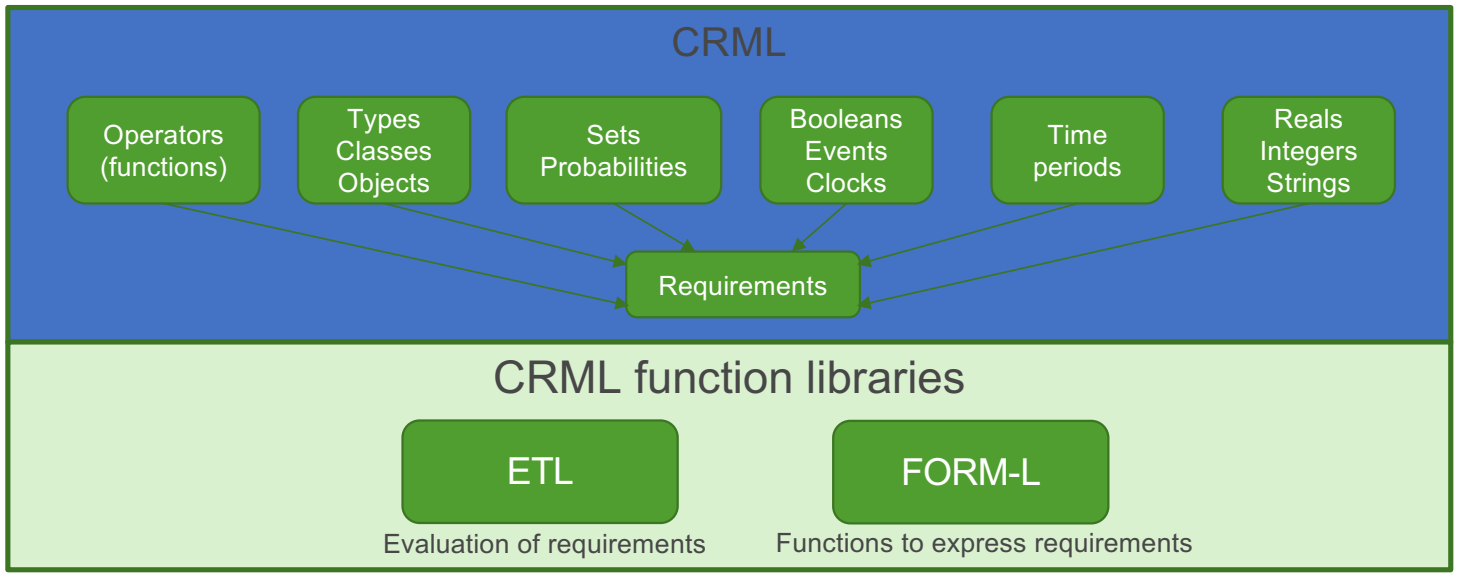



- **FORM-L** (FORMal Requirement Modelling Language) is a CRML library that implements the FORM-L language (T. Nguyen) to **express requirements** in more user-friendly way (i.e., closer to natural language)
- **ETL** (Extended Temporal Language) is a CRML library to **evaluate requirements** (i.e., whether they are satisfied or not)

# CRML language: Semantics and Syntax Overview

More details are available in CRML specification document:
https://www.embrace-project.org/specification/CRML.pdf

# Architecture of CRML




- **FORM-L** (FORMal Requirement Modelling Language) is a CRML library that implements the FORM-L language (T. Nguyen) to **express requirements** in more user-friendly way (i.e., closer to natural language)
- **ETL** (Extended Temporal Language) is a CRML library to **evaluate requirements** (i.e., whether they are satisfied or not)

# CRML Expressions

- Expressions

```
[ [ type ] ident is ] [ value | external ] [; | ,]
```

- Comments

```
// This is a single-line comment

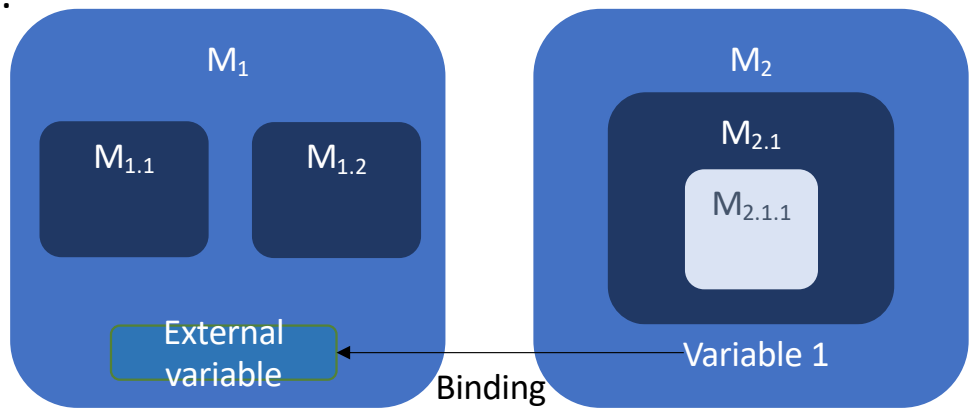
/* This is a
multiline comment */
```

# CRML Variables

- Types
  - `Real var;`
  - `Boolean var;`
  - `Integer var;`
  - `String pathToH; // Path to the observation operator H`
- Variability
  - Variables are function of time
  - Constant variables: variables assumed to be constant in time
    `constant TypeVar var`
  - Parameters: variables assumed to be constant during simulation time, but values could be different from one simulation to another
    `parameter TypeVar var`
- Value definition
  - Either directly via an expression: `TypeVar var is …;`
  - Or via another model: `TypeVar var is external;`

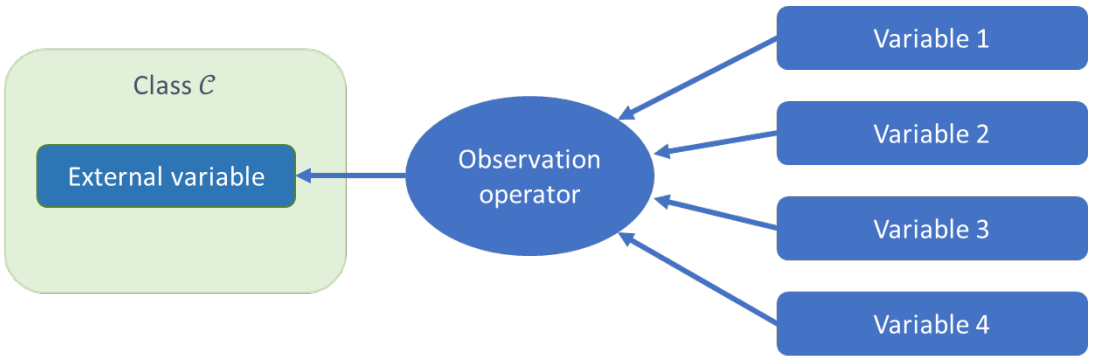
# Information Exchange between Models

- An **external variable** is a variable that gets its values from another model.
- There are two ways of exchanging information between models:
  - **Direct access.** Given an external variable y in model $M_1$ and a variable x in model $M_2$, direct access is obtained by writing $M_1.y = M_2.x$, where the value of x computed in $M_2$ is provided to the external variable y in $M_1$.
  - **Using bindings.** If one cannot write directly $M_1.y = M_2.x$ because some transformation must be done on the information carried by x, then a more general access can be obtained using bindings that generate expressions such as $M_1.O_1.y = H_y(M_2.\ O_2.x)$ where $O_1$ and $O_2$ are objects and $H_y$ an observation operator.



# External Variables and Variable Bindings

- **Bindings** allow an external variable var declared in a class $\mathcal{C}$ to get its values from an observation operator H that in turn gets its inputs from variables defined in other models.
- An external variable var declared in a class $\mathcal{C}$ can be associated with several possible observation operators. This binding is denoted $\mathcal{C}$.var ← [ $H_1$, …, $H_n$ ] where [ $H_1$, …, $H_n$ ] is the ordered set of possible observation operators for C.var.
- Class with binding declaration: $\mathcal{C}$ = { $att_1:\mathbb{A}_1$ ← [ $H_1$, …, $H_n$ ], …, $att_n:\mathbb{A}_n$.}
- The active binding is the observation operator $H_i$ chosen among [ $H_1$, …, $H_n$ ] to provide the values to $\mathcal{C}$.var at run time. The active binding is denoted $\mathcal{C}$.var ← $H_i$.



# CRML Keywords: Types

| Keyword | Semantics | Comments |
|---|---|---|
| **Boolean** | $\mathbb{B}$ | 4-valued Booleans. |
| **Category** | $\mathcal{C}(\mathbb{O}_1 \longrightarrow \mathbb{O}_2)$ | Categories. |
| **class** | $\mathcal{C}$ | Class definitions. |
| **Clock** | $2^{\mathcal{E}}$ or $\mathcal{D}$ | Clocks. |
| **Event** | $\mathcal{E}$ | Events. |
| **Integer** | $\mathbb{Z}$ | Positive and negative integers. |
| **library** | $\mathcal{L}$ | Libraries. |
| **model** | $\mathcal{M}$ | Models. |
| **Operator** | $\mathbb{O}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2)$ | Operators. The names of the domains $\mathbb{D}_1$ and $\mathbb{D}_2$ are given in the declaration of the operator. |
| **package** | $\mathcal{T}$ | Packages. |
| **Period** | $\mathcal{P}$ | Single time periods. |
| **Periods** | $2^{\mathcal{P}}$ | Multiple time periods. |
| **Probability** | $\mathbb{O}(\mathbb{B}_2 \longrightarrow \mathbb{R})$ | Probabilities. |
| **Real** | $\mathbb{R}$ | Real numbers. |
| **String** | $\mathbb{S}$ | Strings. |
| **Template** | $\mathbb{O}(\mathbb{B}^n \longrightarrow \mathbb{B})$ | Templates. |
| **type** | $\mathfrak{T}$ | Type definitions. |

# CRML Keywords: Special Values and Characters

| Values | Semantics |
|---|---|
| **false** | $false$ |
| **true** | $true$ |
| **undecided** | $undecided$ |
| **undefined** | $undefined$ |

| Characters | Semantics | Comments |
|---|---|---|
| **(** | ( | |
| **)** | ) | |
| **[** | [ | |
| **]** | ] | |
| **{** | { | Opens sets. |
| **}** | } | Closes sets. |
| **,** | , | Set element separator. |
| **;** | ; | Set element separator. |
| **.** | . | Decimal point. Path element separator. |
| **"** | " | String delimiter. |
| **'** | | Quote string delimiter. |
| **E** | | Decimal exponent. |
| **e** | | Decimal exponent. |
| **//** | | Start of comment line. |
| **/*** | | Begins comment. |
| ***/** | | Ends comment. |
| **0 \| 1 \| 2 \| 3 \| 4 \| 5 \| 6 \| 7 \| 8 \| 9** | | Digits |

# CRML Keywords: Built-in Operators

| Operators |
|---|
| = |
| + |
| - |
| * |
| / |
| < |
| <= |
| > |
| >= |
| == |
| <> |
| ^ |
| acos |
| alias |
| and |
| asin |
| associate |
| at |
| card |
| constant |
| cos |
| duration |
| element |
| else |
| end |

| Operators |
|---|
| estimator |
| exp |
| extends |
| external |
| filter |
| flatten |
| forbid |
| if |
| integrate |
| is |
| log |
| log10 |
| mod |
| new |
| not |
| on |
| or |
| parameter |
| partial |
| proj |
| redeclare |
| sin |
| start |
| then |
| tick |

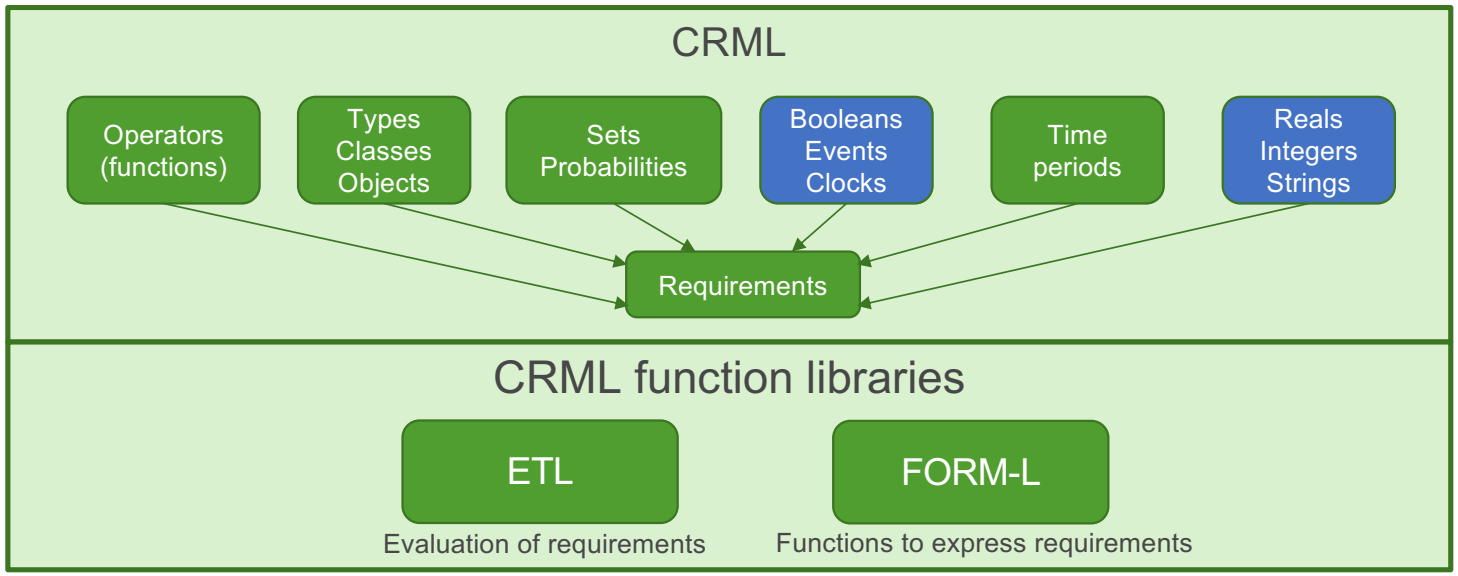
| Operators |
|---|
| time from |
| union |
| variance |
| while |
| with |

Example of operators' combination to "customize" test report

```
model GlobalReports is {

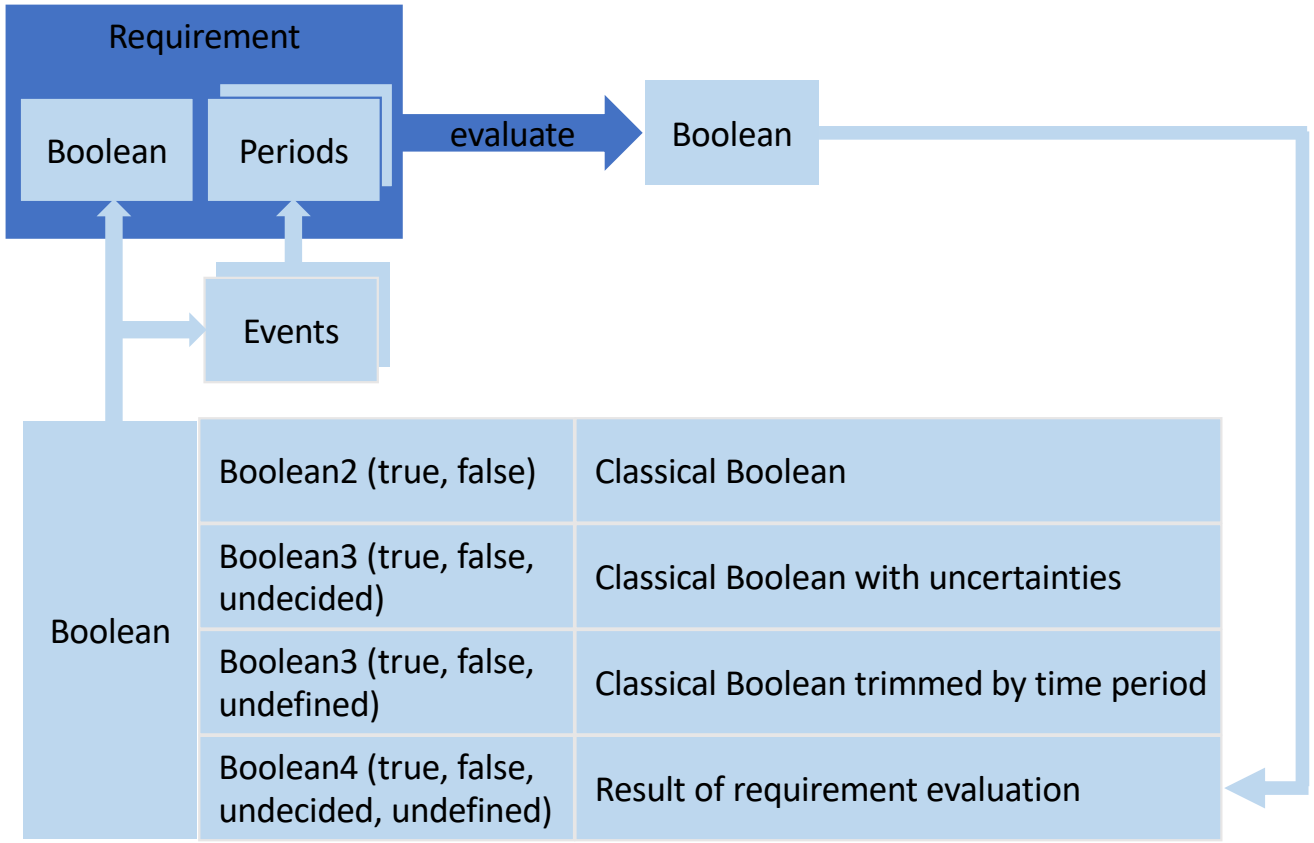
  // Conjunction of all requirements on Served_system
  Boolean globalReport_ForServedSystems is
    and flatten filter (flatten filter sriRequirements (type element == Served_system))
                                                       (type element == Boolean);
};
```

# Architecture of CRML




- **FORM-L** (FORMal Requirement Modelling Language) is a CRML library that implements the FORM-L language (T. Nguyen) to **express requirements** in more user-friendly way (i.e., closer to natural language)
- **ETL** (Extended Temporal Language) is a CRML library to **evaluate requirements** (i.e., whether they are satisfied or not)

# CRML Specific Types for Requirements




| | | |
|---|---|---|
| Boolean | Boolean2 (true, false) | Classical Boolean |
| | Boolean3 (true, false, undecided) | Classical Boolean with uncertainties |
| | Boolean3 (true, false, undefined) | Classical Boolean trimmed by time period |
| | Boolean4 (true, false, undecided, undefined) | Result of requirement evaluation |

# Definition of 4-Valued Booleans (or Boolean4)

- 4-valued Booleans φ satisfy a 4-valued Boolean algebra:

  $\varphi := \varphi \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2$

- The negation (¬) and conjunction (∧) logical operators are defined with truth tables.

$\varphi_1$ and $\varphi_2$ represent the values of two different variables at the same instant in time


| not | Logical negation | | | |
|---|---|---|---|---|
| φ | *true* | *false* | *undecided* | *undefined* |
| ¬φ | *false* | *true* | *undecided* | *undefined* |

| and | Logical conjunction | | | |
|---|---|---|---|---|
| $\varphi_1 \wedge \varphi_2$ | ***true*** | ***false*** | ***undecided*** | ***undefined*** |
| ***true*** | *true* | *false* | *undecided* | *true* |
| ***false*** | *false* | *false* | *false* | *false* |
| ***undecided*** | *undecided* | *false* | *undecided* | *undecided* |
| ***undefined*** | *true* | *false* | *undecided* | *undefined* |

If R1 or if R2 is `undecided`, then R1 ∧ R2 remains `undecided` unless R1 or R2 is `false`

`undefined` has no impact on the decision

If R is `undecided`, then ¬R remains `undecided`

# Definition of 4-Valued Booleans (or Boolean4)

- All other logical operators are defined using the Morgan laws.
  - Logical disjunction

    $\varphi_1 \vee \varphi_2 := \neg(\neg\varphi_1 \wedge \neg\varphi_2)$

  - Logical exclusive disjunction

    $\varphi_1 \oplus \varphi_2 := (\varphi_1 \vee \varphi_2) \wedge \neg(\varphi_1 \wedge \varphi_2)$

  - Logical inference

    $\varphi_1 \Rightarrow \varphi_2 := \neg\varphi_1 \vee \varphi_2$

# Definition of the Continuous Clock

- There is one **continuous clock** denoted $\mathfrak{C}$.
- It represents the Newtonian time. All time instants $t \in \mathbb{R}$.
- All time instants t in $\mathfrak{C}$ have a value which depends on an arbitrary time origin $t_0$.
  Therefore, only time differences (or delays) $t_2 - t_1$ are meaningful.
- Events always occur in the continuous time domain.
- If $R_1\uparrow$ occurs at time $t_1$, and $R_2\uparrow$ occurs at time $t_2$, then the delay between the two events is given by $R_2\uparrow - R_1\uparrow$. There is no need to get the values of $t_1$ and $t_2$ which are meaningless. The time difference is given by $t_2 - t_1 = R_2\uparrow - R_1\uparrow$.

# Definition of Time in the Continuous Clock

- Time t in the continuous clock $\mathfrak{C}$ is defined as the elapsed physical time from an initial event $E_0\uparrow$, which means that $t := \mathfrak{C}\uparrow - E_0\uparrow$ where $\mathfrak{C}\uparrow$ is the continuous clock tick (thus corresponding to the current instant).
- The fact that the value of time t depends on the initial event $E_0\uparrow$ is denoted $t \in \mathfrak{C}(E_0\uparrow)$.


```
Event e is external;
Real d_elapsed_physical_time is time from e;
```

# Definition of Discrete Clocks

- A **discrete clock tick** is an occurrence of an event $E\uparrow$. The discrete clock domain is denoted $\mathcal{D}$.
- A **discrete clock** is a set of ticks ordered in time. A clock can have a predefined or non-predefined number of ticks.

$$\Omega = \{ E_1\uparrow, E_2\uparrow, \ldots, E_i\uparrow, \ldots \}$$

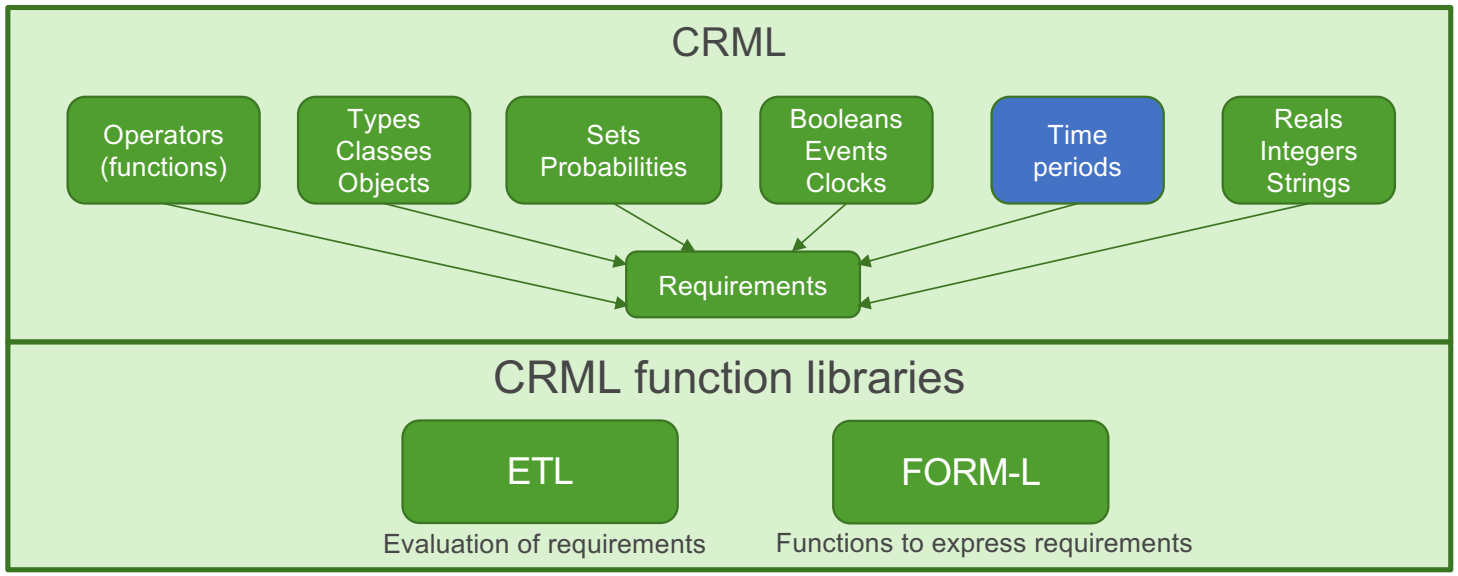
```
Boolean b is external;
Clock c is new Clock b;
```

- Clocks generated by the same Boolean R are denoted

$$\Omega(R) = \{ R(t_1)\uparrow, R(t_2)\uparrow, \ldots \}$$


- Time associated to a discrete clock is given by the sequence number of the clock tick, but each clock tick is associated with its time occurrence in the continuous clock domain. The $i^{th}$ tick of clock $\Omega$ is denoted $\Omega_i = R(t_i)\uparrow$. It is occurring at time $i \in \mathbb{N}$ in the discrete clock domain and at time $t_i \in \mathbb{R}$ in the continuous clock domain.

# Architecture of CRML




- **FORM-L** (FORMal Requirement Modelling Language) is a CRML library that implements the FORM-L language (T. Nguyen) to **express requirements** in more user-friendly way (i.e., closer to natural language)
- **ETL** (Extended Temporal Language) is a CRML library to **evaluate requirements** (i.e., whether they are satisfied or not)

# Definition of Time Periods

- An **event** $E\uparrow$ corresponds to the single occurrence of a 4-valued Boolean φ becoming true.

$$E\uparrow := \varphi\uparrow$$

```
Boolean b is external;
Event e is new Event b;
```

- A **time period** is a time interval between two events. The left and right boundaries can be included or excluded.

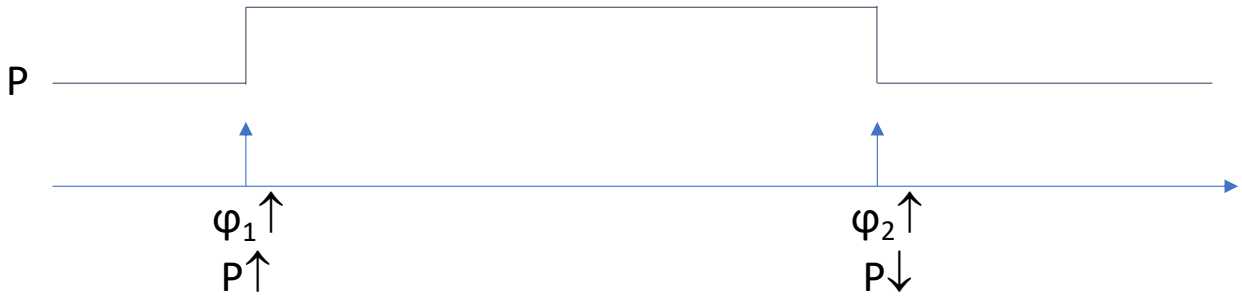
$$P := ([\ |\ ])\ \varphi_1\uparrow, \varphi_2\uparrow\ ([\ |\ ])$$

$\varphi_1\uparrow$ is called the **opening event**. It is denoted $P\uparrow$.
$\varphi_2\uparrow$ is called the **closing event**. It is denoted $P\downarrow$.

$$P := ([\ |\ ])\ P\uparrow, P\downarrow\ ([\ |\ ])$$

```
Event e1 is external;
Event e2 is external;
Period P1 is [ e1, e2 ];
Period P2 is ] e1, e2 ];
Period P3 is [ e1, e2 [;
Period P4 is ] e1, e2 [;
```

```
Period P is external;
Event e is P start;
Event e is P end;
```



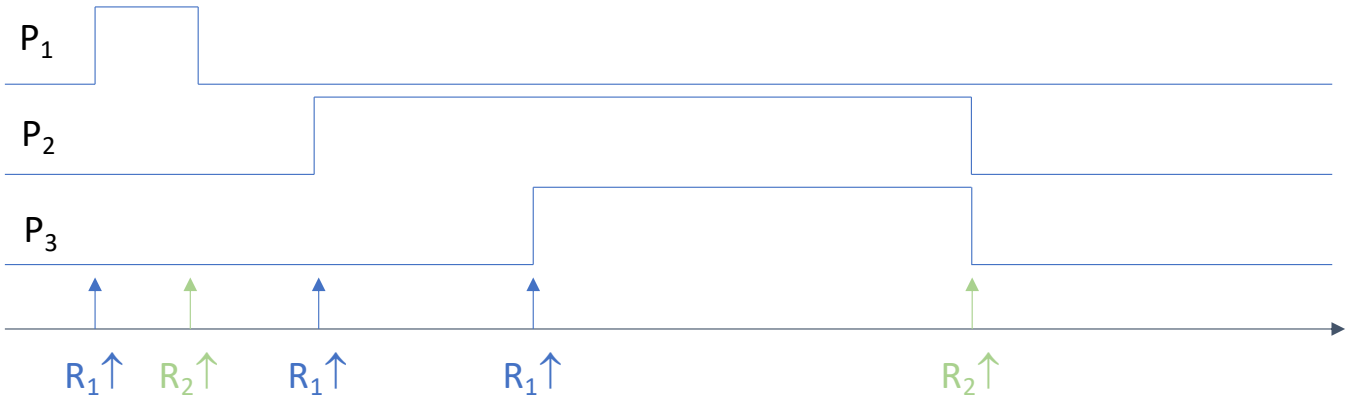
# Definition of Multiple Time Periods

- A **multiple time period** P is a set of time periods $P_i$ ordered by increasing opening events

$$P := \{ P_1, P_2, \ldots, P_n \}$$

- Multiple time periods P containing time periods opened at all occurrences of $R_1\uparrow$ and closed at all occurrences of $R_2\uparrow$ are denoted

$$P := \Pi(([\ |\ ])\ R_1\uparrow, R_2\uparrow\ ([\ |\ ])\ := \{ ([\ |\ ])\ \Omega(R_1), \Omega(R_2)\ ([\ |\ ]) \}$$




```
Period P1 is external;
Period P2 is external;
Period Pn is external;
Periods P3 is { P1, P2, Pn };
Clock c1 is external;
Clock c2 is external;
Periods P4 is [ c1, c2 ];
Periods P5 is ] c1, c2 ];
Periods P6 is [ c1, c2 [;
Periods P7 is ] c1, c2 [;
```

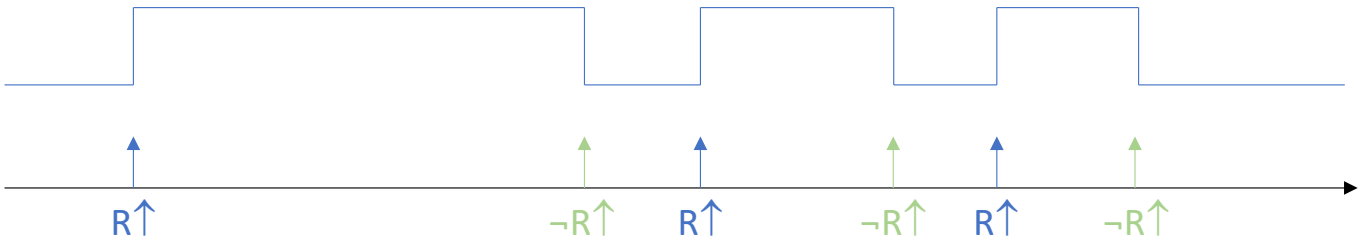
# Definition of Non-Overlapping Multiple Time Periods

- A **non-overlapping multiple time period** P is a multiple time period such that

  $P = \Pi(([\ |\ ])\ R\uparrow, \neg R\uparrow\ ([\ |\ ]))$

  where R is a Boolean.

- A non-overlapping multiple time period contains no overlapping time periods.



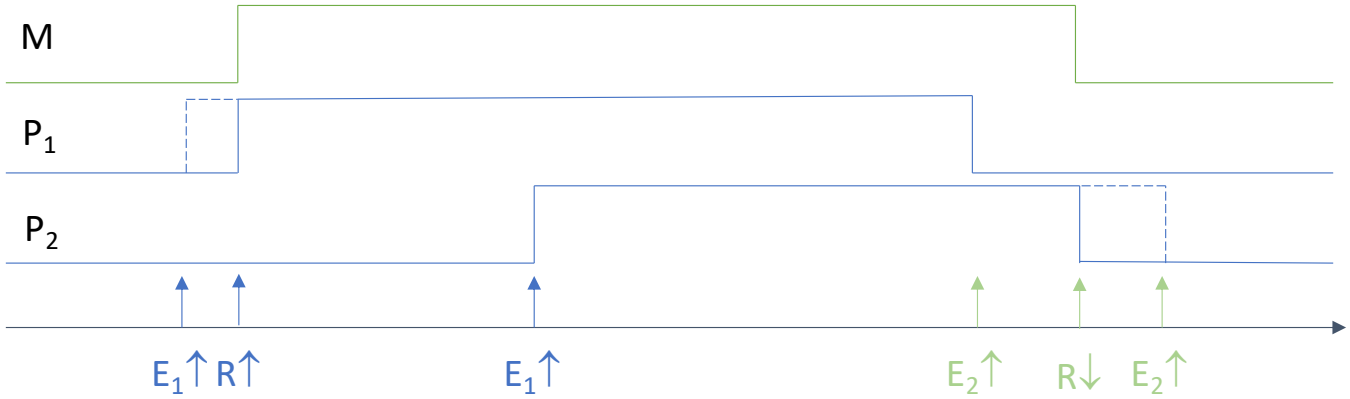
# Definition of Master Time Periods

- A **master time period** M is a time period or a non-overlapping multiple time period that truncates time periods or multiple time periods P. The result of the truncation is a time period if M and P are time periods, or a multiple time period denoted $M \supset P$ otherwise .
- If $M = \Pi(([\ |\ ])\ R\uparrow, \neg R\uparrow\ ([\ |\ ]))$ and $P = [\ E_1\uparrow, E_2\uparrow\ ]$, then

  $M \supset P := [\ (R \wedge E_1)\uparrow, (\neg R \vee E_2)\uparrow\ ]$

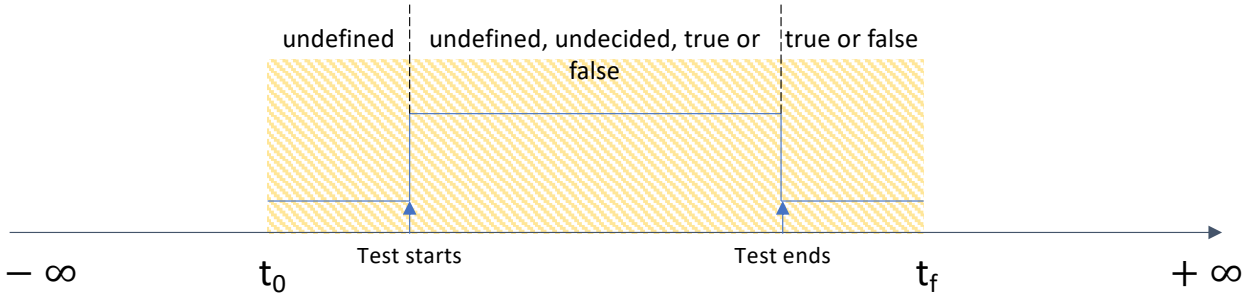
- If $P = \{\ P_1, P_2, \ldots P_n\ \}$ then

  $M \supset P := \{\ M \supset P_1, M \supset P_2, \ldots M \supset P_n\ \}$

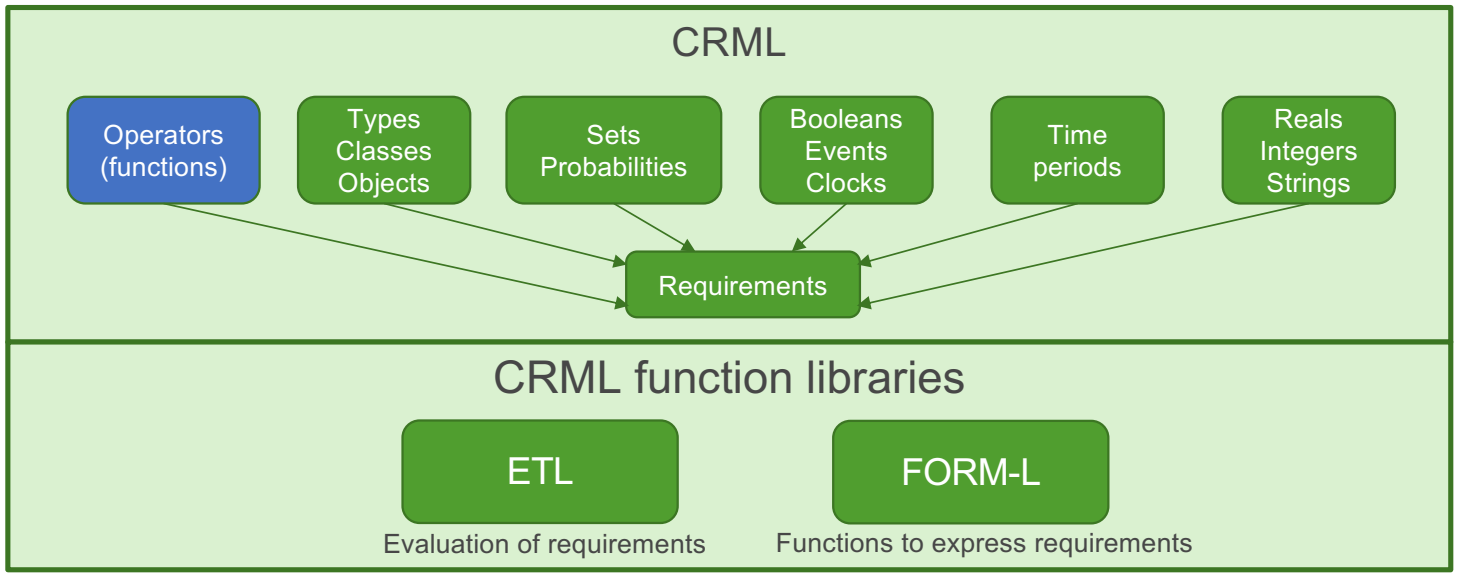


# Start and End of Simulation Time

- ETL formula for evaluating requirements are independent of simulation time window: ETL time spans from $-\infty$ to $+\infty$.
- Simulation time window spans from $t_0$ to $t_f$.
- The user must ensure that all events to be verified occur between $t_0$ and $t_f$, excluding $t_0$ and $t_f$. This can be done by **encapsulating all events in a single master time period**, that defines the beginning and the end of the test-case.
- Before the beginning of the master time period, all requirements are evaluated to undefined: testing has not started. After the end of the master time period, all requirements are evaluated to true or false. If some requirements are still undefined or undecided, that means that not all events are within the master time period and testing is incomplete.



# Architecture of CRML




- **FORM-L** (FORMal Requirement Modelling Language) is a CRML library that implements the FORM-L language (T. Nguyen) to **express requirements** in more user-friendly way (i.e., closer to natural language)
- **ETL** (Extended Temporal Language) is a CRML library to **evaluate requirements** (i.e., whether they are satisfied or not)

# Definition of Operators

- An **operator** is a function between two domains $\mathbb{A}$ and $\mathbb{B}$ (that can be equal or not)

  $f : \mathbb{A} \rightarrow \mathbb{B}$

- Domains $\mathbb{A}$ and $\mathbb{B}$ can be composed of multiple domains:
  - $\mathbb{A} = \mathbb{A}_1 \times \mathbb{A}_2 \times \ldots \times \mathbb{A}_n$
  - $\mathbb{B} = \mathbb{B}_1 \times \mathbb{B}_2 \times \ldots \times \mathbb{B}_n$
- A **unary operator** on domain $\mathbb{A}$ is a function

  $f : \mathbb{A} \rightarrow \mathbb{A}$

- A **binary operator** on domain $\mathbb{A}$ is a function

  $f : \mathbb{A} \times \mathbb{A} \rightarrow \mathbb{A}$

```
// Definition of disjunction of two Booleans.
Operator [ Boolean ] Boolean b1 'or' Boolean b2
    = not (not b1 and not b2);

// Example of function call
Boolean b1 is external;
Boolean b2 is external;
Boolean b is b1 'or' b2;
```

→ Useful e.g., for defining appropriate operators to rigorously evaluate requirements (see ETL library)

# Definition of Templates

- **Templates** are operators on Booleans taking Booleans as values.

  $t: \mathbb{B}_1 \times \mathbb{B}_2 \times \ldots \times \mathbb{B}_n \rightarrow \mathbb{B}$

- Example: mission change. If requirement $R_1$ fails, then requirement $R_2$ should be satisfied at most 1h after $R_1$ fails.

  missionChange ($R_1$, $R_2$) := $\neg R_1 \Rightarrow R_2 \otimes [R_1\downarrow, R_1\downarrow + 1*h]$

  If mission change fails, then switch to R3

  missionChange (missionChange ($R_1$, $R_2$), $R_3$)

```
// Definition of disjunction of two Booleans.
Template b1 'or' b2 = not (not b1 and not b2);

// Logical inference
Template b1 'implies' b2 = not b1 'or' b2;
```

- Example: if assumption $A_1$ is verified then $R_1$ should be satisfied, else $R_2$ should be satisfied

  rule ($A_1$, $R_1$, $R_2$) := $(A_1 \Rightarrow R_1) \wedge (\neg A_1 \Rightarrow R_2)$

→ Useful e.g., for defining combination of requirements (see logical inference)

# Definition of Categories

- **Categories** are operators on operators. They are introduced to handle the computation of operator $a(\varphi)$ in

$$\varphi \otimes P := \int_{\varphi \in P} a(\varphi)\, d\varphi$$

- A category is defined as an operation on a domain $\mathbb{O}$ of operators

$$\begin{array}{ccc} \mathbb{O}(\mathbb{D}_1 \rightarrow \mathbb{D}_2) & \rightarrow & \mathbb{O}(\mathbb{D}_1 \rightarrow \mathbb{D}_2) \\ o & \rightarrow & C(o) \end{array}$$

- Example from fragment code of ETL library

→ Useful e.g., for defining the early decision event and have a requirement diagnosis (if possible) before the end of the time periods (see ETL library)

```
// Decide
Operator 'decide' is
    Operator [ Boolean ] 'decide' Boolean phi 'over' Period P = phi 'or' (P end));

Category c1 is Category increasing1
    = { (>, >), (>=, >=), (<, >=), (<=, >), (==, >), (<>, >) };
Category {} C1 is associate increasing1 with 'decide';

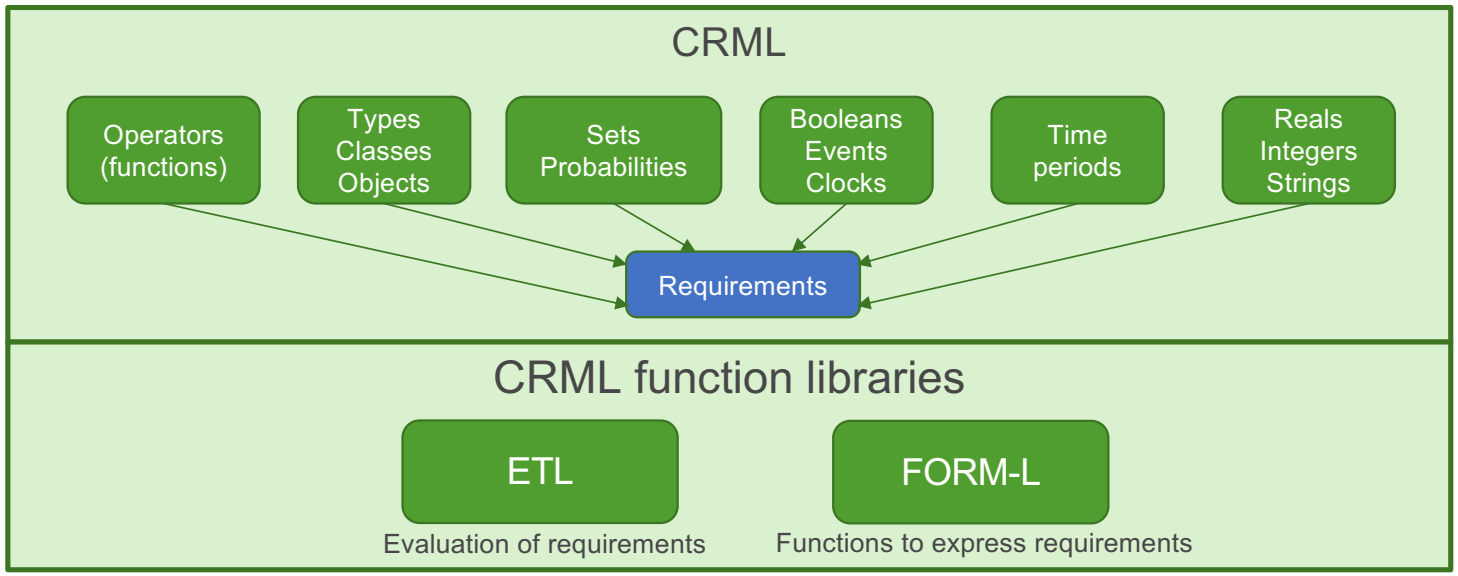
Category c2 is Category increasing2
    = { (>, >), (>=, >=), (<, >=), (<=, >) };
Category {} C2 is associate increasing2 with 'decide';

Operator [ Boolean ] 'id' Boolean b = b;
Operator [ Boolean ] 'cte_false' Boolean b = false;
Operator [ Boolean ] 'cte_true' Boolean b = true;

Category c3 is Category varying1 = { ('id', 'cte_false') };
Category {} C3 is associate varying1 with 'decide';

Category c4 is Category varying2 = { ('id', 'cte_true') };
Category {} C4 is associate varying2 with 'decide';
```

# Architecture of CRML




- **FORM-L** (FORMal Requirement Modelling Language) is a CRML library that implements the FORM-L language (T. Nguyen) to **express requirements** in more user-friendly way (i.e., closer to natural language)
- **ETL** (Extended Temporal Language) is a CRML library to **evaluate requirements** (i.e., whether they are satisfied or not)

# Definition of a Requirement

- A **requirement** R is a 4-valued Boolean obtained by combining a 4-valued Boolean φ with a time period P:

  $R := \varphi \otimes P$

- Therefore, the ⊗ operator that creates requirements goes from the cross-product of the domains of Boolean and time periods to the domain of Booleans:

$$\otimes: \mathbb{B} \times \mathcal{P} \rightarrow \mathbb{B}$$

- The value of $R = \varphi \otimes P$ is:

$$\varphi \otimes P := \int_{\varphi \in P} a(\varphi)\, d\varphi$$

$a(\varphi)$ depends on the condition φ.
It filters out the events dφ that are not decision events.

- Value of R varies with time until period P ends and stays constant after period P closed.
- Examples

$\varphi := \text{count}(E\uparrow, P) \leq 2$
$a(\varphi) := \text{count}(E\uparrow, P) > 2 = \neg\varphi$

$\varphi := \text{count}(E\uparrow, P) \geq 3$
$a(\varphi) := \text{count}(E\uparrow, P) \geq 3 = \varphi$

# Definition of Decision Events

- A **decision event** is an event that occurs
  when the decision whether R is satisfied or not can be made.
- The **decision events** for condition $\varphi \in \mathbb{B}$ and single time period $P \in \mathcal{P}$ constitute
  a clock whose ticks are denoted $\mathrm{dev}(\varphi, P)\uparrow$.

$$\mathrm{dev}(\varphi, P)\uparrow = \big(a(\varphi, P)\times\varphi \vee \neg(a(\varphi, P)\times\varphi)\big)\uparrow$$

- They occur when $a(\varphi, P)\times\varphi$ becomes true or false.
- In principle, the value $\varphi \otimes P$ should be the same at all decision events,
  but there may be some situations where the value of $\varphi \otimes P$ can differ from one decision event to the other.

# Events Filtering

- The **filter** $a(\varphi, P)$ filters out the events of $\varphi$ which are not decision events and ensures that the closing event of $P$ is always a decision event if $\varphi \in \mathbb{B}_2$.
- The filter is composed of two terms, one that depends only on $\varphi$ and the other that depends only on $P$:

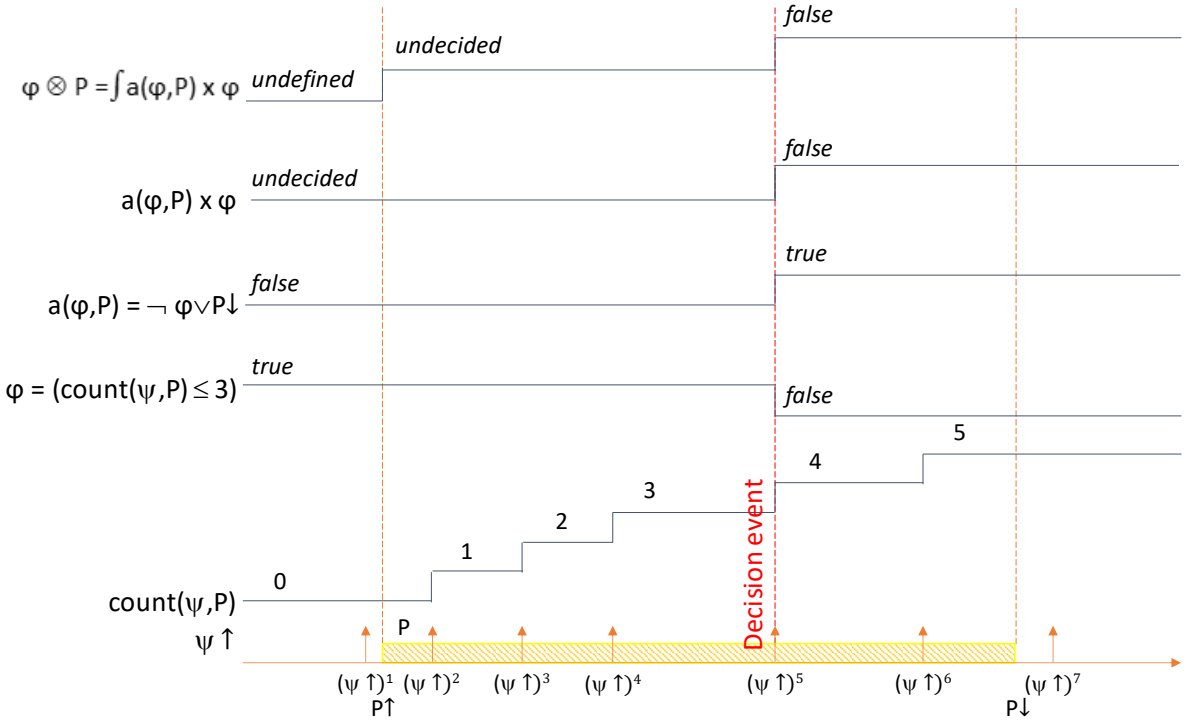
$$a\colon\ \mathbb{B}\times\mathcal{P} \longrightarrow\ \mathbb{B}$$
$$a(\varphi, P) = a'(\varphi) \vee P\downarrow$$

- $\mathrm{dev}'(\varphi, P)\uparrow = \big(a'(\varphi, P)\times\varphi \vee \neg(a'(\varphi, P)\times\varphi)\big)\uparrow$ is called the **early decision event** because if it occurs, it occurs before $P\downarrow$.
- There is no general algorithm for computing $a'(\varphi)$ from the condition $\varphi$, but a general procedure is given in CRML specification.

# Requirements' Evaluation

Example 1

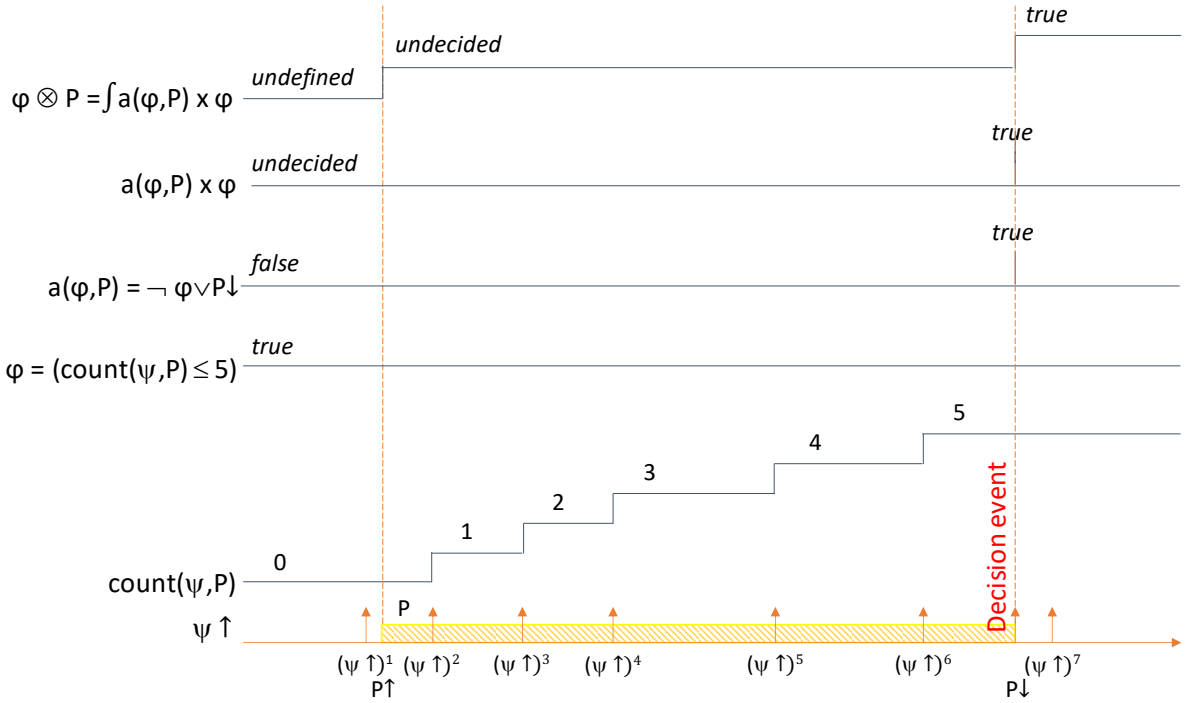
with the decision event
at the end of the time period



# Requirements' Evaluation

Example 2

with the decision event
at the end of the time period



# Definition of Satisfaction of a Requirement

- A requirement $R = \varphi \otimes P$ is satisfied iff $\varphi \otimes P = true$ at the decision event $\mathrm{dev}(\varphi, P)\uparrow$.
- The **satisfaction of $R$** is denoted $P \vDash \varphi$
  which means that condition $\varphi$ is satisfied over time period $P$:

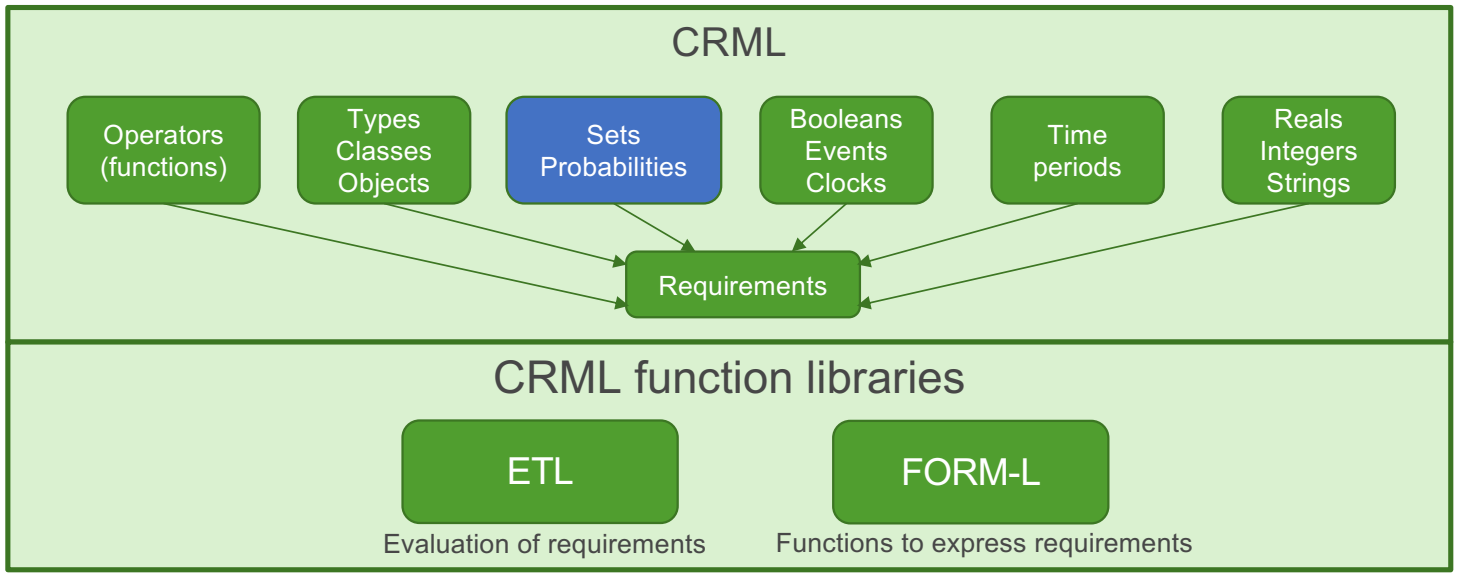
$$\vDash: \ 2^{\mathcal{P}} \times \mathbb{B} \longrightarrow \ \mathbb{B}_2$$
$$(P, \ \varphi) \longmapsto P \vDash \varphi := (\varphi \otimes P)(@\mathrm{dev}(\varphi, P)\uparrow) = true)$$

- $P \vDash \varphi$ is a 2-valued Boolean:
  - If $R = \varphi \otimes P$ is satisfied, then $(\varphi \otimes P)(@\mathrm{dev}(\varphi, P)\uparrow) = true)$ and $P \vDash \varphi = true$.
  - If $R = \varphi \otimes P$ is not satisfied, then $(\varphi \otimes P)(@\mathrm{dev}(\varphi, P)\uparrow) = false)$ and $P \vDash \varphi = false$.
- Therefore : $P \vDash \varphi = (\varphi \otimes P)(@\mathrm{dev}(\varphi, P)\uparrow)$
- The value of $(\varphi \otimes P)(t)$ is:

$$(\varphi \otimes P)(t) = \begin{cases} undefined \ \text{if} \ t < @P_1\uparrow \\ undecided \ \text{if} \ @P_1\uparrow \leq t < @\mathrm{dev}(\varphi, P)\uparrow \\ P \vDash \varphi \ \text{if} \ t \geq @\mathrm{dev}(\varphi, P)\uparrow \end{cases}$$

`undefined` till time period P is not open
`undecided` till decision event hasn't occurred within P
`satisfied` or `not satisfied` when decision event occurred

# Architecture of CRML




- **FORM-L** (FORMal Requirement Modelling Language) is a CRML library that implements the FORM-L language (T. Nguyen) to **express requirements** in more user-friendly way (i.e., closer to natural language)
- **ETL** (Extended Temporal Language) is a CRML library to **evaluate requirements** (i.e., whether they are satisfied or not)

# Definition of Sets

- A **set** S is a finite collection of n elements $a_i$ from the same domain $\mathbb{A}$:

  $S := \{ a_1, a_2, \ldots a_n, a_i \in \mathbb{A}, 1 \leq i \leq n \}$

- There are no duplicates in a set.
- If set $S_1$ and set $S_2$ have the same elements, they are equal even if their elements are listed in different orders.
- Three kinds of sets
  - Typed
  - Empty
  - Special
- **Special sets** are sets with special operators
  - Sets of requirements
  - Clocks: sets of events
  - Multiple time periods: sets of time periods
  - Sets of objects

```
// Non-empty typed set
Real e1 is external;
Real e2 is external;
Real en is external;
[Real {} S1 is] { e1, e2, en };
// Empty typed set
[Boolean {} S2 is] {};
// Special set
model S3 is { e1, S2 };
```

# Sets of Requirements

- Requirements can be grouped into sets of requirements

$$S := \{ R_1, R_2, \ldots, R_n \}$$

- Boolean operators can be iteratively applied to all elements of a set in the following way:
  - Unary operators:

$$op\ S := \{ op\ R_1, op\ R_2, \ldots, op\ R_n \}$$

$$\neg S := \{ \neg R_1, \neg R_2, \ldots, \neg R_n \}$$

```
// Negation of a set of Booleans.
Boolean R1 is external;
Boolean R2 is external;
Boolean R3 is external;
Boolean R4 is external;
Boolean S1 is not { R1, R2, R3, R4 };
// The value of S1 is { not R1, not R2, not R3, not R4}.
// Note that { R1, R2, R3, R4 } is an anonymous set.

Boolean S2 is { not R1, not R2, not R3, not R4};
```

  - Binary operators:

$$op\ S := R_1\ op\ R_2 \ldots op\ R_n$$

$$\wedge S := R_1 \wedge R_2 \ldots \wedge R_n$$

$$\vee S := R_1 \vee R_2 \ldots \vee R_n$$

$$\oplus S := R_1 \oplus R_2 \ldots \oplus R_n$$

# Sets of Requirements

- In general, requirements are sets of requirements because they are in general associated with multiple time periods. It is therefore also possible to associate multiple requirements with time periods and with multiple time periods.
- The multiple time period associated with $R = \varphi\otimes\{P_1, P_2, \ldots, P_n\}$ is denoted $\otimes R$

  $\otimes R := \{ P_1, P_2, \ldots, P_n \}$
- Condition $\varphi$ associated with a multiple time period

  $R = \varphi\otimes\{P_1, P_2, \ldots, P_n\} := \{ \varphi\otimes P_1, \varphi\otimes P_2, \ldots, \varphi\otimes P_n \}$
- Multiple conditions associated with a time period P

  $R = \{ \varphi_1, \varphi_2, \ldots, \varphi_p \}\otimes P := \{ \varphi_1\otimes P, \varphi_2\otimes P, \ldots, \varphi_p\otimes P \}$
- Multiple condition associated with a multiple time period

  $R = \{ \varphi_1, \varphi_2, \ldots, \varphi_p \}\otimes\{ P_1, P_2, \ldots, P_n \}$
  $:= \{ \varphi_1\otimes\{P_1, P_2, \ldots, P_n\}, \varphi_2\otimes\{ P_1, P_2, \ldots, P_n \}, \ldots, \varphi_p\otimes\{ P_1, P_2, \ldots, P_n \}\}$
  $= \{ \varphi_i\otimes P_j, 1 \le i \le p, 1 \le j \le n \}$
  $= \{\{ \varphi_1, \varphi_2, \ldots, \varphi_p \}\otimes P_1 \}, \{\{ \varphi_1, \varphi_2, \ldots, \varphi_p \}\otimes P_2 \}, \ldots, \{\{ \varphi_1, \varphi_2, \ldots, \varphi_p \}\otimes P_n \}\}$

# Definition of Probabilistic Requirements

- The satisfaction $x$ of requirement $R = \varphi \otimes P$ is defined as $x = P \vDash \varphi$, where $x$ is a 2-valued Boolean that takes the value $true$ if $R$ is satisfied, or $false$ otherwise.
- A **realistic requirement** cannot be satisfied with absolute certainty.
  To that end, the satisfaction of a **probabilistic requirement** $R$ is defined as a condition to be satisfied on the probability $p$ that $x = true$,
  e.g. $p \geq 99.9\%$, or $p \geq f(t)$ where $f(t)$ is a function of time.
- Then $x \in \mathbb{B}_2$ is replaced by a random variable $X$ that associates probabilistic events to the outcomes $true$ or $false$:

$$X\colon \Omega \longrightarrow \mathbb{B}_2$$

where $\Omega$ denotes the domain of probabilistic events.
$\Omega$ is the randomized version of domain $2^{\mathcal{P}} \times \mathbb{B}$.

- Examples of probabilistic events can be whether a tank level subject to random fluctuations exceeds a maximum level within a given time period, or whether a system goes out of its authorized operating domain according to measurements subject to random errors.

# Definition of Probabilities

- The probability that requirement $R = \varphi \otimes P$ is satisfied is then $\mathbb{P}(X = true | \mathrm{dev}(\varphi, P) \uparrow)$, the outcomes of $X$ being $x = P \vDash \varphi$.
- Probabilities

```
// External inputs
Boolean b is external;
Clock c is external;

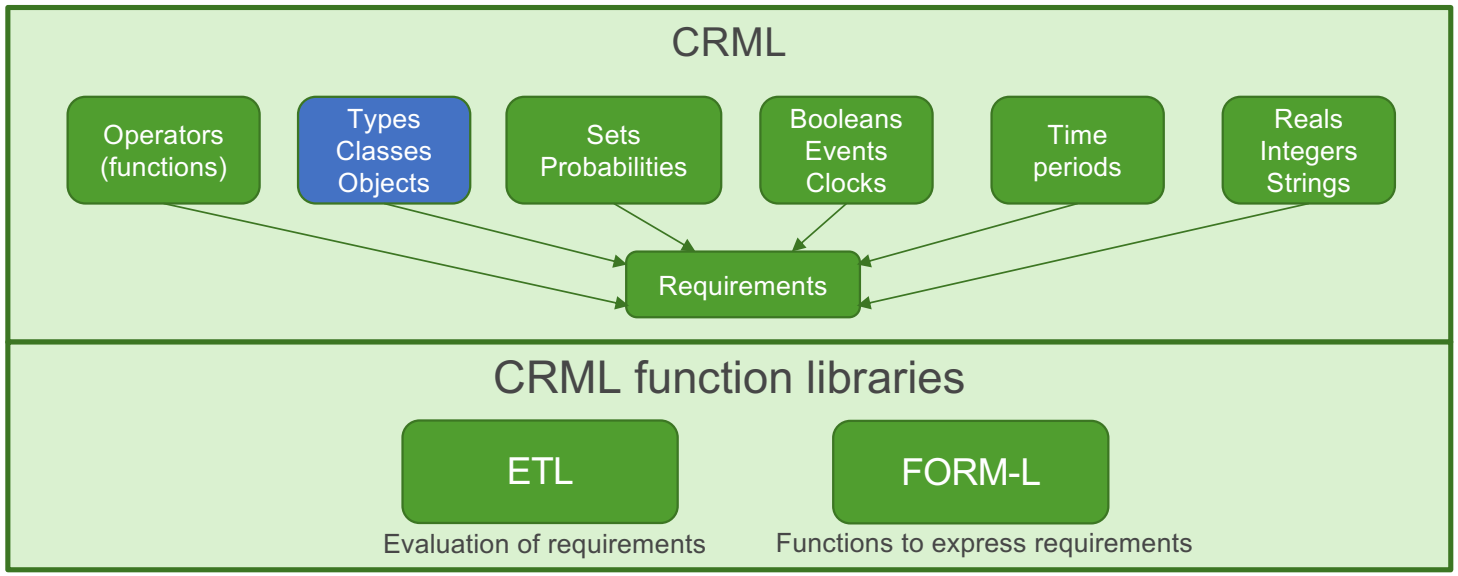
// Probability that b is true at all time instants t
Probability p_of_b is Probability b;

// Probability that b is true at all ticks of clock c
Probability p_of_b_at_c is Probability b at c;
```

- Estimate of mean values and variances of random variables

```
Probability x is external;
Real y_estimator_of_x is estimator x;
Real y_estimator_of_variance_of_x is estimator variance x;
```

# Architecture of CRML




- **FORM-L** (FORMal Requirement Modelling Language) is a CRML library that implements the FORM-L language (T. Nguyen) to **express requirements** in more user-friendly way (i.e., closer to natural language)
- **ETL** (Extended Temporal Language) is a CRML library to **evaluate requirements** (i.e., whether they are satisfied or not)

# Definition of Classes

The 4 requirements can be encapsulated into a generic class « Pump »

```
class Pump is {
    String ident;
    Boolean isStarted is external;
    Real temperature is external;

    // R1: While the system is in operation, the pump must not be started more than twice.
    Requirement R1 is
        'during' system.inOperation 'check count' (pump.isStarted 'becomes true') '<=' 2;
    // R2: At least one hour must separate two consecutive pump startups.
    Requirement R2 is
        'after' pump.isStarted 'for' 1*h 'check count' (pump.isStarted 'becomes true') '==' 0;
    // R3: While the pump is in operation (i.e. started), its temperature must always stay below 50°C.
    Requirement R3 is
        'during' pump.isStarted 'ensure' pump.temperature < 50*degC;
    // R4: While the system is in operation, after the pump temperature rises above 40 °C,
    // the temperature must not stay above for a duration of more than 1 mn cumulated over the next 15 mn.
    Requirement R4 is
        'during' system.inOperation 'after' pump.temperature > 40*degC 'for' 15*mn
            'check duration' (pump.temperature > 40*degC) '<' 1*mn;
};

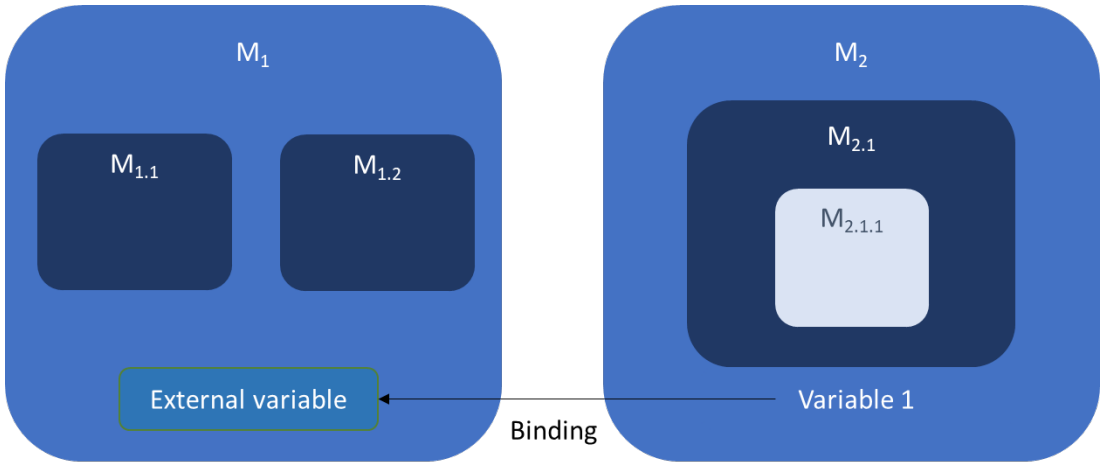
class System is {
    Pump {} pumps;
    Boolean inOperation is external;
};

System system is System{ Pump (ident = "PO1"),  Pump (ident = "PO2"), Pump (ident = "PO3") };
```

When the system is defined, each requirement is automatically instanciated for each pump

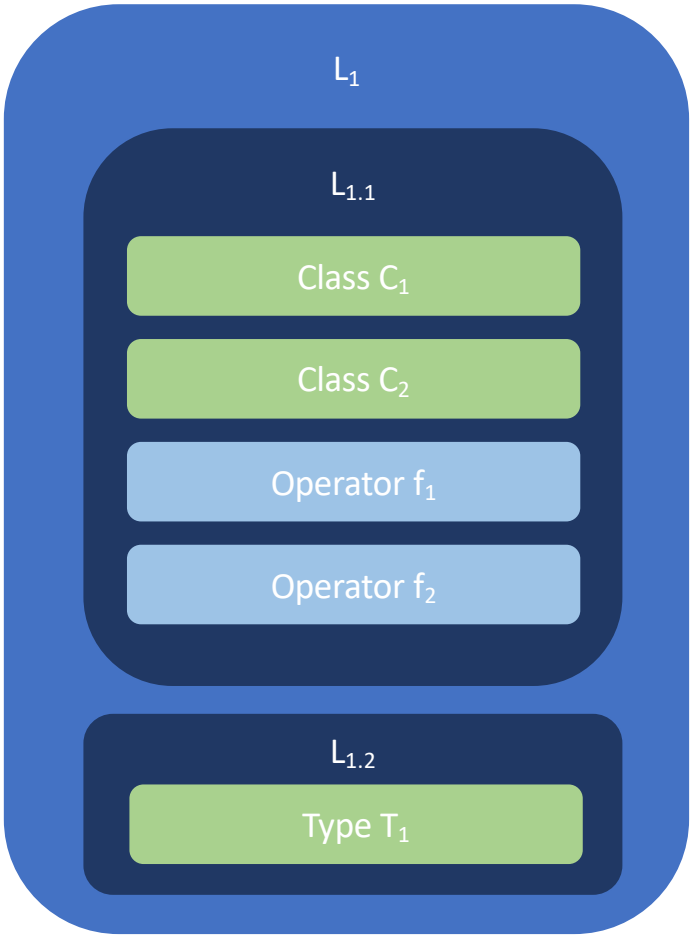
# Definition of a Model

- A model M is a container that can contain the following entities:
  - Models M' ≠ M
  - Types T and classes $\mathcal{C}$
  - Objects O and sets of objects S
  - Variables x
  - Operators f and templates t
  - Calls to operators f and templates t
  - Requirements R (i.e., calls to operators $\mathbb{B} \times \mathcal{P} \rightarrow \mathbb{B}$)



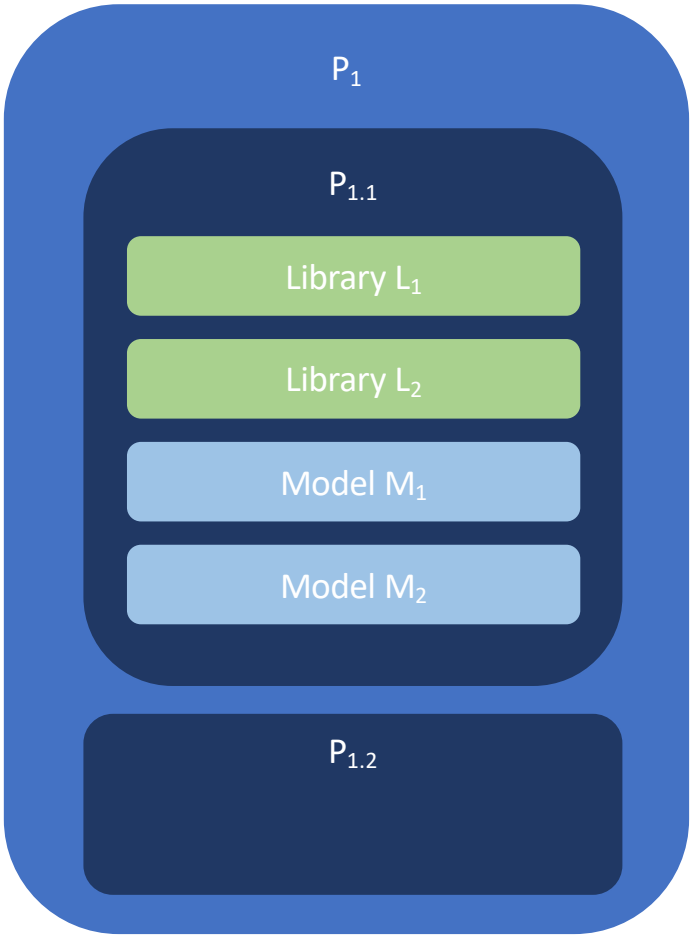
# Definition of a Library

- A library L is a container that can contain the following entities:
  - Libraries L' ≠ L
  - Types T
  - Classes $\mathcal{C}$
  - Operators f
- An entity E that belongs to a library L is denoted E ∈ L or L.E.
  - Ex: $L_1.L_{1.1}.C_1$



# Definition of a Package

- A package P is a container that can contain the following entities:
  - Packages P' ≠ P
  - Libraries L
  - Models M
- An entity E that belongs to a package P is denoted E ∈ P or P.E.
  - Ex: $P_1.P_{1.1}.L_1$



# Definition of Customized Types

- Example 1:
  Definition of a new type for requirements

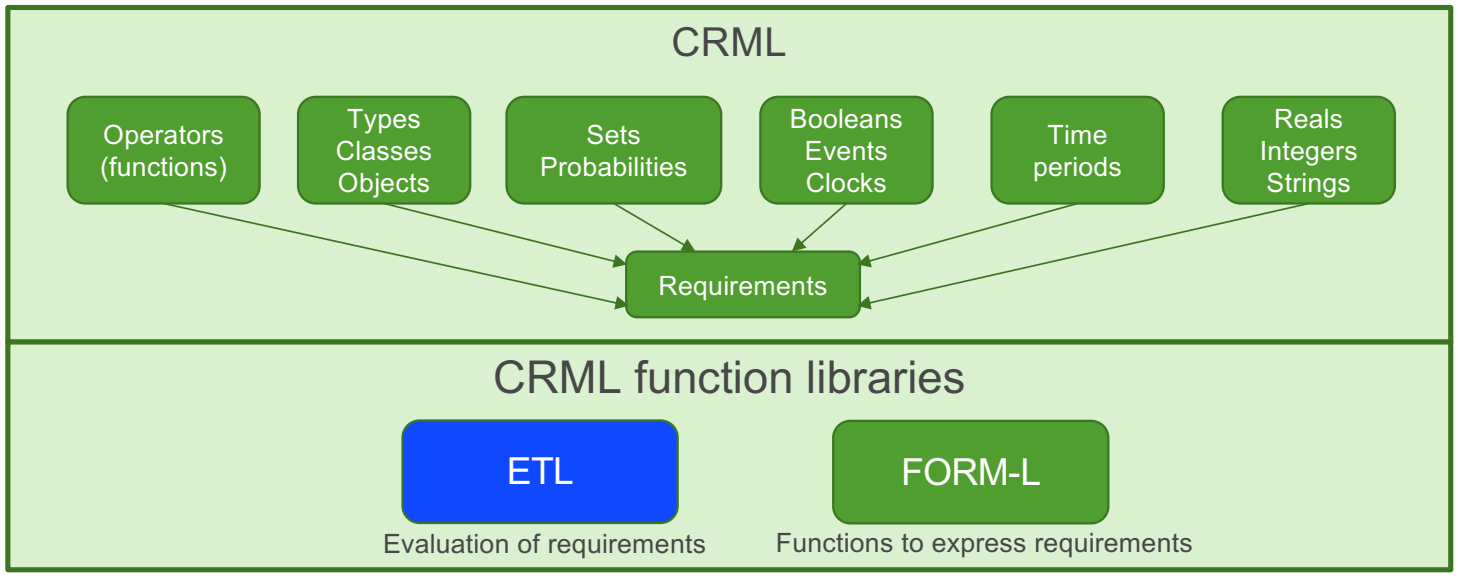
```
// Type Requirement is aimed at providing a dedicated keyword
// to requirements and forbid the use of temporal operators by the user.
type Requirement is Boolean forbid { *, +, integrate };
```

- Example 2:
  Definition of SI units

```
/* Define type PhysicalQuantity to handle physical units */
partial type PhysicalQuantity is (Real q is rate*u + offset) {
  String SIUnit;                    // SI unit for quantity q
  String userUnit;                  // User unit for quantity q
  String SIDimension;               // Unit for quantity q expressed in the fundamental units
  String SIFUnits = "[m][kg][s][K]";    // Fundamental units of the chosen unit system
  Real u;                           // Quantity q expressed in user units
  Real rate;                        // Conversion rate between user units and SI units
  Real offset;                      // offset between user units and SI units
};

/* Physical unit:  temperature */
partial type Temperature is PhysicalQuantity (SIUnit = "K", SIDimension= "[K]");
type TemperatureKelvin is Temperature (userUnit = "K", rate = 1, offset = 0) alias K;
type TemperatureCelsius is Temperature (userUnit = "Celsius", rate = 1, offset = 273.15) alias Celsius;
```

# Architecture of CRML




- **FORM-L** (FORMal Requirement Modelling Language) is a CRML library that implements the FORM-L language (T. Nguyen) to **express requirements** in more user-friendly way (i.e., closer to natural language)
- **ETL** (Extended Temporal Language) is a CRML library to **evaluate requirements** (i.e., whether they are satisfied or not)

# ETL: From 2-value logic to 4-value logic

| | |
|---|---|
| Requirement | R := (*true* \| *undecided* \| *undefined* \| a \| R)⊗P |
| Time period | P := ([ \| ]) event, event ([ \| ]) |
| Event | event := R↑ (R becomes *true*) |
| Boolean algebra | R := R \| ¬R \| R1 ∧ R2 |

New temporal operators to evaluate requirements subject to time periods :

x : filters events to keep decision events
+ : accumulates decision events to make the final decision (satisfied or not)

LINKÖPING UNIVERSITY EDF

# ETL: Requirement Evaluation by Example

**Time period:** `P = 'during' system.inOperation`

**Condition:** `ϕ = 'check count' (pump.isStarted 'becomes true') '<=' 2`

**Requirement:** `ϕ⊗P = 'during' system.inOperation 'check count'(pump.isStarted 'becomes true') '<=' 2`

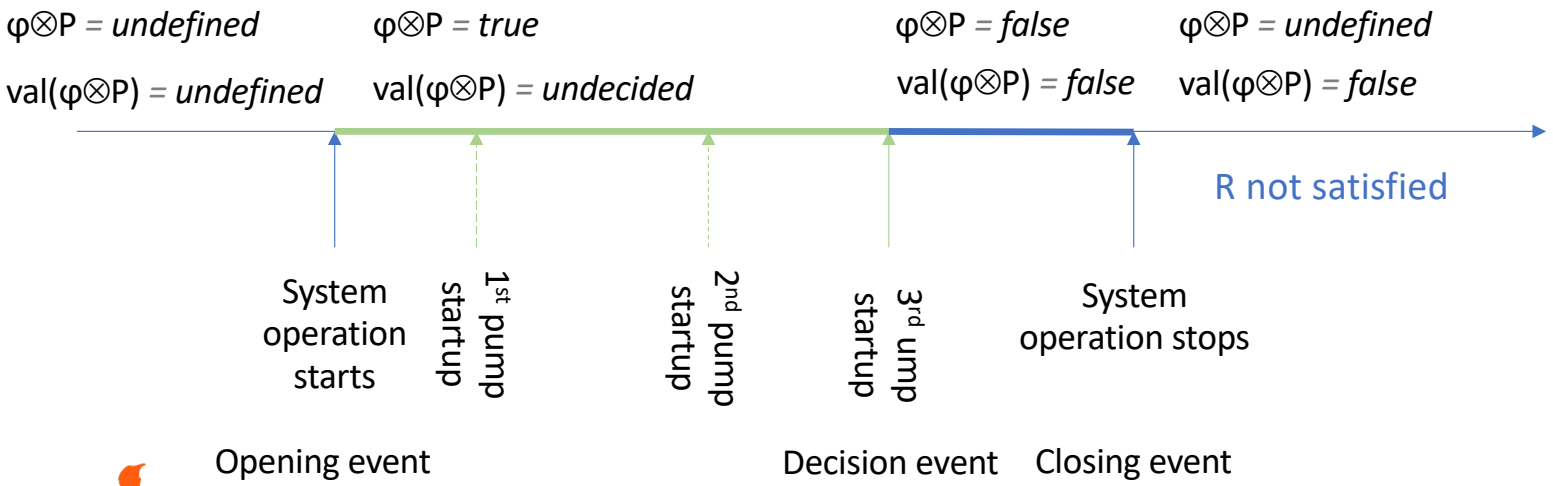
**Value of the requirement:** **val(φ⊗P)**

$$P \models \varphi \Leftrightarrow val(\varphi \otimes P) = true$$

$val(\varphi \otimes P) \neq \varphi \otimes P$

**Test case 1**

Condition φ satisfied at the end of time period P




LINKÖPING UNIVERSITY

EDF

# ETL: Requirement Evaluation by Example

**Time period:** `P = 'during' system.inOperation`

**Condition:** `ϕ = 'check count' (pump.isStarted 'becomes true') '<=' 2`

**Requirement:** `ϕ⊗P = 'during' system.inOperation 'check count'(pump.isStarted 'becomes true') '<=' 2`

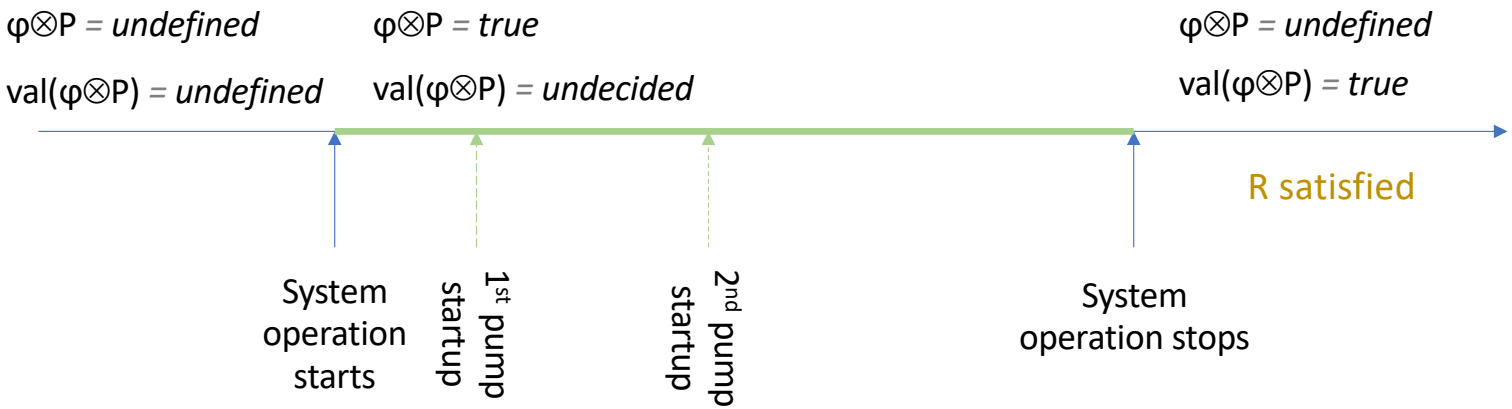
**Value of the requirement:** **val(φ⊗P)**

$$P \models \varphi \Leftrightarrow val(\varphi \otimes P) = true$$

$val(\varphi \otimes P) \neq \varphi \otimes P$

**Test case 2**

Condition φ satisfied at the end of time period P




LINKÖPING UNIVERSITY

eDF

# Relationship between 2- and 4-value Boolean algebras

- 2-value Boolean algebra tautology $\varphi \vee \neg\varphi = true$
  is not verified any more for 4-value Boolean algebra.
- $\varphi$ is never undefined if $\varphi$ is always inside a time period.
- $\varphi$ is never undecided if the end of the time period equals the start of the time period.
- Therefore, $\varphi$ is never undefined nor undecided if time periods are infinitely small and infinitely close together. This corresponds to limiting situations where decisions can be made instantaneously at any time instant.
- For Boolean algebras $val(\varphi \otimes P) = \varphi \otimes P$

# ETL: Temporal Operators

| × | Filter operator | | | |
|---|---|---|---|---|
| $a \times \varphi$ | *true* | *false* | *undecided* | *undefined* |
| *true* | *true* | *false* | *undecided* | *undefined* |
| *false* | *undecided* | *undecided* | *undecided* | *undefined* |
| *undecided* | *undecided* | *undecided* | *undecided* | *undefined* |
| *undefined* | *undefined* | *undefined* | *undefined* | *undefined* |

| + | Accumulate operator | | | |
|---|---|---|---|---|
| φ1 + φ2 | *true* | *false* | *undecided* | *undefined* |
| *true* | *true* | *false* | *true* | *true* |
| *false* | *false* | *false* | *false* | *false* |
| *undecided* | *true* | *false* | *undecided* | *undecided* |
| *undefined* | *true* | *false* | *undecided* | *undefined* |

φ1 and φ2 represent the values of the same variable at different instants in time


→ **Filters events to keep decision events**

$$a \times \varphi = \begin{cases} \textit{undecided} & \text{iff filter } a \text{ is on} \\ \varphi & \text{iff filter } a \text{ is off} \end{cases}$$

$$\varphi \otimes P := \int_{\varphi \in P} a(\varphi)\, d\varphi$$

→ **Defines the decision policy for requirement satisfaction**

When φ becomes false, R(φ) remains false even if φ becomes true later on.

*undefined* and *undecided* do not change the decision.

LINKÖPING UNIVERSITY

EDF

# ETL Library: Operators on Booleans, Clocks and Events

```
library ETL is {
    // Operators on Boolean
    // Logical disjunction
    Template b1 'or' b2 = not (not b1 and not b2);

    // Exclusive logical disjunction
    Template b1 'xor' b2 = (b1 'or' b2) and not (b1 and b2);

    // Logical inference
    Template b1 'implies' b2 = not b1 'or' b2;

    // Operators on clocks
    // Filter clock ticks inside a time period
    Operator [ Clock ] Clock C 'inside' Period P
        = C filter (tick >= P start) and (tick <= P end);

    // Count the occurrences of events inside a time period
    Operator [ Integer ] 'count' Clock C 'inside' Period P = card (C 'inside' P);

    // Operators on events
    // Events generated when a Boolean becomes true
    Operator [ Clock ] Boolean b 'becomes true' = Clock b;

    // Events generated when a Boolean becomes false
    Operator [ Clock] Boolean b 'becomes false' = not b 'becomes true';

    // Events generated when a Boolean becomes true inside a time period
    Operator [ Clock] Boolean b 'becomes true inside' Period P
        = (b 'becomes true') 'inside' P;

    // Events generated when a Boolean becomes false inside a time period
    Operator [ Clock] Boolean b 'becomes false inside' Period P
        = (b 'becomes false') 'inside' P;
```

# ETL Library: Temporal Operators to Evaluate Requirements

```
// Operators for the evaluation of requirements
// Check
Operator [ Boolean ] 'check' Boolean phi 'over' Periods P
    = and ('evaluate' phi 'over' P);

// Evaluate
Operator [ Boolean ] 'evaluate' Boolean phi 'over' Period P
    = integrate (('decide' phi 'over' P) * phi) on P;

// Decide
Operator 'decide' is
    Operator [ Boolean ] 'decide' Boolean phi 'over' Period P = phi 'or' (P end));

Category c1 is Category increasing1
    = { (>, >), (>=, >=), (<, >=), (<=, >), (==, >), (<>, >) };
Category {} C1 is associate increasing1 with 'decide';

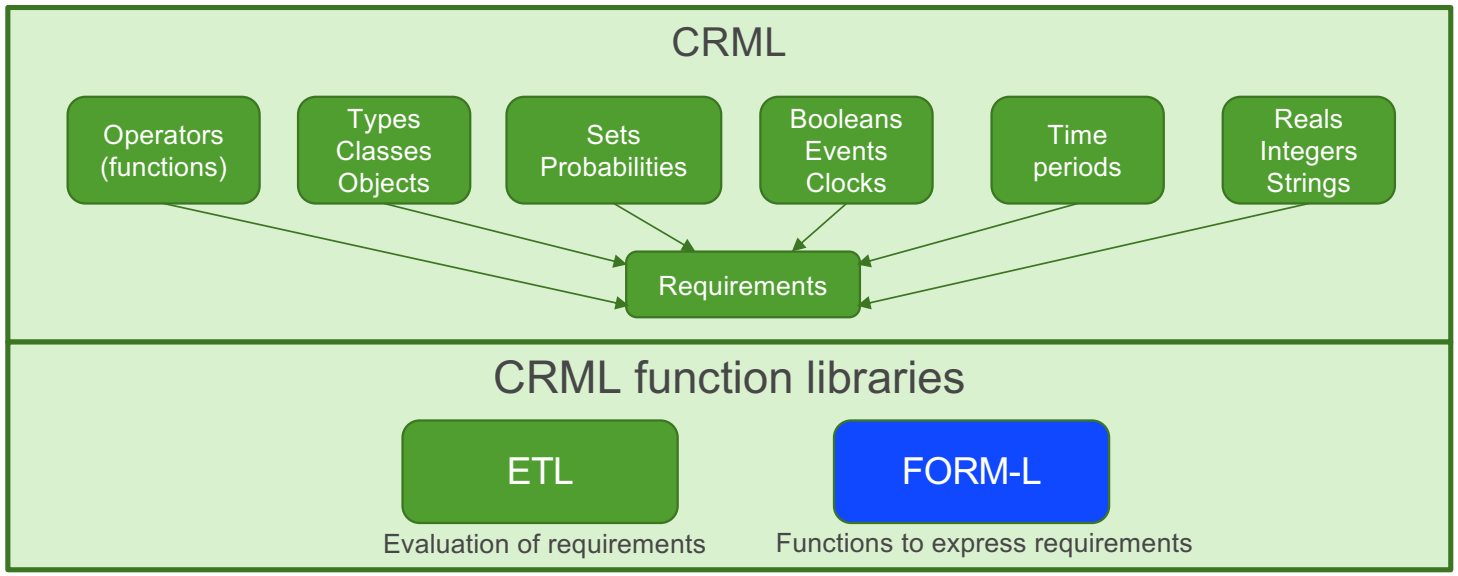
Category c2 is Category increasing2
    = { (>, >), (>=, >=), (<, >=), (<=, >) };
Category {} C2 is associate increasing2 with 'decide';

Operator [ Boolean ] 'id' Boolean b = b;
Operator [ Boolean ] 'cte_false' Boolean b = false;
Operator [ Boolean ] 'cte_true' Boolean b = true;

Category c3 is Category varying1 = { ('id', 'cte_false') };
Category {} C3 is associate varying1 with 'decide';

Category c4 is Category varying2 = { ('id', 'cte_true') };
Category {} C4 is associate varying2 with 'decide';
```

# Architecture of CRML




- **FORM-L** (FORMal Requirement Modelling Language) is a CRML library that implements the FORM-L language (T. Nguyen) to **express requirements** in more user-friendly way (i.e., closer to natural language)
- **ETL** (Extended Temporal Language) is a CRML library to **evaluate requirements** (i.e., whether they are satisfied or not)

# FORM-L Library: Time Periods

```
library FORM_L is ETL union {
    // Operators to define time periods
    // From events occur
    Operator [ Periods ] 'from' Clock ev = Periods [ ev, new Clock false ];
    // After events occur
    Operator [ Periods ] 'after' Clock ev = Periods ] ev, new Clock false ];
    // Before events occur
    Operator [ Periods ] 'before' Clock ev = Periods [ new Clock false, ev [;
    // Until events occur
    Operator [ Periods ] 'until' Clock ev = Periods [ new Clock false, ev ];
    // While a Boolean is true
    Operator [ Periods ] 'during' Boolean b = Periods [ Clock b, Clock not b ];
    // After events occur and before events occur
    Operator [ Periods ] 'after' Clock ev1 'before' Clock ev2 = Periods ] ev1, ev2 [;
    // After events occur and until events occur
    Operator [ Periods ] 'after' Clock ev1 'until' Clock ev2 = Periods ] ev1, ev2 ];
    // After events occur and for an elapsed time
    Operator [ Periods ] 'after' Clock ev 'for' Real d = Periods ] ev, d ];
    // After events occur and within an elapsed time
    Operator [ Periods ] 'after' Clock ev 'within' Real d = Periods ] ev, d [;
    // From events occur and before events occur
    Operator [ Periods ] 'from' Clock ev1 'before' Clock ev2 = Periods [ ev1, ev2 [;
    // From events occur and until events occur
    Operator [ Periods ] 'from' Clock ev1 'until' Clock ev2 = Periods [ ev1, ev2 ];
    // From events occur and for an elapsed time
    Operator [ Periods ] 'from' Clock ev 'for' Real d = Periods [ ev, d ];
    // From events occur and within an elapsed time
    Operator [ Periods ] 'from' Clock ev 'within' Real d = Periods [ ev, d [;
    // When events occur
    Operator [ Periods ] 'when' Clock ev = Periods [ ev, ev ];
```

# FORM-L Library: Operators to Evaluate Requirements

```
// Operators for the evaluation of requirements

// Checking that a requirement is satisfied at the end of a time period
Operator [ Boolean ] Periods P 'check at end' Boolean b = 'check' varying1 'id' b 'over' P;
// Checking that a requirement is satisfied at any time instant of a time period
Operator [ Boolean ] Periods P 'check anytime' Boolean b = 'check' varying2 'id' b 'over' P;
// Ensuring that a requirement is satisfied all along a time period
Operator [ Boolean ] Periods P 'ensure' Boolean b
    = ('check' ('count' (b 'becomes true') == 0) 'over' P) and (P 'check anytime' b);

// Checking that the number of event occurrences at the end of a time period
//  is lower or higher than a threshold
Operator [ Boolean ] Periods P 'check count' Clock E '<' constant Integer n
    = 'check'(('count' E 'inside' P) increasing1 '<' n) 'over' P;
Operator [ Boolean ] Periods P 'check count' Clock E '<=' constant Integer n
    = 'check'(('count' E 'inside' P) increasing1 '<=' n) 'over' P;
Operator [ Boolean ] Periods P 'check count' Clock E '>' constant Integer n
    = 'check'(('count' E 'inside' P) increasing1 '>' n) 'over' P;
Operator [ Boolean ] Periods P 'check count' Clock E '>=' constant Integer n
    = 'check'(('count' E 'inside' P) increasing1 '>=' n) 'over' P;
Operator [ Boolean ] Periods P 'check count' Clock E '==' constant Integer n
    = 'check'(('count' E 'inside' P) increasing1 '==' n) 'over' P;
Operator [ Boolean ] Periods P 'check count' Clock E '<>' constant Integer n
    = 'check'(('count' E 'inside' P) increasing1 '<>' n) 'over' P;

// Checking that the duration of a condition at the end of a time period
// is lower or higher than a threshold
Operator [ Boolean ] Periods P 'check duration' Boolean b '<' constant Real d
    = 'check' (('duration' b 'on' P) increasing2 '<' d) 'over' P;
Operator [ Boolean ] Periods P 'check duration' Boolean b '<=' constant Real d
    = 'check' (('duration' b 'on' P) increasing2 '<=' d) 'over' P;
Operator [ Boolean ] Periods P 'check duration' Boolean b '>' constant Real d
    = 'check' (('duration' b 'on' P) increasing2 '>' d) 'over' P;
Operator [ Boolean ] Periods P 'check duration' Boolean b '>=' constant Real d
    = 'check' (('duration' b 'on' P) increasing2 '>=' d) 'over' P;
```

# Increased Readability of FORM-L Requirements

- Natural expression

  *"When the system is in operation check the pump starts no more than 2 times and the pump temperature is below 50 °C"*

- FORM-L translation vs. ETL translation

```
// FORM-L translation
Boolean R_forml is
    'during' system.inOperation 'check count' (pump.isStarted 'becomes true') '<=' 2
        and 'during' pump.isStarted 'ensure' pump.temperature < 50*degC;

// ETL translation
Periods P1 is [system.inOperation 'becomes true', system.inOperation 'becomes false'];
Periods P2 is [pump.isStarted 'becomes true', pump.isStarted 'becomes false'];

Boolean R_etl is
    check (count (pump.isStarted 'becomes true' 'inside' P1) <= 2 over P1)
        and check ( count( pump.temperature >= 50*degC 'becomes true' 'inside' P2) == 0) over P2;
```

LINKÖPING UNIVERSITY

EDF

# Practical exercises

- How to formalize textual requirements?
  - Traffic light example
  - Pumping system example
  - Mission change case
  - Typical realistic requirement

- How to define new operator or template?
- How to define library?

→ Inspire yourself from ETL and FORM-L librairies

# Example: Traffic Light

- **List of informal requirements**
  - *"Req1: After green, next step is yellow"*
  - *"Req2: Step green should stay active for at least 30 seconds"*
  - *"Req3: After green becomes active + 30 seconds, next step should turn yellow within 0.2 seconds"*

- **Exercise**: Try to formalize requirements above in CRML
- **Advice**: Use ETL and FORM-L libraries

Notepad++ could be used for code editing
Installation of `crml.xml` enables syntax highlighting

# Example: Traffic Light

- A possible translation in CRML

```
model Spec is ETL union FORM_L union {
// Requirement model for the traffic light example

  // Definition of Requirement type
  Type Requirement is Boolean forbid { *, +, integrate };

  // List of external variables
  Boolean red is external;
  Boolean yellow is external;
  Boolean green is external;

  // Definition of requirements
  // req1: "After green, next step is yellow"
  Requirement req1 is
    'after' (green 'becomes true') 'before' (yellow 'becomes true')
    'check count' (red 'becomes true') '==' 0;

  // req2: "Step green should stay active for at least 30 seconds"
  Requirement req2 is
    'after' (green 'becomes true') 'for' 30
    'ensure' green;

  // req3: "After green becomes active + 30 seconds,
  //        next step should turn yellow within 0.2 seconds"
  Requirement req3 is
    'after' (green 'becomes true' + 30) 'for' 0.2
    'check' yellow;

};
```

# Example: Pumping System

Extract from IEEE SysCon'18 article
(doi: 10.1109/SYSCON.2018.8369502)

- **List of informal requirements**

R1: While the system is in operation, a pump must not be started more than twice.

R2: At least one hour must separate two consecutive pump startups.

R3: While the pump is in operation (i.e. started), its temperature must always stay below 50°C.

R4: While the system is in operation, after the pump temperature rises above 40 °C, the temperature must not stay above this value for a duration of more than 1 mn cumulated over the next 15 mn.

- **Exercise**: Try to formalize requirements above in CRML
- **Advice**: Use ETL and FORM-L libraries

Notepad++ could be used for code editing
Installation of `crml.xml` enables syntax highlighting

# Example: Pumping System

Extract from IEEE SysCon'18 article
(doi: 10.1109/SYSCON.2018.8369502)

R1: While the system is in operation, a pump must not be started more than twice.

```
Requirement R1 is
    'during' system.inOperation 'check count' (pump.isStarted 'becomes true') '<=' 2;
```

R2: At least one hour must separate two consecutive pump startups.

```
Requirement R2 is
    'after' pump.isStarted 'for' 1*h 'check count' (pump.isStarted 'becomes true') '==' 0;
```

R3: While the pump is in operation (i.e. started), its temperature must always stay below 50°C.

```
Requirement R3 is
    'during' pump.isStarted 'ensure' pump.temperature < 50*degC;
```

R4: While the system is in operation, after the pump temperature rises above 40 °C, the temperature must not stay above this value for a duration of more than 1 mn cumulated over the next 15 mn.

```
Requirement R4 is
    'during' system.inOperation 'after' pump.temperature > 40*degC 'for' 15*mn
        'check duration' (pump.temperature > 40*degC) '<' 1*mn;
```

# Example: Mission Change

- **Natural expression (with a logical combination of requirements)**

  *“Requirement R:*
  *Requirement R2 should be satisfied at most n seconds after requirement R1 is violated”*

- **Exercise**: Try to formalize the expressions above in CRML
- **Advice**: Use ETL and FORM-L libraries

Notepad++ could be used for code editing
Installation of `crml.xml` enables syntax highlighting

# Example: Mission Change

- **Natural expression**

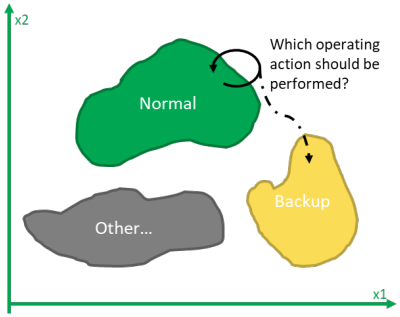
  *"Requirement R:*
  *Requirement R2 should be satisfied at most n seconds after requirement R1 is violated"*

- **Mathematical formulation**
  - R = (¬R1 ⇒ R2⊗[R1 *becomes false*, R1 *becomes false* + *n* seconds])⊗P
  - R = (¬R1 ⇒ R2⊗[R1↓, R1↓ + *n* s])⊗P
- **A possible formulation in CRML**

```
Requirement R1 is external;
Requirement R2 is external;
Real n is external;

//Requirement R2 should be satisfied at most n seconds after requirement R1 is violated
Boolean systemInOperation is external;
Requirement R is
    'during' systemInOperation
        'ensure' ((not R1) 'implies' R2 'becomes true inside' [R1 'becomes false'; R1 'becomes false' + n seconds])
```

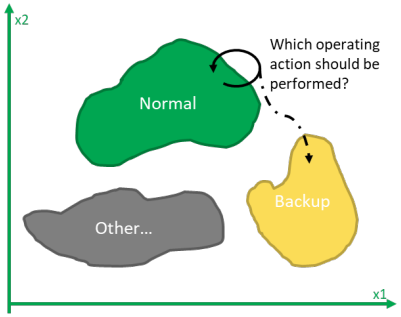
# Example: Typical Requirement




- **Natural expressions**
  1. *The system should stay within its normal operating domain.*
  2. *If partial requirement 1 above fails, then the system should go back to its normal operating domain within a given time delay.*
  3. *If partial requirement 2 above fails, or if partial requirement 1 fails with a too high failure rate, then the system should go to a safe backup state within a given time delay.*
  4. *The complete requirement made of the conjunction of partial requirements 1, 2 and 3 should be satisfied with a given probability (e.g., > 99.99%).*
- **Exercise**: Try to formalize the expressions above in CRML
- **Advice**: Use ETL and FORM-L libraries

Notepad++ could be used for code editing
Installation of `crml.xml` enables syntax highlighting

# Example: Typical Requirement




- A possible solution in CRML

```
class TypicalRequirement is ETL union FORM_L union {

    type Requirement is Boolean forbid { *, +, integrate };

    Boolean inOperation is external;
    Boolean inNormalDomain is external;
    Boolean inBackupDomain is external;
    Boolean inSystemOperatingLife is external;

    Real x, y is external;
    Integer n is external;

    Real p is 0.99;

    // r1 is "During operation, the system should stay within its normal domain."
    Requirement r1 is 'during' inOperation 'ensure' inNormalDomain;

    // r2 is "If the system fails to stay within its operating domain, then it should not stay outside of its normal domain for more than x minutes."
    Requirement r2 is 'during' inOperation 'ensure' (not inNormalDomain 'implies' r2_outside);
    Requirement r2_outside is 'during' [inNormalDomain 'becomes false', inNormalDomain 'becomes false' + x mn] 'check at end' b;
    Boolean b is inOperation 'implies' inNormalDomain;

    // r3 is "The system should not go outside its normal domain more than n times per year."
    Requirement r3 is 'count' ((b 'becomes false') on [b 'becomes false', b 'becomes false' + 1 year] <= n;

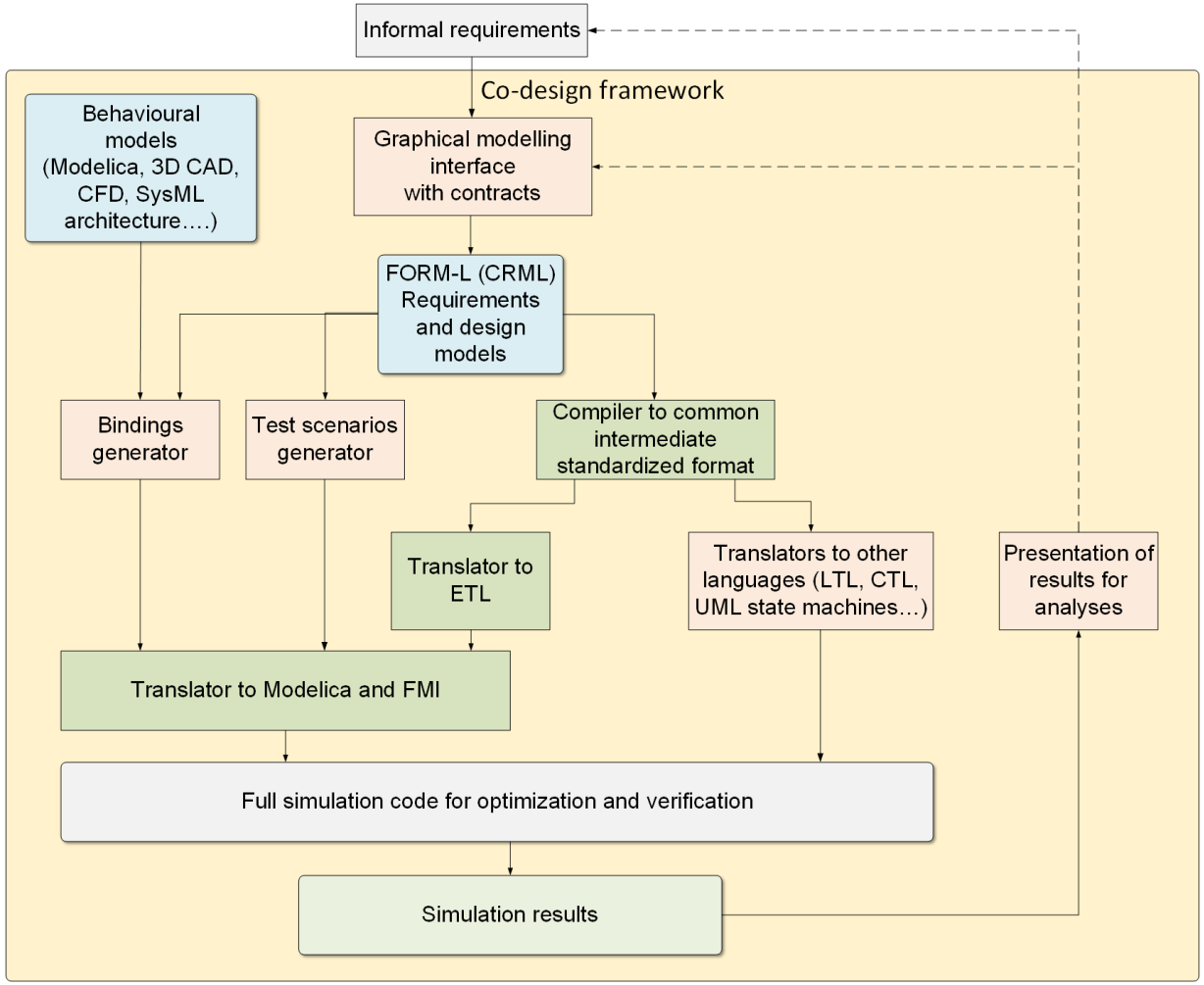
    // r4 is "If (r1 and r2 and r3) fail, then the system should go to its backup domain within y minutes as soon as the failure is detected."
    Requirement r4 is not (r1 and r2 and r3) 'implies'
                        'during' [(r1 and r2 and r3) 'becomes false', (r1 and r2 and r3) 'becomes false' + y mn] 'check at end' inBackupDomain;

    // R is "During system operating life, r1 and r2 and r3 and r4 should be satisfied with a probability of success of p%."
    Real prob is estimator Probability (r1 and r2 and r3 and r4) at inSystemOperatingLife 'becomes false';
    Requirement R is 'during' inSystemOperatingLife 'check at end prob' > p;
};
```

# CRML toolchain

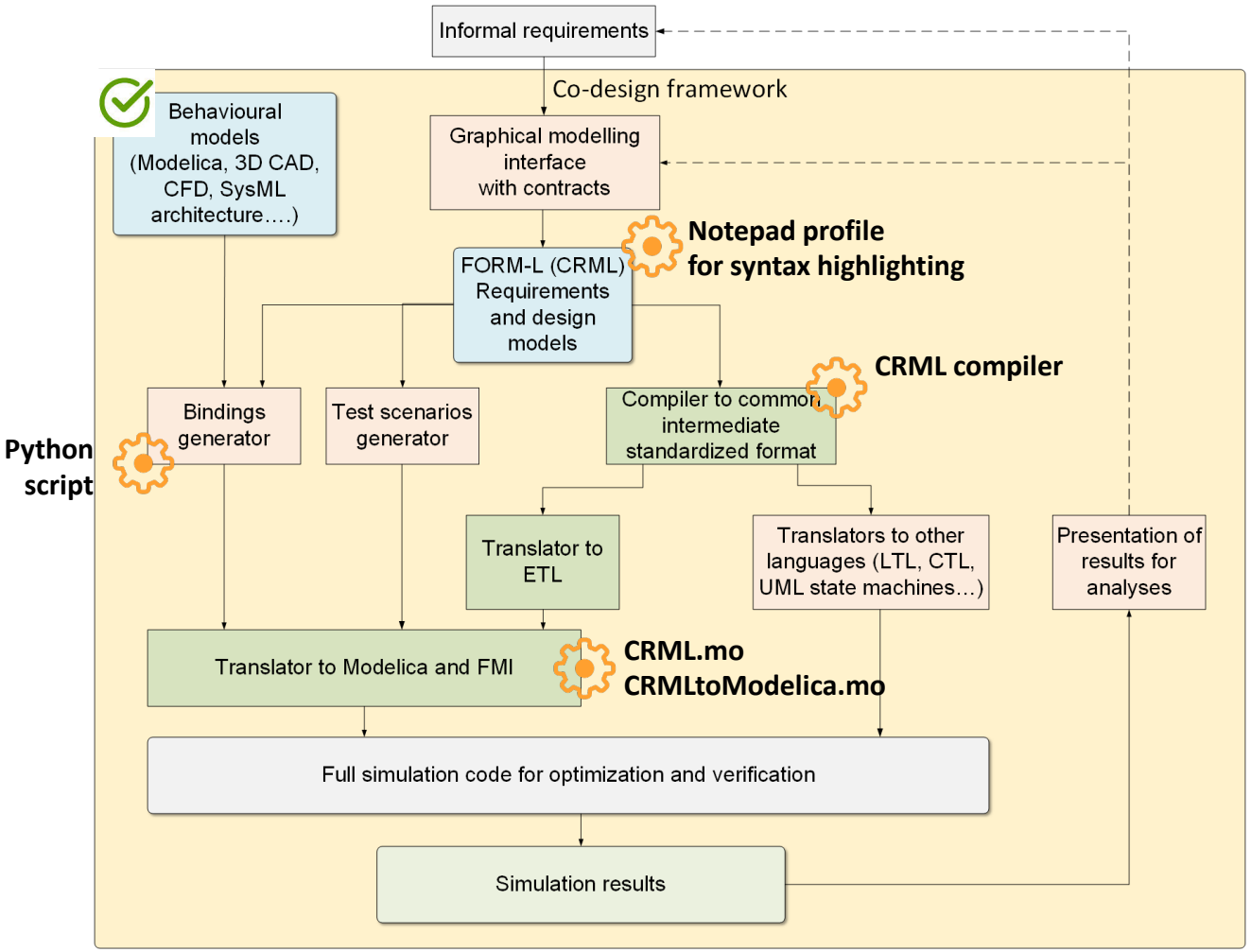
# Outlook of a CRML framework

*CRML Framework is integrated with Modelica*



# Outlook of a CRML framework

*CRML Framework is integrated with Modelica*

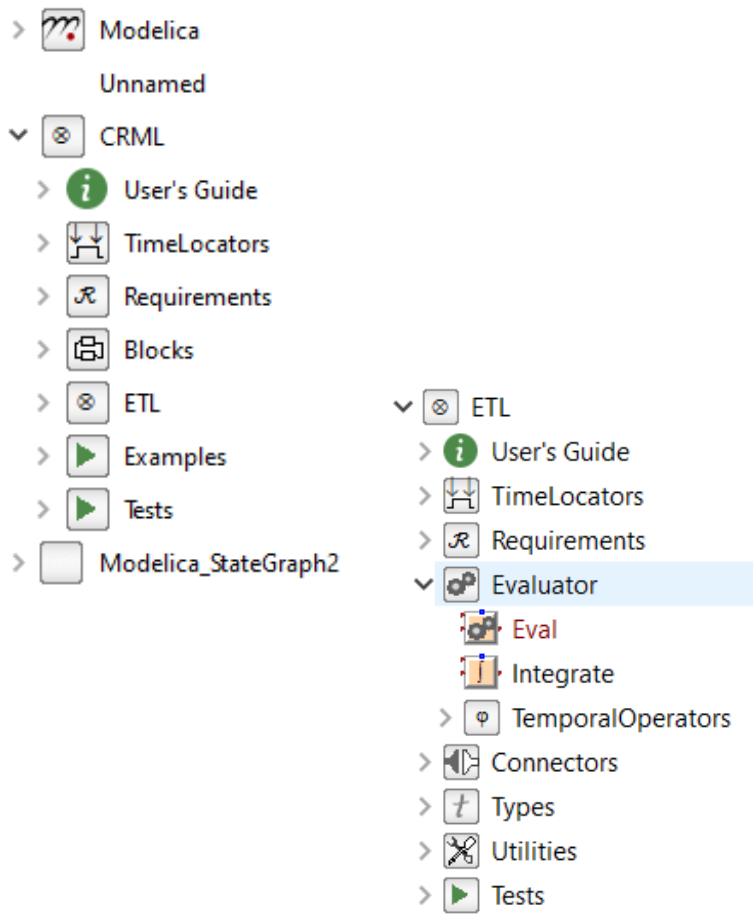



Tools available

Prototypes available

# CRML Modelica Library

- Library developed to verify the correctness of ETL semantics for evaluating requirements
- The library contains blocks to express time locators, conditions and probabilities
- Requirements are built by connecting the blocks together in the form of block diagrams



# Using the CRML Modelica Library

- Let's take again the pumping system example

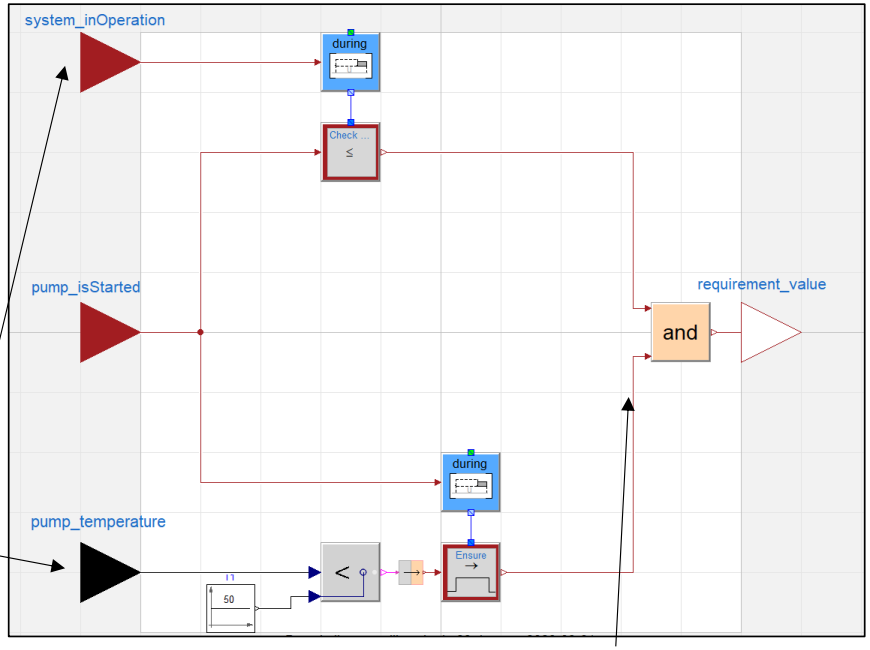
A requirement is built by connecting a time locator block to a condition block

R1: While the system is in operation, a pump must not be started more than twice.

```
Requirement R1 is
    'during' system.inOperation 'check count' (pump.isStarted 'becomes true') '<=' 2;
```

R3: While the pump is in operation (i.e. started), its temperature must always stay below 50°C.

```
Requirement R3 is
    'during' pump.isStarted 'ensure' pump.temperature < 50*degC;
```



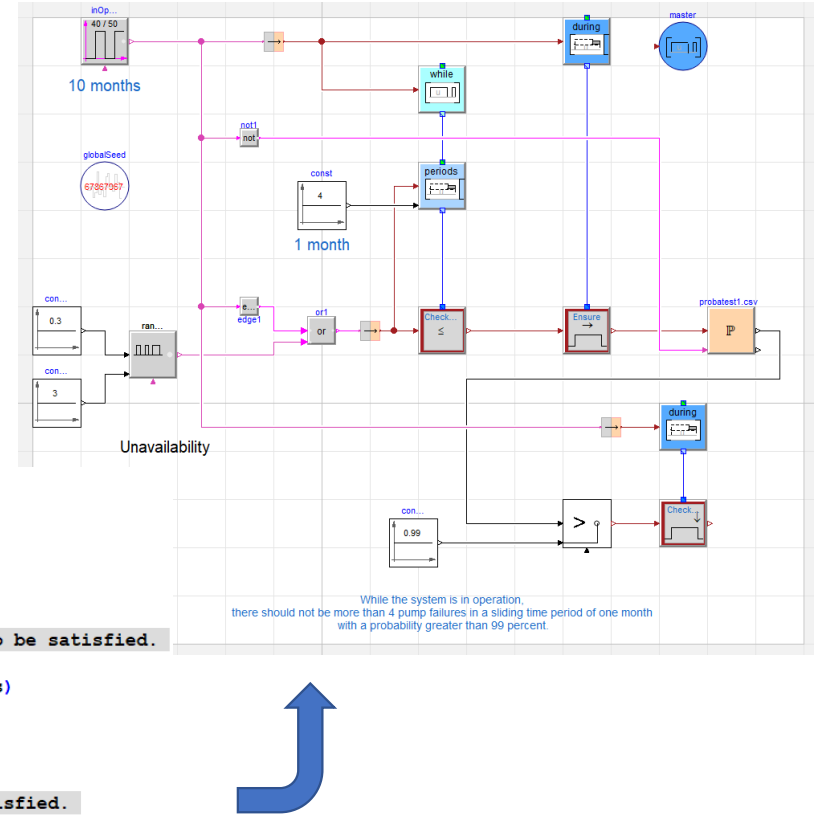

External variables are provided by behavioral model

Requirements can be connected together

# Using the CRML Modelica Library

- A more complicated example with probabilistic requirement
- One can easily observe that expressing requirements as block diagrams is not convenient!

```
class Pump is {
    // External variable that tells whether the system is in operation
    Boolean inOperation is external;
    // External variable that tells whether the pump is in a failure state
    Boolean failure is external;
    Events failures is new Clock failure;
    // No-start non-probabilistic requirement.
    // It is not declared as a requirement because it is not the final requirement to be satisfied.
    Boolean nostart is ('during' inOperation 'excluding closing events')
        'ensure' (('during' 1 month 'sliding while' inOperation 'for' Events failures)
            'check count' Events failures '<=' 2);
    // Probability that requirement no-start is true at end of the system operation
    Real p is estimator Probability noStart at inOperation 'becomes false';
    // Probabilistic no-start requirement.
    // It is declared as a requirement because it is the final requirement to be satisfied.
    Requirement noStartProb is 'during' inOperation 'check at end' p > 0.99;
};
```



# CRML compiler

*From requirement modelling to Modelica code*

- Developed by Linköping University within ITEA3 EMBrACE project
- Takes CRML models as input and produces Modelica models as output (based on the CRMLtoModelica.mo library).
- Works only on a subset of CRML (development still ongoing)
- If you are using a VM, the compiler repository is already cloned in:

  `/data/crml-compiler`

- If you are running the compiler on your own machine, you can clone it from the github repository:

  `https://github.com/lenaRB/crml-compiler`

# Running the compiler

- The tutorial examples and a copy of these slides can be found in
  `resources/crml_tutorial`

- You will also find the Notepad ++ profile with syntax highlighting for CRML in
  `resources/notepad_profile`

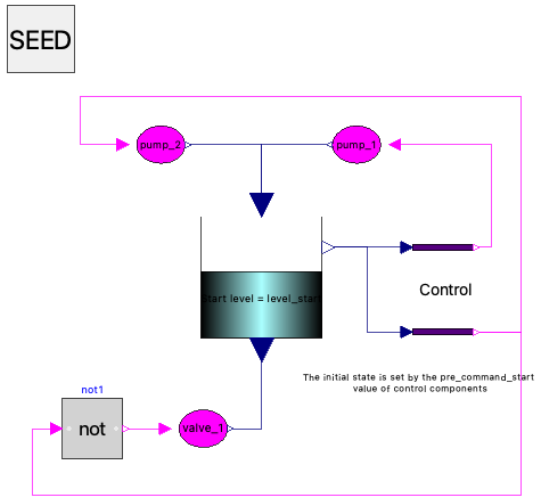
- To run the compiler go to the compiler directory, you can use the makefile and run:
  `make translate DIR=<file or directory>`

  Where you provide the path to either a single .crml file you want to translate or to a directory containing multiple .crml files

  For example: `make translate DIR=resources/crml_tutorial/BooleanNegation.crml`

  The .mo files are generated in the `generated` directory in the root

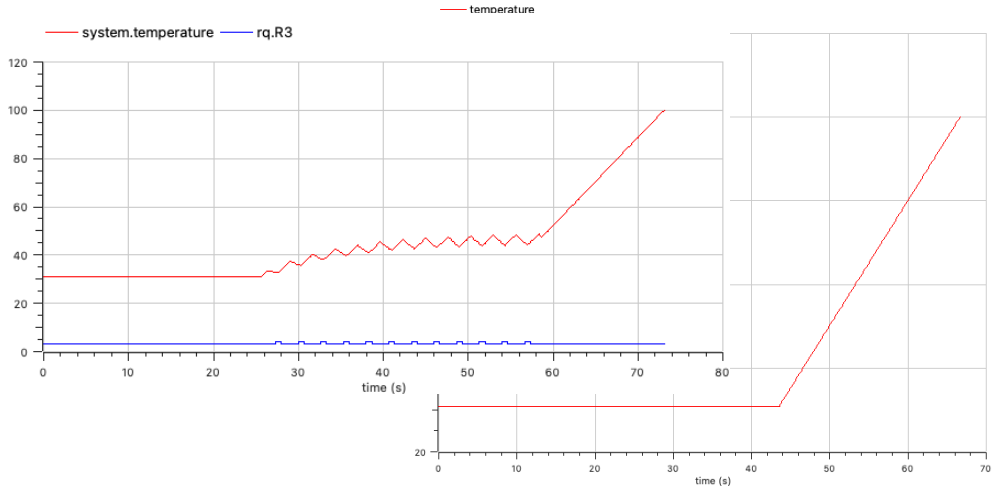
# Running a CRML compiler example




- Using Notepad++ you can open the crml requirement model
- Simulate with

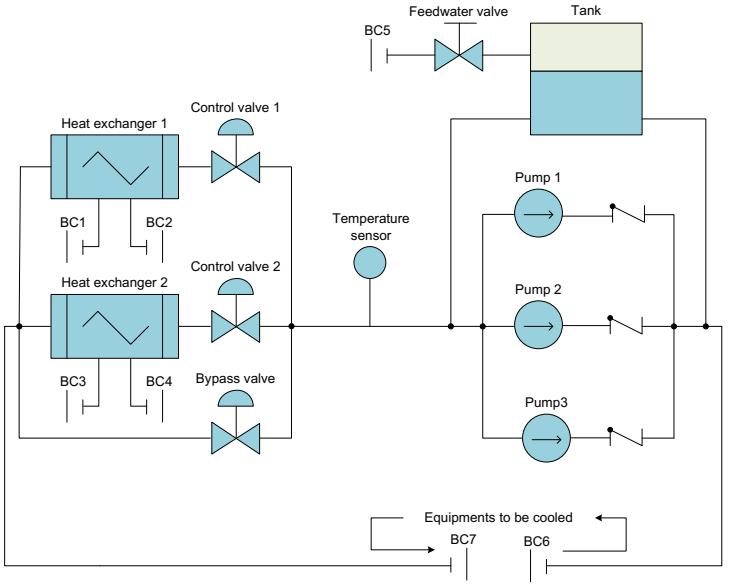
```
make translate DIR=<file path>
```

- Load the models and library in `heated_tank`



# Bindings: Purpose

- The purpose of bindings is to solve the general problem of evaluating at simulation time the formal requirement expressions that contain quantifiers on **external sets** and conditions on **external variables**.
- Example:
  - ```
    Requirement R2 is 'for all' p 'in' pumps 'such that' p.inOperation 'check' not p.cavitate;
    ```
  - External sets:
    - Set pumps of objects Pump
  - External variables:
    - Variable p.inOperation defined in object Pump
    - Variable p.cavitate defined in object Pump

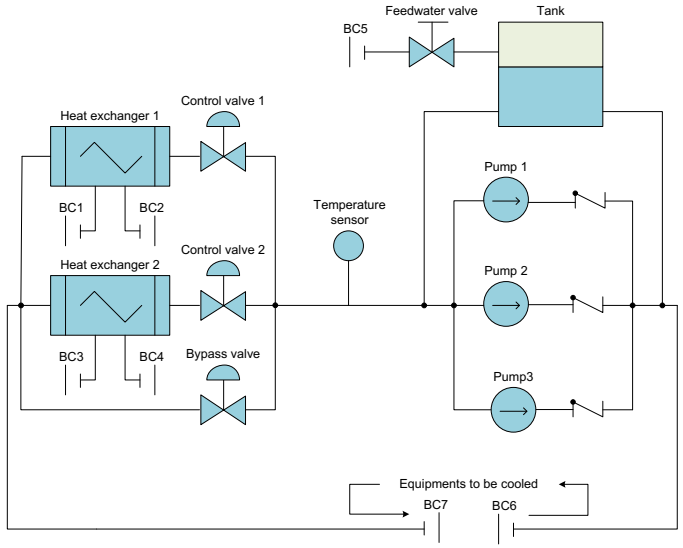



LINKÖPING UNIVERSITY

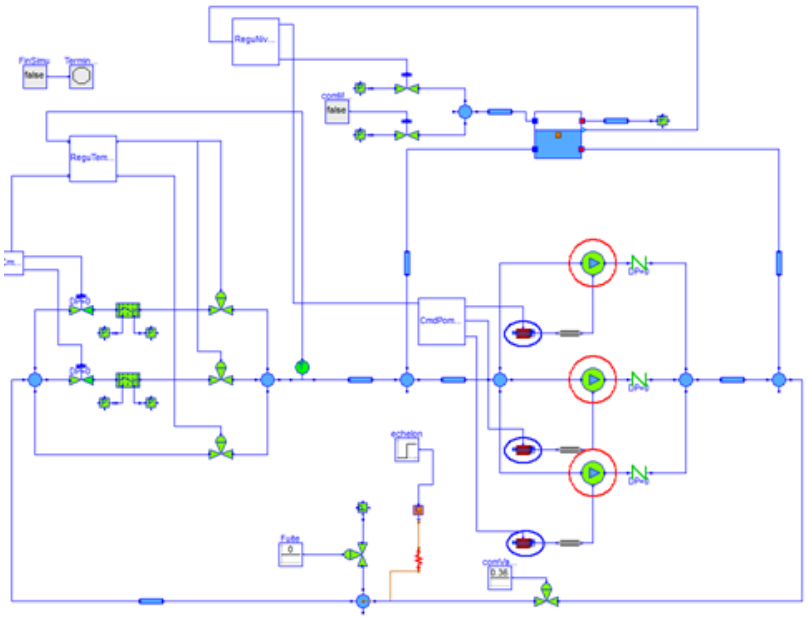
EDF
    ```

# Bindings: Question to be Solved?

- Where to find the information to feed the external variables?
- How to do this as automatically as possible?




Architectural model

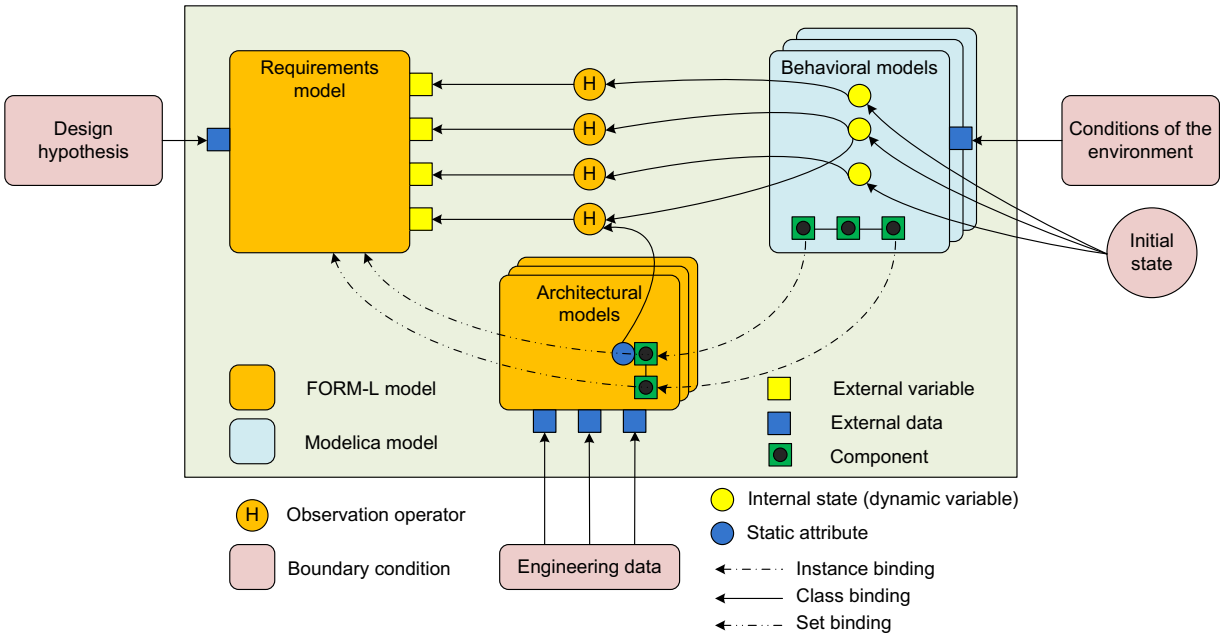


Behavioral model

From the functional side → to the physics: “Converters” are needed!

# Bindings: Main Concepts

- **External variable**: variable used in the requirement model whose value is elaborated by observation operators.
- **Observation operator**: function that take as inputs variables defined in other models.
- Static variable: variable elaborated in an **architectural model**.
- Dynamic variable: variable elaborated in a **behavioral model**.




LINKÖPING UNIVERSITY

EDF

# Bindings: Formal Problem

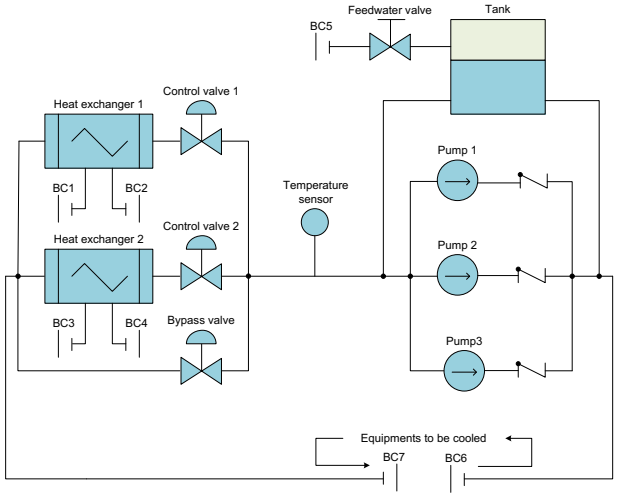
- How to construct as automatically as possible expressions such as
  - $p.y = H_y(e_1.x, \ldots, e_n.x)$ (ex: p.cavitate)
- Where:
  - y is the external variable define in object p of an external set S
  - $H_y$ is the observation operator that computes y
  - $e_1.x, \ldots, e_n.x$ are the variables defined in objects $e_1, \ldots, e_n$ located in different models $M_1, \ldots, M_n$ ?
- Solution
  - **Set bindings** provide the elements p to the external sets S.
  - **Class** and **instance bindings** provide the values for external variable y.
  - Define as much as possible binding at the class level, and go to the instance level only when class binding is not possible:
    - Binding at the class level is called *class binding*.
    - Binding at the instance level is called *instance binding*.
    - Class bindings are stored in libraries and reused across test scenarios to build bindings.
    - Instance bindings complement class bindings for bindings that cannot be done at the class level and must be specified for each test scenario.

# Set Bindings

- S is a set that is involved in a generic requirement (ex: all pumps in S must not cavitate).
- For a test given scenario, S must be populated with elements. This is done by providing a binding to S: the binding tells which are the elements of S for that particular test scenario.
- The binding to S is denoted S ← { $p_1$, $p_2$, ….$p_n$ }. $p_1$, $p_2$, ….$p_n$ are the objects from architectural or requirement models $M_1$, $M_2$, ….$M_n$ assigned to S by the binding.
- { $p_1$, $p_2$, ….$p_n$ } is called the target of S for that binding: target(S ← { $p_1$, $p_2$, ….$p_n$ }) = { $p_1$, $p_2$, ….$p_n$ }.
- S may have several bindings, but only one is active for a given scenario.

```
Requirement R2 is
 'for all' p 'in' pumps 'such that' p.inOperation
 'check' not p.cavitate;
```

pumps ← { Pump1, Pump2, Pump3 }

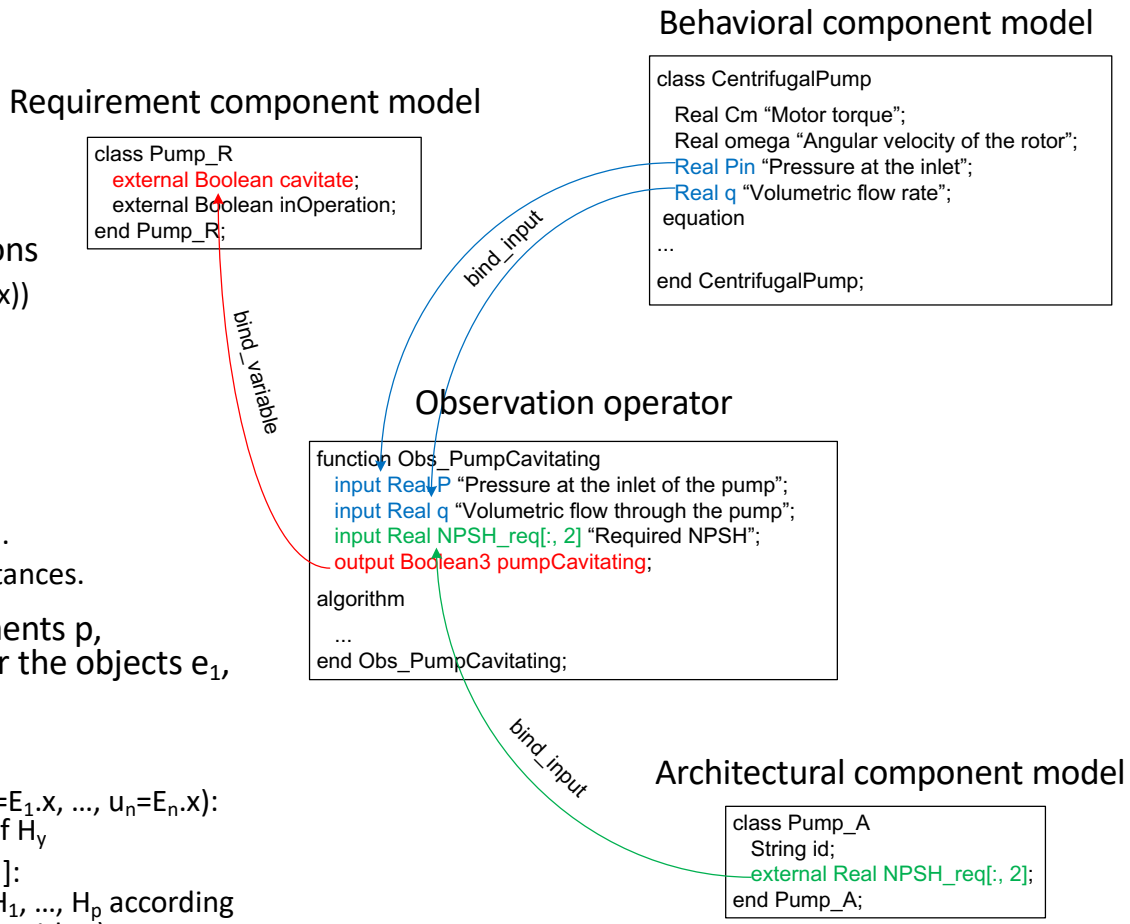



LINKÖPING UNIVERSITY

EDF

# Class Bindings

- The purpose of class binding is to form expressions
  - P.y = $H_y$ ($E_1$.x, …, $E_n$.x) (p.y = $H_y$ ($e_1$.x, …, $e_n$.x))
- Where:
  - P is the class of p
  - $E_1$, …, $E_n$ are the classes of $e_1$, …, $e_n$
- To compute p.y = $H_y$ ($e_1$.x, …, $e_n$.x):
  - The **right** observation operator $H_y$ must be found.
  - The classes must be replaced by their proper instances.
- **Set bindings** perform this operation for set elements p, and **instance bindings** perform this operation for the objects $e_1$, …, $e_n$
- **Class binding** is performed in 2 steps:
  - **Input binding** of $H_y$, denoted $H_y$ ($u_1$, …, $u_n$) ← ($u_1$=$E_1$.x, …, $u_n$=$E_n$.x): $E_1$.x, …, $E_n$.x are declared as the inputs $u_1$, …, $u_n$ of $H_y$
  - **Variable binding** of $H_y$, denoted P.y ← [ $H_1$, …, $H_p$ ]: $H_y$ will be chosen among observation operators $H_1$, …, $H_p$ according to a matching algorithm (step 1 of the binding algorithm).




LINKÖPING UNIVERSITY

EDF

# Class Bindings: Example of Multiple Observers

- Example of multiple observation operators $H_1$, …, $H_p$ for a given external variable y.
  - Let's consider the requirement model uses the functional variable y = p.inOperation that tells whether a pump p is started or not.
  - But the notion of a started pump is not present in the behavioral model.
- The behavioral model contains variables that represent the physical state of the pumps:
  - $P_1$: pressure at the inlet of the pump casing
  - $P_2$: pressure at the outlet of the pump casing
  - Q: mass flow rate through the pump casing
  - T: torque provided to the shaft of the pump
  - V: voltage provided to the motor of the pump b
- The observation for p.inOperation can be:
  - $H_1$: $P_2 - P_1 >$ threshold
  - $H_2$: $Q >$ threshold
  - $H_3$: $T_m >$ threshold. Can be chosen if the shaft of the pump is represented in the model.
  - $H_4$: $V >$ threshold. Can be chosen if the motor of the pump is represented in the model.

LINKÖPING UNIVERSITY eDF

# Instance Bindings

- The purpose of **instance binding** is to form the final expressions

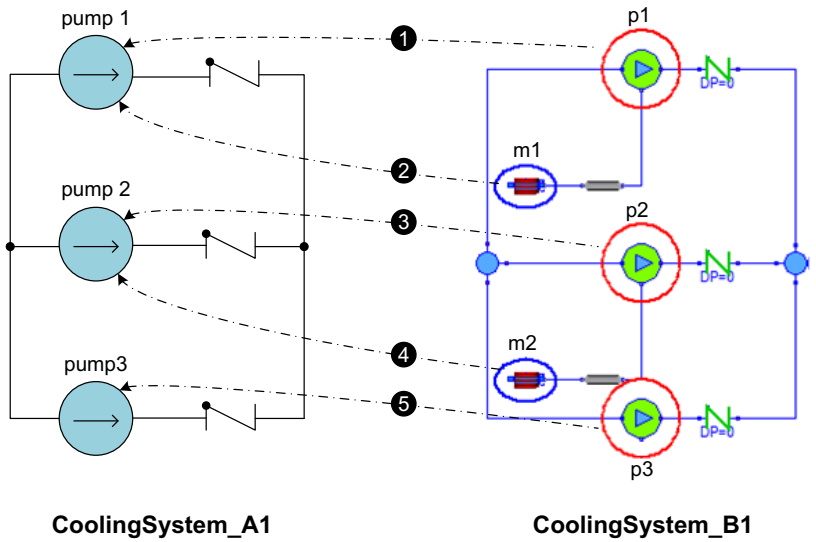
  $p.y = H_y(e_1.x, \ldots, e_n.x)$

- from the expressions $P.y = H_y(E_1.x, \ldots, E_n.x)$

  where p is the proper instance of P and $e_1, \ldots, e_n$ are the proper instances of $E_1, \ldots, E_n$.

- Instance binding of object a is denoted $a \leftarrow \{ e_1, \ldots, e_n \}$: instances $e_1, \ldots, e_n$ are bound to instance a.

- Object a may have several declared bindings but may have at most one active binding.

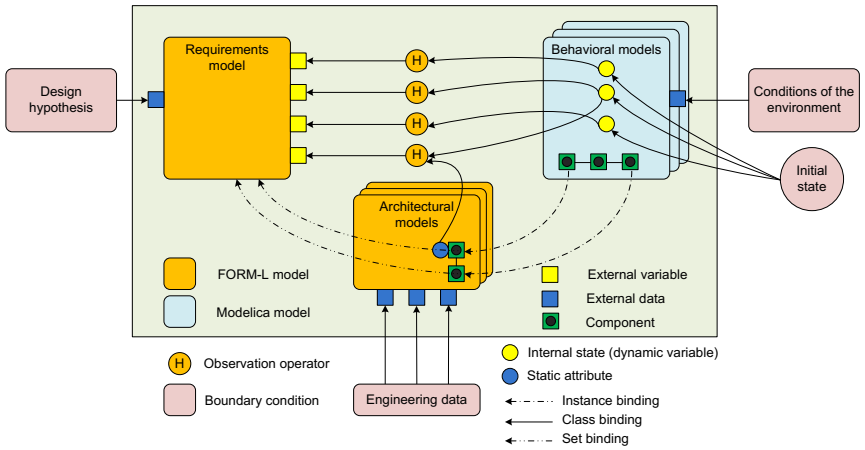



pump1 ← { $p_1$, $m_1$ }
pump2 ← { $p_2$, $m_2$ }
pump3 ← { $p_3$ }

LINKÖPING UNIVERSITY

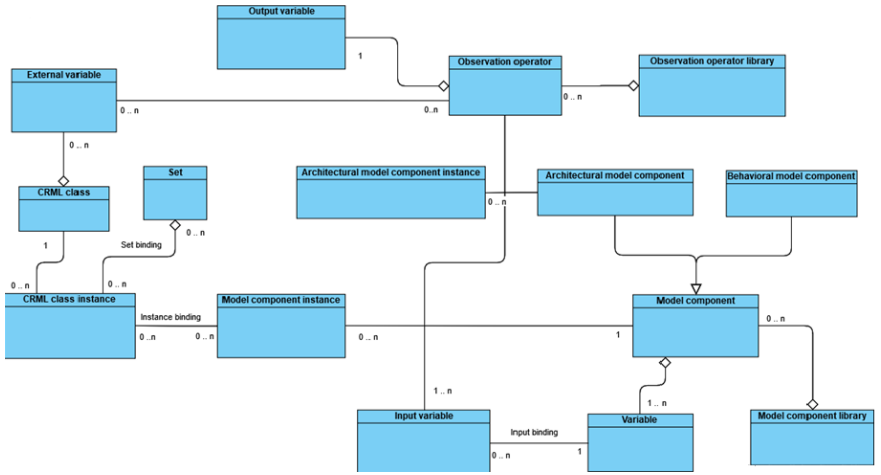
EDF

# Bindings: Current Implementation




- Bindings equations can be generated with a Python code which:
  - Uses Excel files to describe the bindings:
    - Set bindings, Class bindings
    - Instance bindings
  - Produces automatically the Modelica verification model.

- Matching algorithm for choosing observation operators has been published in deliverable of ITEA project
- Class diagram has been developed



# Running Python Script to Automate Bindings

```
Requirement R2 is 'for all' p 'in' pumps 'such that' p.inOperation 'check' not p.cavitate;
```

**Step 1:**
Automatic choice of observator Hi for each pump i

+ call to each Hi with appropriate inputs

```
model ref_binding_model

 SRIN4_v3.SRI_v3_init Behavior_B1;

 Pump_A Architecture_A1;

 ObsSysPro.PumpInOperation1 pump1_inOperation;
 ObsSysPro.PumpInOperation1 pump2_inOperation;
 ObsSysPro.PumpInOperation2 pump3_inOperation;
 ObsSysPro.PumpCavitating pump1_cavitation;
 ObsSysPro.PumpCavitating pump2_cavitation;
 ObsSysPro.PumpCavitating pump3_cavitation;

equation
 pump1_inOperation.V = Behavior_B1.Moteur1.Um;
 pump2_inOperation.V = Behavior_B1.Moteur2.Um;
 pump3_inOperation.C = Behavior_B1.PompeCentrifugeDyn3.Cm;
 pump1_cavitation.He = Behavior_B1.PompeCentrifugeDyn1.hn;
 pump1_cavitation.T = Behavior_B1.PompeCentrifugeDyn1.pro.T;
 pump1_cavitation.d = Behavior_B1.PompeCentrifugeDyn1.rho;
 pump1_cavitation.NPSHreq = Architecture_A1.pump1.NPSHr;
 pump2_cavitation.He = Behavior_B1.PompeCentrifugeDyn2.hn;
 pump2_cavitation.T = Behavior_B1.PompeCentrifugeDyn2.pro.T;
 pump2_cavitation.d = Behavior_B1.PompeCentrifugeDyn2.rho;
 pump2_cavitation.NPSHreq = Architecture_A1.pump2.NPSHr;
 pump3_cavitation.He = Behavior_B1.PompeCentrifugeDyn3.hn;
 pump3_cavitation.T = Behavior_B1.PompeCentrifugeDyn3.pro.T;
 pump3_cavitation.d = Behavior_B1.PompeCentrifugeDyn3.rho;
 pump3_cavitation.NPSHreq = Architecture_A1.pump3.NPSHr;

end ref_binding_model;
```

```
model ref_verification_model

 extends Pump_R;
 ref_binding_model BindingModel;


equation
 pump1.inOperation = BindingModel.pump1_inOperation.y;
 pump2.inOperation = BindingModel.pump2_inOperation.y;
 pump3.inOperation = BindingModel.pump3_inOperation.y;
 pump1.cavitation = BindingModel.pump1_cavitation.y;
 pump2.cavitation = BindingModel.pump2_cavitation.y;
 pump3.cavitation = BindingModel.pump3_cavitation.y;

end ref_verification_model;
```

**Step 2:**
Connection of observator outputs to external variables of the requirement model

# Practical exercises

- With CRML Modelica library
  - Traffic light example
  - Pumping system example (part)

- With CRML compiler
  - Tank with failure

LINKÖPING UNIVERSITY

EDF

# Example: Traffic Light

- **List of informal requirements**
  - *"Req1: After green, next step is yellow"*
  - *"Req2: Step green should stay active for at least 30 seconds"*
  - *"Req3: After green becomes active + 30 seconds, next step should turn yellow within 0.2 seconds"*

- **Exercise**: Try to translate CRML requirements as Modelica block diagrams
- **Advice**: Don't forget to declare inputs

- **Formulation in CRML**

```
model Spec is ETL union FORM_L union {
// Requirement model for the traffic light example

  // Definition of Requirement type
  Type Requirement is Boolean forbid { *, +, integrate };

  // List of external variables
  Boolean red is external;
  Boolean yellow is external;
  Boolean green is external;

  // Definition of requirements
  // req1: "After green, next step is yellow"
  Requirement req1 is
    'after' (green 'becomes true') 'before' (yellow 'becomes true')
    'check count' (red 'becomes true') '==' 0;

  // req2: "Step green should stay active for at least 30 seconds"
  Requirement req2 is
    'after' (green 'becomes true') 'for' 30
    'ensure' green;

  // req3: "After green becomes active + 30 seconds,
  //        next step should turn yellow within 0.2 seconds"
  Requirement req3 is
    'after' (green 'becomes true' + 30) 'for' 0.2
    'check' yellow;

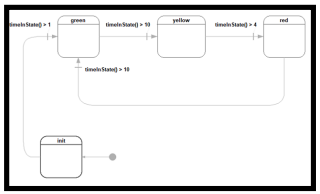
};
```

Library CRML.mo could be used

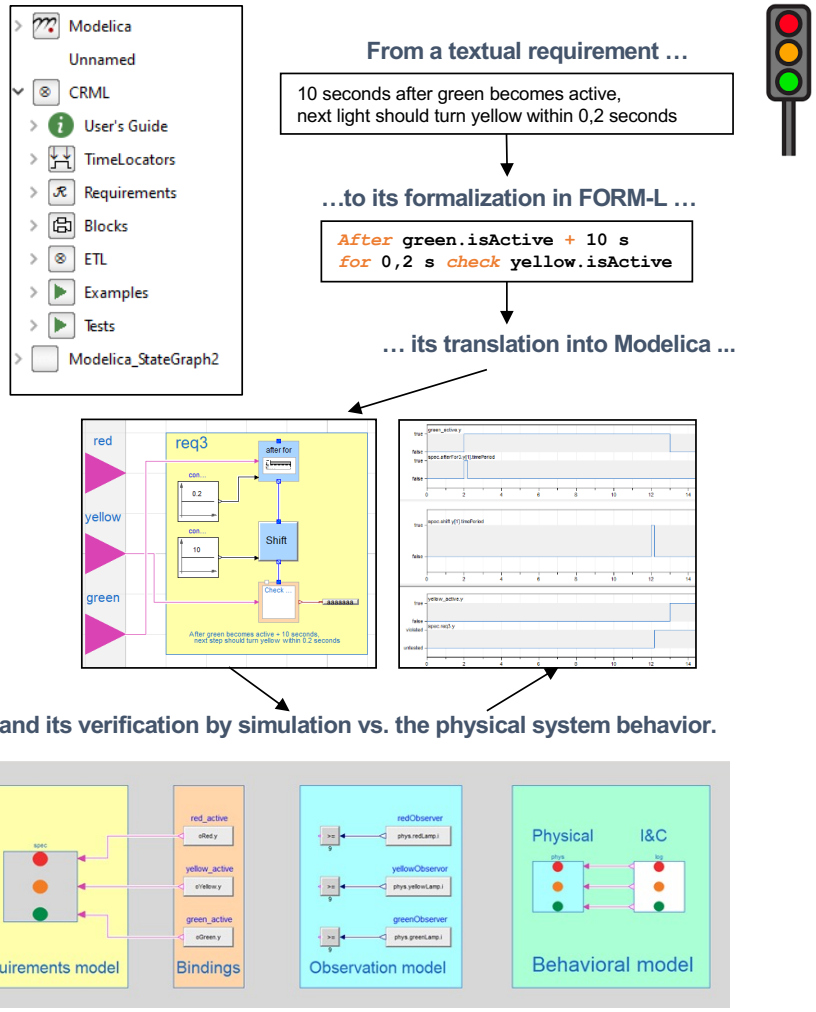
# Example: Traffic Light

- **Exercise**: Try to connect the previous requirement model with some inputs and analyze the results
- **Advice**: Test with different implementation of the traffic light control part

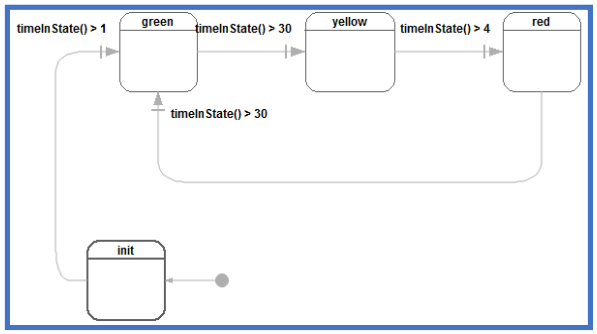
Model of control part



Some behavioral models (physics+control) can be found in the “example” package of CRML.mo



# Example: Traffic Light




Behavioral model of control part

Provides inputs

Requirement model

```
// req3: "After green becomes active + 30 seconds,
//        next step should turn yellow within 0.2 seconds"
Requirement req3 is
  'after' (green 'becomes true' + 30) 'for' 0.2
  'check' yellow;
```

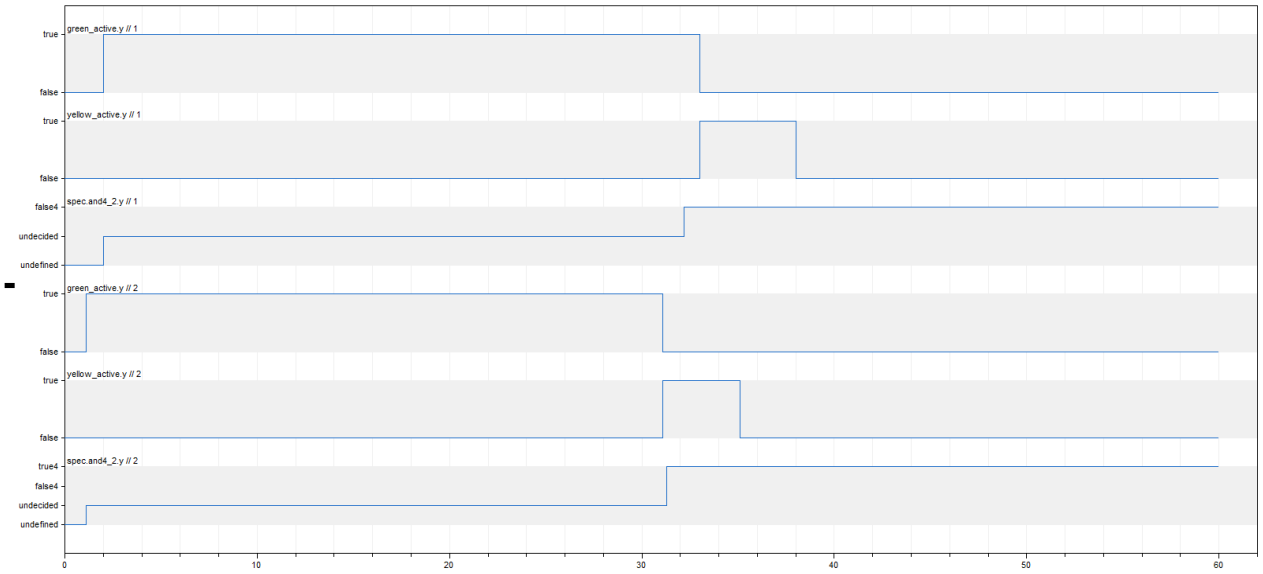
Clock = 1 s

req3 not satisfied

---

Clock = 0.1 s

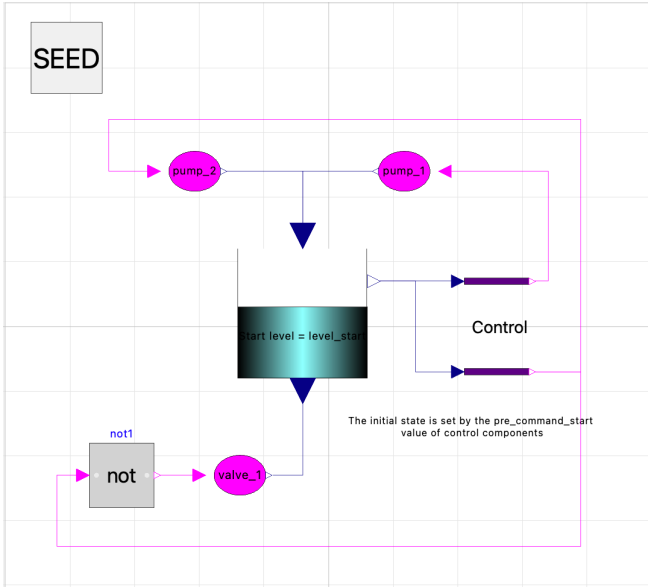
req3 satisfied




LINKÖPING UNIVERSITY

EDF

# Example: Tank with Failure




```
model requirements_tank is {

    class TankModel is {
        Real level is external ;
        Real temperature is external;
    };

    TankModel tankm is new TankModel;

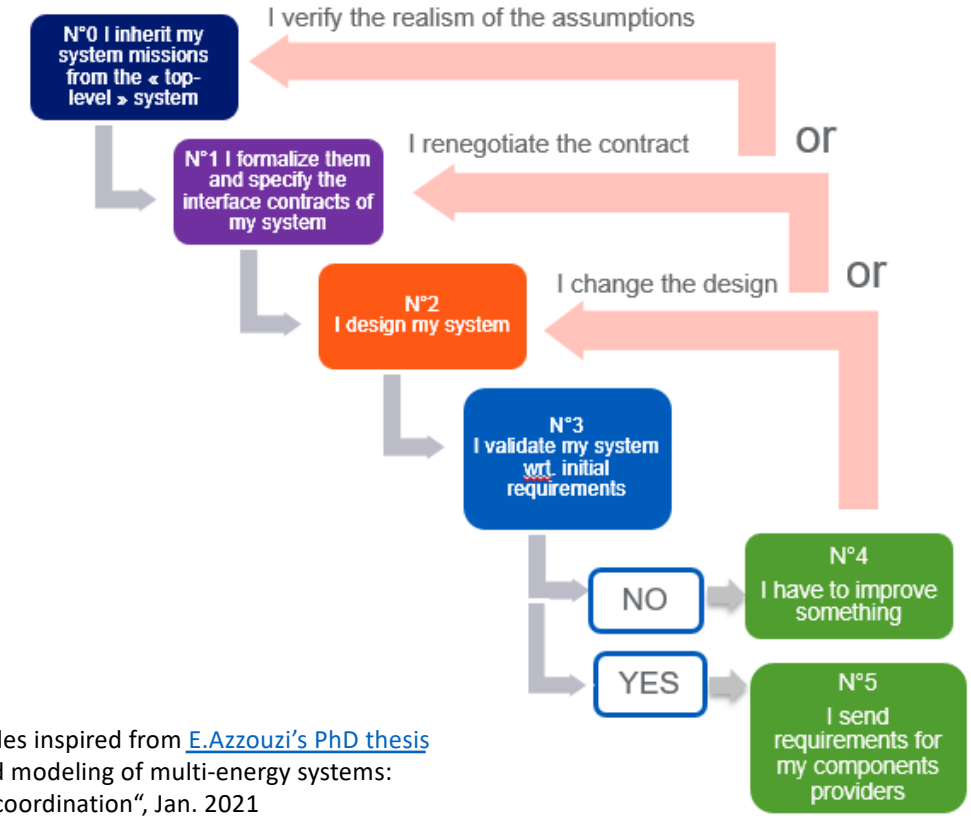
    // TankModel tankm;

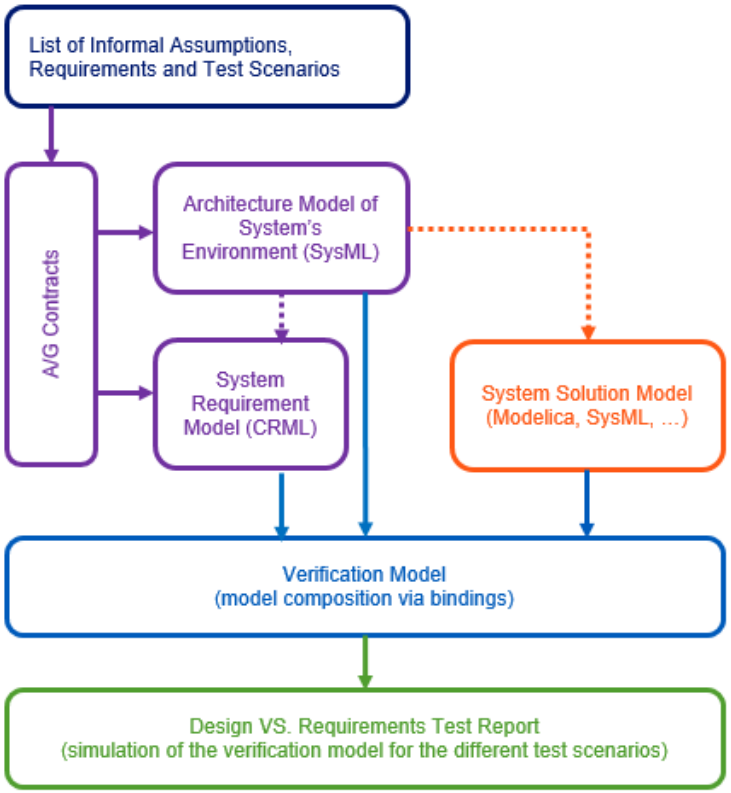
    Boolean R1 is tankm.level >= 7.5;
    Boolean R2 is tankm.level >= 10;
    Boolean R3 is R1 and (not R2);
    Boolean R4 is tankm.temperature <= 100;

};
```

# CRML methodology by example

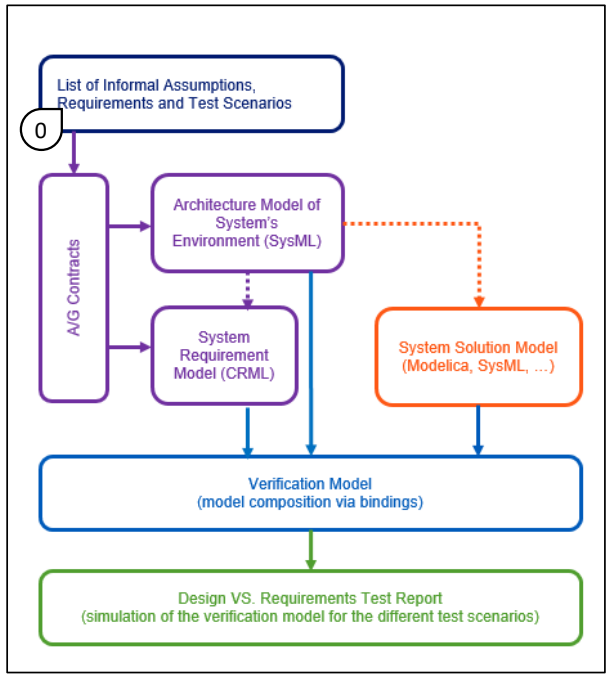
# How to Use CRML in Practice?




*Main principles inspired from E.Azzouzi's PhD thesis "Multi-faceted modeling of multi-energy systems: Stakeholders coordination", Jan. 2021

# Demo on ICS



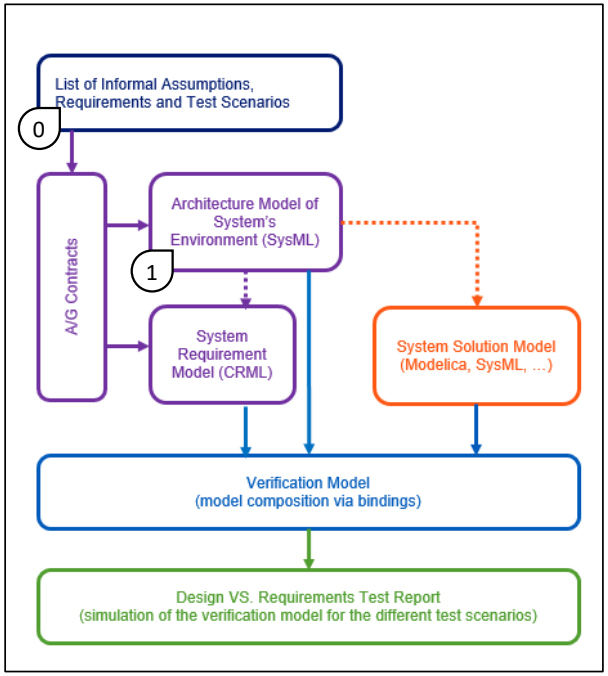

**Use-Case: Intermediate Cooling Subsystem (ICS)**
A cooling system used in French nuclear power plant

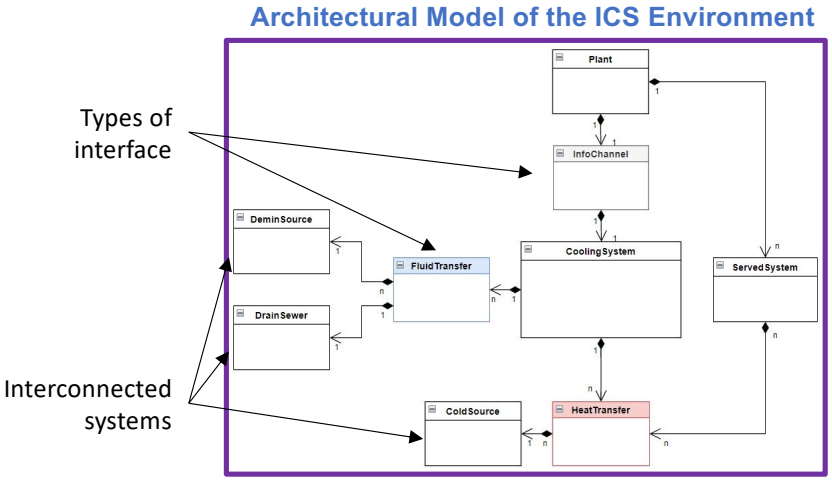
**Scenario: "Design justification"**
Rigorously demonstrate that the ICS fulfils its mission in compliance with the safety & physical constraints at an acceptable availability rate.

## Main System Mission

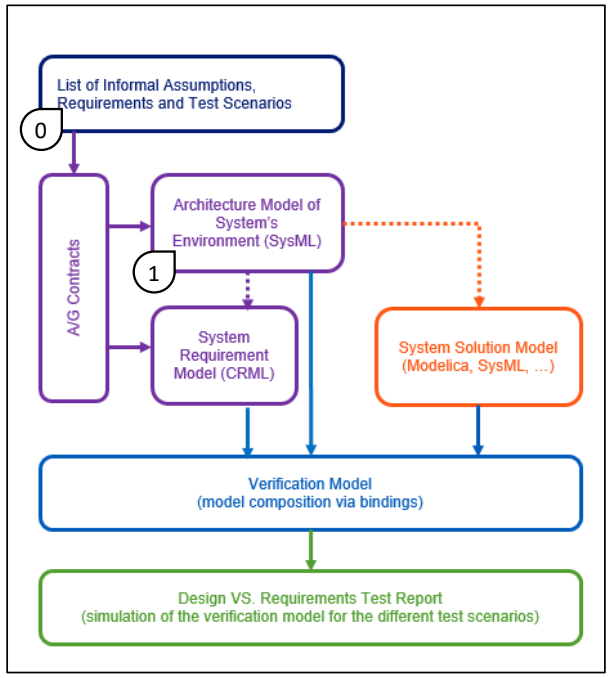
- Evacuate the heat produced by several served systems when they are in operation
- With the use of demineralized water (water coming directly from the cold source -sea or river- could damage equipment)
- At a good availability rate
  (the overall plant should be shutdown if the cooling system does not properly fulfil its mission)

# Demo on ICS

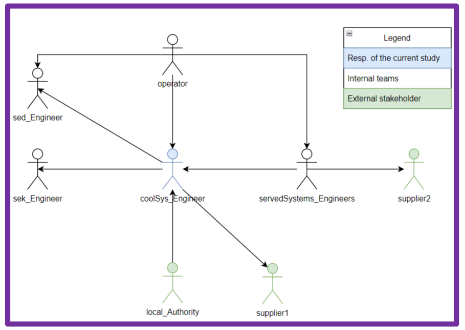

List of Informal Assumptions, Requirements and Test Scenarios
0
A/G Contracts
Architecture Model of System's Environment (SysML)
1
System Requirement Model (CRML)
System Solution Model (Modelica, SysML, ...)
Verification Model (model composition via bindings)
Design VS. Requirements Test Report (simulation of the verification model for the different test scenarios)


Architectural Model of the ICS Environment
Types of interface
Interconnected systems
Plant
InfoChannel
DeminSource
FluidTransfer
CoolingSystem
ServedSystem
DrainSewer
ColdSource
HeatTransfer

# Demo on ICS




**Stakeholder Interaction Model of the ICS Environment**

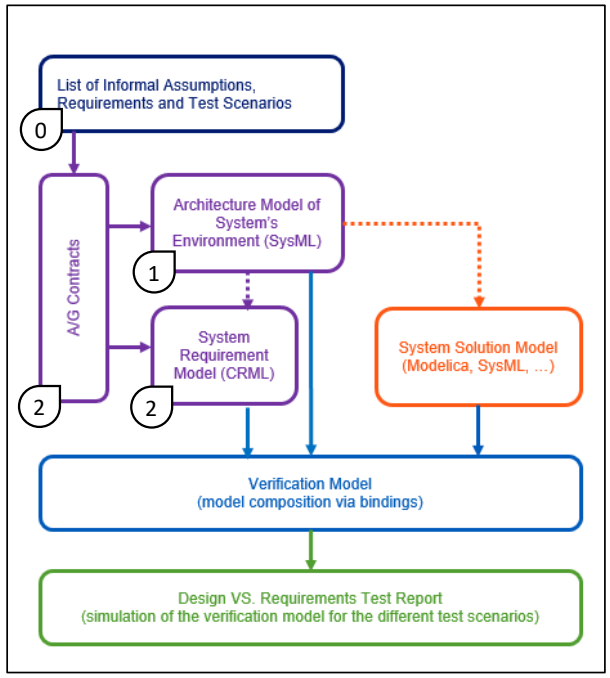



| Company | Stakeholder | Role(s) | System under responsability |
|---|---|---|---|
| Plant owner | coolSys_ Engineer | Design of the cooling system Specifications of heat exchangers | coolSys |
| | operator | Operation of the cooling system | coolSys |
| | servedSys1_ Engineer | Design of served system 1 | servedSys1 |
| | servedSys2_ Engineer | Design of served system 2 | servedSys2 |
| | servedSys3_ Engineer | Design of served system 3 | servedSys3 |
| | sed_ Engineer | Design of the demineralized water source | sed |
| | sek_ Engineer | Design of the drain sewer | sek |
| Local Authority | local_ Authority | Responsible of the integrity of the cold-water source (sea or river) | sen |
| Suppliers | suppliers | Furniture of components such as heat exchangers, pumps, ... | supplied components |

At least 9 stakeholders are involved although ICS is quite a mid-size system

→ Difficulty for the engineers to well define system's interfaces
→ Difficulty to track all possible changes and provide « on-purpose » test reports for each stakeholder without any method
→ Let's see how CRML can help automate the verification tasks

# Demo on ICS




## CRML requirement model

**Contract between the ICS and the served systems**

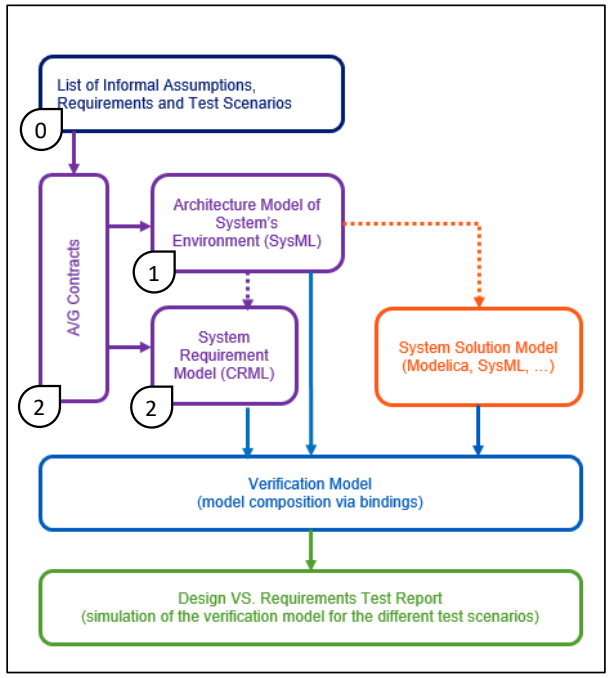
```
class Served_system is {
  [...]

  // Mission: When the served system is operating,
  // all the heat produced should instantly be evacuated by the sri
  constant Real epsilon is 0.001; // tolerance
  Boolean cool_served_system_0 is
  during inOperation ensure 0 <= sri.W - W <= epsilon

  // Requirement: The served system should not be
  // supercooled by the sri (to avoid thermal stresses).
  Boolean cool_served_system_1 is
    during sri.tRW <= (tRWMin - 0.1 Celsius) ensure not open;

  // Requirement: The served system should be cooled by the sri
  //  as soon as the sri water temperature is above the minimum acceptable.
  Boolean cool_served_system_2 is
    during sri.tRW >= tRWMin ensure open;
};
```

# Demo on ICS




**CRML requirement model**

**Contract between the ICS and the served systems**

```
class Served_system is {
  [...]

  // Mission: When the served system is operating,
  // all the heat produced should instantly be evacuated by the sri
  constant Real epsilon is 0.001; // tolerance
  Boolean cool_served_system_0 is
  during inOperation ensure 0 <= sri.W - W <= epsilon

  // Requirement: The served system should not be
  // supercooled by the sri (to avoid thermal stresses).
  Boolean cool_served_system_1 is
                                    nsure not open;

                                  cooled by the sri
                                  above the minimum acceptable.
```

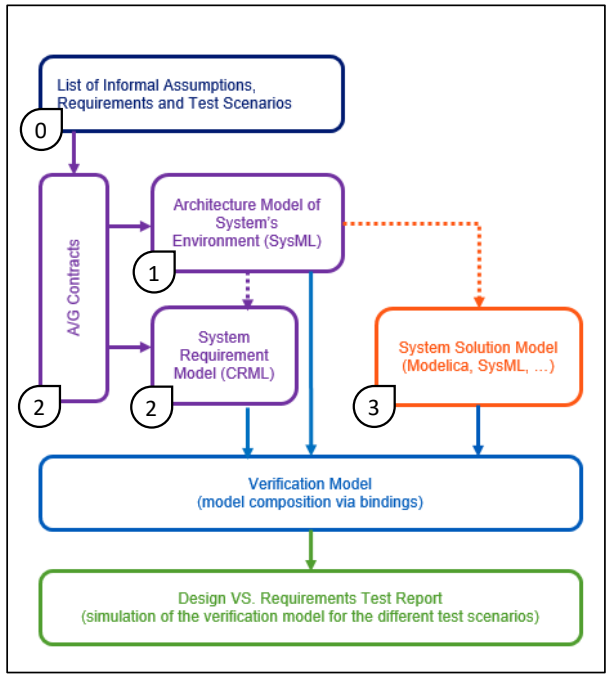
```
class Cooling_system is {
    [...]

    // Requirement: When the sri is operating,
    // the temperature of the cold water source should
    // be below its acceptable maximum (no overheat of the sea or river).
    Boolean coldW_sri_1 is
      during not (state_stopped or state_stopping)
      ensure sen.tCW <= sen.tCWMax;

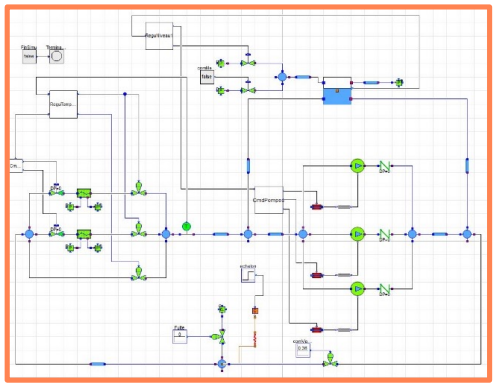
    // Requirement: When the sri is operating,
    // the temperature increase of the cold water source should
    // be below its acceptable maximum.
    Boolean coldW_sri_2 is
      during not (state_stopped or state_stopping)
      ensure tWW < (sen.tCW + sen.deltaTMax);
```

**Contract between the ICS and the cold source**

# Demo on ICS




**ThermoSysPro model of one possible design solution**

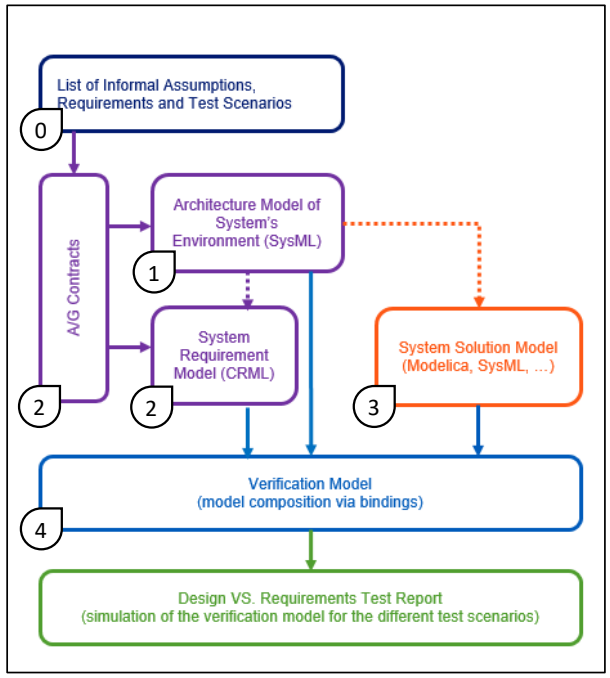


**Refinement of CRML model**

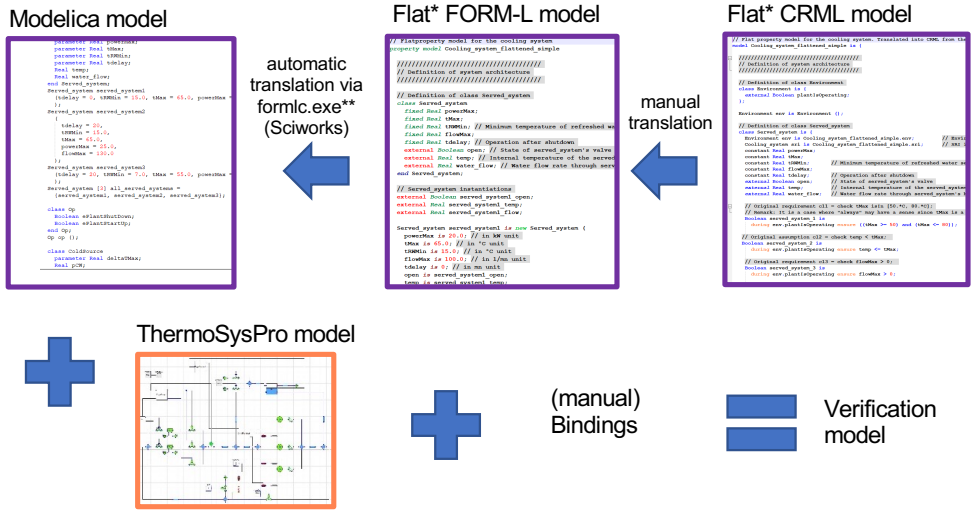
```
class Cooling_system is {
  [...]
  // Kpi: When operating, sri temperature should be around 17°C
  // to optimize thermal transfer in exchangers
  constant Temperature tolerance is 0.5 Celsius;
  Boolean kpi_1 is
    during inOperation ensure tRW >= (17 Celsius - tolerance)
    and tRW <= (17 Celsius + tolerance);
};
```

Possibility to refine the CRML model by adding new constraints such as some KPI for ICS performance

# Demo on ICS

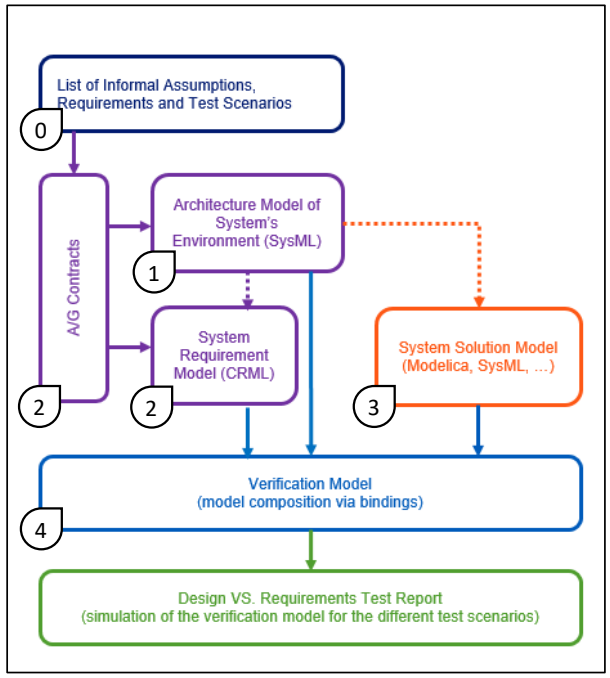



## Construction of the verification model



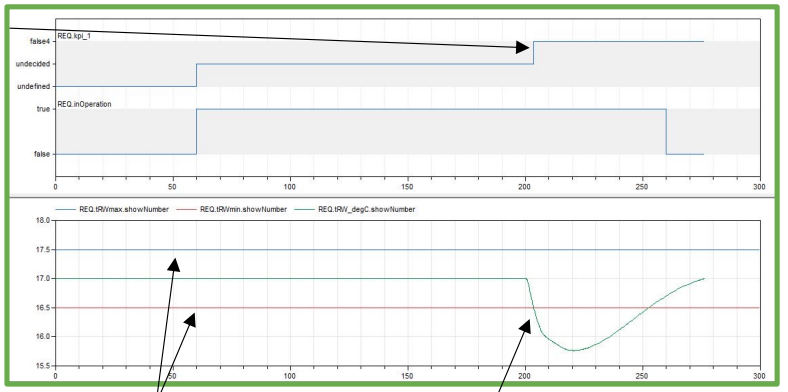

* All requirements are modelled in one file
** Will be replaced by LIU compiler for the final demo
*** Will be automated for the final demo (Python script already available)

# Demo on ICS




## Simulation results of the verification model

Violation of "kpi1" computed by the CRML model




Tolerance interval for temperature

Temperature computed by the ThermoSysPro model

# First demos for industrial use

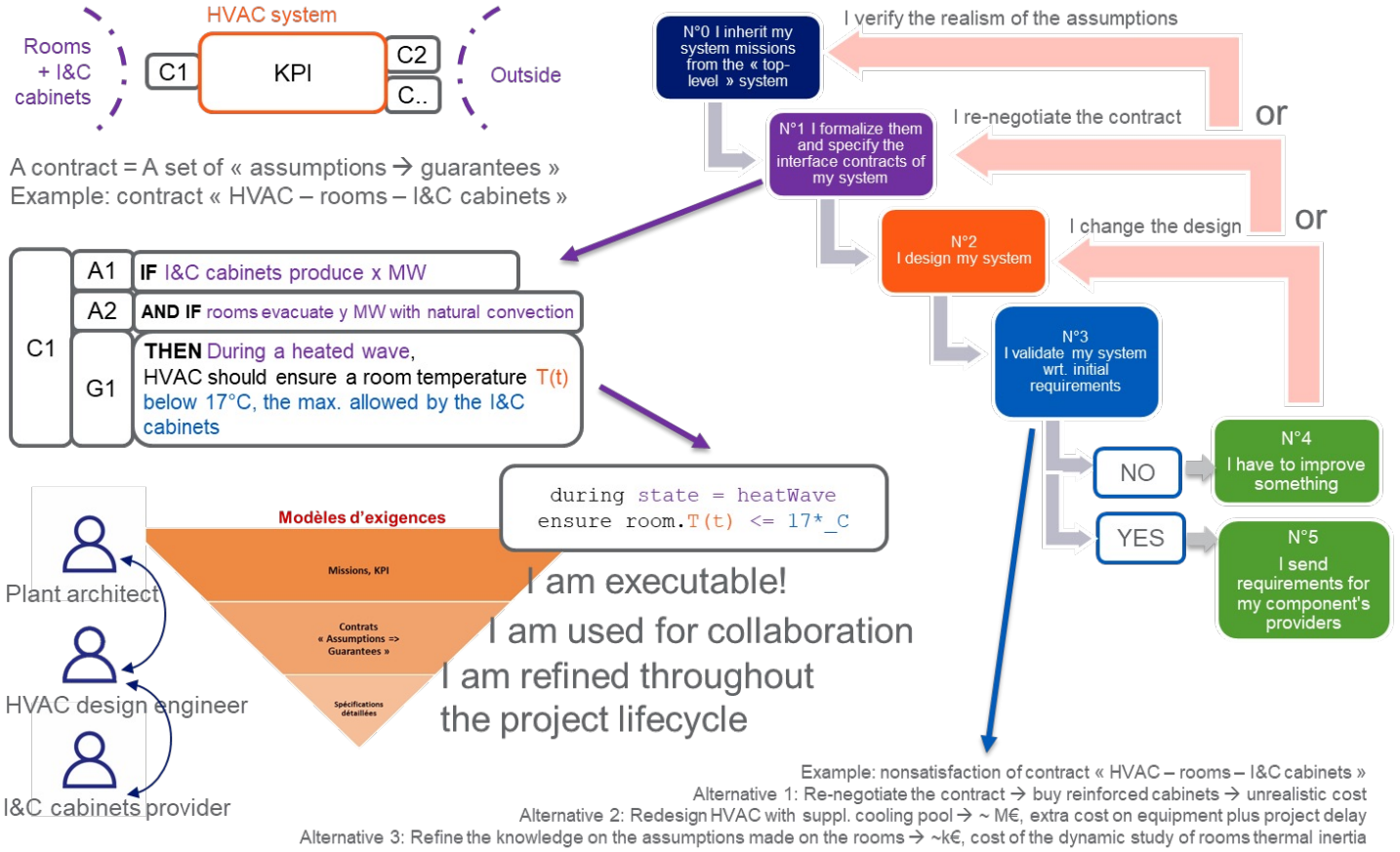
# Challenges Relative to Energy Systems' Engineering

- Interconnected systems with stringent physical constraints to ensure grid balancing
- Long system lifecycles: new solutions build on existing ones (they are not created from scratch)
- Compliance with strict safety and environmental rules
- Compliance with dependability and availability constraints (to ensure security of energy supply)
- Involvement of multiple stakeholders: clients, regulatory authorities, grid operators, energy providers, insurers, urban and land-use planning, plant operators..., with different and possibly contradictory objectives
- Moving context with increasing uncertainties (due to lack of energy policy coordination between countries, climate change, energy market instabilities, evolution of demand wrt. new usages...).

**Energy systems are globally over constrained.**
Methods & tools are needed to help engineers **find the best compromise**

# CRML to Demonstrate the Correct Behavior While Finding the Best Compromise

**Use-Case:**
Heat, ventilation and air conditioning system in charge of cooling control cabinets in nuclear power plants



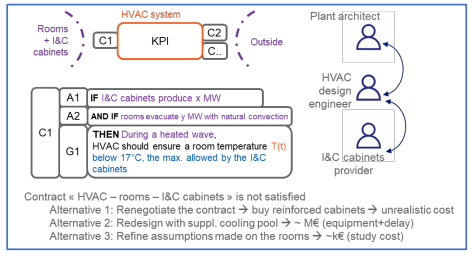
# Automating FMECA with CRML to Perform Safety Analyses Earlier in Design Lifecycle

- FMECA: Failure Mode and Effect Critical Analysis
- Objective: identify simple failures that could lead to unacceptable consequences for system's missions
- Current state-of-the-art FMECA are performed manually and consists in:
  - Listing system's components and, for each component, listing possible defects
  - Then for each defect evaluating with static models what are the consequences on the system's mission.
- It results in a table displaying the criticality level for each configuration which is costly, hardly verifiable and quickly outdated (since project data evolves rapidly).
- With a CRML model it is possible to generate such table automatically (see table for HVAC system previously described)

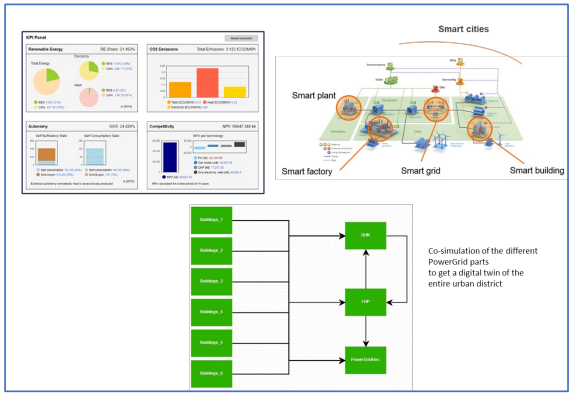
| | DWL Division1 | Failure Mode | Normal Operation | Duration until LHVAC(mn) | From LLCU to LHVAC((mn) |
|---|---|---|---|---|---|
| Scenario | | | | | |
| 1 | Protection Grid | Fails Blocked (FB) | 1 | 0.0 | 0.0 |
| 2 | Fan | Failure to Start (FS) | 2 | 10.0 | 0.0 |
| 3 | Fan | Failure to Run (FR) | 2 | 60.0 | 0.0 |
| 4 | Fan | Failure to Restart (FRS) | 2 | 20.0 | 0.0 |
| 5 | Fan | Spurious Operation (SO) | 1 | 0.0 | 0.0 |
| 6 | Fan | Run-On (RO) | 1 | 0.0 | 0.0 |
| 7 | Manuel Valve | Valve Left in Wrong Position (EW) | 2 | 80.0 | 0.0 |
| 8 | Manuel Valve | Rupture (RU) | 3 | 189.4 | 179.4 |
| 9 | Cooling Coil | Fails Blocked (FB) | 1 | 0.0 | 0.0 |
| 10 | Temperature Sensor | Failure to Provide Signal (FPZ) - untested | 1 | 0.0 | 0.0 |
| 11 | Temperature Sensor - Maximum Temperature Thres... | Several Failures - untested | 1 | 0.0 | 0.0 |
| 12 | Temperature Sensor - Minimum Temperature Thres... | Several Failures - untested | 1 | 0.0 | 0.0 |

# Usages Tested with CRML (and previous related work on FORM-L)

- Evaluation of solution alternatives
  - Use of requirement models as an objective comparison criteria
- Automation of FMECA studies
  - Use of requirement models to define the impact of a faulty component on system's missions and its criticality
- Automation of impact analysis
  - Use of requirement models to propagate changes in design assumptions
- Stakeholder coordination
  - Use of requirement models to conceive large-scale systems and to prepare model interfaces for the assembly of their digital twins
- …




Solution comparison for HVAC design




Energy renovation of an urban district

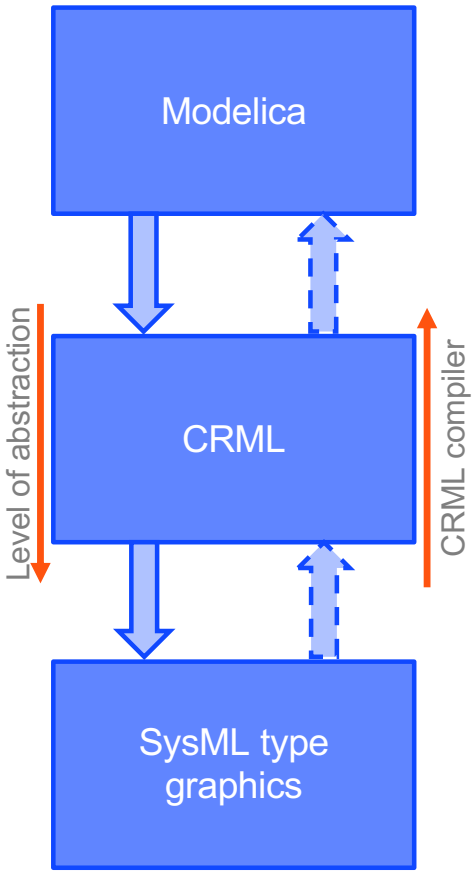
# Wrap-Up & Perspectives

# Adequacy of CRML to Realistic Engineering Problems

- CRML allows to deal with the complexity of realistic requirements on CPS
- First experiments on energy systems showed that CRML allows to automate some engineering tasks with relatively poor human-added value (e.g., manual evaluation of a change in design assumptions for tens till hundreds of sizing configurations)
- Consequently, more verifications can be performed all along the project and engineers may focus on the solution dynamic behavior to retrieve some margins (time is source of cost reductions and cost overruns, see HVAC experiment)

# A First CRML Tool Support But Still Limited

- Small requirement models can be developed with Modelica (CRML.mo)
- Larger models need CRML
  - Close to natural language syntax.
  - Objects with redefinition, functions and sets.
- Engineering activities need a graphical design method
  - Method development started with an intern on SMR design



# How to Promote CRML?

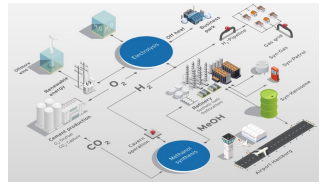
- Rethinking historical engineering practices is still a long journey
  but things are changing (internal digital transformation program with adoption of PLM solutions, constitution of “textual” requirement databases…)
- CRML can foster such acculturation to System Engineering practices by offering a better integration with disciplinary computation tools → requirements are really used to “relieve the pain” and do not appear anymore as an extra-activity “just” made for documentation

- (Large) Adoption of CRML requires:
  - Communication efforts → see perspectives for new demos
  - More mature supporting tool(s) → see perspectives for a better toolchain
  - Persistent tools / ecosystem → see standardization efforts
  - Better progress on the link with SysML v2 to ease adoption by the System Engineering community and to provide graphical patterns for end-users to ‘transparently’ generate CRML

# What's Next for Demos?

- Optimized design of new nuclear plants
  - E.g., reduce the cost of some EPR subsystems
  - Conceive SMR as a product: modular design methodology for an adaptable SMR to export and new usages such as combined production of electricity, $H_2$, heat, water desalinization, …
    => EURATOM TANDEM project just started, internal internship ongoing

- Optimized design and operation of $H_2$ stations
  - E.g., offer new services to industrials and public entities
    - for low-carbon and renewable $H_2$ production
    - for distribution to industrial processes and fleet of heavy vehicles
  - Efficient design of $H_2$ production plants to best fit local needs (renewables, needs to valorize co-products such as heat, $O_2$...): use of models to reinforce integration engineering and guarantee performance all along the payback period
    => internal R&D activity starting in 2023



[The « Westküste 100 » Real-world Laboratory]

# What's Next to Improve the Toolchain?

- Specify the tool architecture for bindings in a cross platform compatible workflow (mainly a text-based format for binding specification and a Modelica tool-agnostic workflow)

- Integration with the work on variable data structures

- Support of Libraries, Types and Units by the CRML compiler

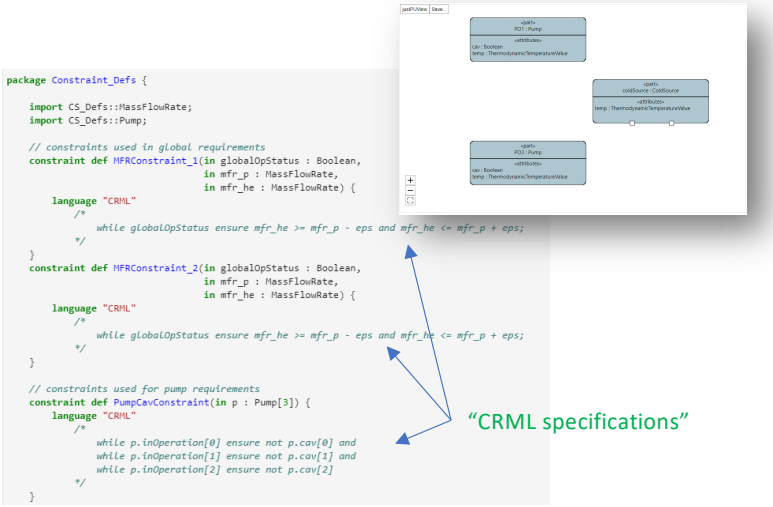
# Standardization efforts

- Target to standardize CRML as Modelica Association project
- Efforts on integration with the SysMLv2 standard for architecture specification
- Provide a bridge between SysML v2 and M&S

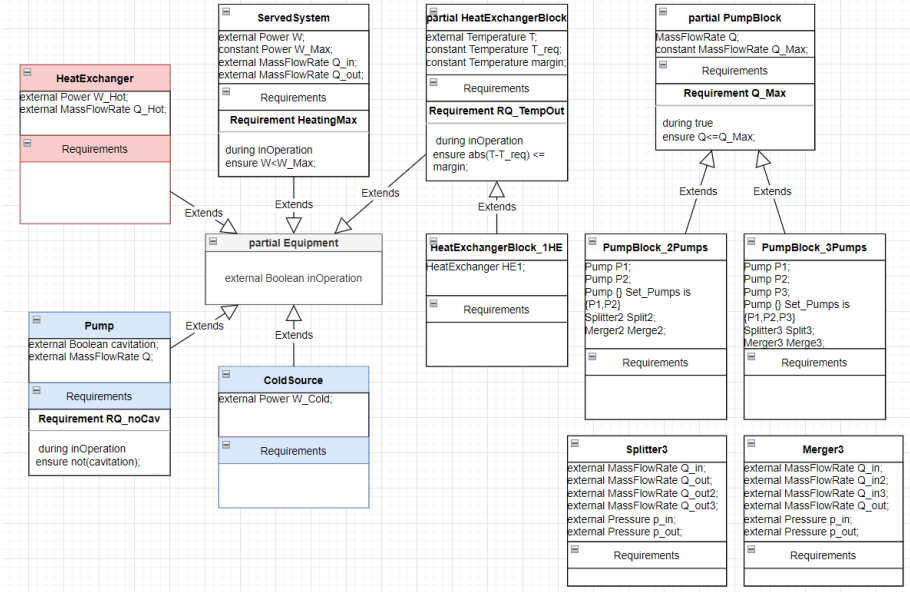
# Integration within SysML v2

**CRML support by integration within SysML v2 ecosystem:** can we support SysML v2 within generative engineering tool, and use it to embed CRML specifications?

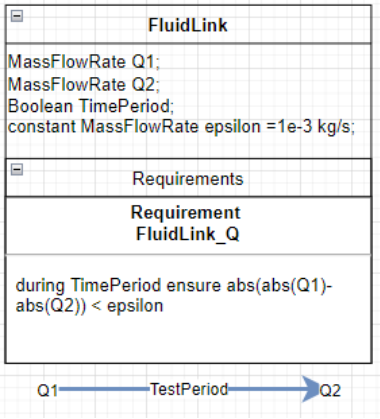
Work of Siemens Belgium partner within EMBrACE project

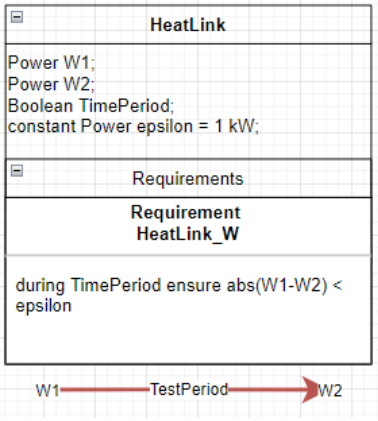


# Towards a Graphical Method to Ease CRML Coding



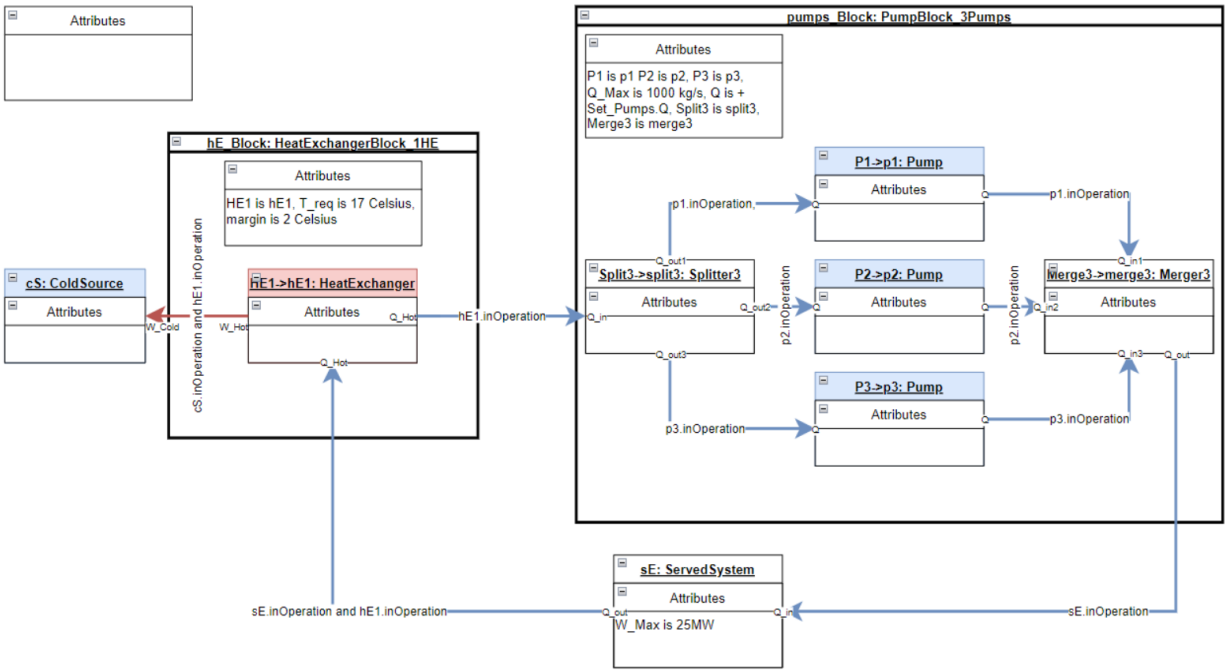

Let's take the cooling system (ICS) example again!




- Preliminary description of the cooling system architecture using a class diagram
- Physical constraints on heat and water exchanges are put in generic classes

# Towards a Graphical Method to Ease CRML Coding




- Then the class instances diagram (interaction diagram) is defined

# Towards a Graphical Method to Ease CRML Coding

```
ColdSource cS2 is new ColdSource();

HeatExchanger hE1 is new HeatExchanger();

HeatExchangerBlock_1HE hE_Block is new HeatExchangerBlock_1HE(HE1 is hE1, T_req is 17 Celsius, margin is 2 Celsius);


class FluidLink is {
    MassFlowRate Q1;
    MassFlowRate Q2;
    Boolean TimePeriod;
    constant MassFlowRate epsilon = 1e-3 kg/s ;

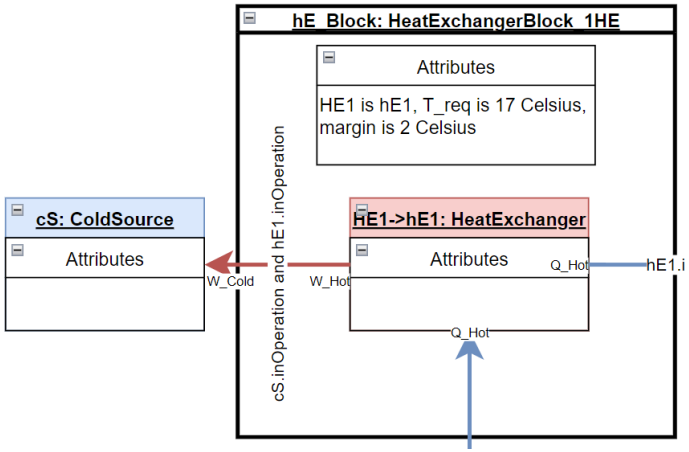
    Requirement FluidLink_Q is during TimePeriod ensure abs(abs(Q1)- abs(Q2))<epsilon;
};

class HeatLink is {
    Power W1;
    Power W2;
    Boolean TimePeriod;
    constant Power epsilon = 1 kW;

    Requirement HeatLink_W is during TimePeriod ensure abs(W1-W2) < epsilon ;
};

FluidLink f1 is new FluidLink(Q1=split3.Q_out1,Q2=p1.Q,TimePeriod=p1.inOperation);
FluidLink f2 is new FluidLink(Q1=split3.Q_out2,Q2=p2.Q,TimePeriod=p2.inOperation);
FluidLink f3 is new FluidLink(Q1=split3.Q_out3,Q2=p3.Q,TimePeriod=p3.inOperation);
FluidLink f4 is new FluidLink(Q1=p1.Q,Q2=merge3.Q_in1,TimePeriod=p1.inOperation);
FluidLink f5 is new FluidLink(Q1=p2.Q,Q2=merge3.Q_in2,TimePeriod=p2.inOperation);
FluidLink f6 is new FluidLink(Q1=p3.Q,Q2=merge3.Q_in3,TimePeriod=p3.inOperation);
FluidLink f7 is new FluidLink(Q1=merge3.Q_out,Q2=sE.Q_in.Q,TimePeriod=sE.inOperation);
FluidLink f8 is new FluidLink(Q1=sE.Q_out,Q2=hE1.Q_Hot,TimePeriod=sE.inOperation and hE1.inOperation);
FluidLink f9 is new FluidLink(Q1=hE1.Q_Hot,Q2=split3.Q_in,TimePeriod=hE1.inOperation);

HeatLink h1 is new HeatLink(W1=hE1.W_Hot,W2=cS.W_Cold,TimePeriod=cS.inOperation and hE1.inOperation);
```




Could be generated automatically

# References

- CRML
  - Specification v1.1 : https://www.embrace-project.org/specification/CRML.pdf
  - Compiler prototype: https://github.com/lenaRB/crml-compiler

- Modelling architecture
  - Bouskela D., Nguyen T. and Jardin A. (2017), “Toward a Rigorous Approach for Verifying Cyber-Physical Systems Against Requirements,” Canadian J. of Electrical and Computer Engineering, Vol. 40-2, pp. 66-73.
  - Bouskela D. and Jardin (2018), “ETL: A New Temporal Language for the Verification of Cyber-Physical Systems,” 2018 Annual IEEE International Systems Conference (SysCon).
  - Dong Y., Bouskela D., Jardin A., Bindings: a New Approach for co-simulation applied to System Design Verification Against Requirements, 12th MODPROD Workshop on Model-Based Cyber-Physical Product Development. February 2018.
  - Bouskela D., Nguyen T. and Jardin A. (2015), “MODRIO Project Modeling Architecture for the Verification of Requirements,” MODRIO deliverable D2.1.1, EDF R&D report n°H-P1C-2014-15188-EN.
- State of the art
  - Azzouzi E., Jardin A., Bouskela D., Mhenni F., Choley J.-Y. 2019. “A Survey on Systems Engineering Methodologies for Large Multi-Energy Cyber-Physical Systems”. In *Proceedings of the 13th IEEE International Systems Conference*, Orlando, Florida, USA
- Applications
  - E. Azzouzi. Multi-Faceted Modelling of Multi-Energy Systems : Stakeholders Coordination. PhD thesis, Université Paris-Saclay, 2021.
  - Bouquerel M., Kremers E., Thuy N., Jardin A. 2019. "Requirements Modelling to Help Decision Makers to Efficiently Renovate Energy Systems of Urban Districts". In Proceedings of the 29th Summer Simulation Conference, Berlin, Germany

# Thank you for your attention!

**Contact:**
lena.buffoni@liu.se
audrey.jardin@edf.fr

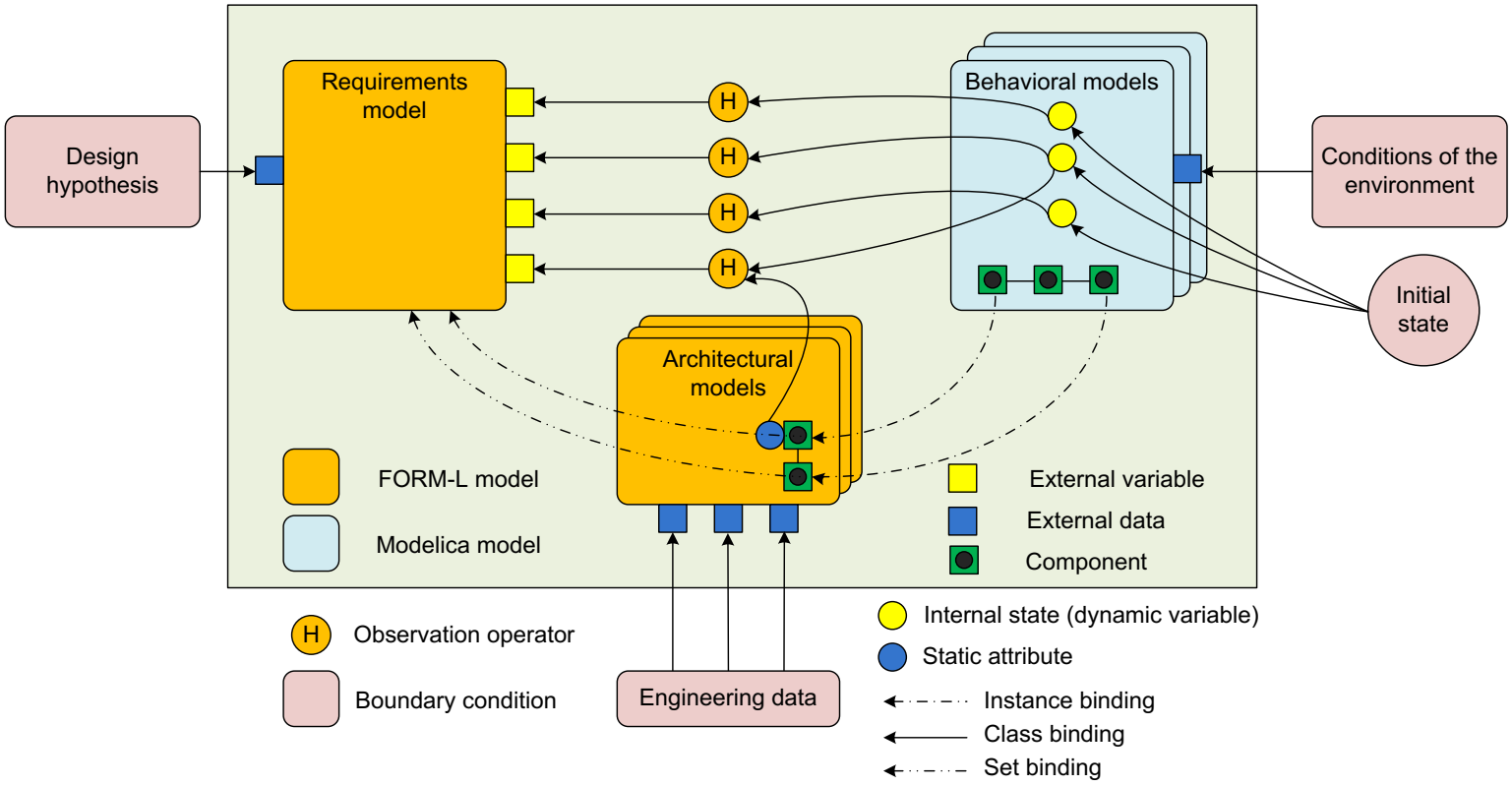
LINKÖPING UNIVERSITY EDF

# Appendix

# Methods Used For Requirement Verification

- **Inspection**.
  Examination of the product or system using human senses.
- **Analysis**.
  Validation of a product or system using calculations and models.
- **Demonstration**.
  Use the product or system as it is intended to be used and check that it does what the user requirements say that it should do.
- **Test**.
  Check through multiple scenarios that the product behaves as specified under a set of carefully specified test conditions.

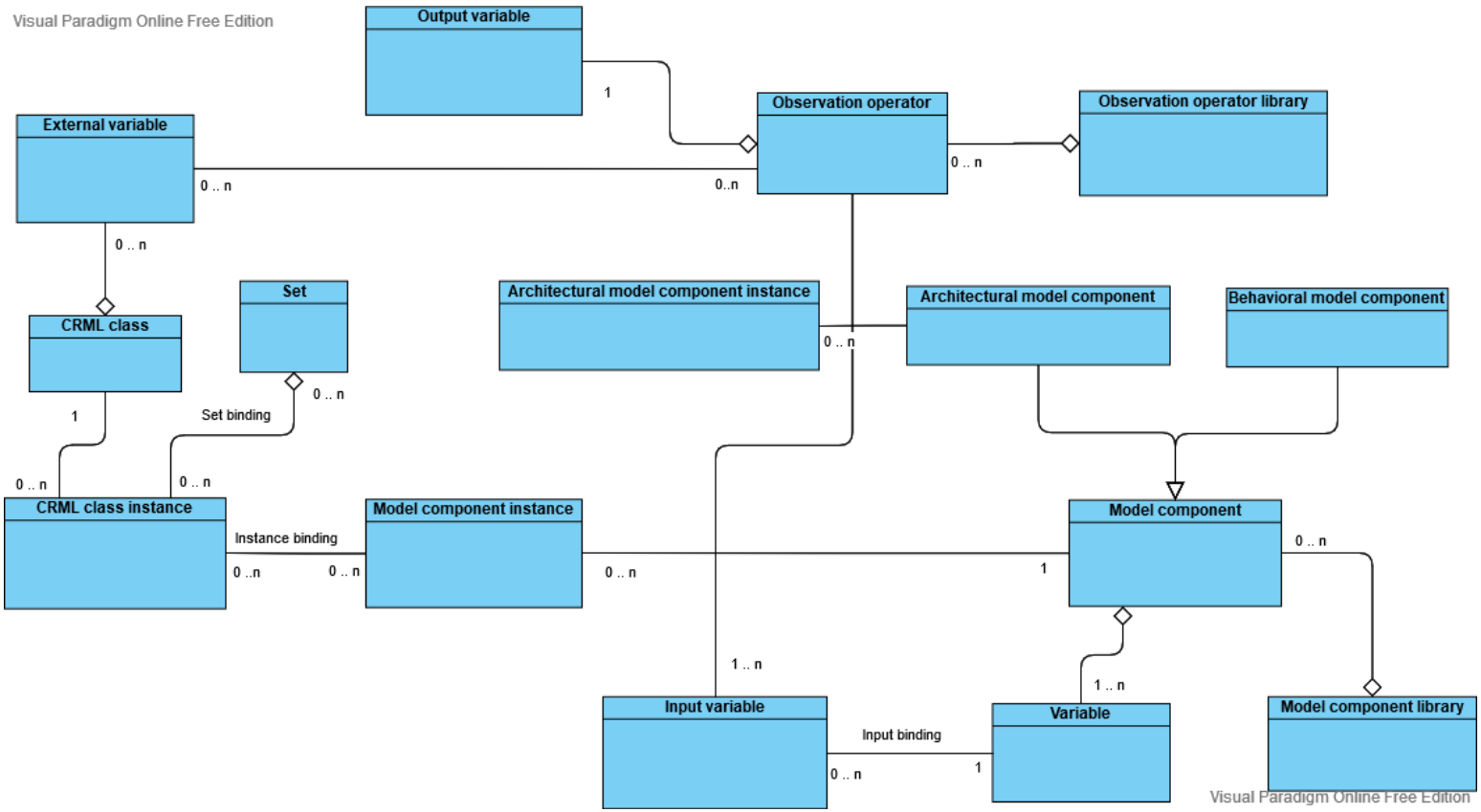
# Bindings: Verification model




LINKÖPING UNIVERSITY

EDF

# Bindings: Meta-model



# Bindings: Algorithm

The binding algorithm constructs automatically the verification model from the requirement, architectural and behavioral models, and the set, class and instance bindings.

Let us consider:

a. an external variable y declared in a class R: R.y,
b. an external set s of objects of type R declared in an object O: O.s

The problem is to form the expressions $p.y = H_y(e_1.x, \ldots, e_n.x)$ for all objects p in set O.s.

Active bindings used as input of the algorithm

a. Set binding $O.s \leftarrow \{ p_1, \ldots, p_n \}$
b. Variable binding $R.y \leftarrow [ H_1, \ldots, H_r ]$
c. Input bindings $H_i(u_1, \ldots, u_n) \leftarrow (u_1=E_1.x, \ldots, u_n=E_n.x)$ for $i = 1$ to $i = r$
d. Instance bindings $p \leftarrow \{ e_1, \ldots, e_n \}$ for all p in target$(O.s \leftarrow \{ p_1, \ldots, p_n \})$

Algorithm

For each p in target$(O.s \leftarrow \{ p_1, \ldots, p_n \})$

1. Find H of lowest rank in $R.y \leftarrow [ H_1, \ldots, H_r ]$ such that $sig(H) \subset sig(p \leftarrow \{ e_1, \ldots, e_n \})$. The result is $H_y$.
2. From the binding $H_y(u_1, \ldots, u_n) \leftarrow (u_1=E_1.x, \ldots, u_n=E_n.x)$, form symbolically the expression $p.y = H_y(u_1=E_1.x, \ldots, u_n=E_n.x)$.
3. For each $E_i$ in $\{ E_1, \ldots, E_n \}$
   a. If there exists a unique instance $e_i$ in the target of $p \leftarrow \{ e_1, \ldots, e_n \}$ such that $class(e_i) = E_i$, then keep $e_i$. If $role(e_i, p \leftarrow \{ e_1, \ldots, e_n \})$ is different from $role(R.y \leftarrow [ H_1, \ldots, H_r ])$, a warning may be raised.
   b. Else if there exists multiple instances $e_i$ in the target of $p \leftarrow \{ e_1, \ldots, e_n \}$ such that $class(e_i) = E_i$, then keep $e_i$ such that $role(e_i, p \leftarrow \{ e_1, \ldots, e_n \}) = role(R.y \leftarrow [ H_1, \ldots, H_r ])$.
   c. If no such unique $e_i$ may be found, then raise an error.
4. Replace in $p.y = H_y(u_1=E_1.x, \ldots, u_n=E_n.x)$ the classes $E_i$ by the found instances $e_i$. The result is $p.y = H_y(e_1.x, \ldots, e_n.x)$.

LINKÖPING UNIVERSITY

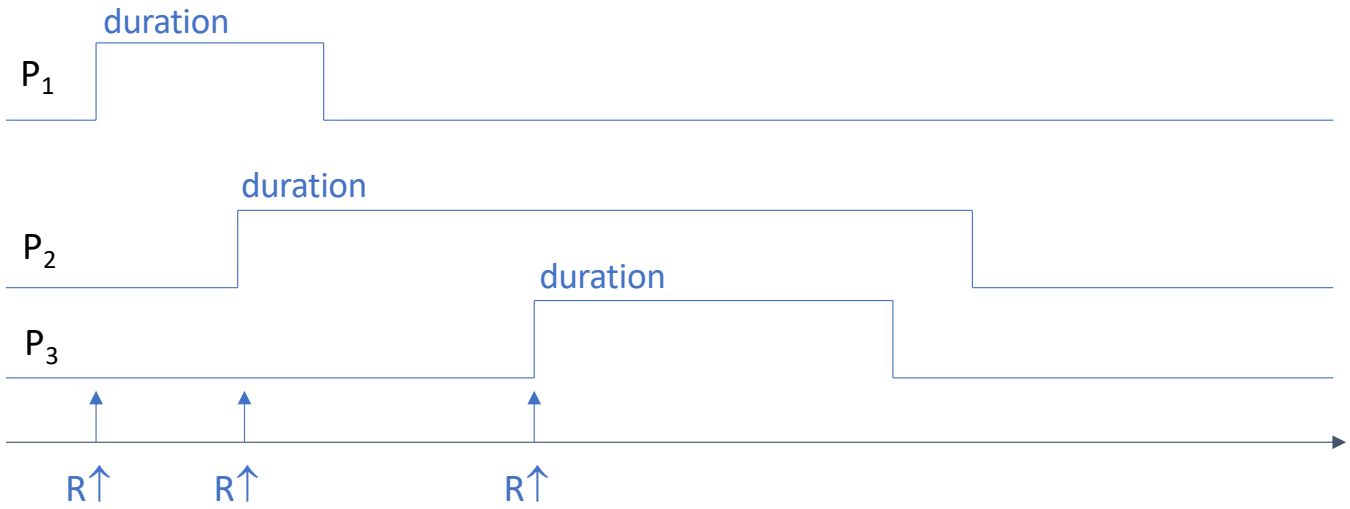
EDF

# Definition of Multiple Time Periods In the Continuous Clock Domain

- Multiple time periods P containing time periods opened at all occurrences of R↑ and closed at all occurrences of R↑ + duration are denoted

  P := $\Pi$(([ | ]) R↑, duration ([ | ]) := { ([ | ]) $\Omega$(R), $\Omega$(R) + duration ([ | ]) }

  The value of duration is taken at the instant of occurrence of R↑: duration := duration(R↑).
  duration $\in \mathbb{R}$

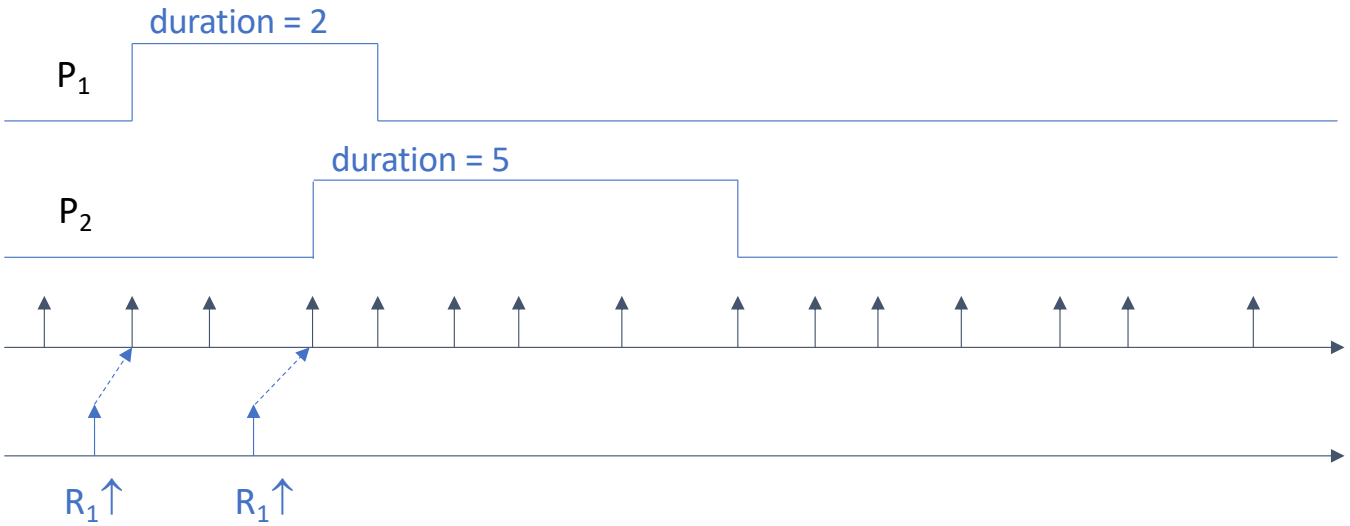


# Definition Of Multiple Time Periods in a Discrete Clock Domain

- Multiple time periods P containing time periods opened at all occurrences of $R\uparrow$ and closed at all occurrences of $R\uparrow$ + duration in the discrete clock domain $\Omega$ are denoted

$$P := \Pi(([\ |\ ])\ R\uparrow/\Omega,\ \text{duration}\ ([\ |\ ]) := \{([\ |\ ])\ \Omega(R\uparrow/\Omega),\ \Omega(R\uparrow/\Omega) + \text{duration}([\ |\ ])\ \}$$

The value of duration is taken at the instant of occurrence of $R\uparrow/\Omega$ : duration := duration($R\uparrow/\Omega$)
duration $\in \mathbb{N}$

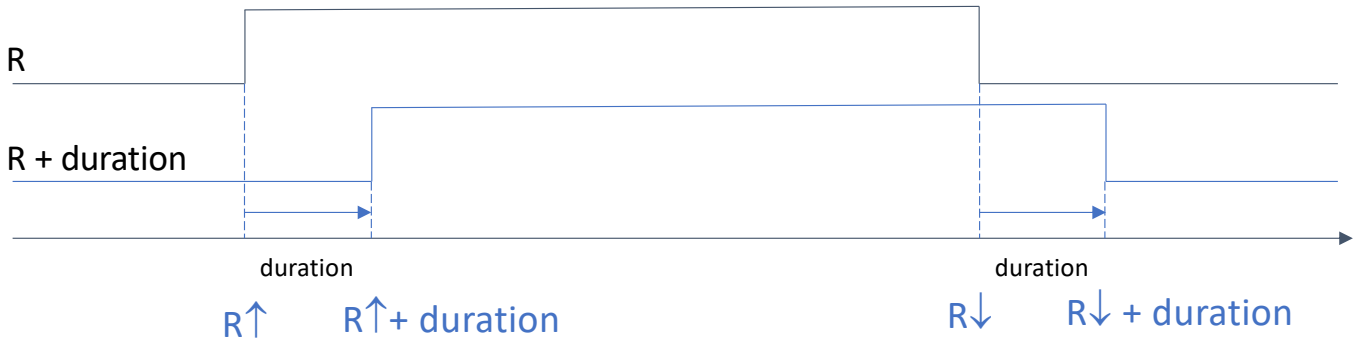


# Operations on Booleans in the Continuous Clock Domain

- Delay: R := R + duration

  (R + duration)(t) := R(t – duration)

The value of duration is taken at the instant of occurrence of R↑: duration := duration(R↑)




R↓ = ¬R↑

# Operations on Booleans in the Continuous Clock Domain

- The duration of a Boolean φ over a time period P is the elapsed time φ is true inside P

$$\text{duration}(\varphi, P) := \int_{t \in P} \text{int}(\varphi)\, dt$$

- int is the function

$$\text{int}(\varphi) := \begin{cases} 1 \text{ if } \varphi = \text{true} \\ 0 \text{ if } \varphi \neq \text{true} \end{cases}$$

# Operations on Events in the Continuous Clock Domain

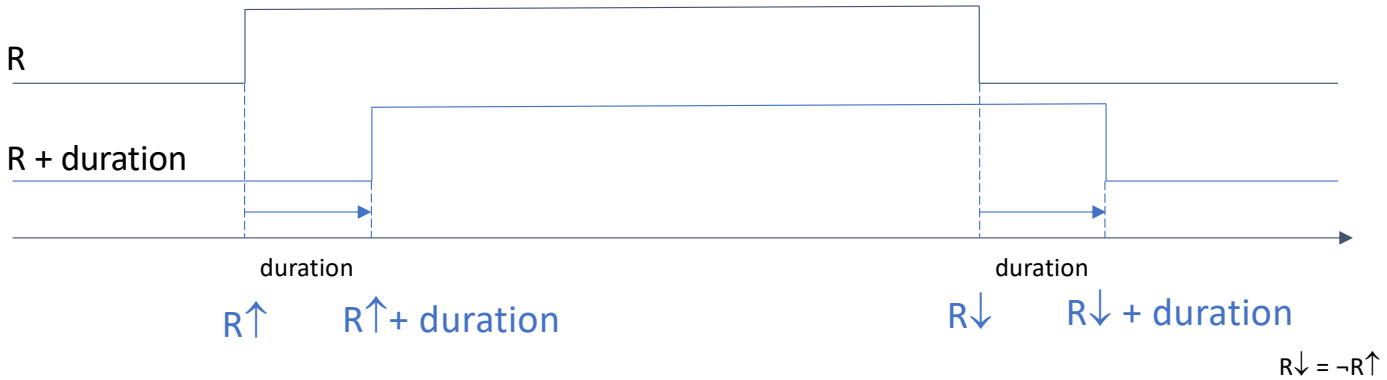
- Delay: $R_2\uparrow := R_1\uparrow + \text{duration}$ with duration $\in \mathbb{R}$

  $R\uparrow + \text{duration} := (R + \text{duration})\uparrow$

- Elapsed time: $\text{duration} := R_2\uparrow - R_1\uparrow$ with duration $\in \mathbb{R}$

  $\text{duration} = (R + \text{duration})\uparrow - R\uparrow$

The value of duration is taken at the instant of occurrence of $R\uparrow$: duration := duration($R\uparrow$)



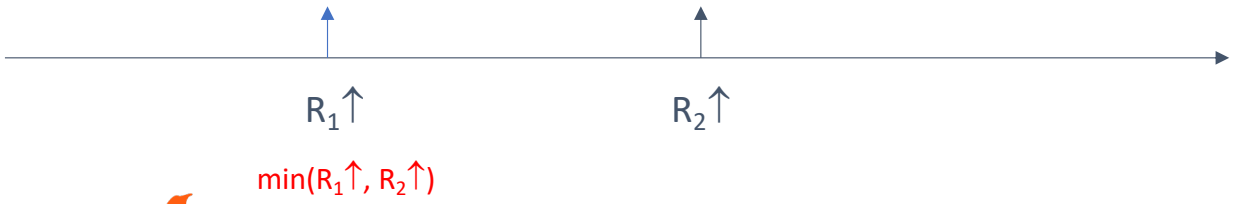
# Operations on Events in the Continuous Clock Domain

- First: $\min(R_1\uparrow, R_2\uparrow)$

$$\min(R_1\uparrow, R_2\uparrow) := \begin{cases} R_1\uparrow \text{ if } R_1\uparrow \text{ occurs before } R_2\uparrow \\ R_2\uparrow \text{ if } R_2\uparrow \text{ occurs before } R_1\uparrow \end{cases}$$

- Definition using clocks:

$$\min(R_1\uparrow, R_2\uparrow) := \Omega(R1\uparrow \vee R2\uparrow)(\bullet == 1)$$

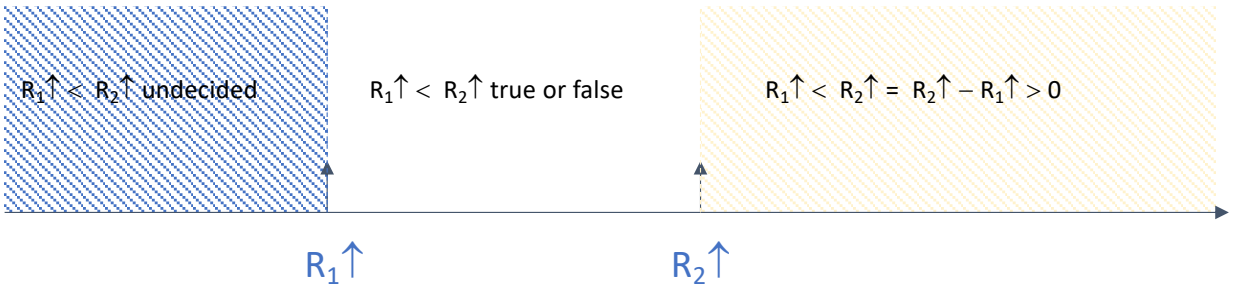


# Operations on Events in the Continuous Clock Domain

- Strictly before: $R_1\uparrow < R_2\uparrow$.

- The value of $R_1\uparrow < R_2\uparrow$ is decided at time instants when $R_1\uparrow$ or $R_2\uparrow$ have occurred. If $R_1\uparrow$ and $R_2\uparrow$ have occurred, then $R_1\uparrow < R_2\uparrow = R_2\uparrow - R_1\uparrow > 0$

$$R_1\uparrow < R_2\uparrow := \begin{cases} \text{for } t \in [\, \min(R_1\uparrow, R_2\uparrow), +\infty\, [ & \begin{cases} \text{true if } R_1\uparrow \text{ occurs strictly before } R_2\uparrow \\ \text{false otherwise} \end{cases} \\ \text{for } t \in ]\, -\infty, \min(R_1\uparrow, R_2\uparrow)\, [ & \text{undecided} \end{cases}$$

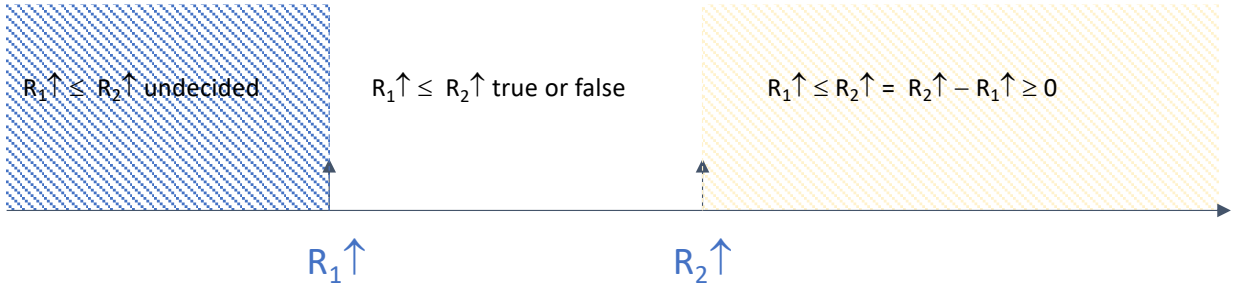


# Operations on Events in the Continuous Clock Domain

- Before: $R_1\uparrow \leq R_2\uparrow$.

- The value of $R_1\uparrow \leq R_2\uparrow$ is decided at time instants when $R_1\uparrow$ or $R_2\uparrow$ have occurred. If $R_1\uparrow$ and $R_2\uparrow$ have occurred, then $R_1\uparrow \leq R_2\uparrow = R_2\uparrow - R_1\uparrow \geq 0$

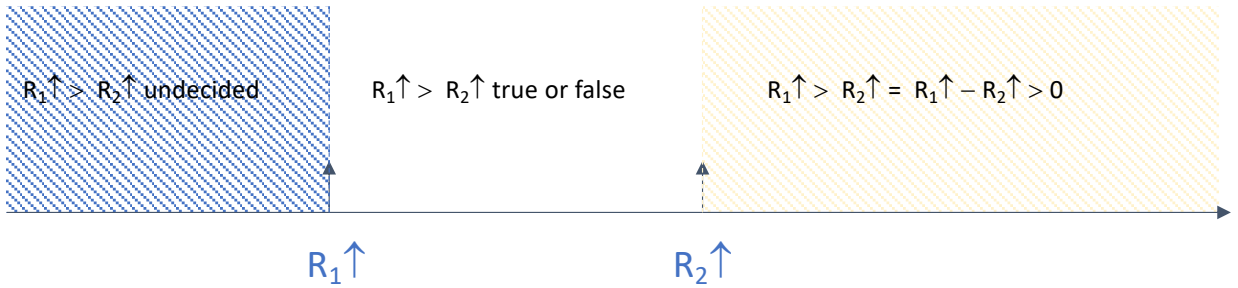
$$R_1\uparrow \leq R_2\uparrow := \begin{cases} \text{for } t \in [\, \min(R_1\uparrow, R_2\uparrow), +\infty \,[ & \begin{cases} \text{true if } R_1\uparrow \text{ occurs before } R_2\uparrow \\ \text{false otherwise} \end{cases} \\ \text{for } t \in ]\, -\infty, \min(R_1\uparrow, R_2\uparrow) \,[ & \text{undecided} \end{cases}$$



# Operations on Events in the Continuous Clock Domain

- Strictly after: $R_1\uparrow > R_2\uparrow$.
- The value of $R_1\uparrow > R_2\uparrow$ is decided at time instants when $R_1\uparrow$ or $R_2\uparrow$ have occurred. If $R_1\uparrow$ and $R_2\uparrow$ have occurred, then $R_1\uparrow > R_2\uparrow = R_1\uparrow - R_2\uparrow > 0$

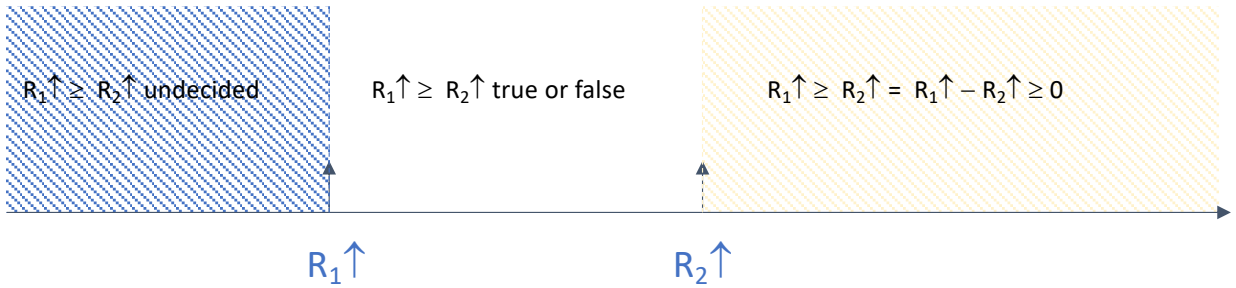
$$R_1\uparrow > R_2\uparrow := \begin{cases} \text{for } t \in [\, \min(R_1\uparrow, R_2\uparrow), +\infty \,[ & \begin{cases} \text{true if } R_1\uparrow \text{ occurs strictly after } R_2\uparrow \\ \text{false otherwise} \end{cases} \\ \text{for } t \in ]\, -\infty, \min(R_1\uparrow, R_2\uparrow) \,[ & \text{undecided} \end{cases}$$



# Operations on Events in the Continuous Clock Domain

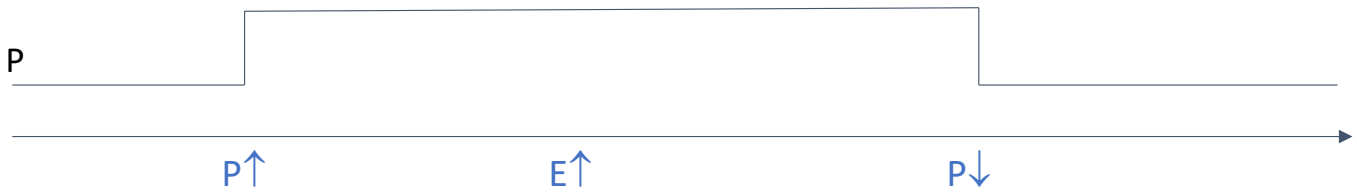
- After: $R_1\uparrow \geq R_2\uparrow$.
- The value of $R_1\uparrow \geq R_2\uparrow$ is decided at time instants when $R_1\uparrow$ or $R_2\uparrow$ have occurred. If $R_1\uparrow$ and $R_2\uparrow$ have occurred, then $R_1\uparrow \geq R_2\uparrow = R_1\uparrow - R_2\uparrow \geq 0$

$$R_1\uparrow \geq R_2\uparrow := \begin{cases} \text{for } t \in [\, \min(R_1\uparrow, R_2\uparrow), +\infty \,[ & \begin{cases} \text{true if } R_1\uparrow \text{ occurs after } R_2\uparrow \\ \text{false otherwise} \end{cases} \\ \text{for } t \in ]\, -\infty, \min(R_1\uparrow, R_2\uparrow) \,[ & \text{undecided} \end{cases}$$



# Events Occurring Inside a Time Period

- Event $E^{\uparrow}$ occurring inside time period P is denoted $E^{\uparrow} \in P$.
- $E^{\uparrow} \in P := E^{\uparrow} \geq P^{\uparrow}$ and $E^{\uparrow} \leq P^{\downarrow}$.
- $\geq$ and $\leq$ must be replaced by $>$ and $<$ if boundaries are not included.



# Requirement Satisfaction Over Multiple Time Periods

- The satisfaction of $R = \varphi \otimes \{ P_1, P_2, \ldots, P_n \}$ is defined as

  $$P \models \varphi := \wedge \{ P_1 \models \varphi, P_2 \models \varphi, \ldots P_n \models \varphi \}$$

- The satisfaction of $R = \{ \varphi_1, \varphi_2, \ldots, \varphi_p \} \otimes P$ is defined as

  $$P \models \varphi := \wedge \{ P \models \varphi_1, P \models \varphi_2, \ldots P \models \varphi_n \}$$

- The satisfaction of $R = \{ \varphi_1, \varphi_2, \ldots, \varphi_p \} \otimes \{ P_1, P_2, \ldots, P_n \}$ is therefore

  $$P \models \varphi = \wedge \{ P_j \models \varphi_i, 1 \leq i \leq p, 1 \leq j \leq n \}$$

# 4-Valued Logical Equality Operator

- The equality of Boolean $\varphi_1$ with Boolean $\varphi_2$ when event $E\uparrow$ occurs is

$$(\varphi_1 = \varphi_2)@E\uparrow := \begin{cases} \text{for } t \in [\, E\uparrow, E\uparrow\,] & \begin{cases} \text{true if } \varphi_1(t) = \varphi_2(t) \\ \text{false if } \varphi_1(t) \neq \varphi_2(t) \end{cases} \\ \text{for } t \notin [\, E\uparrow, E\uparrow\,] & \text{undefined} \end{cases}$$