Domain-Specific Modeling for Architecture-Centric Software Engineering

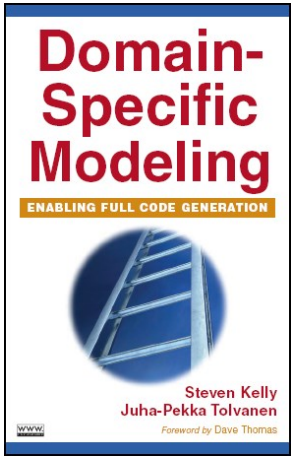
ECSA 2023

Juha-Pekka Tolvanen, Steven Kelly

https://metacase.com/papers/ECSA_2023_tutorial.pdf

About me: Juha-Pekka Tolvanen

- Works for MetaCase
  - Provider of modeling and code generation tool MetaEdit+
- Acts as a consultant for creating DSLs
  - 100+ DSL solutions
- Co-author of a book on Domain-Specific Modeling, IEEE-Wiley
- PhD in computer science, adjunct professor
- Enjoys sailing and skiing



Route today

1 Introduction

Languages and architecture, domain-specific languages

2 Cases and examples

Industrial experience reports, Detailed demonstration

3 Elements of language

Abstract syntax, concrete syntax, semantics, metamodels

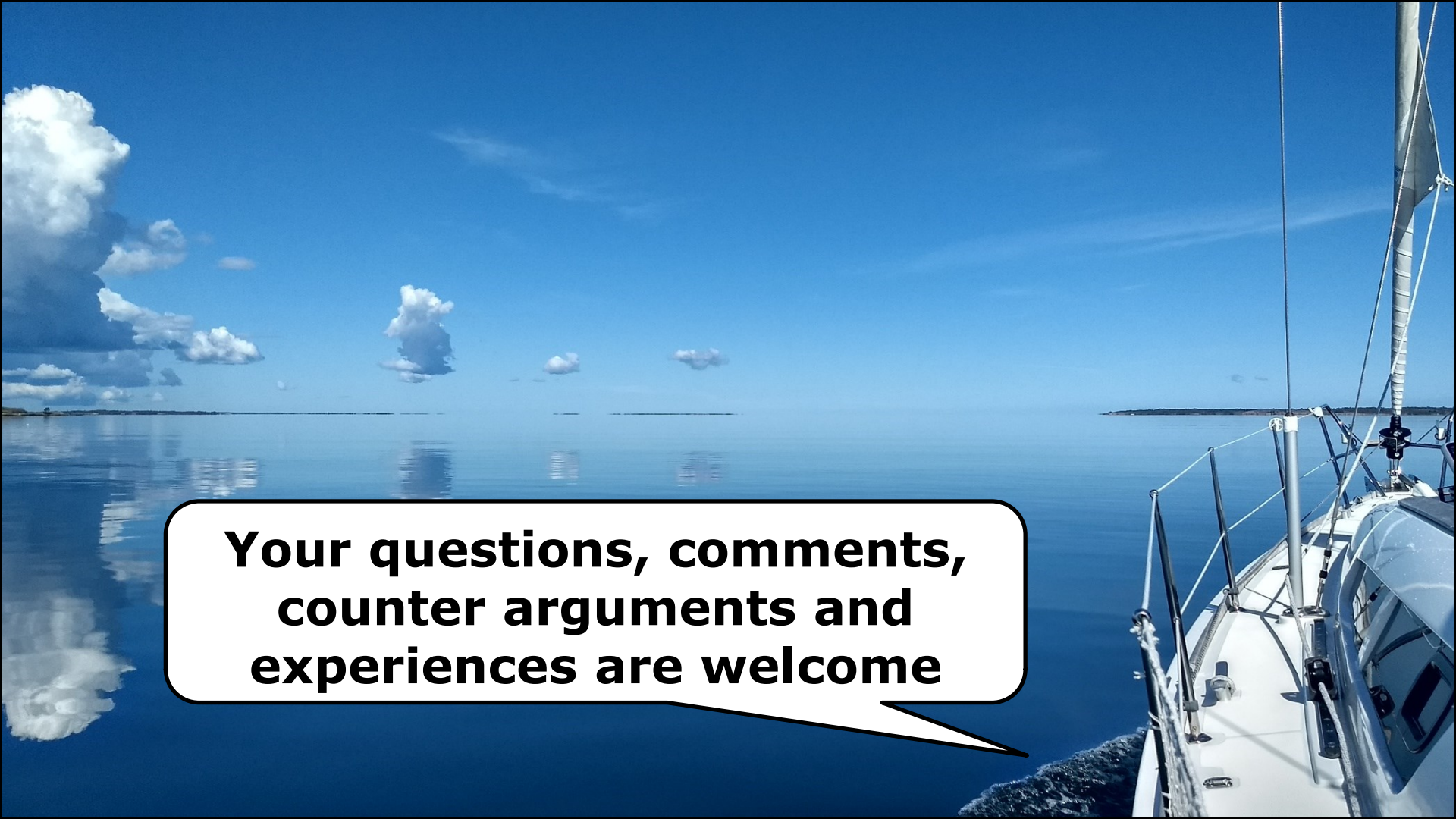
4 How to create a modeling language

How to start, language definition steps, exercise

5 Generators/transformations

How to, different approaches, examples, integration

6 Summary and discussion


Your questions, comments, counter arguments and experiences are welcome

1 Introduction

Languages and software architecture

- Languages to specify architectures, ADLs
- Architecture in the language

Domain-specific modeling languages

Languages and architecture

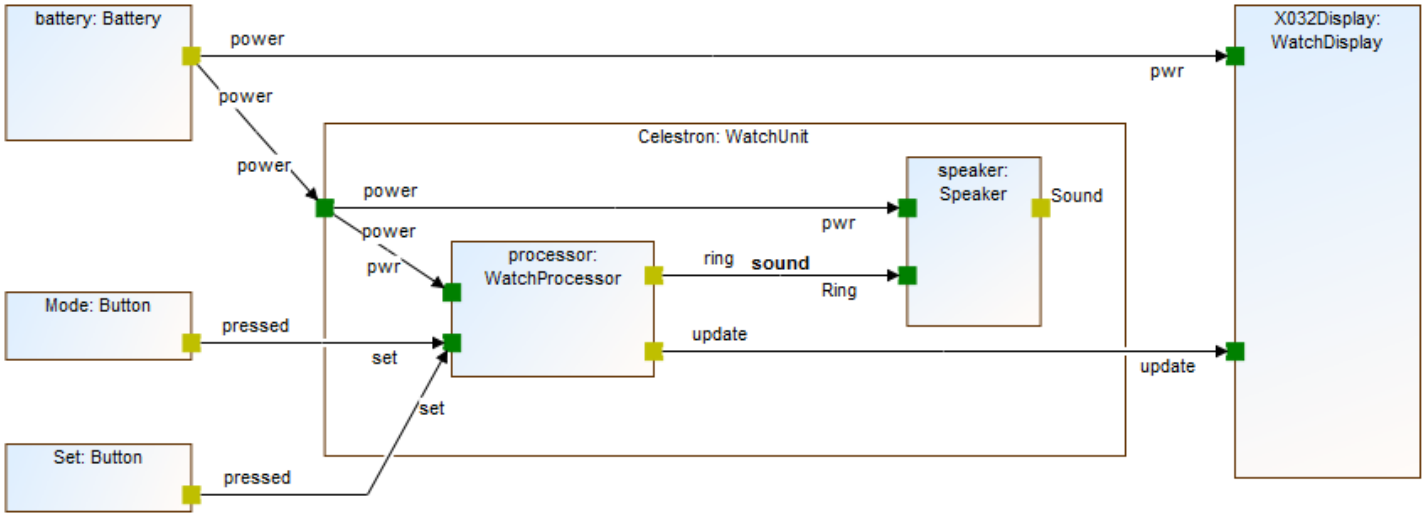
ADLs and a bit of background

- Various languages to design, analyze, document, simulate, integrate with other languages are proposed*
- No common understanding what should be presented, especially in behavior side from architecture
  - Various needs and requirements
- Different levels of formality
  - Informal, semi-formal, formal
- Different representation styles
  - Matrix, diagram, text, hybrid
- ADLs have typically fixed metamodel/grammar/structure
  - **In this tutorial we can fully change them, or create new**

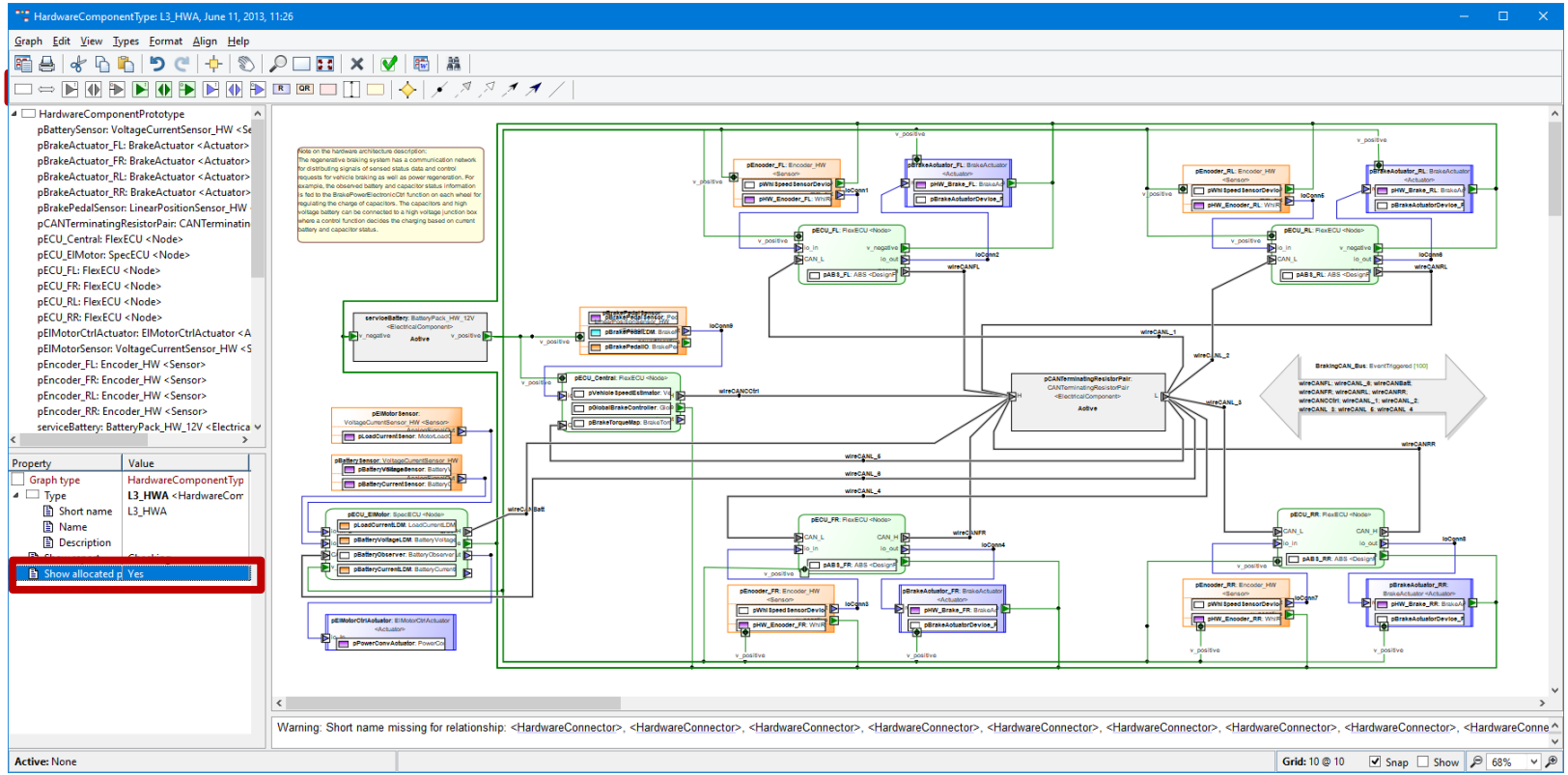
* Ozkaya, M. The analysis of architectural languages for the needs of practitioners. *Softw Pract Exper*. 2018

Common: Components & connections

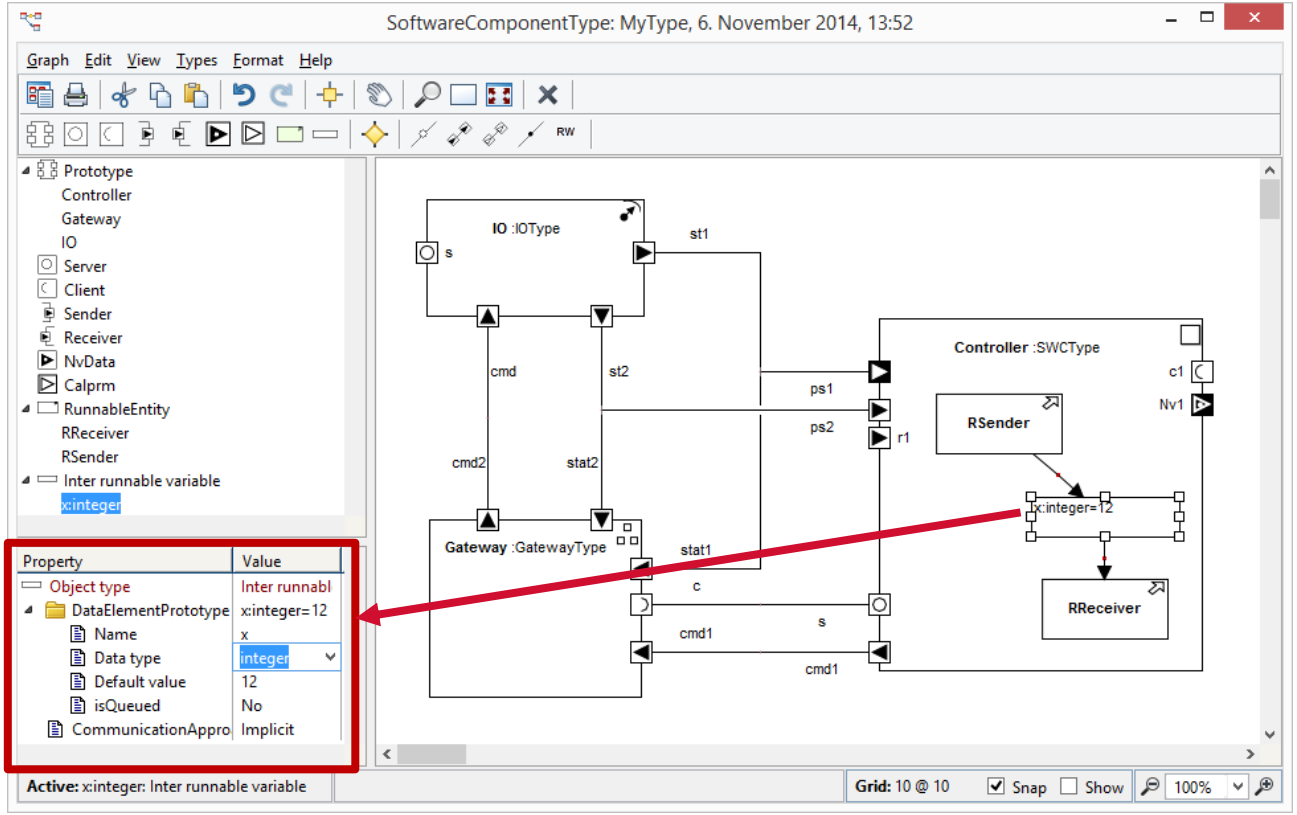
Others: +ports, +nesting, type system?



Example, HW function architecture (EAST-ADL in automotive)

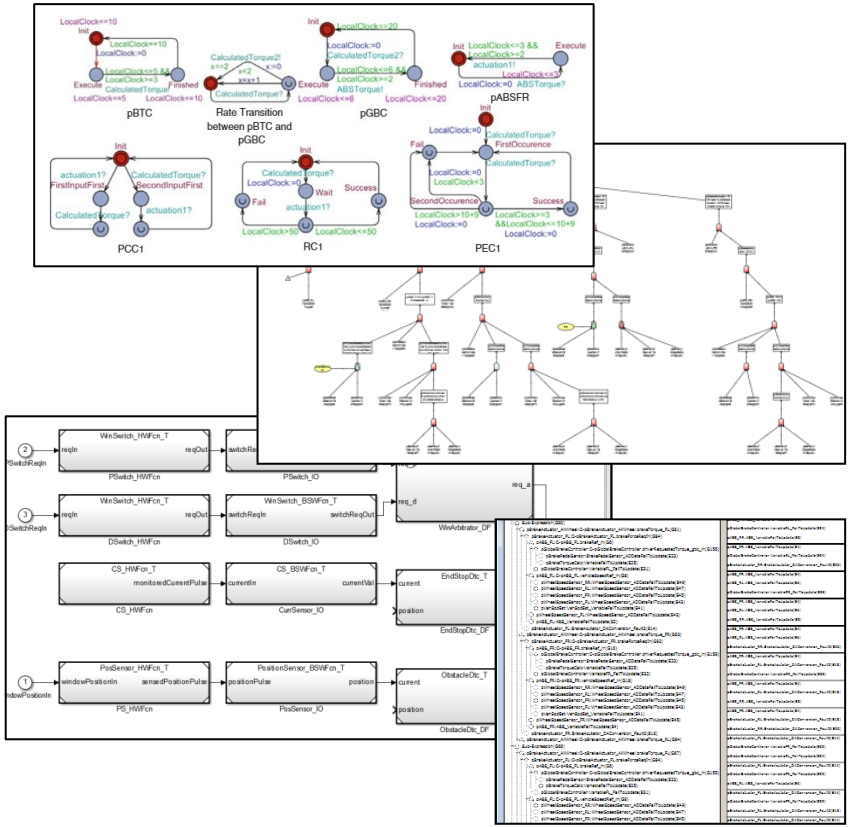


AUTOSAR
(automotive software architecture)



Narrow focus enables wanted usage

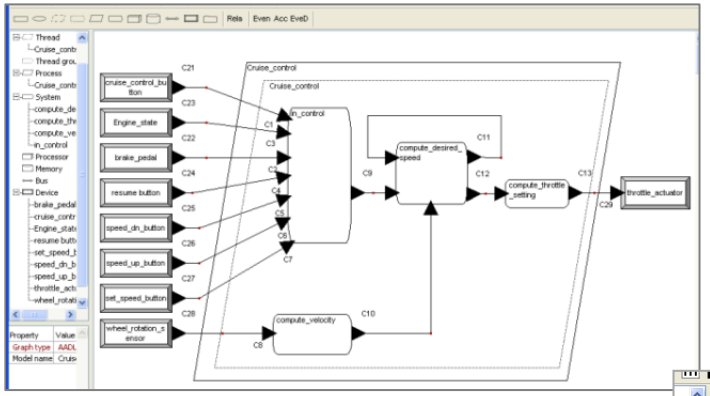
- EAST-ADL targeting automotive EE systems
  - Generate Simulink
  - Check with UPPAAL, SPIN
  - Support for functional safety Assist failure mode and effect analysis (FTA/FMEA)
  - Generate AUTOSAR (arxaml)
  - Trace with requirements
  - Integrate various views/designs
    - safety, security, variability, dependability

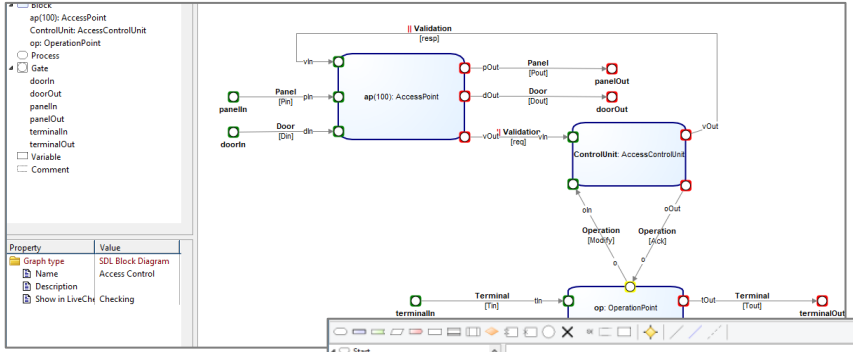


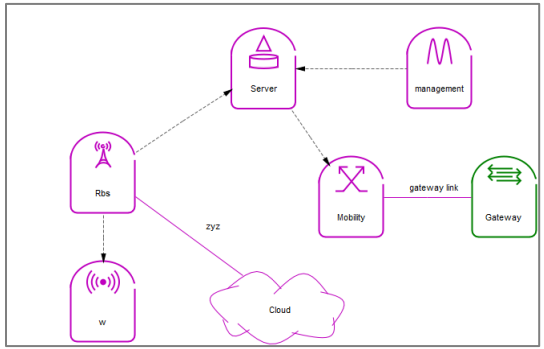
Some narrow, domain-specific ADLs

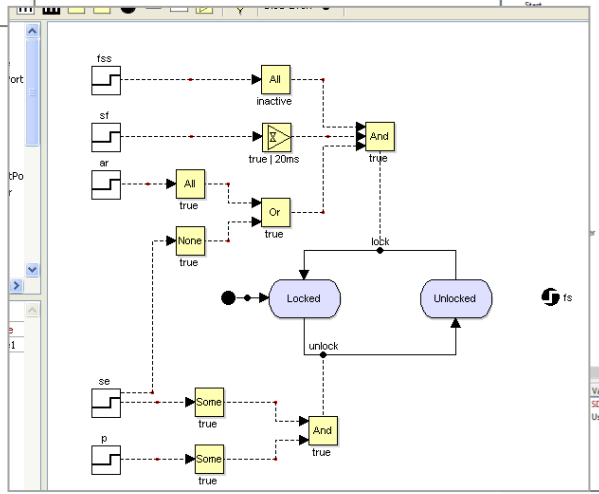
- Applied in the industry, developed by consortiums:
  - EAST-ADL
  - AUTOSAR
  - AADL
- Company-specific
  - Koala at Philips
  - Network architectures at Ericsson
  - Telecom system architecture at Nokia
  - Embedded software at Honeywell
  - Printer Data Path Architectures at Océ/Canon
  - Architecture definition at NASA

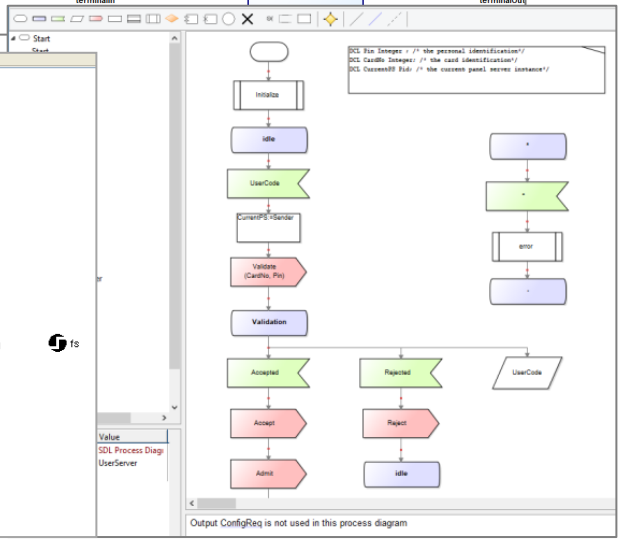
Some examples



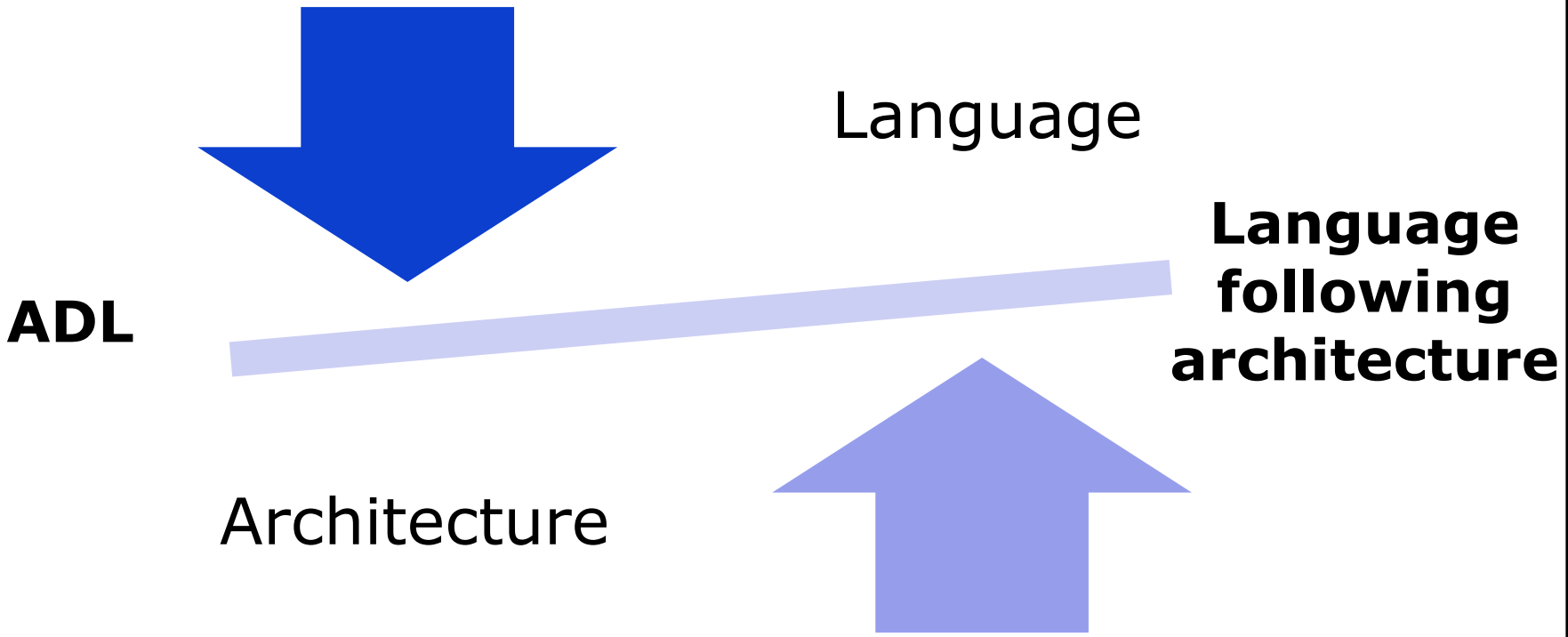








Architecture ⟷ Language



2 Cases and examples

ADLs

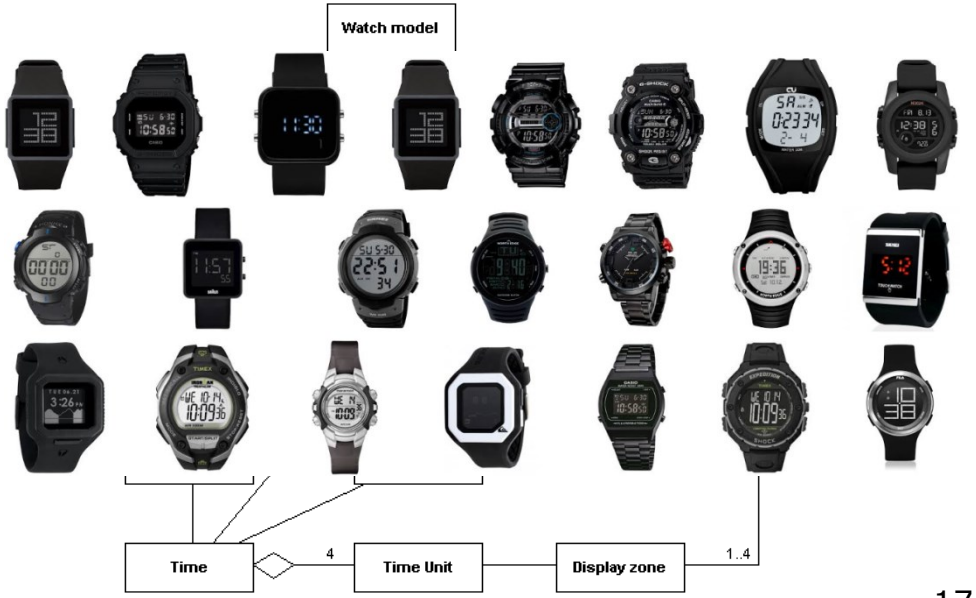
Architecture in the language

EAST-ADL, AUTOSAR,…

- Running demonstrations

Running example: watches

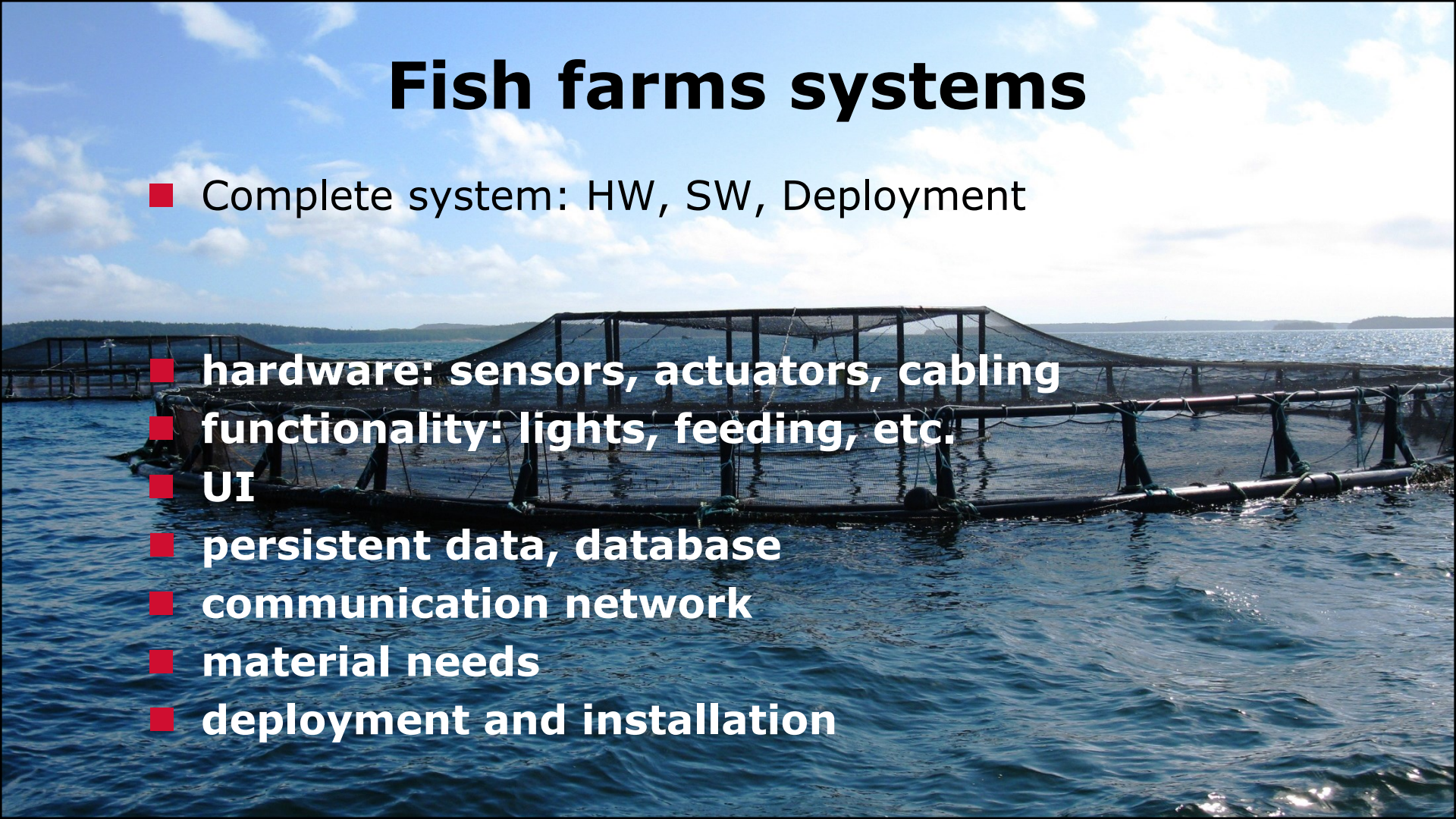
- Models: His, Hers, Sport, Kid, Traveler, Diver…
- Reusable component applications
  - Time, Alarm, Timer, WorldTime, StopWatch…
- Product contains:
  - Buttons
  - Icons
  - Time units
  - Alarms
  - Sound
  - Applications
  - Application behavior



Demo on watch products

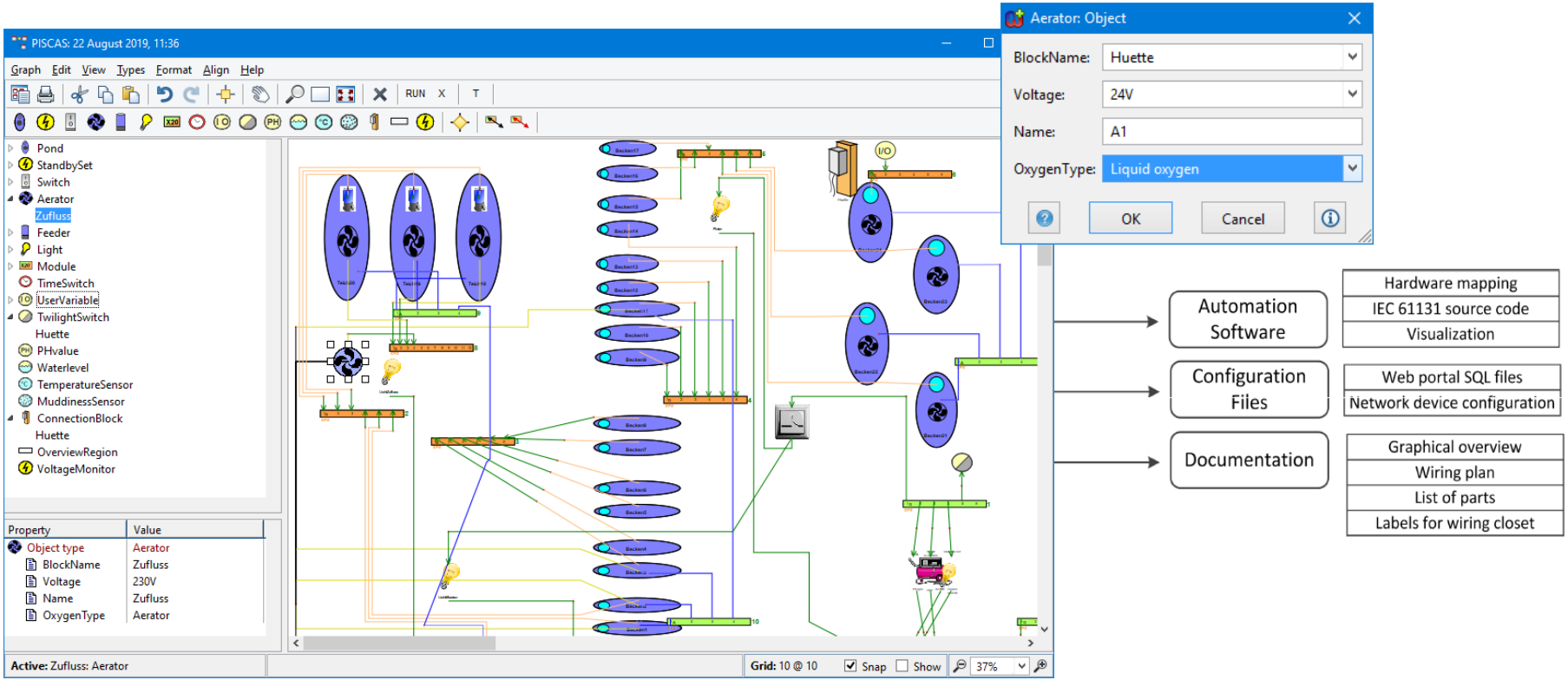
Fish farms systems

- Complete system: HW, SW, Deployment



- **hardware: sensors, actuators, cabling**
- **functionality: lights, feeding, etc.**
- **UI**
- **persistent data, database**
- **communication network**
- **material needs**
- **deployment and installation**

Fish farm automation system



Industry experiences on productivity increase with DSM

Panasonic

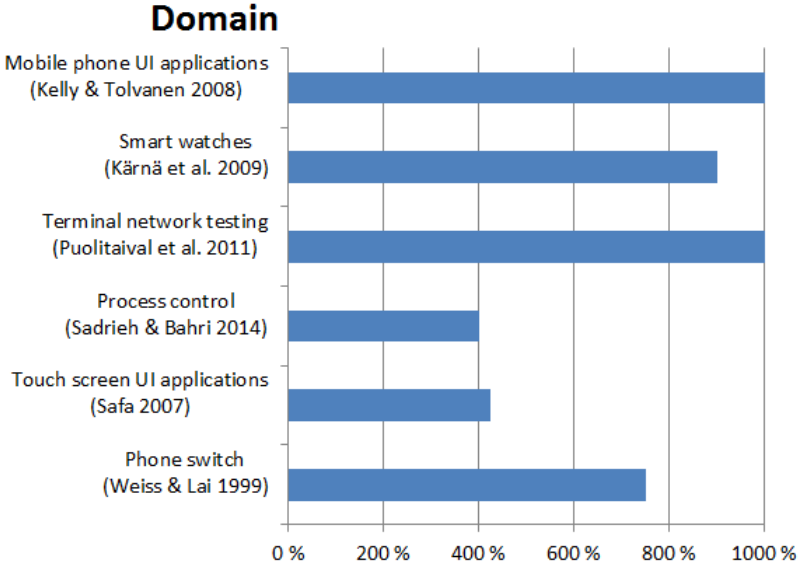
"**5-fold** productivity increase when compared to standard development methods"

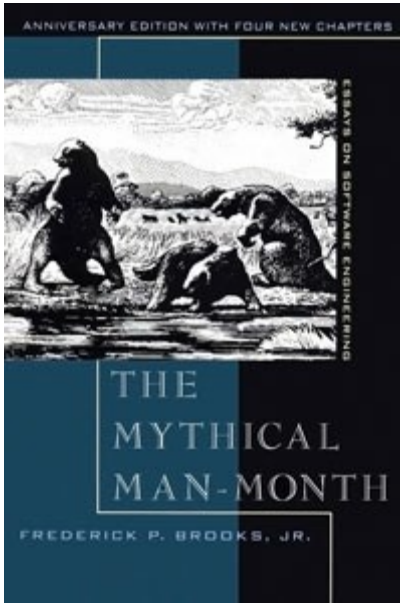
POLAR

"**750%** increase in developer productivity, and **greatly** improved code quality"

EADS

"The quality is clearly better, simply because the modeling language **rules out errors**"




ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS
ESSAYS ON SOFTWARE ENGINEERING
THE MYTHICAL MAN-MONTH
FREDERICK P. BROOKS, JR.

- Productivity seems constant in terms of elementary statements, a conclusion that is reasonable in terms of the thought a statement requires and the errors it may include.[11]
- Programming productivity may be increased as much as five times when a suitable high-level language is used.[12]

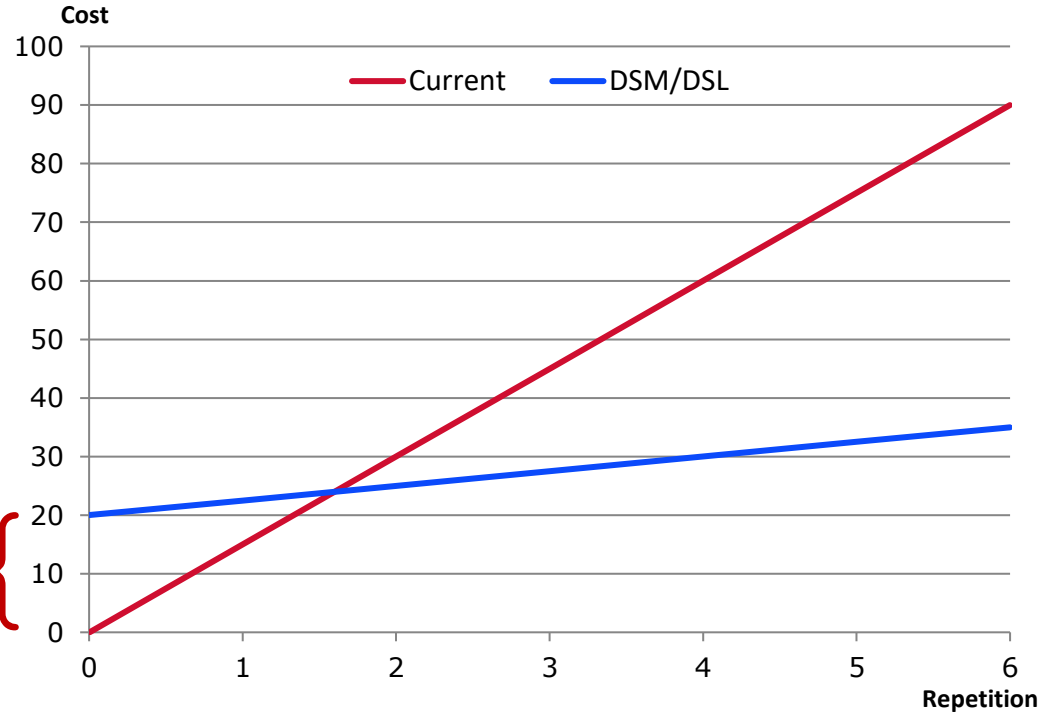
9
Ten Pounds in a Five-Pound Sack

https://archive.org/details/mythicalmanmonth00fred/page/94

Investment and ROI

Existing language
Customize language
Own language

- Investment



3 Elements of language

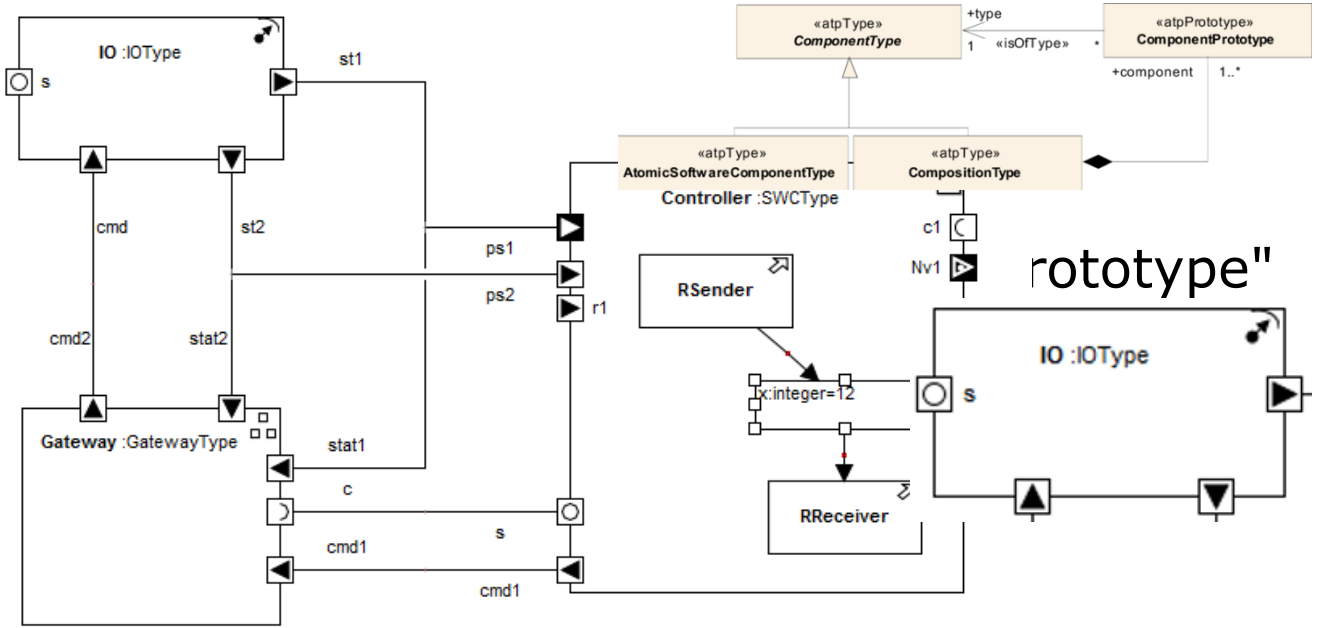
Abstract syntax, concrete syntax, semantics

Metamodeling

Tool support for language definition

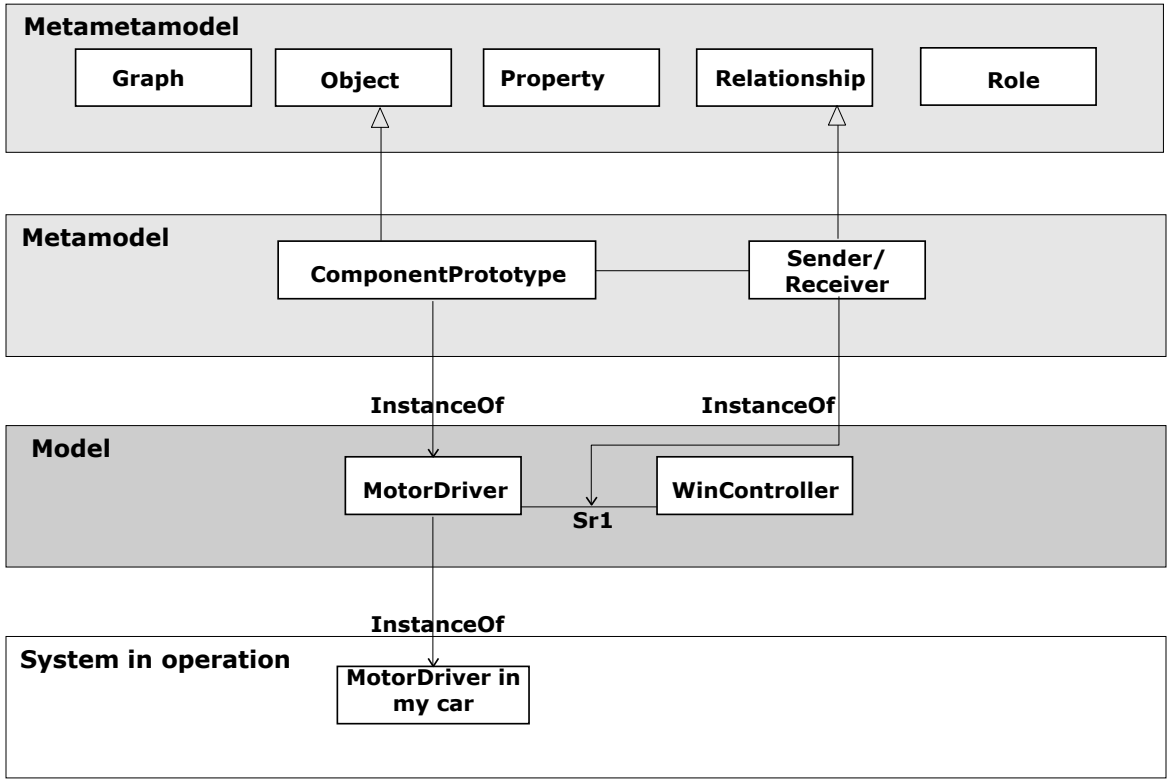
Language

- Language =
  abstract syntax + concrete syntax + semantics
- Abstra
  - defi
- Concre
  - not
- Semar
  - typi
    and

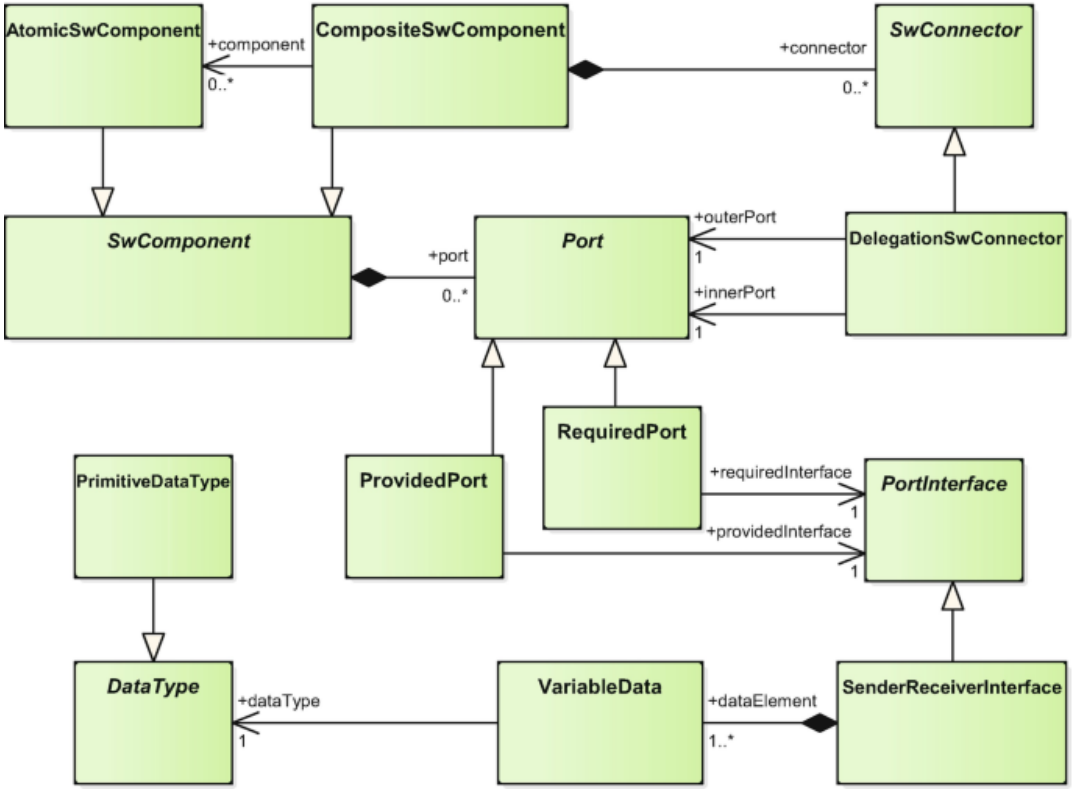


4 levels


«atpType»
ComponentType
+type
1
«isOfType»
*
«atpPrototype»
ComponentPrototype
+component
1..*
«atpType»
AtomicSoftwareComponentType
«atpType»
CompositionType


Metametamodel
Graph
Object
Property
Relationship
Role
Metamodel
ComponentPrototype
Sender/
Receiver
InstanceOf
InstanceOf
Model
MotorDriver
WinController
Sr1
InstanceOf
System in operation
MotorDriver in
my car

Metamodel example - overview

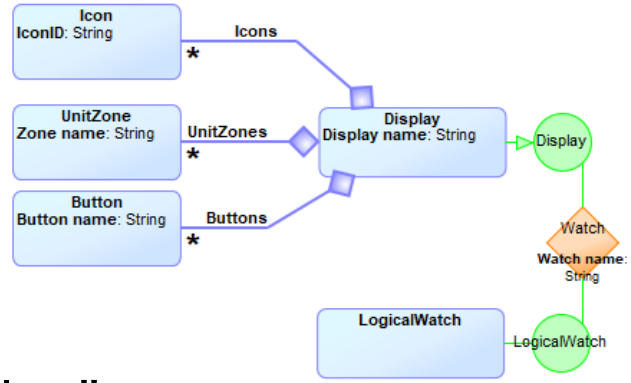


- + Details:
  - Properties
  - Constraints
    - Uniqueness
    - Mandatory
    - Naming conventions
    - Connections
  - Views/ sublanguages
  - Reuse

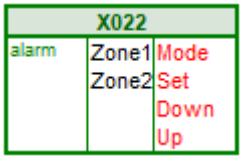
Language

- Language =
  abstract syntax + concrete syntax + semantics

- Abstract syntax
  - defined via a metamodel
- Concrete syntax
  - notational symbols
- Semantics
  - C, Java, C# generators

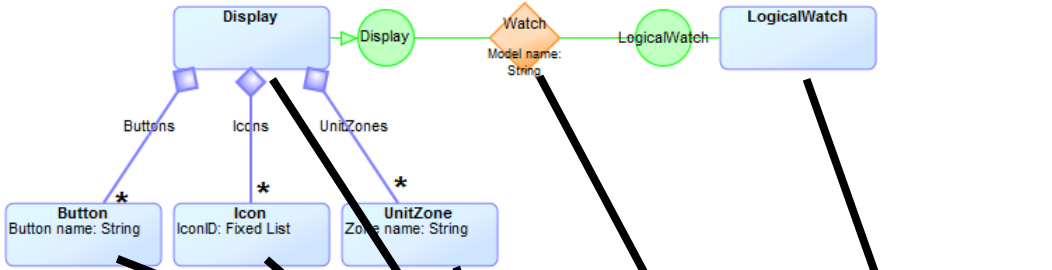


"Display"

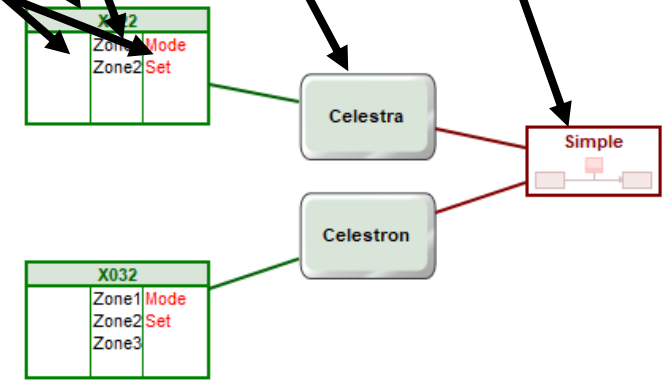


Example: Watch-Specific Language, 1

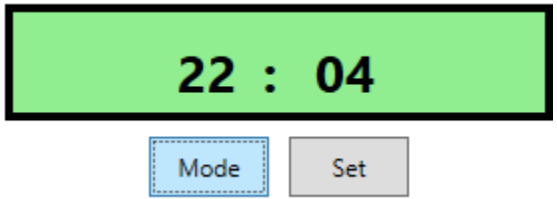
Metamodel



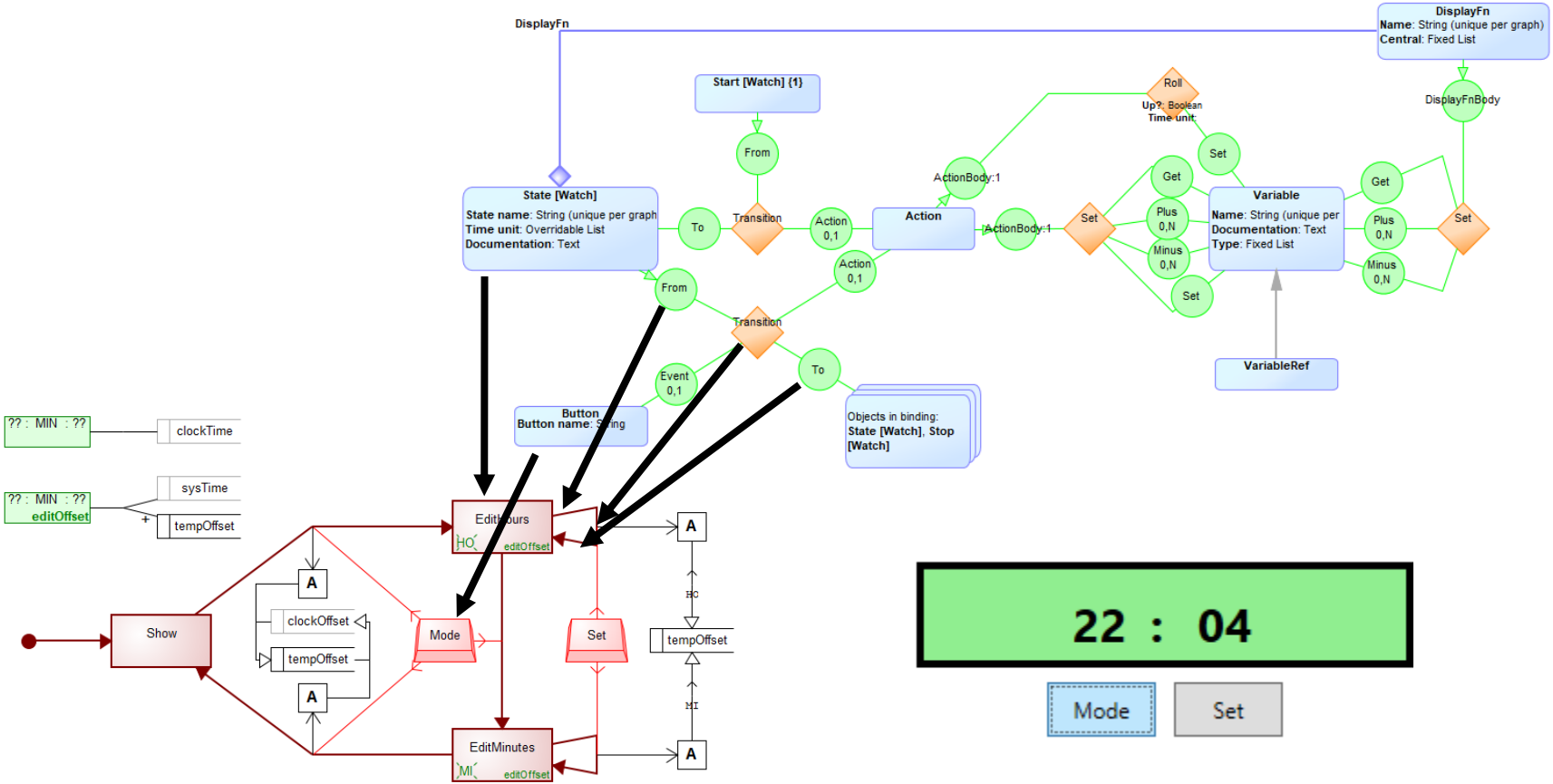
Model



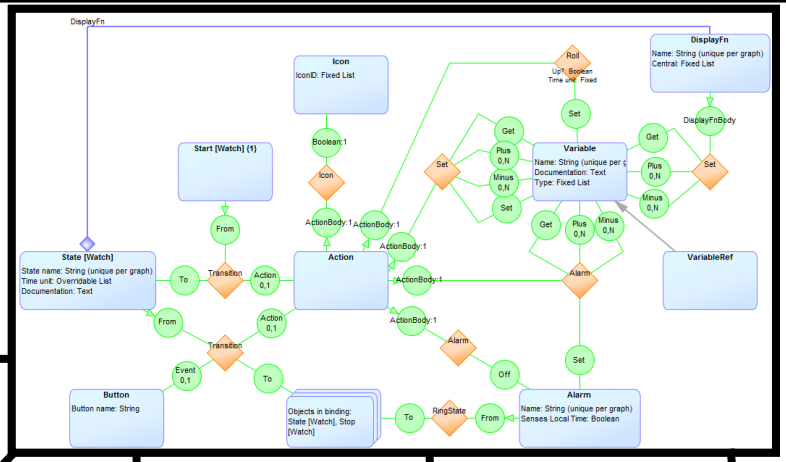
Celestra product

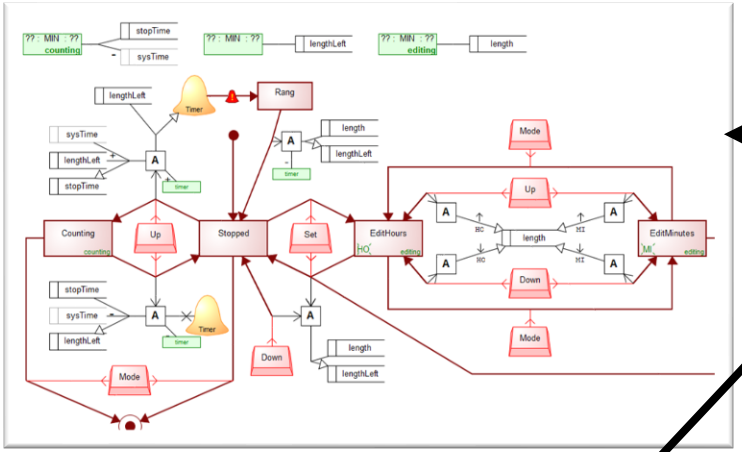


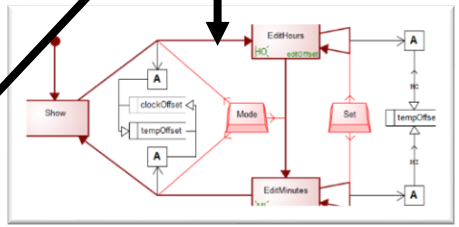
Watch-Specific Language, 2

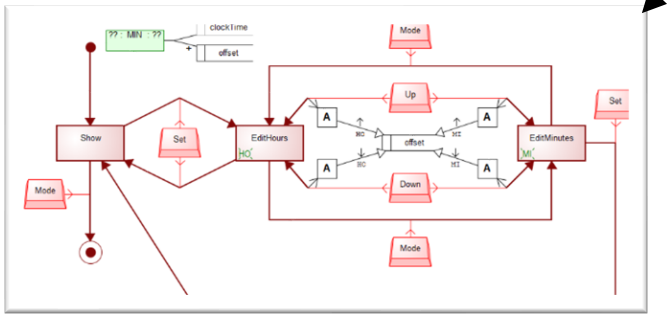


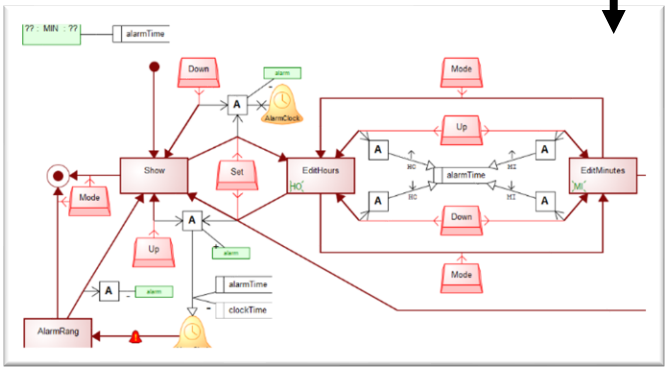
1 metamodel, many models

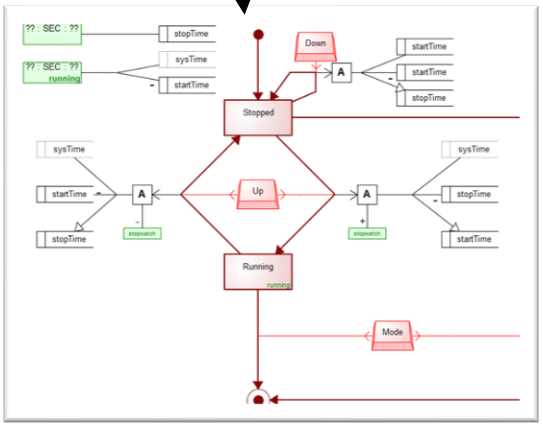




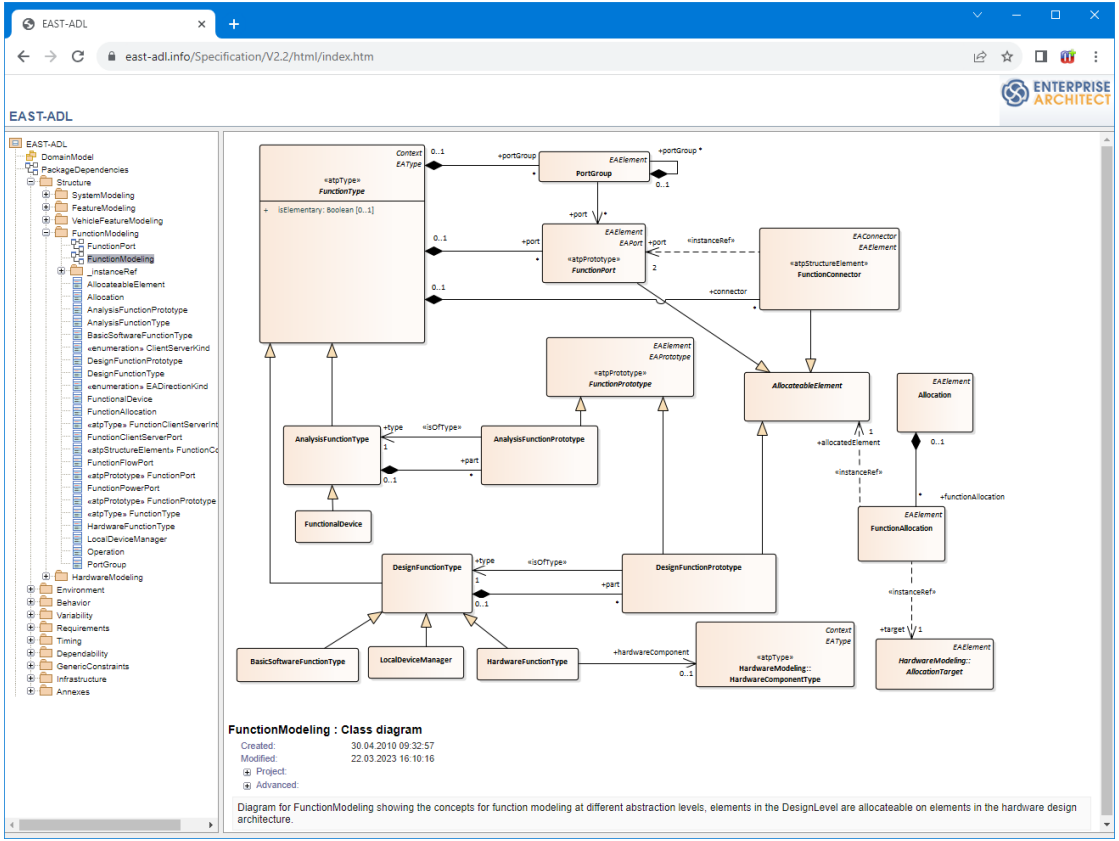








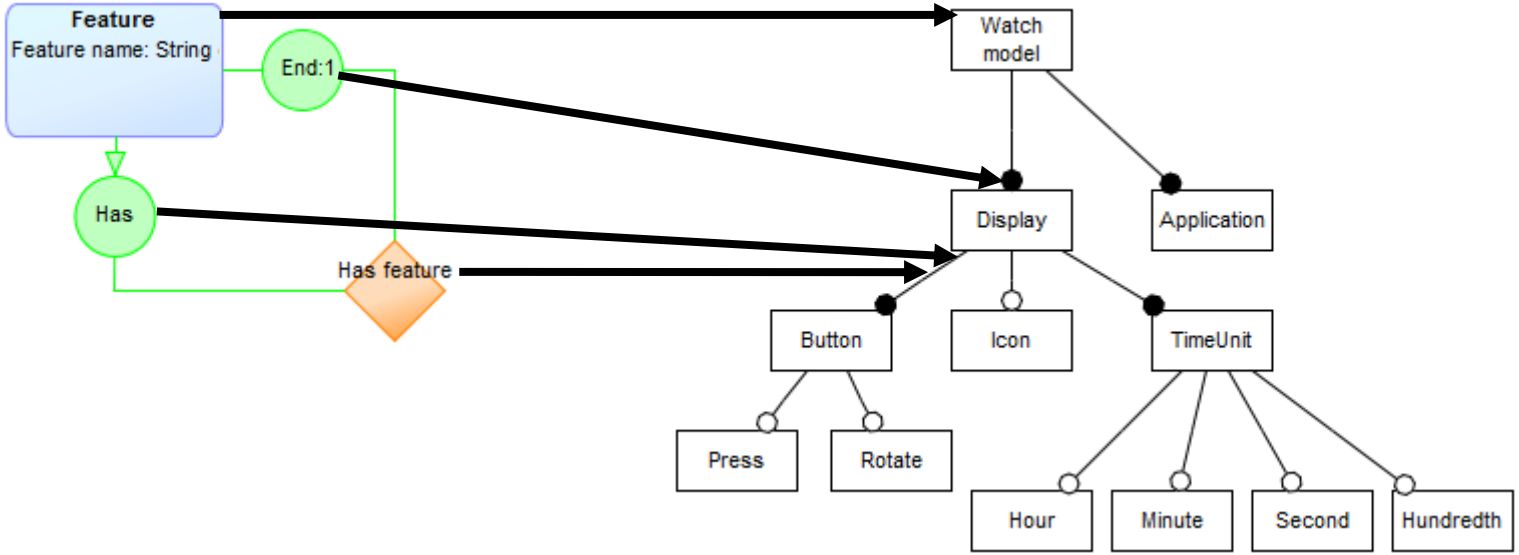
Example of a large metamodel



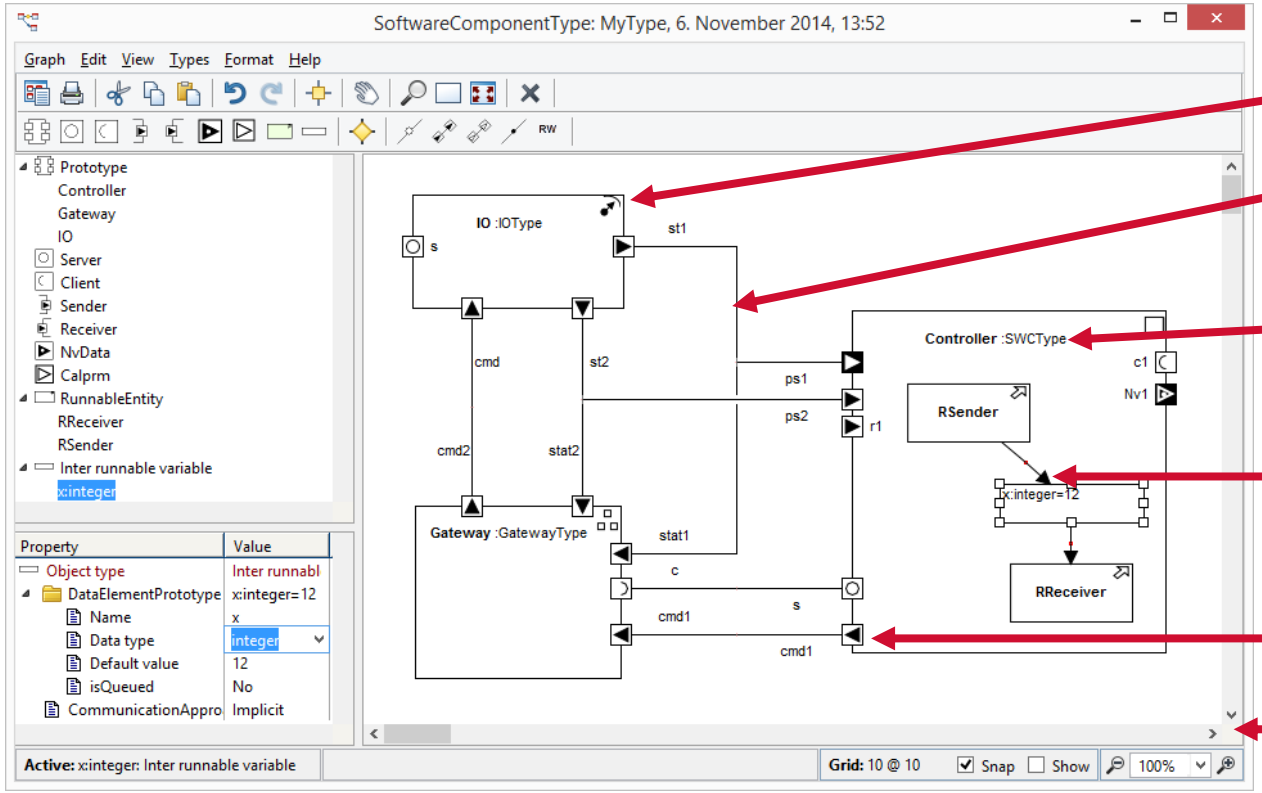
Feature model example

Metamodel

Model



Metamodel



Object/Class

Relationship/ Connection

Property/ Attribute

Role/ Association end

Port

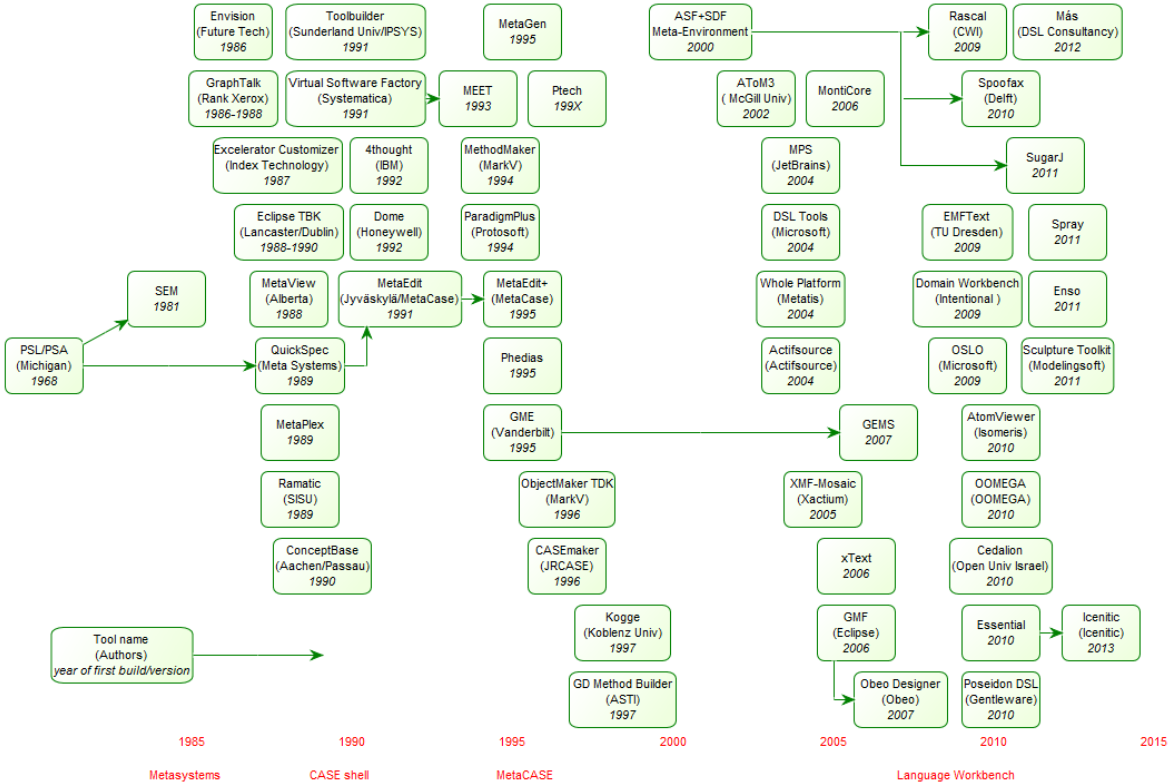
Graph/Language

Tooling to support your languages

- 6 ways to get the tools we need for DSM
  1. Write own modeling tool from scratch
  2. Write own modeling tool based on frameworks
  3. Metamodel, generate modeling tool skeleton, add code
  4. Metamodel, generate full modeling tool over a framework
  5. Metamodel, output configuration for generic modeling tool
  6. Integrated modeling and metamodeling environment

- Good tools minimize resource use (few man-weeks)
  - creating modeling tools and generators data-like, not code
  - guide in language definition
  - allow testing the language etc.

Various metamodeling tools

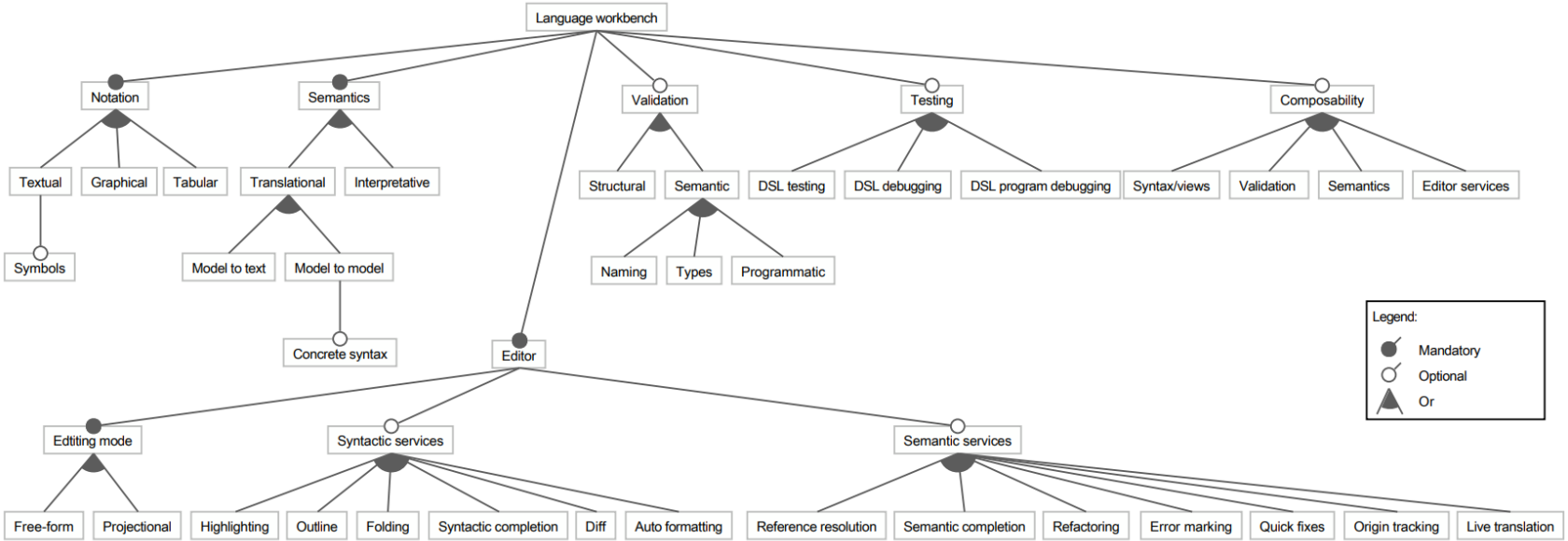


Some metatools

- Research
  - ConceptBase
  - (Web)GME
- Commercial
  - MetaEdit+
  - Microsoft DSL tools
- Open source
  - Eclipse (various frameworks and specific tools)
    - EMF, GEF, GMF, Sirius etc.
  - MPS
- See Erdweg et al. State of the Art in Language Workbenches for details: https://hal.inria.fr/hal-00923386

Features of tools

- Erdweg et al.



- + collaboration, evolution, versioning, scalability etc.

Key functions for a metatool, 1

1. Definition of metamodels
   - specify the concepts, rules, and symbols of individual modeling techniques as well as their interconnection rules.
2. Create/provide modeling tools
   - different kinds of editors, toolbars, dialogs, help, etc.
3. Repository/storage format
   - to store the models based on the new language
4. Maintenance and evolution support
   - change metamodels and models
   - modify tool support

Key functions for a metatool, 2

5. Definition for model transformations/generators
   - definition of various model analyses, checking, code generation and model documentation reports

6. Metamodel management
   - Similar to model management
     - browsers, documentation tools, libraries for metamodels, backups, versioning, configuration management and access rights for language specifications or for their parts

Key functions for a metatool, 3

7. Management of language updates
   - Transformation rules between language versions
     - e.g. v1.5 to v2.0
   - Update designs (semi-)automatically to correspond to the new language version

8. Interchange format
   - Importing and exporting of both models and metamodels
     - safety, avoid tool locking
   - Importing should be incremental:
     - previously imported data from the same exporter should be updated automatically, rather than creating duplicates

4 How to create a modeling language

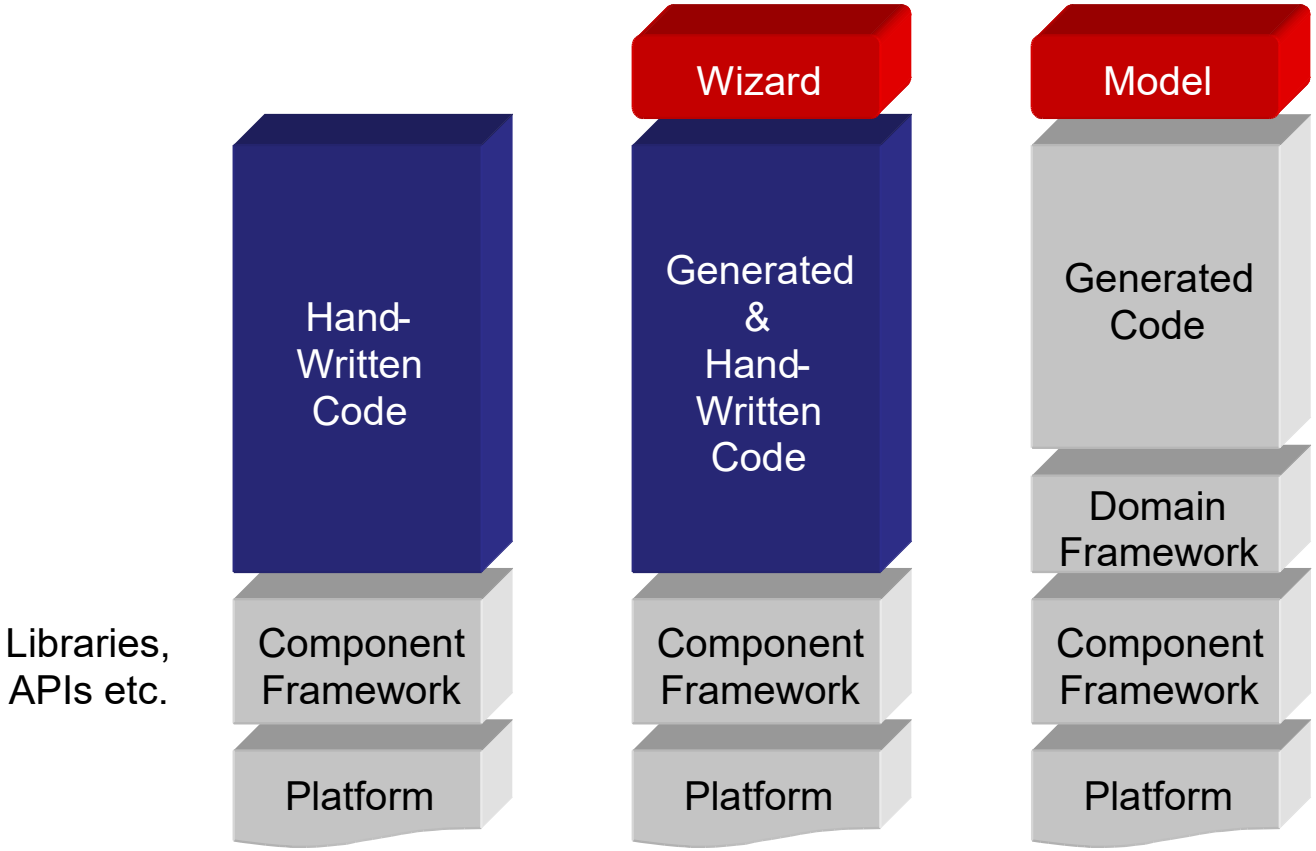
How to identify language concepts and rules
Defining language via metamodels
Defining notations
Language creation exercise

Language definition steps

1. Identify abstractions
   - Concepts and how they work together
2. Specify the metamodel
   - Language concepts and their rules
3. Create the notation
   - Representation of models
4. Define the generators
   - Various outputs and analysis of the models

- The process is iterative: try solution with examples
  - Define part of language, model with it, define more...

How to find language concepts?



Approaches to identify DSL concepts

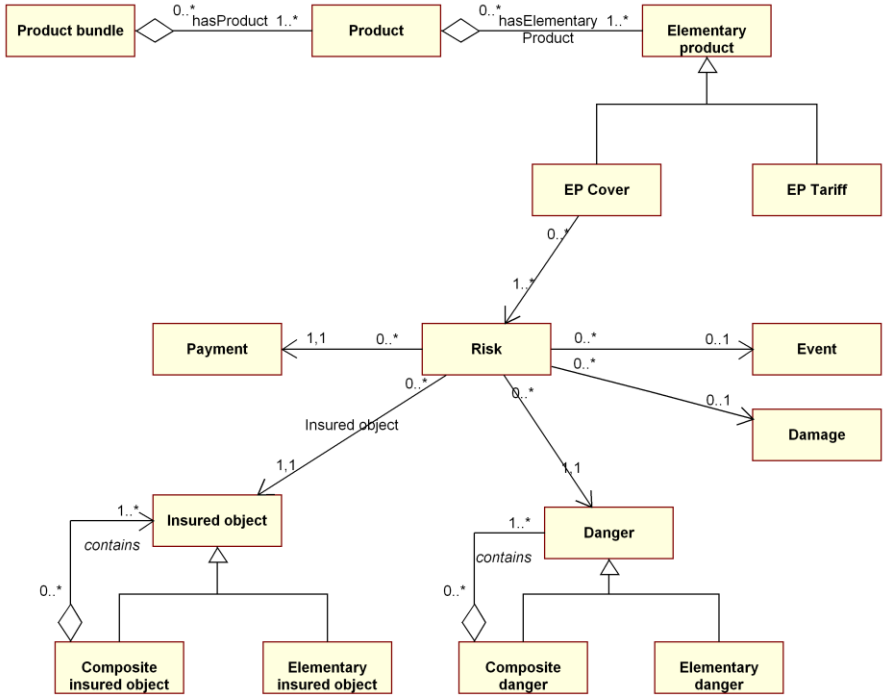
- "How do I start creating language?"
  - Hard problem for beginners
  - Analyzed 23 cases to find good toolbox of approaches

- Initial analysis suggested five approaches:
  1. Domain expert's or developer's concepts
  2. Generation output
  3. Physical structure
  4. Look and feel of the system built
  5. Variability space

Problem domain	Solution domain/ generation target	Approach
Telecom services	Configuration scripts	1
Insurance products	J2EE	1
Business processes	Rule engine language	1
Industrial automation	3 GL	1, (2)
Platform installation	XML	1, (2)
Medical device configuration	XML	1, (2)
Machine control	3 GL	1, 2
Call processing	CPL	2, (1)
Geographic Information System	3 GL, propriety rule language, data structures	2
SIM card profiles	Configuration scripts and parameters	2
Phone switch services	CPL, Voice XML, 3 GL	2, (4)
eCommerce marketplaces	J2EE, XML	2, (4)
Automation network	C	3, 4
Crane operations	C/C++	3, (5)
SIM card applications	3 GL	4
Applications in microcontroller	8-bit assembler	4
Household appliance features	3 GL	4
Smartphone UI applications	Scripting language	4
ERP configuration	3 GL	4, 5
ERP configuration	3 GL	4, 5
Handheld device applications	3 GL	4, 5
Phone UI applications	C	5, (4)
Phone UI applications	C++	5, (4)

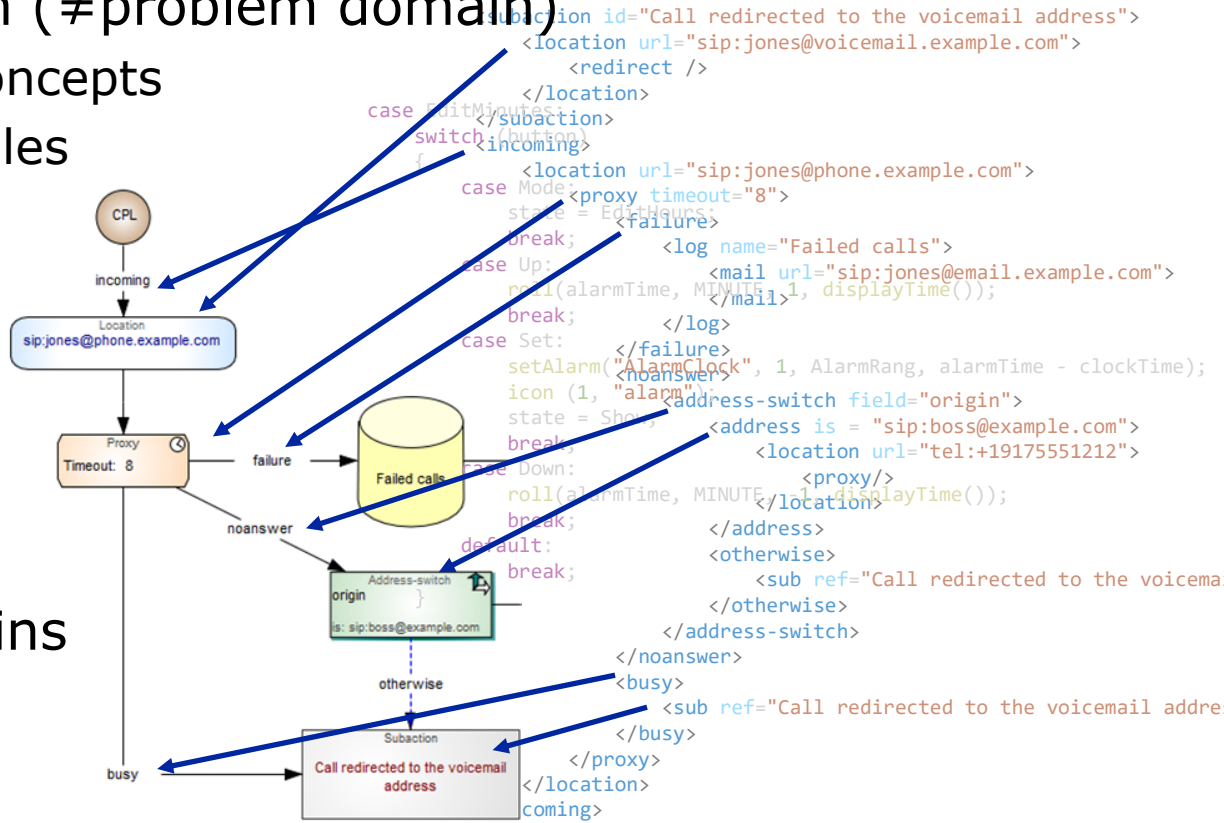
1. (Some) domain concepts exists

- A good start, but needs revision as often differs from metamodel/grammar
  - Lack details
  - Few constraints only
  - No consideration of reuse
  - No concrete syntax
- Refine with examples
  - legal
  - illegal



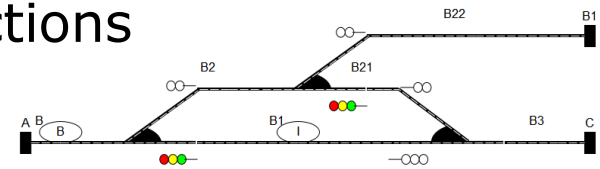
2. Generation output

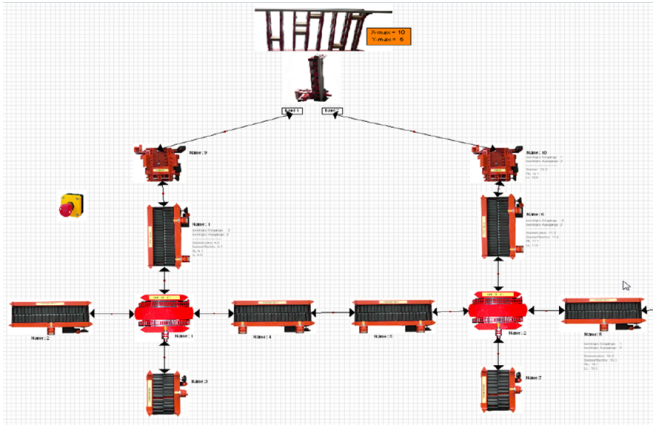
- Low abstraction (≠problem domain)
  - No domain concepts
  - No domain rules
  - Notation?
- Danger: Little productivity gains

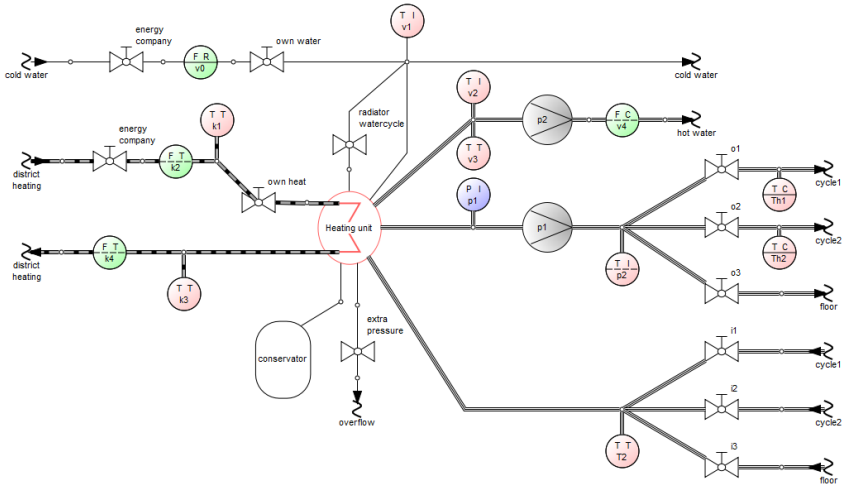


3. Physical structure as a base

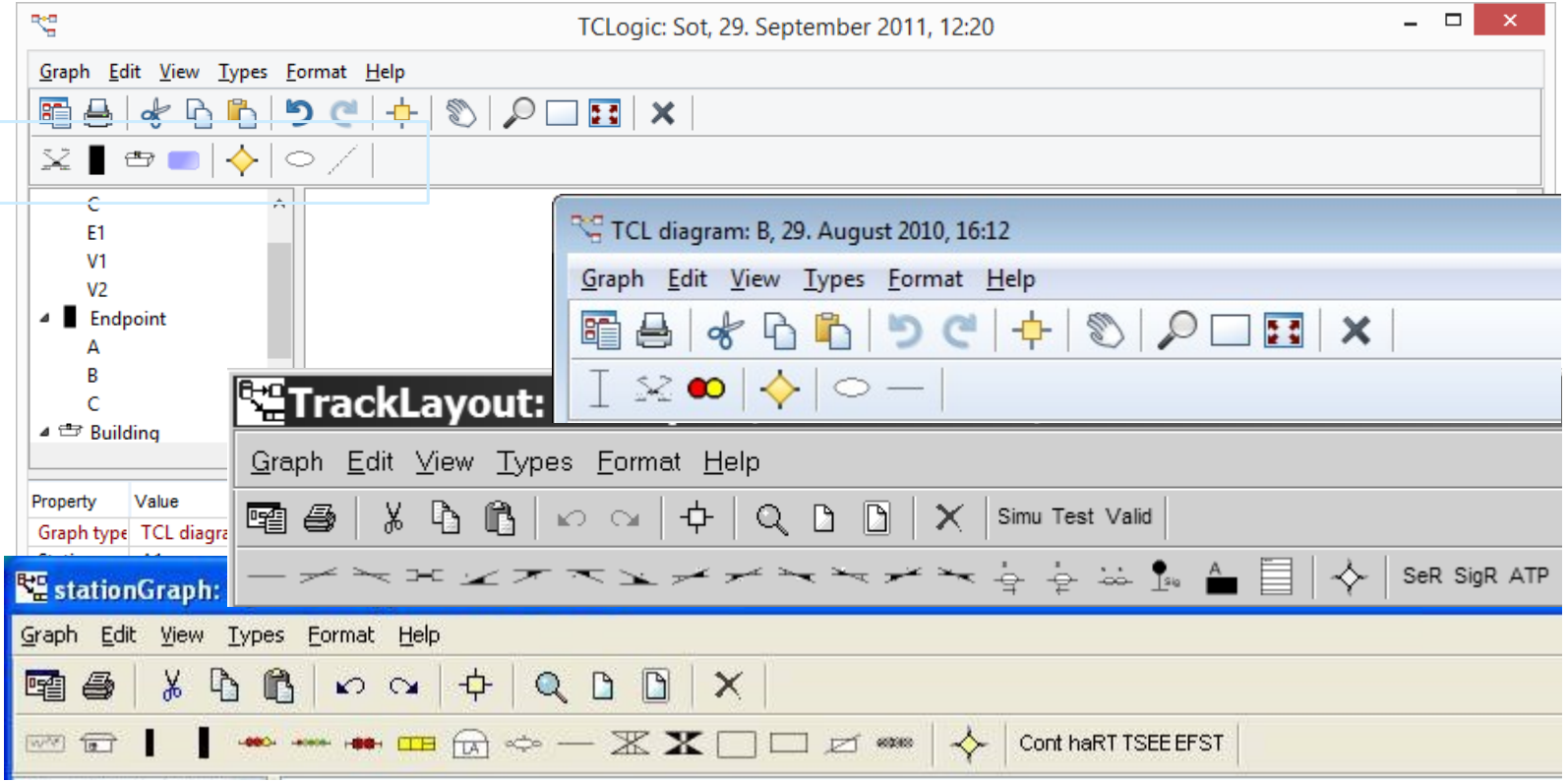
- Great as mimic higher level of abstractions
  - Do not cover constraints
  - Enable creating different examples
  - Easier for external language engineers
  - Suggest a notation







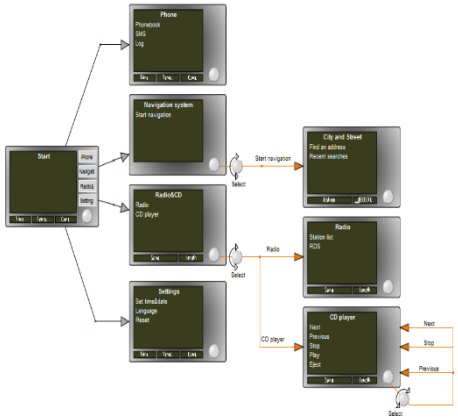
One DSL per domain?

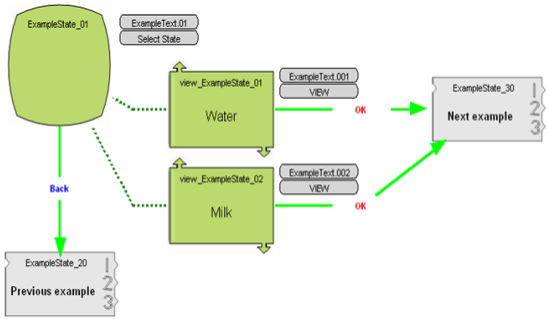


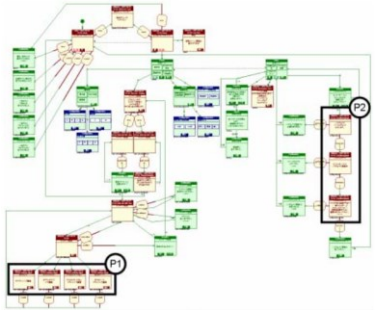
4. Look and feel of end system

- High level of abstraction
  - Domain concepts visible
  - Notation can mimic the "real world"
  - Finalize by applying all UI concepts in examples
- Often state machine as a basis
  - Extend with data & control flow

4. Look and feel of end system

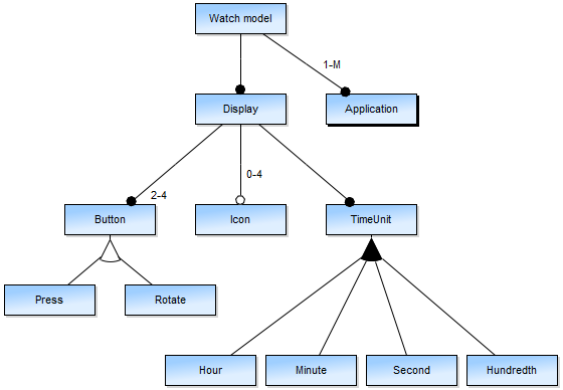






5. Variability space

- Domain Engineering
  - Language concepts capture variability space
- Modeler makes variant choices
  - Composition, relationships, values
- Infinite variability space (Czarnecki)
  - Not just feature tree: unbounded product family
- Static variance easy,
  dynamic harder
- Predict future variability
  → high level of abstraction



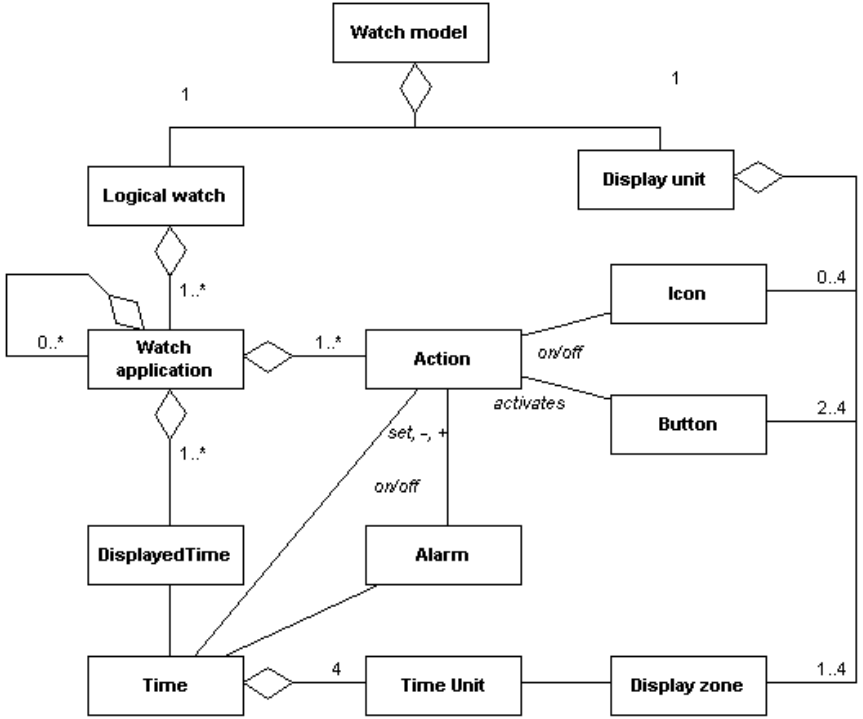
Task 1

- We need to support different button pressing policies -- other than single button press

- Q: How would you change the language?

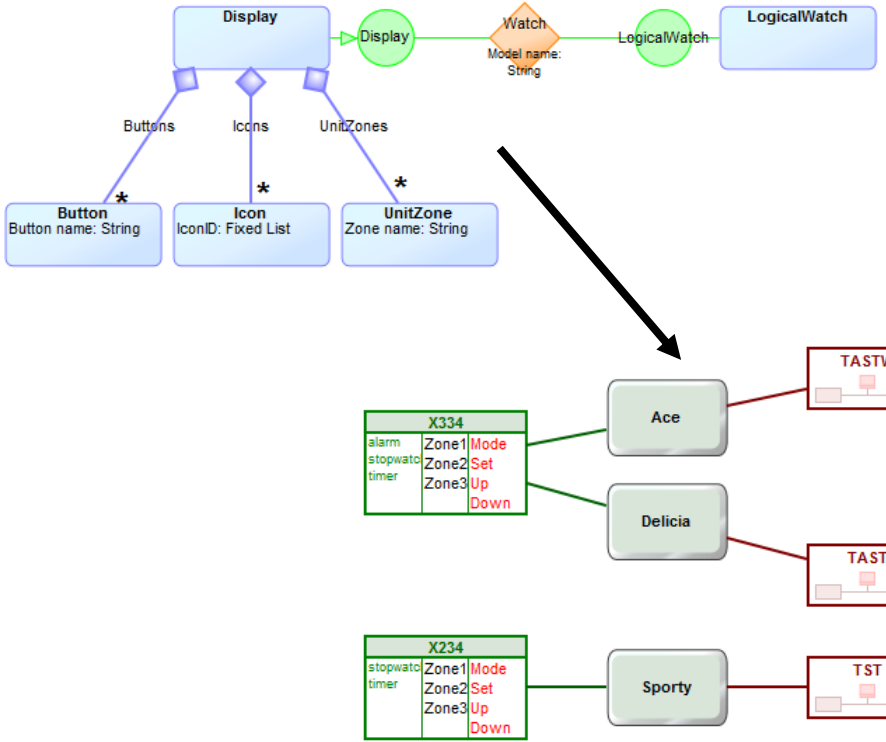
Back to our watch product line

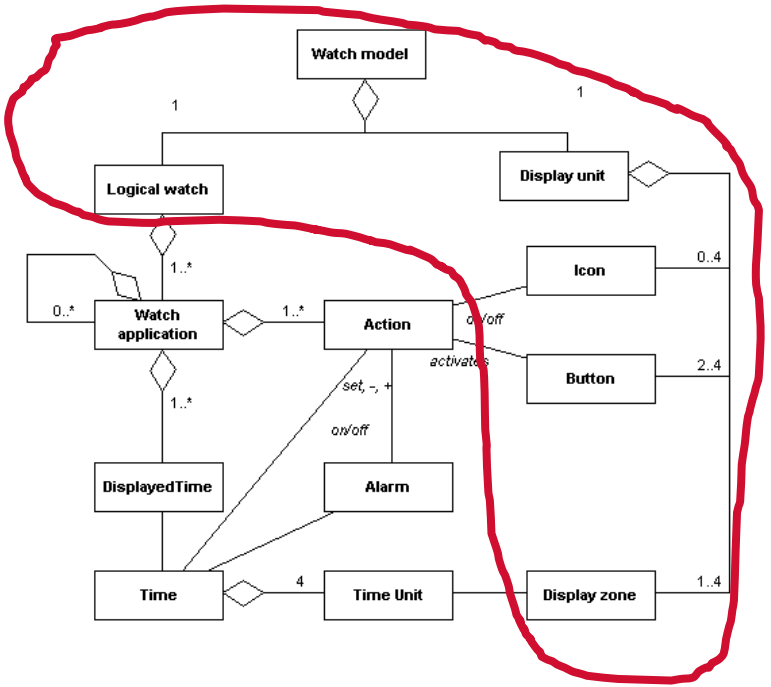
1. Understand the domain (conduct domain analysis)
2. Identify variation
3. Map domain concepts and variation to DSL



Two integrated languages: Structure and Applications

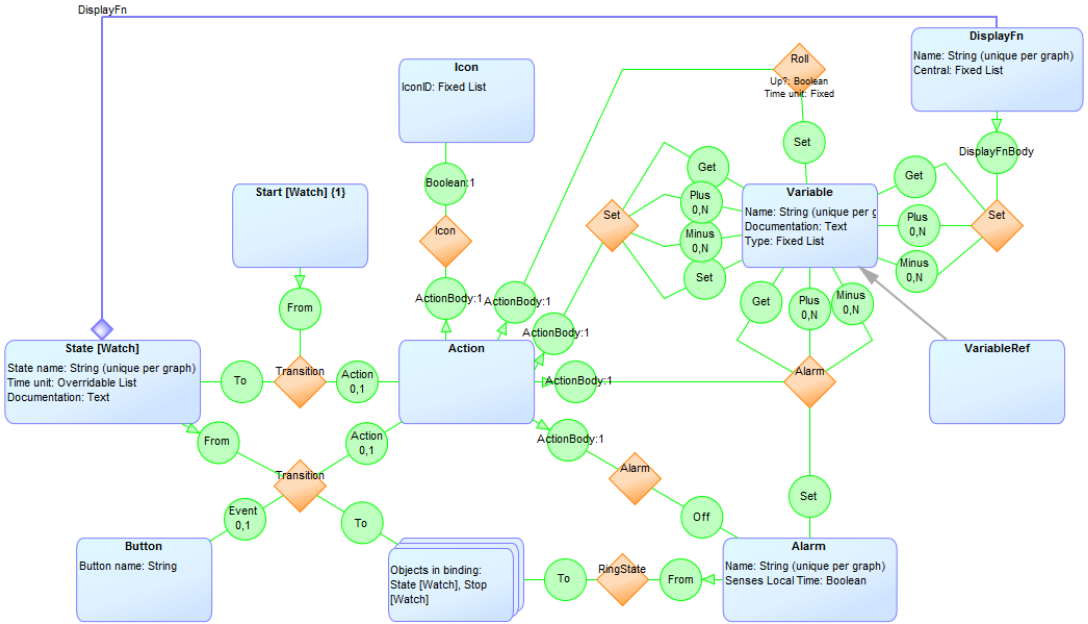
- Structure

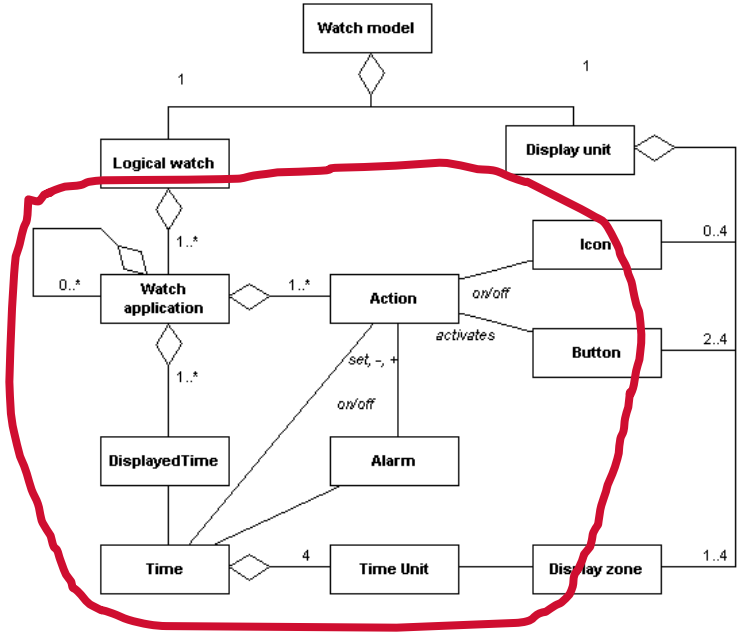


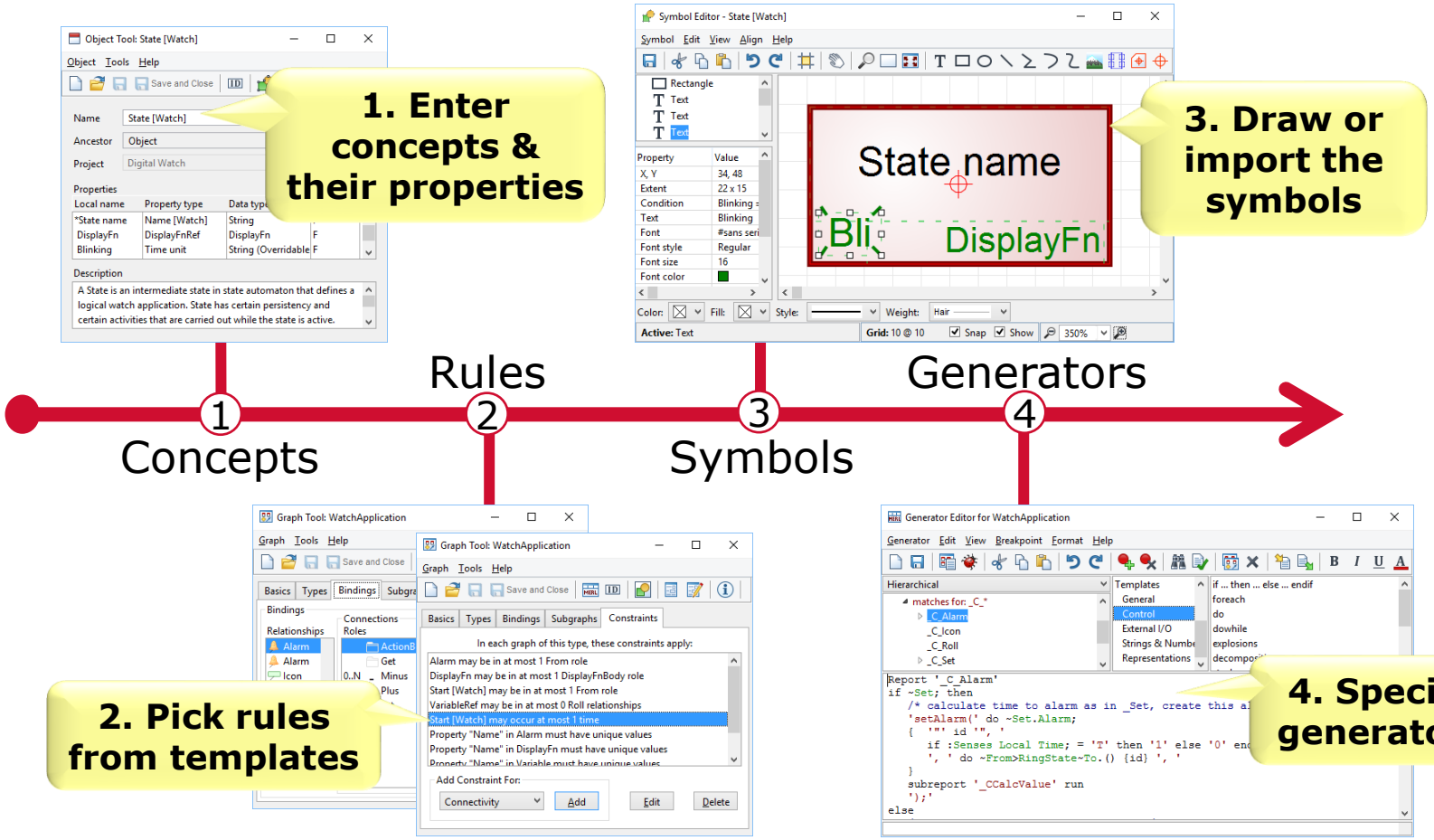


Two integrated languages: Structure and Applications

- Applications and behavior

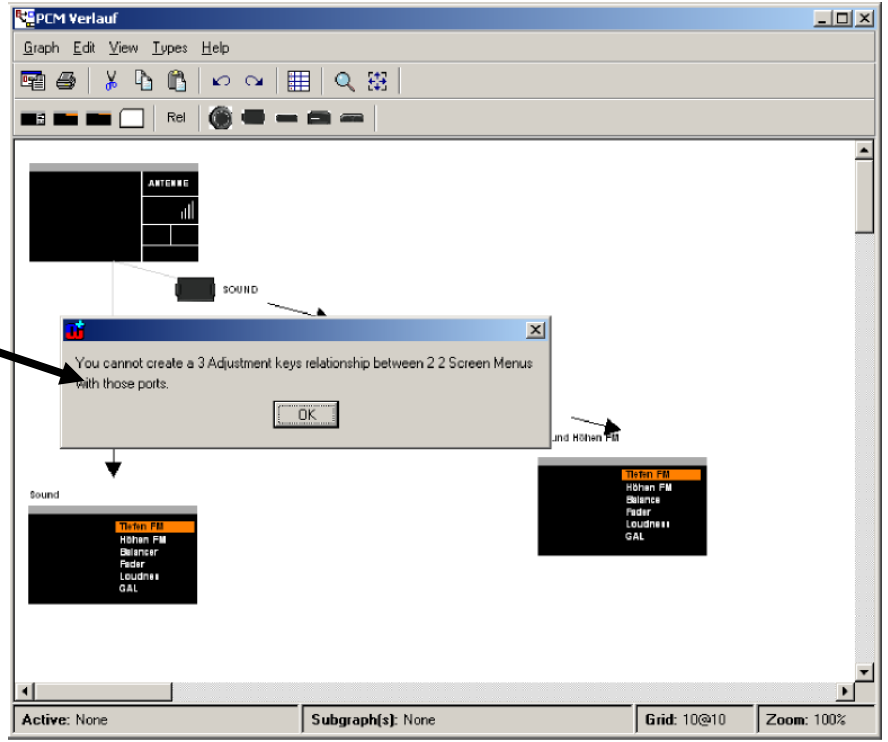





1. Enter concepts & their properties
3. Draw or import the symbols
Rules
Generators
1
2
3
4
Concepts
Symbols
2. Pick rules from templates
4. Specify generators

Language implementation

About following domain rules in a language

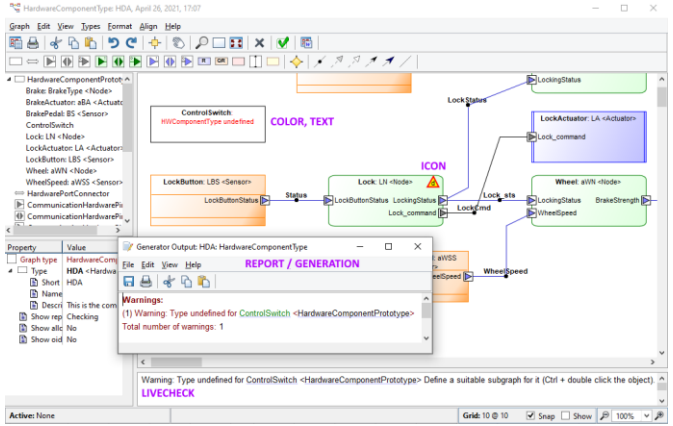


Rules [1/2]

- Language definition can cover the rules of the domain
- Complete and correct models are relevant for product derivation (code generation)
- Putting the domain rules into the language allows
  - preventing creation of illegal models
  - informing about missing data
  - keeping models consistent
- Prefer having rules as part of metamodel to having separate checker
  - Support early error prevention and provide guidance
  - But going overboard can hinder flow of modeler

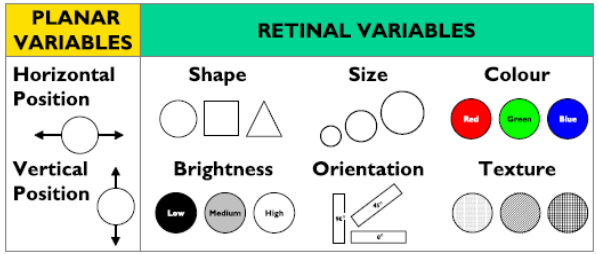
Rules [2/2]

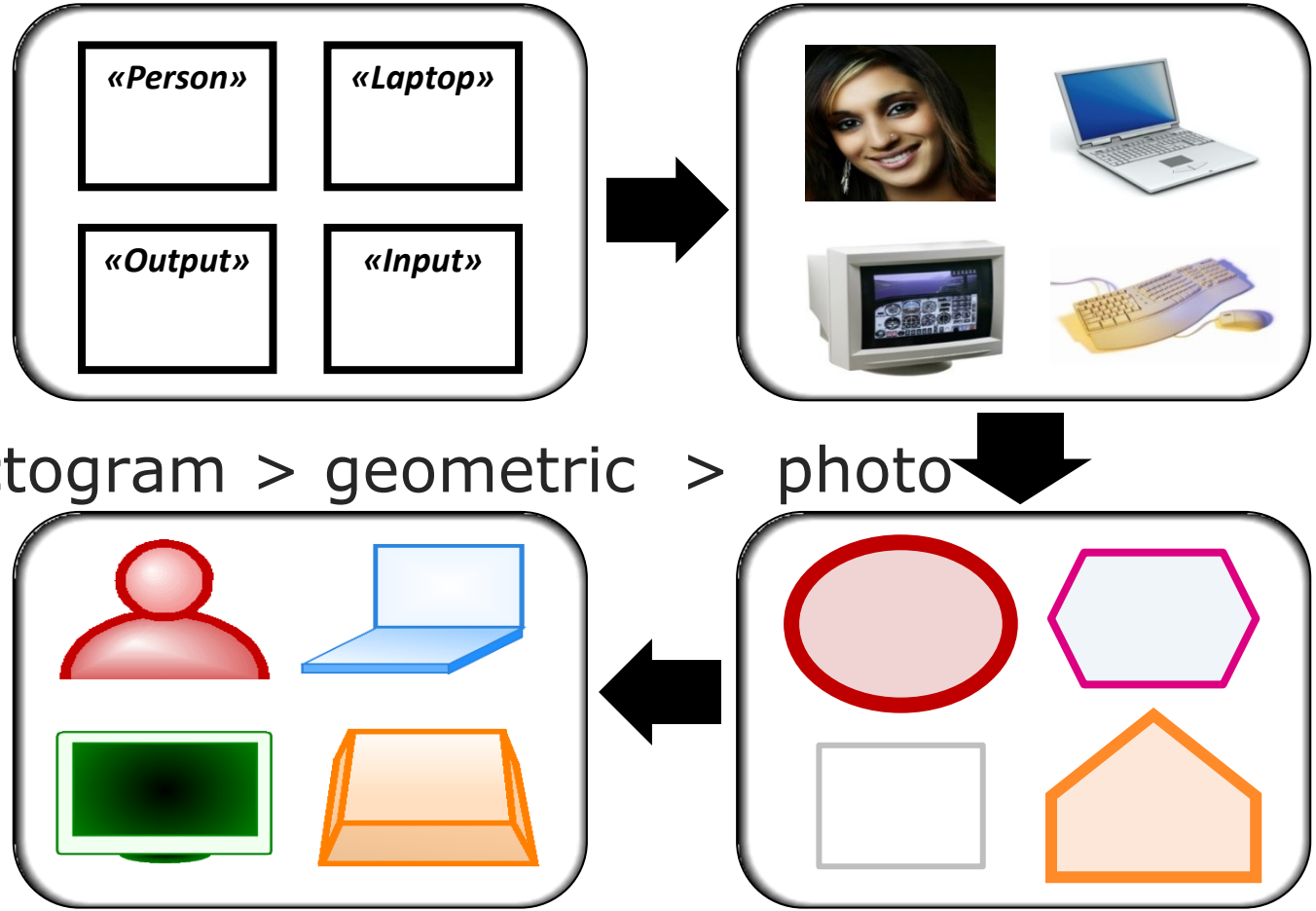
- How rules are visible for modelers
  - During modeling action
  - Inform when illegal design is made
  - In a separate model check window
  - By highlighting element(s) with errors or missing data
- When to run a separate model checking
  - Whenever wanted
  - After certain model editing actions
  - Before code generation
  - Show in produced review documentation
  - Before versioning etc.



Concrete syntax: Guidelines for defining notation [1/2]

- Vital for acceptance and usability
- Symbols can vary from boxes to photorealism
  - Best to resemble closely the actual domain representation
  - Worst is having everything a box and special text to show the difference (cf. stereotypes)
- Don’t create notation from scratch
  - Use known/existing elements
  - Apply full range of visual variables
- Hint: ask users to define the notation
  - It is much easier to introduce their own language than something you created alone
  - Remember also model readers: customers, managers etc.

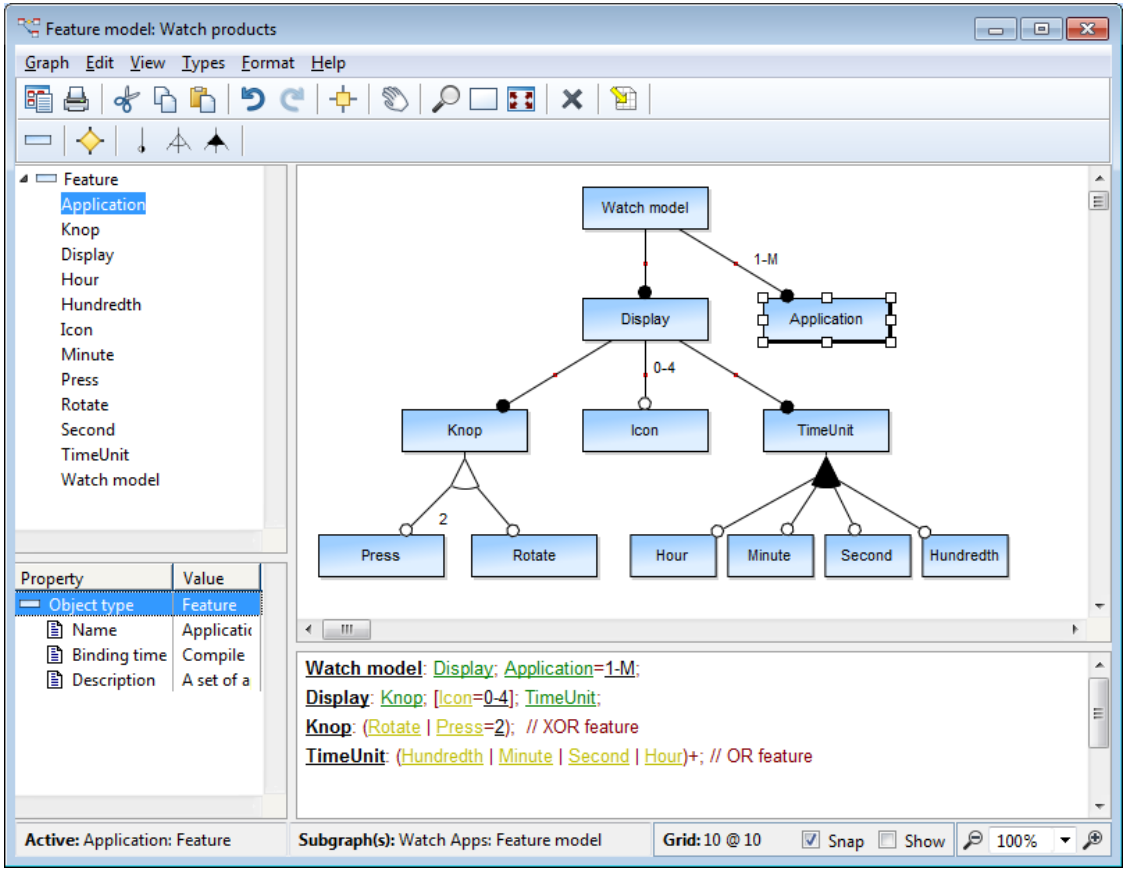



«Person»
«Laptop»
«Output»
«Input»
pictogram > geometric > photo

Guidelines for defining notation [2/2]

- Borrow directly from corporate documentation standards
- Use eye candy conservatively
  - Shadows and fountain fills pretty on screen, poor otherwise
  - Clarity in use more important than coolness in first impressions
- Notation can also show other than pure design:
  - Different views, levels of detail
  - Show errors and warnings, e.g. missing data, default value not used etc.
  - Provide guidance, e.g. indicate if submodel, reused etc.
  - Give feedback from running/debug apps, animate

Several concrete syntaxes



On reusing models and model elements

- Typically start by creating one specification
  - Normal for the first product, easy to grasp, version etc.
  - Most typical approach (Tolvanen & Kelly SPLC 2019)
- Later created models may contain parts already defined
  - People want to reuse existing specification models
    - For fixing bugs once only, for speeding development etc.
  - Language extended to include support for reuse, e.g.
    1. Has linking to existing reusable models or model elements
    2. Adds configuration data on how elements are reused
    3. Has core specifications which can be extended and customized in a predefined way

5 Generators

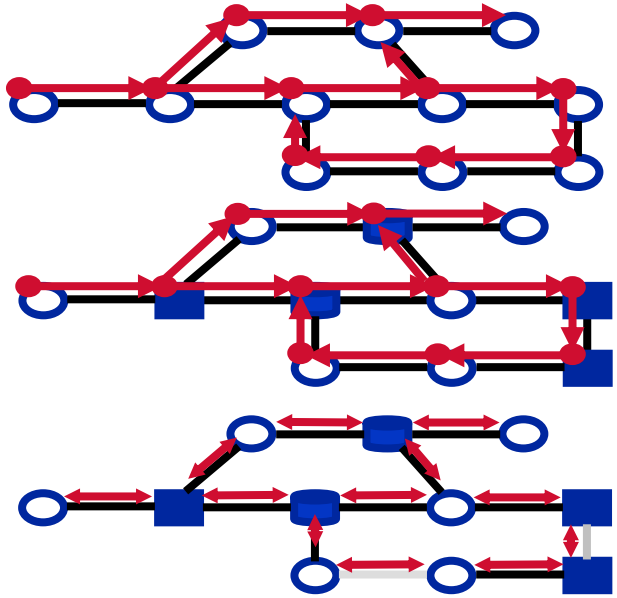
Different generator approaches
How to define generators
Examples
Exercise

Generators (M2T, M2Code)

- Generator translates the computational model into a required output
  1. **crawls** through the models
     → navigation according to metamodel
  2. **extract** required information
     → access data in models
  3. **translates** it as the code
     → translation semantics and rules
  4. using some **output format**
     → possibility to define output format

Model navigation and translation

- Multiple ways to navigate
  - Using some start elements
  - Based on certain types
    - Object types
    - Relationship types
    - Objects with certain connections
    - Objects with certain submodels, etc.
  - Based on certain instance values
- Different computational implementations possible
  - Sequential, Function calls, Switch-case structure, Transition tables etc.



Implementing domain-specific generators

- All code can never be generated so it is essential to decide what to generate (and what not)
- What to generate is conditioned by
  - Applied modeling language (i.e. metamodel)
    - Concepts, rules, semantics
  - Required code size and performance
  - Chosen implementation platform
    - Programming language, components, OS, HW
- Generator should operate directly with domain concepts
  - Shift abstraction
  - Rules of the metamodel can guarantee that input (=model) to code generator is correct
  - Generator definition (and maintenance) becomes easier

Let’s see some generators executed…

- 8-bit assembler for microcontroller (flow)
- PLC code for automation system (state machine)
- 3 GL (C, C#, Java) (state machine)

A task

How to design a generator, 1

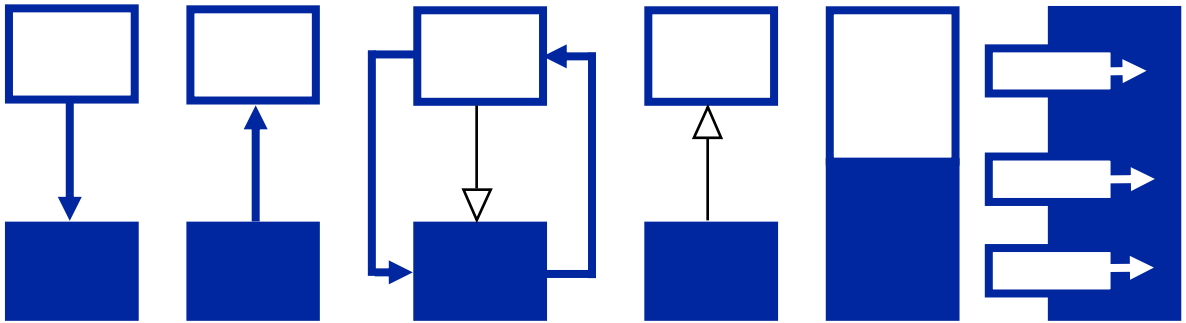
- Make generator for your situation only
  - Trying to make general purpose generator often fails
- Make generation process complete, target 100% output
  - Never modify the generated code
    - want to change assembler after compiling?
  - Correct the generator or common framework instead
    - no round-trip-related problems
- Use modeling languages to raise abstraction
  - Don’t model code, model variation
- Put domain rules up-front to the language
  - Generator definition becomes easier when the input is correct
  - Models should be impossible to draw wrongly for generation (unlike having constraints or code attached to model elements)

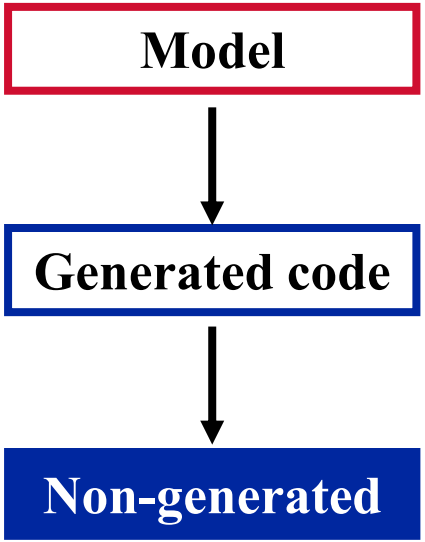
How to design a generator, 2

- Try to generate as little code as possible
  - Glue code only, rest in common part (framework/platform)
- Keep generator as simple as possible
  - Raise variation to the specification language
  - Push low-level common implementation issues to the framework
- Keep generator modular to reflect changes, e.g.
  - structure generator based on modeling concepts
  - generator per file or section in a file
  - use common generator subroutines
- Make generated code readable (“good looking”)
  - To be used later while debugging the code, executing it in a simulator, and while implementing the generator

Combining generated code and other code

- call existing code, instantiate data structures
- be called from existing code
- be subclassed from existing code
- form base classes
- be part of the class (partial class in C#)
- fill templates in existing code
- include protected regions for manual code

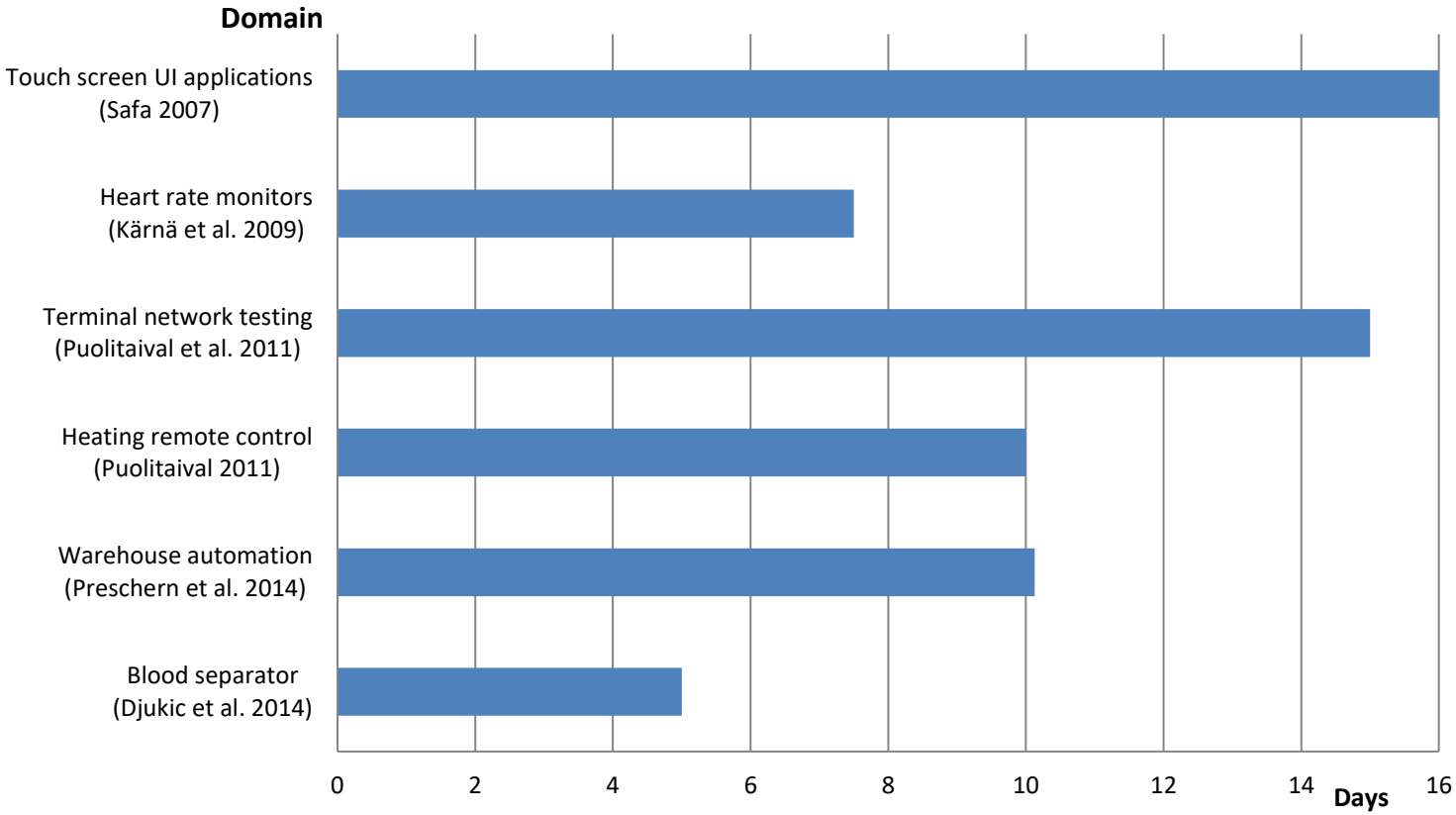




Other-than-code generators

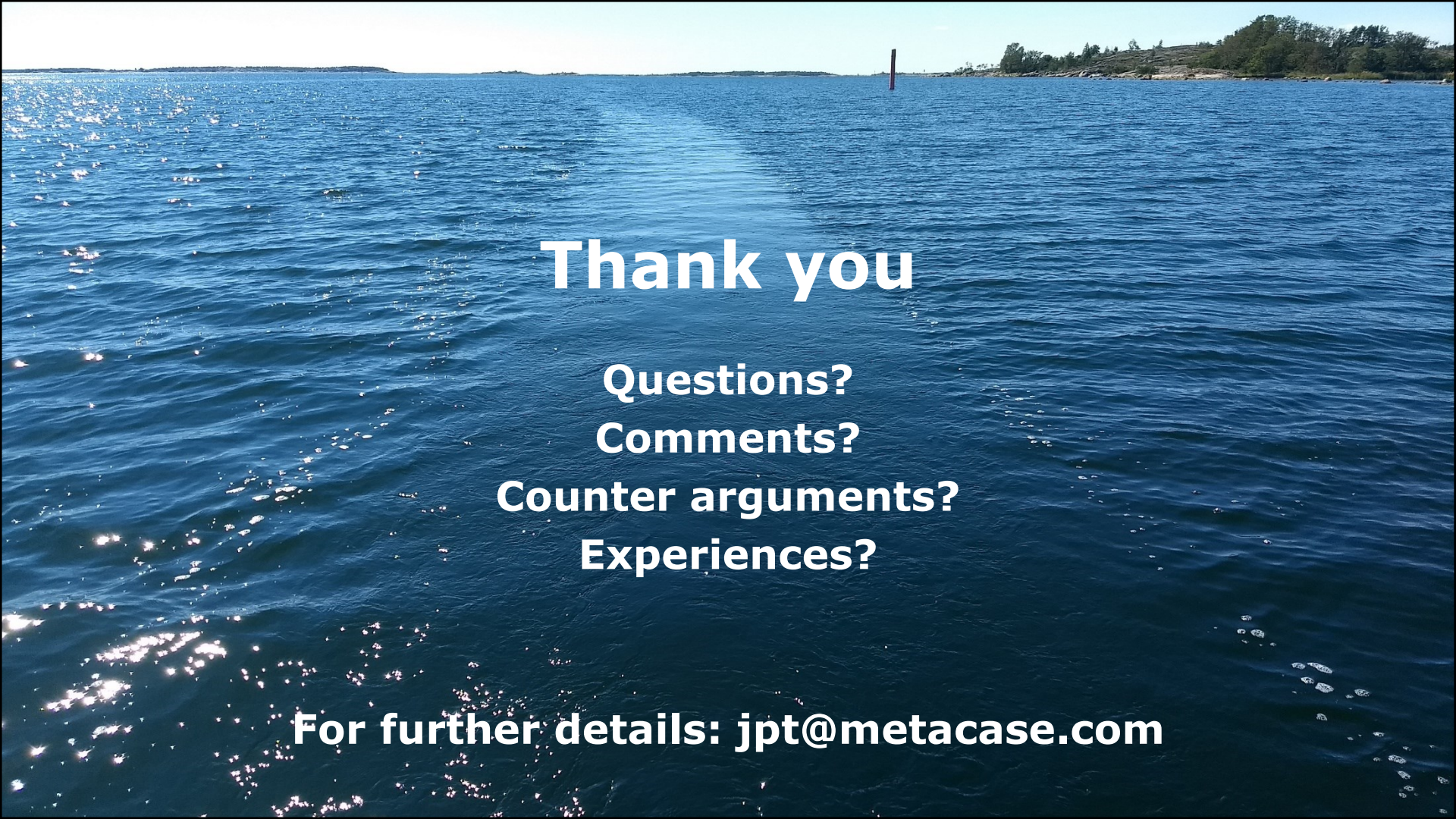
- Checking completeness and uniformity
- Configuration
- Testing and analysis
- Automated build → automating compile and execution
- Installation and deployment
- Help text
- User guides
- Documentation and review

Cost of DSL creation: industry cases



6 Summary and discussions

- Languages that narrow the domain can raise the level of abstraction
  - Domain = narrow area of interest, a problem domain
  - DSM/DSL is typically tightly related to the architecture
- DSM/DSLs are applied in practice
  - A proven way for automating development
- Build your languages and generators incrementally
  - Provide immediately results to benefit the organization
- A variety of tools available
  - Development effort 1-3 weeks
- Building automation is great fun for experts


Thank you
Questions?
Comments?
Counter arguments?
Experiences?
For further details: jpt@metacase.com

References

- Kelly, S., Tolvanen, J.-P., Domain-Specific Modeling: Enabling Full Code Generation, Wiley, 2008. DSMbook.com
- Moody, *D.,* The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering, IEEE Transactions on software engineering, Vol. 35, 5, 2009
- Ozkaya, M. The analysis of architectural languages for the needs of practitioners. Softw Pract Exper. 2018;
- Sprinkle et al. (eds) IEEE Software, July/Aug, 2009, including: Kelly & Pohjonen, Worst Practices for DSM, tinyurl.com/worstDSM
- Tolvanen, J.-P., Kelly, S., Effort Used to Create Domain-Specific Modeling Languages. ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems, ACM, 2018.
- Tolvanen, J.-P., Kelly, S., How Domain-Specific Modeling Languages Address Variability in Product Line Development: Investigation of 23 Cases, 23rd SPLC, ACM, 2019
- Tolvanen, J.-P., Kelly, S., Model-Driven Development Challenges and Solutions, Int. Conf. on Model-Driven Engineering and Software, 2016