RAMDE

Requirements and Model Drive Engineering

Introducing Formal Languages & Methods in Requirements Engineering

CISTER – Research Centre in Real-Time & Embedded Computing Systems

What we will be looking at now

- Why **do we need** Formal Methods in Requirements Engineering
- What do Formal Methods **have to offer**

> **"It is clear to all the best minds in the field that a more mathematical approach is needed for software to progress much."**
>
> *Bertand Meyer, author of the concept of "Design by Contract"*

What are Formal Methods?

A very, very lighweight introduction…

What are Formal Methods?

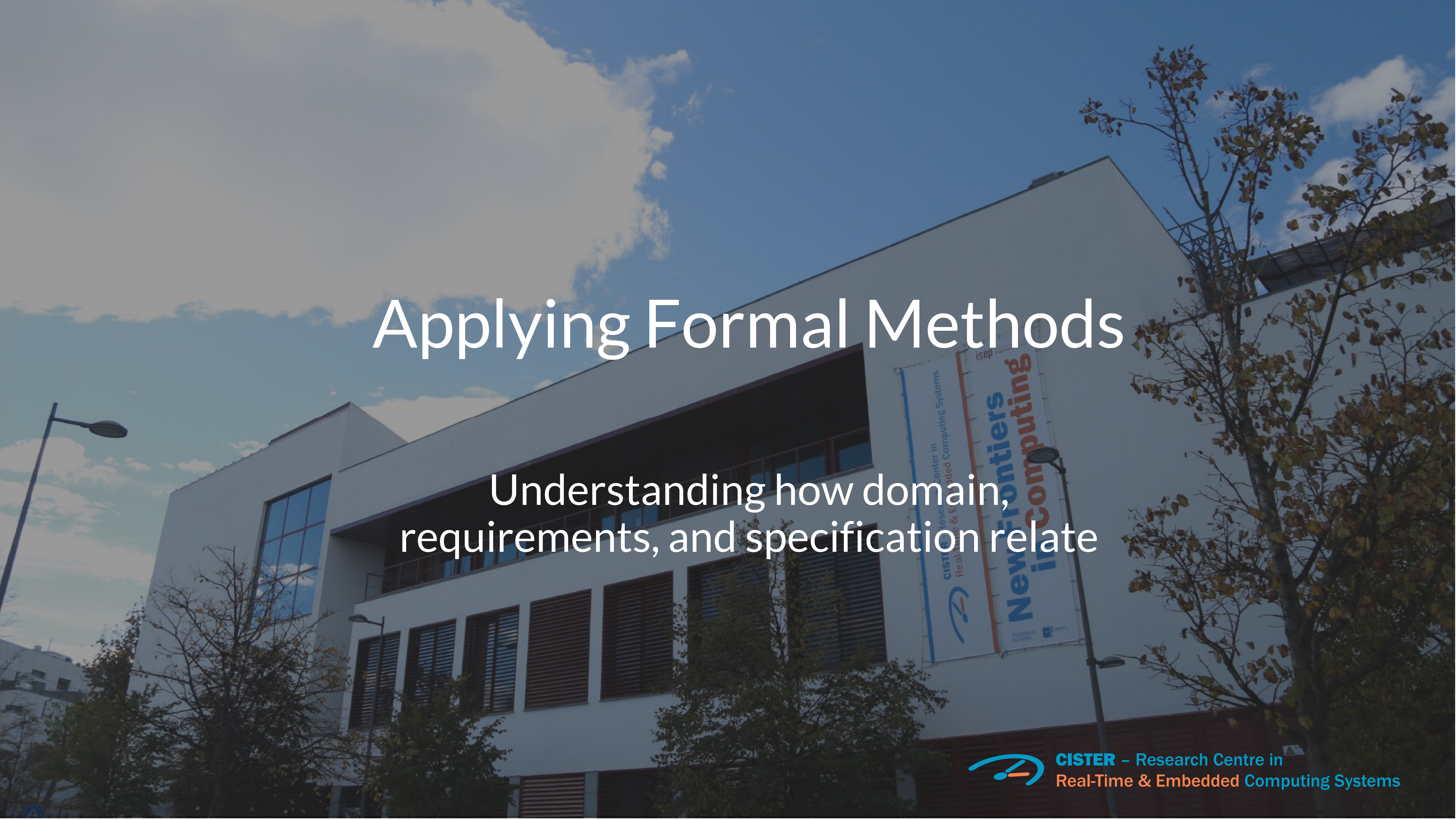
- **General view:**
  - application of mathematics to software engineering
  - concerned with specifying, modelling, and analysing the system using and underlying, precise, mathematically based notation
- **More narrow view:**
  - use formal languages
  - engage on formal reasoning about formulae of the language

Formal Methods and Requirements

- **How do Formal Methods relate to requirements engineering?**
- **Essentially, through formal notation:**
  - Formal **set of rules** that define the **syntax** and **semantics** of the language
  - The rules that determine if the formulae are **well-formed** and to **prove properties** about those formulae

Highlight Example

- First Order Logic (also known as Predicate Logic)
  - primitives for **expressions**: *variables, constants, functions, predicates*
  - logical **connectives**: *and* (∧), *or* (∨), *implies* (→), *equality* (=), *negation* (¬)
  - **quantifiers**: *for all* (∀), *exists* (∃)
  - **deduction** rules: *we will look into this in the next lessons*
- Examples of expression/formulae:
  - $(x > y) \wedge (y > z) \longrightarrow x > z$
  - $\forall x, \forall y, \forall z, (x > y) \wedge (y > z) \longrightarrow x > z$
  - $\forall x, \forall y, \forall z, [\, \boldsymbol{Node}(x, y, z) \longrightarrow \boldsymbol{Leaf}(x) \longrightarrow \boldsymbol{Leaf}(z)] \longrightarrow x < y \wedge z \geq y$
  - $\forall h, \forall l, \boldsymbol{List}(l) \longrightarrow \boldsymbol{List}(h :: l)$
  - $\forall p, \forall s, \boldsymbol{Parent}(p, s) \longrightarrow \neg \boldsymbol{Parent}(s, p)$



Applying Formal Methods

Understanding how domain, requirements, and specification relate

CISTER – Research Centre in Real-Time & Embedded Computing Systems

What does correctness mean?

- Let us consider the following representation

Application Domain

S - Specification

Machine Domain

D – Domain Properties

R - Requirements

C – Computer

P - Program

- Some important information:
  - ***Domain properties*** are characteristics of the **application domain**
  - ***Requirements*** are conditions/features that to be **made true** by the system
  - ***Specification*** is a description of the behaviours the **system must have** in order to **satisfy the requirements**

What does correctness mean?

- **Correctness criteria:**
  - the program running on a given computer **satisfies** the **specification**
  - the **specification**, in the context of the domain properties, **satisfies** the requirements

- **Completeness criteria:**
  - We **discovered** all the **important requirements**
  - We **discovered** all the **relevant domain properties**

Understanding the Differences

- Requirement **R**:

  The database shall only be accessible by authorized personnel

- Domain Properties **D**:

  Authorized personnel have passwords

  Passwords are never shared with non-authorized personnel

- Specification **S**:

  Access to the database shall only be granted after the user types and authorized password

D + S *implies* R

When things go wrong

- not that rare
- caused by:
  - bugs, misunderstood specification, poor configuration management, poor control, etc.
  - …
- caught by:
  - testing, simulation, **formal verification** against spec
  - code inspection and walkthoughs

- very rare
- cause by:
  - power failure, circuit/chip failure, …
- caught by:
  - desing analysis, testing, certification through usage, …

When things go wrong

S - Specification

Application Domain

D – Domain Properties

R - Requirements

Machine Domain

C – Computer

P - Program

- common
- caused by:
  - misunderstood requirements
  - wrong choice of specification language
  - ambiguous, inconsistent/incomplete specification
- caught by:
  - inspection, **formal verification**
  - end-to-end testing

When things go wrong

- **very common**
- **caused by:**
  - lack of domain expertiese
  - unquestioned assumptions
  - insufficient domain analysis
- **caught by:**
  - failure analysis
  - talking to the right experts
  - …

- **common**
- **caused by:**
  - insufficent/bad coomunication with stakeholders
  - lack of analysis
  - failure to handle change
- **caught by:**
  - inspections, customer reviews
  - modelling, **formal verification,** prototyping

Where do Formal Methods Apply

- **Formalize the specification**:
  - as a precise baseline to **verify** the **program** against
  - as a **model of program behavior** to compare against **requirements**
- **Formalize the domain knowledge**:
  - **reason** about whether it is **complete**
  - **reason** about how it **affects** the proposed system
- **Formalize the requirements**:
  - **animate** them
  - test **logical coherence**
  - check for **completeness** **against** an underlying mathematical **model**

Why formalise in Requirements Eng?

- **To remove ambiguity and improve precision**
- **Provides a basis for verification:**
  - That a **program** meets its **specification**
  - That a that **specification captures** the **requirements** adequately
- **Allows us to reason about the requirements**
  - **Properties** of formal requirements models can be **checked automatically**
  - Can test for **consistency**, explore the **consequences**, etc.
- **Allows us to animate/execute the requirements**
- **Will have to formalize eventually anyway**
  - **bridging** from the informal **world** to a formal **machine** domain

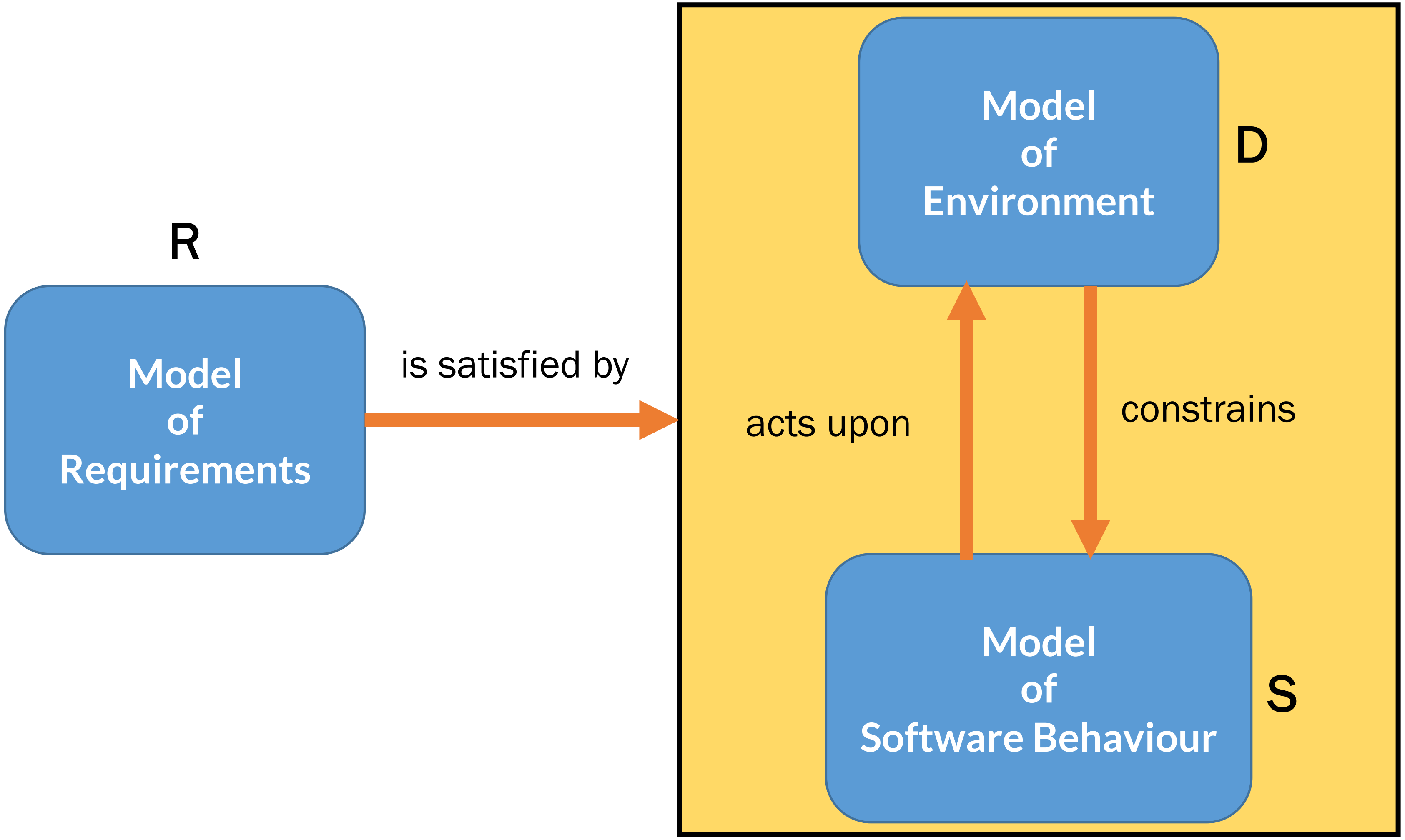
Why is not usually done?

- **Tend to exhibit lower level than other techniques**
  - inclusion of a **lot of detail** and the **system boundaries** already well **established**
- **Tend to concentrate on consistent, correct models**
- **Confusion about which tools are appropriate**
  - advocates generally **too tied** to a **certain tool**, and **scalability** is still not possible with many tools/prototypes
- **Require more effort**
  - mostly due to the necessary **mathematical training** and **experience**
- **Are not appropriate in many projects…**

Modelling

And how it relates to Formal Methods

So, too many models involved?



Modelling can help!

- **Can guide elicitation**:
  - understanding what **questions** to make, **identify hidden** requirements
- **Can provide a measure of progress**:
  - understand if **completeness** of the model implies **finished elicitation**
- **Uncover problems:**
  - **inconsistency** of the model **revealing** other interesting **aspects** of the system (*e.g.*, *infeasible requirements*, *terminology confusion*, *disagreement between stakeholders*)
- **Check our own understanding:**
  - model **has the right** properties, or **reason** about its **consequences**

Types of Model

- **Informal, written in natural language**
  - extremely **expressive** and **flexible**, **poor** at capturing the **semantics** of the model, good for **elicitation** and to **annotate** models for communication
- **Semi-formal**
  - captures **structure** and **some semantics**
  - allow for some **reasoning**, **consistency** checking, etc.
- **Formal**
  - very **precise semantics**, possible to conduct **reasoning**

Good Properties of Modelling Languages

- **Indepent of Implementation**
  - **does not** consider **internal representation**, abstracts data types
- **Abstraction capabilities**
  - extracts the **essential aspects** of the system, allowing to have a "big pic"
- **Formality**
  - precise **syntax** and rich **semantics**
- **Constructability/Composability**
  - can **compose/decompose** pieces of the model to handle **complexity** and size

Good Properties of Modelling Languages

- **Ease of analysis**
  - ability to **analyze** for **ambiguity**, **incompleteness**, **inconsistency**
- **Traceability**
  - ability to **cross-reference elements**, ability to **link** to **design**, etc.
- **Executability**
  - can **animate** the model, to **compare** it to **reality**
- **Minimality**
  - no **redundancy** of **concepts**, **no extraneous choices** of how to represent **concepts**

Model Validation

- **Consistency Analysis and Typechecking**
  - **well-formedness first** that most of other methods and represent real-world integrity
- **Validation**
  - **animation** of small models
  - **formal queries** about **implications** of know properties, particular requirements, or changes
  - **system properties**
- **Verifying design refinement**
  - does the **design meets the requirements**

Some differences between Formal Methods

Formal methods differ considerably on their nature, notably in what concerns the following aspects.

- Ontology
  - fixed vs extensible
- Mathematical foundations
  - logic vs algebraic languages vs set theory vs state machines
- Treatment of time
  - discrete sequences of events vs continuous time vs quantified intervals

Some Traditions/Schools

- **Formal Specification Languages**
  - inception on **program verification**, suitable for **specifying** the **behaviour** of **program units**
  - key technologies being **type checking**, **theorem proving**
- **Reactive System Modelling**
  - capture **dynamic models** of **system behaviour**, with focus on **reactive systems** (e.g., safety and liveness in real-time embedded systems)
  - key technologies: **model checking**, **consistency checking**
- **Formal Conceptual Modeling**
  - focus on **modelling domain entities**, **activities**, **agents**, **assertions**
  - key technologies: **formal ontologies**, **first order logic**

Formal Specification Languages

- **Three families:**
  - Operational - specification is executable abstraction of the implementation
  - State-based - views a program as a (large) data structures whose state can be altered by procedure calls
  - Algebraic - views a program as a set of abstract data structures with a set of operations
- **Developed for specifying programs**
  - **Programs** are **formal**, man-made **objects**
  - these languages are typically **not very appropriate** for Requirements Engineering (**requirements specification ≠ program specification**)

Reactive Systems Modelling

- **Modeling how a system should behave**
  - model the **environment** as a **state machine**
  - model the **system** as a **state machine**
  - model **safety**, **liveness** properties of the machine as **temporal logic assertions**
  - check whether the **properties hold** of the system **interacting** with its **environment**
- **We will look into this during the course, mostly in the 2nd semester**
  - class on formal verification of software
  - in the current class you will get the basis using algebraic specifications

From Notations to Methods

- **Notation:**
  - a representation **language** for **expressing things**
- **Technique:**
  - prescribes **how to perform** a particular activity and **its product** in a **particular notation**
- **Method:**
  - definition on **how to perform** a collection of **activities**, focusing on **integration of techniques** and **guidance** about their use

Conclusions

- **Formal languages and methods** are here to **help** in the very **complex and challenging** process of Requirement Engineering
- **Requirements** valid as a **consequence** of **domain properties** and **software properties & behaviour**
- **Several flavours** of formal methods available, which we must **choose wisely**
- **First Order Logic as highlight method**
  - *we will start looking at it in the next lesson, picking up on the basic mathematical notations, and proceeding to syntax, semantics, and deductive rules that support reasoning*

The End


CISTER - Research Center in
Real-Time & Embedded Computing Systems
isep
New Frontiers
in Computing