Modulare Werkzeugarchitektur

A critical discussion on Domain-Specific Languages

Disclaimer

- DSLs are a topic that is currently finding its research community
- There are only very few scientifically consolidated insights
- However, may publications are still on a „position paper“ level, filled with many unproven claims

Agenda

- Introduction to DSLs
- Language Engineering vs. Program Engineering
- Language and Tool Architecture

Programming Languages

“Programming languages are a programmer's most basic tools ” *Tony Hoare*

Classification-Dimensions:

- Paradigm (procedural, functional, object-oriented,…)
- Textual vs. graphical
- Imperative vs. deklarative
- Use Case: Embedded programming, Business Information Systems, …

Definitions

“A DSL is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.“*

Metaphor:

- General-purpose Programming Language = Craftman‘s toolbox (usable for many problems but not efficient).
- DSL = Factory (only usable for a very specific product but with high efficiency).

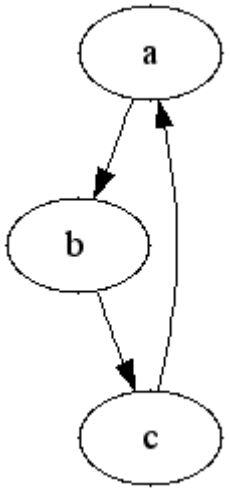
*Arie van Deursen, Paul Klint, Joost Visser

Examples (1): „Classic DSLs“

Textual

- Lex (RegExps), Yacc (BNF)
- SQL
- HTML, MathML, VRML, SGML, ...
- Teapot: Cache-Coherence Protocols.
- Dot:

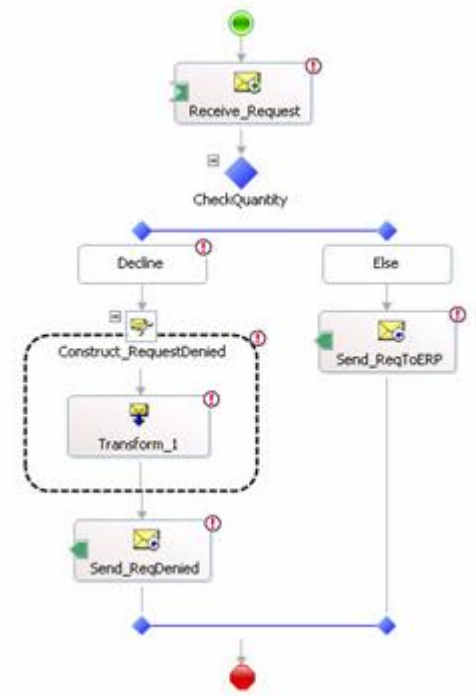
```
digraph Cycle {
  a -> b
  b -> c
  c -> a
}
```

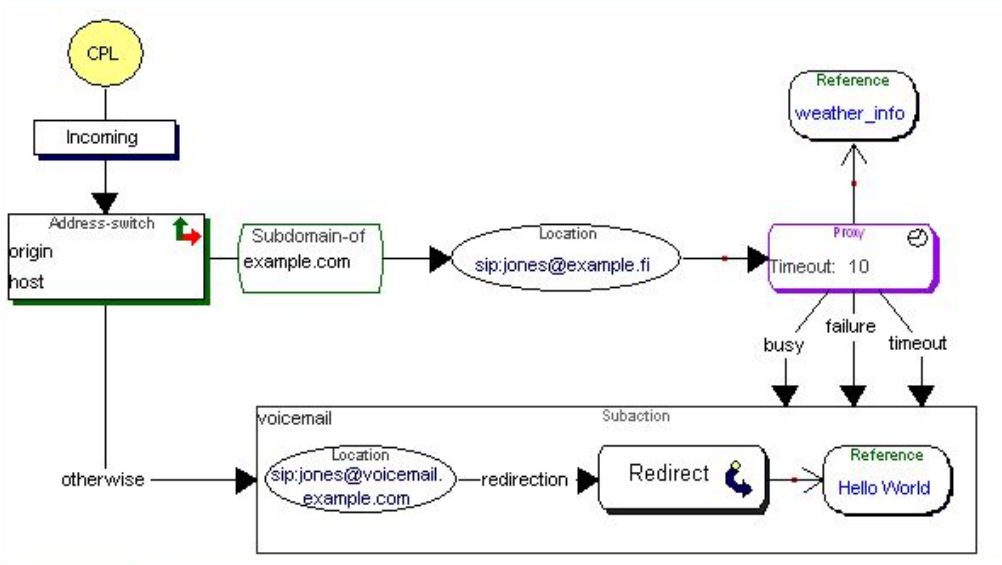


Examples (2): More specific domains

Graphical

- GUI Builder
- Biztalk Orchestration Designer
- CPL: Internet Telephony Services*





*Internet Engineering Task Force

Old hat?

DSLs are (almost) as old as computer science itself

- (first so called DSL: „Automatically Programmed Tools“ von 1959.)

Why are they now of current interest?

- Hope to repeat the success of general-purpose programming languages:

  „Surely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high-level languages for programming. Most observers credit an increase in productivity with at least a factor of five.“ *
- Increasing abstraction forces a reduction of expressiveness => DSL

*Fred Brooks, 1975/1995, „No Silver Bullet“

Claimed Advantages

- Domain-specific notation

  => Self-documenting

  => Are understood by domain-experts (without programming skills)

- High level-of-abstraction

  => Smaller „programs“

  => Less bugs (because of smaller programs)

  => Increased productivity

Claimed Advantages (2)

- Limited Expressiveness

  => Better possibilities for analyses, verification, optimization, …

- Reuse of domain-knowledge

- Do not have to be executable

  => You do not notice defects ;)

  „Pictures, as opposed to programs, don‘t crash“*

*David Garlan

Claimed Disadvantages (1): Usage

- For many domains
  - No DSL available
  - If so, then often very proprietary ones
  - (Almost) no experiences with DSLs
- Users must learn a new language (but: they must learn domain-knowledge anyway)
- Many fundamental questions remain still unanswered
- Potentially lower performance than manually crafted (and optimized) solution

Claimed Disadvantages (2): Construction

- Costs for design, implementation and maintenance of a DSL
- Domain-knowledge as well as compiler-construction skills necessary
- Difficulties of finding the „right scope“
- Maintenance of DSLs not yet fully understood

Reduction of „Accidential Complexity“

Classification of the difficulties during software design:*

Essential: Data structures, relationships between entities, algorithms, functions

= Problem-inherent Complexity

Accidential: historical, artificial, not problem-inherent complexity.

= „Artificial Complexity“

Example: GUI Builder, Parser Generator

*Fred Brooks, 1975/1995, „No Silver Bullet“

Avoidance of redundancy

In most systems you find

Domain concepts (e.g. customer) implemented at different places in the solution. (GUI, BL, DB)

$\Rightarrow$ Update-anomaly: A chance in the domain concept customer forces many changes in the solution.

$\Rightarrow$ DSL at the „right“ level of abstraction helps to avoid these kind of redundancy

Examples: CDL, „Application Generators“

Analyzability by reducing expressiveness

Approach:

- Reduce the contructs of the language and their expressiveness to gain better analyzability.

Examples:

- Protocol verification
- ConQAT: Load Time Type Checking
- Giotto

Isolation of Variability

Approach:

- Bundling of similar information with high expected change rate

Examples:

- CSS: Layout informationen in a central file.
- JBoss Rules: Business Logic in terms of rules. This does not avoid redundancy. But chances of the business logic can be done locally at one point.

Current Approaches

- Generative Programming (Czarnecki, Eisenecker)
- Domain Specific Modeling (Tolvanen, MetaCase)
- Model Driven Software Development (Völter, oAW)
- Model Integrated Computing (Vanderbuilt, GEM)
- Language Oriented Programming (JetBrains, MPS)
- Software Factories (Microsoft, DSL Tools)

- Model Driven Architecture
- Software Productlines

Open Questions

- In which cases makes the application of a DSL economical sense, in which cases is GPPL cheaper / more efficient?
- Which methods should I use to develop a DSL?
- What is a good DSL?
- How can we maintain a DSL efficiently? (Coupled evolution of metamodel, model and tools?)
- Which „Technical Space“ fits best for which scenario? (Grammar, Metamodel, XML-Schema, …)

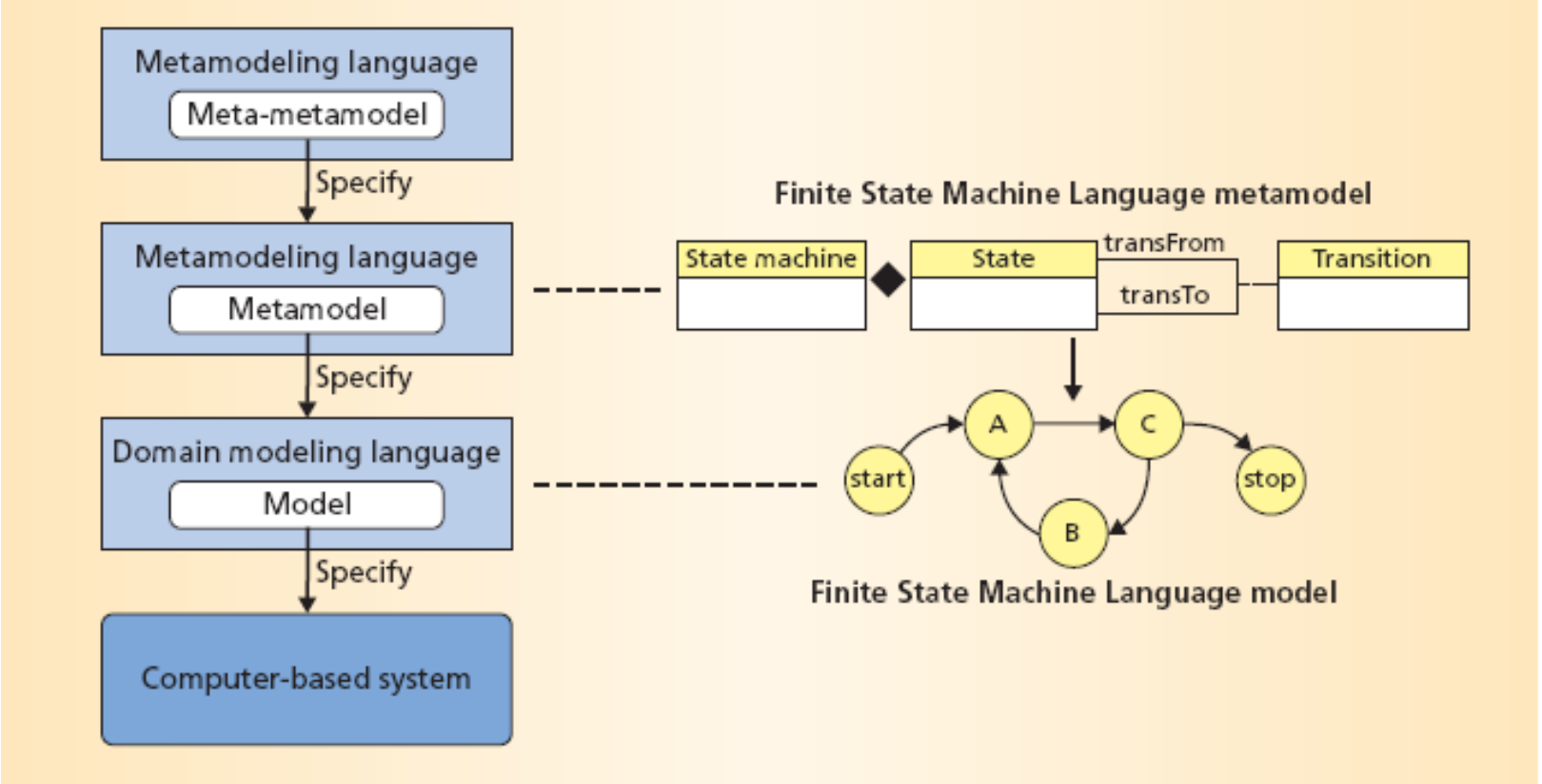
Language Engineering
vs.
Program Engineering

Language Engineering

- State-of-the-art
  - Syntax definition using grammars/data models and constraints
  - Definition of „translational semantics“

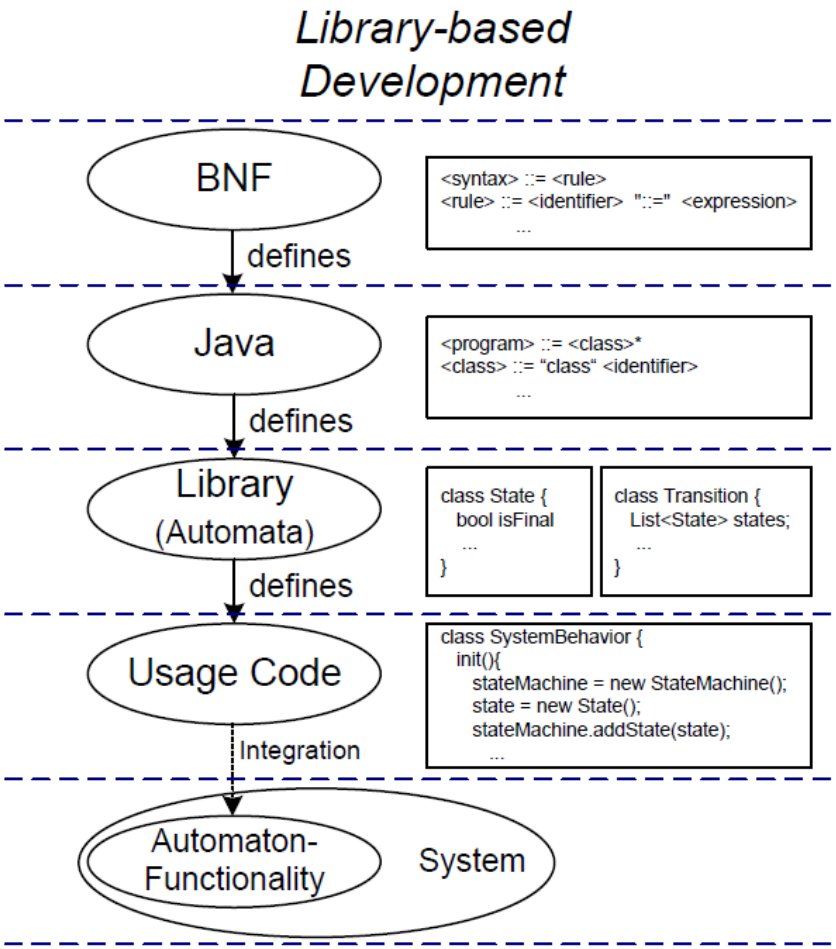
- Challenges:
  - Diverse metamodeling dialects, incompatibility
  - Few approaches to build components/modules, usually huge flat metamodels
  - Evolution problem of models/programs if the language changes
  - High tool development efforts: editors, generators, but also debugger/simulator, profiler, …
  - Diff/Merge is usually cumbersome when using graphical languages

Metamodeling



3-layered Metamodel-Architecture (MOF)

A Comparison …

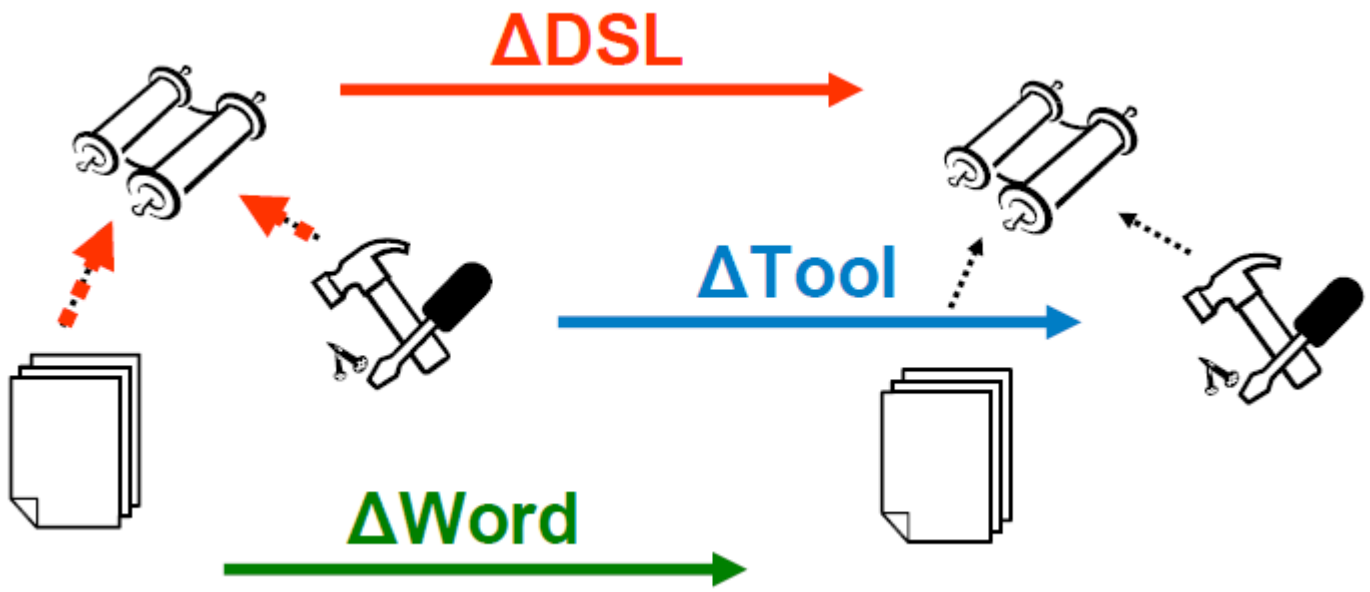


Some Discussion …

	Language Engineering	Program Engineering
Elimination of redundancy	DSLs can be created to avoid redundancy	Good design with low redundancy often possible, in some cases repetitive code inevitable
Effort	Development and maintenance of several tools, risk of over engineering	Focus efforts on the product, not on the tools to develop the product
Memory Footprint of executables	Good generators enable generating small code bases with low memory footprint (unfortunately many do not)	Libraries often come up with much functionality that is not needed for a specific application
Product Lines	Describe variability in the language, generate only the code needed for the individual product	Provide the full functionality and dynamically configure the system at start-up.

Language Maintenance

- DSL: Grammar, Metamodel
- Tool: Generators, Editors
- Word: Models/Programs

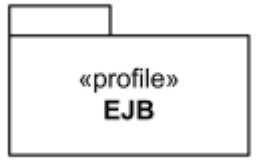


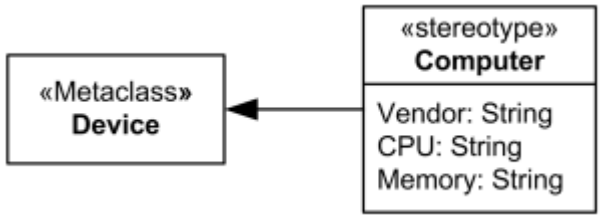
Language and Tool Design

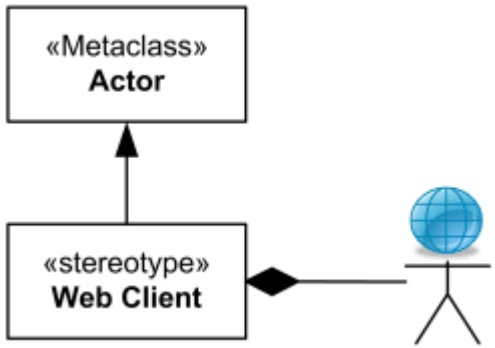
UML Profiles

What are UML Profiles?

- Extension of the UML Standard diagrams with custom entities using stereotypes
- Starting point is a package declaration
- Definition of metaclasses (concepts/entities) optional with custom symbols/icons



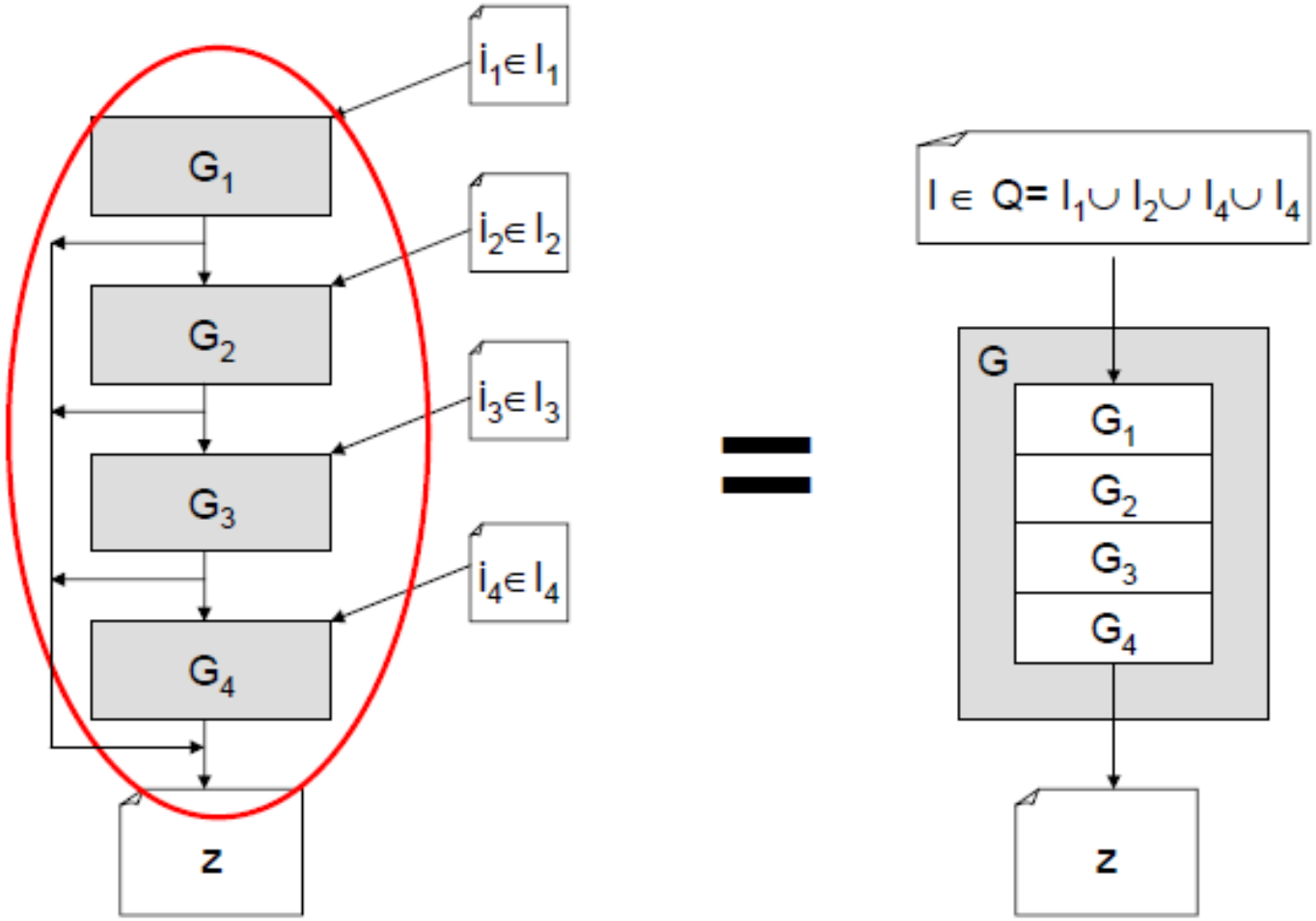




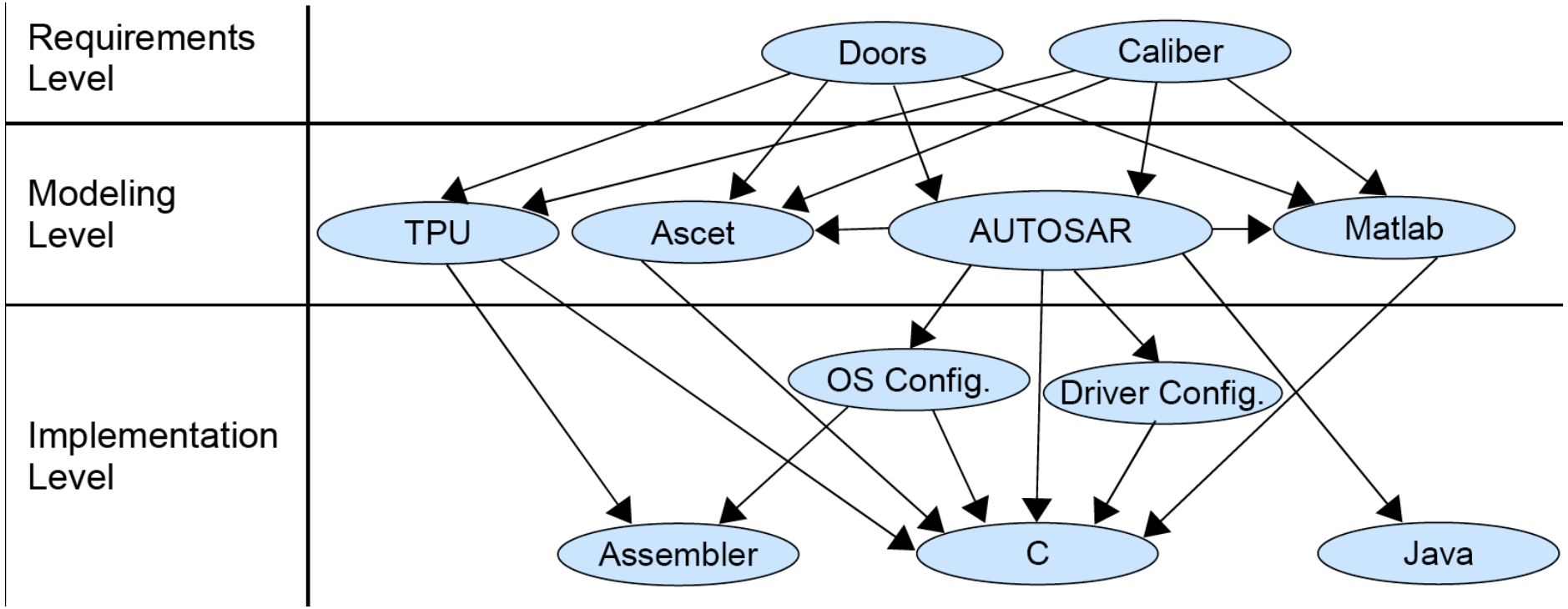
Syntax: UML Profiles vs. Metamodeling

- UML profiles are more lightweighted than metamodeling
- Profiles do not define contraints on the syntax
  - Diagrams possible that are not even correct on a syntactic level
- UML diagrams may be a well-known notation
  - Stereotypes may give them completely different semantics
- UML is a standard
  - Without defined semantics
- There are standard tools for UML
  - There is also tooling for Metamodels (e.g. metamodels)

Staged Generation/Model Transformation



Tool Integration: e.g. Automotive Tooling today



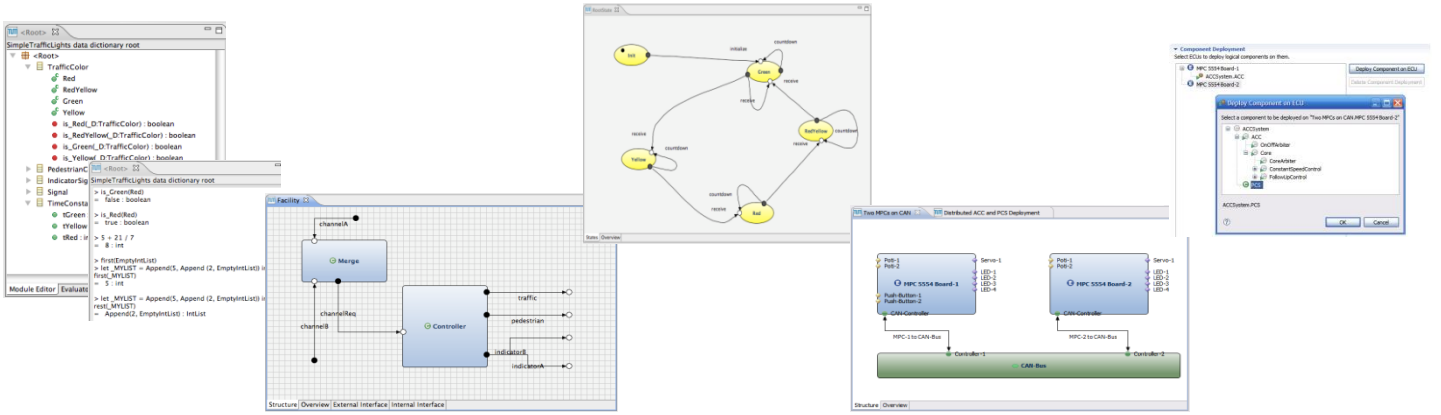
Integration of languages

State-of-the-art:

- Integration using the target programming language (using glue code)
- Integration using one language as Strings in another language (e.g. web programming)
- Extendable Compilers (z.B. Stratego/SDF, MetaBorg, Silver, …) → only academic prototypes

Component-based Language Engineering

Example: Tool-Architecture of AutoFOCUS 3



Editors

Metamodel

Generators

Typesystem

Component Network

Automaton Specification

Hardware Topology

Deployment

...

...

Generic Toolingframework

eclipse

EMF, GEF, …

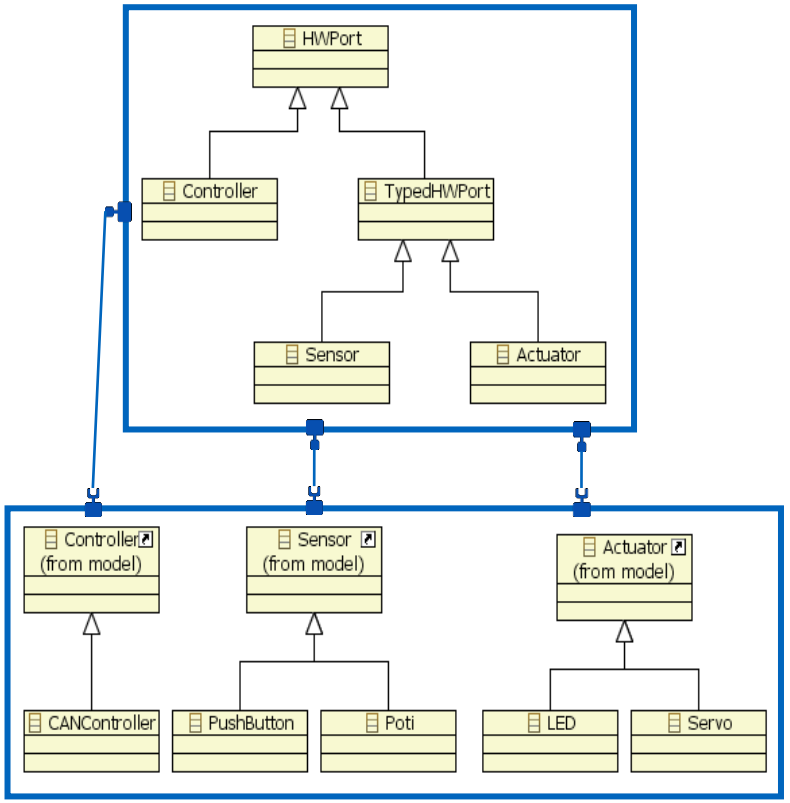
Component-based Language Engineering

Four Levels of Integration

- Abstract Syntax: In-Memory representation of the words
  - Metamodell, Grammar
- Concrete Syntax: Human read-/editable representation
  - Diagrams/graphical Editors, Text/Text Editors
- Transformation: Translation to other artifacts
  - Template-based (text generation), Model-to-Model transformation
- Semantics: Common definition of the meaning of the concepts
  - Automatons, Focus, …

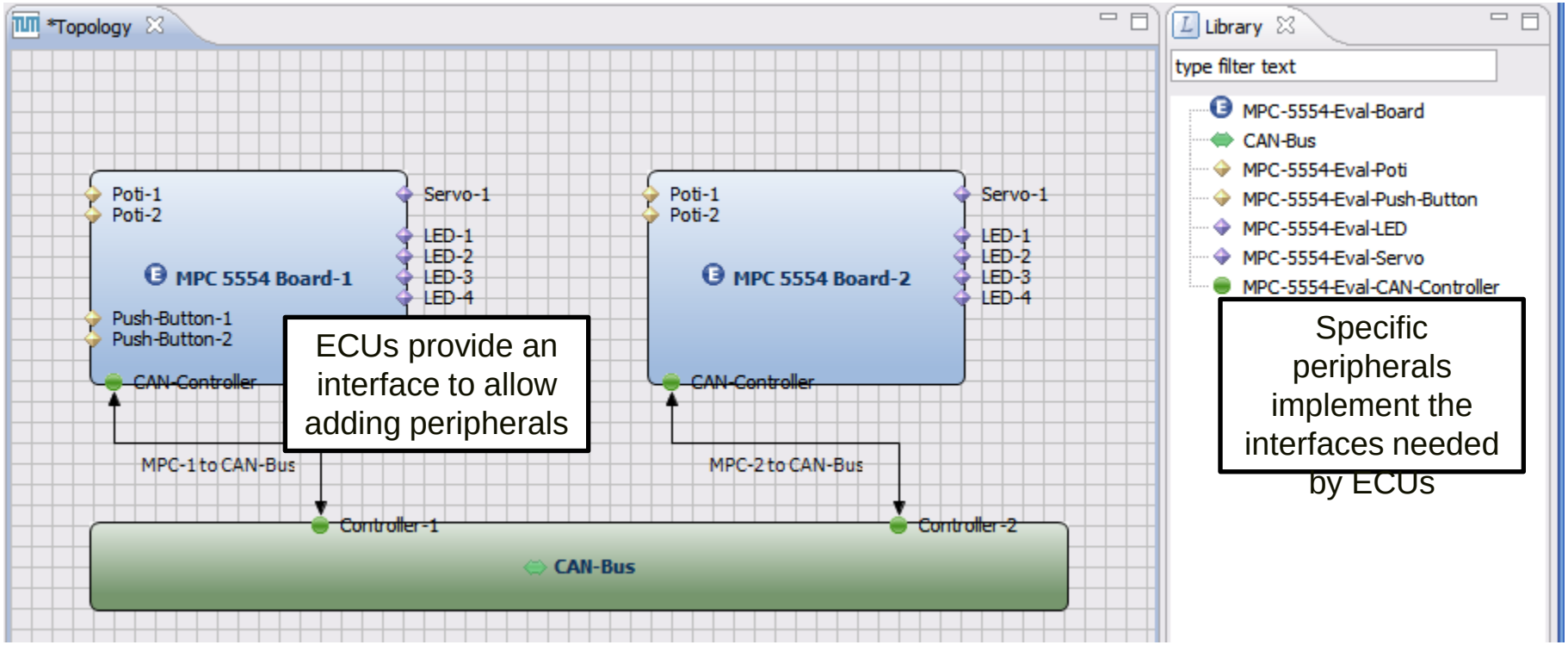
Language Components: Abstract Syntax

Use inheritance to extend abstract concepts from another component



Language Components: Concrete Syntax

- Provide interfaces (extension points) to integrate the concepts of other languages



Language Components: Transformation

- Define only the code-representation of the concepts of the lang. component (a good code-level architecture needed!)
- Define interfaces (extension points) where other information is needed

```
«DEFINE Root FOR data::DataModel»
    «EXPAND Entity FOREACH entity»
«ENDDEFINE»

«DEFINE Entity FOR data::Entity»
    «FILE name+".java"»
        public class «name» {
            «FOREACH attribute AS a»
                public «a.type» «a.name»;
            «ENDFOREACH»
        }
    «ENDFILE»
«ENDDEFINE»
```

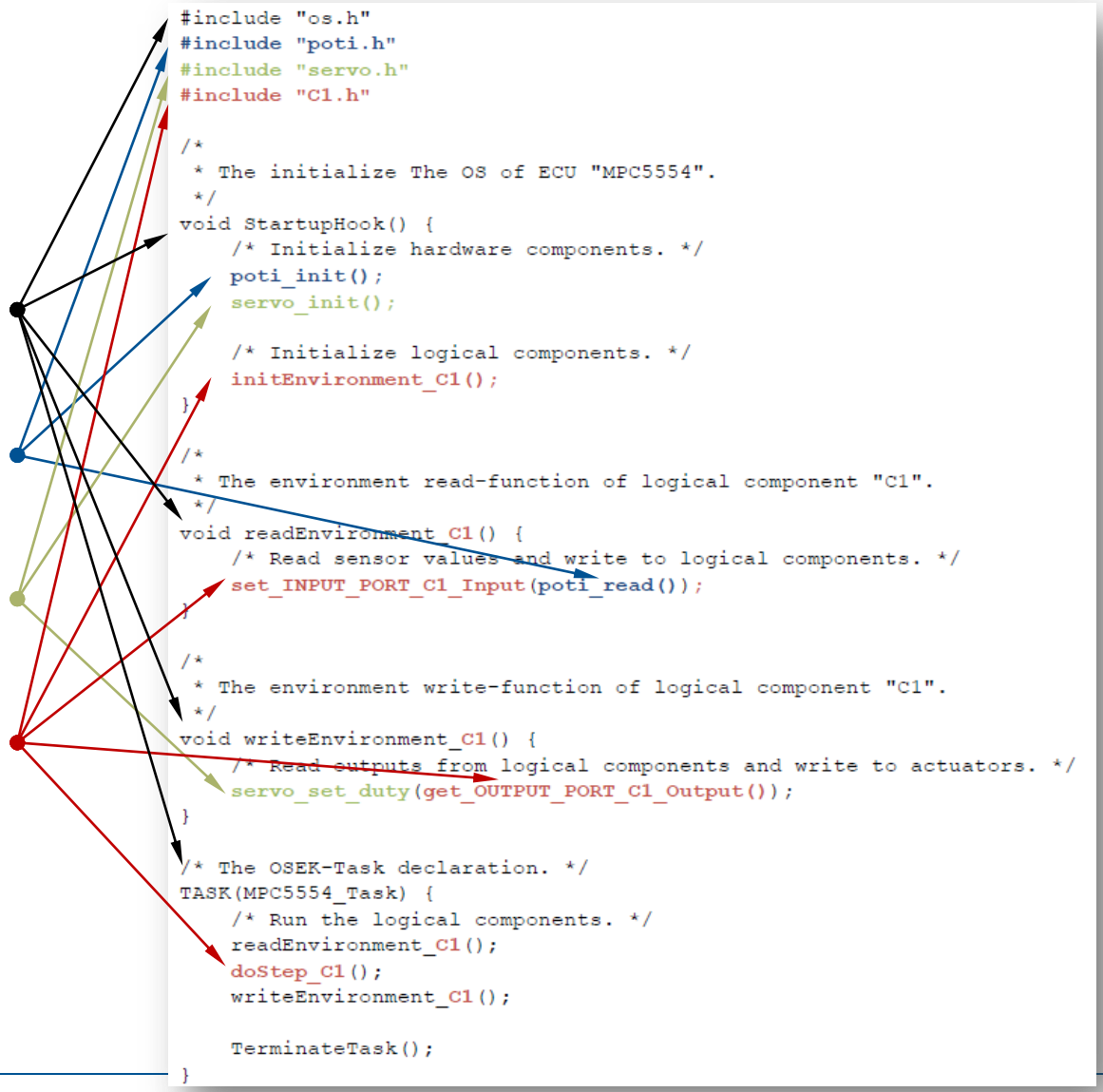
Modular Code-Generation II

MPC5544-OSEK-Generator

Potentiometer-Generator

Servo-Generator

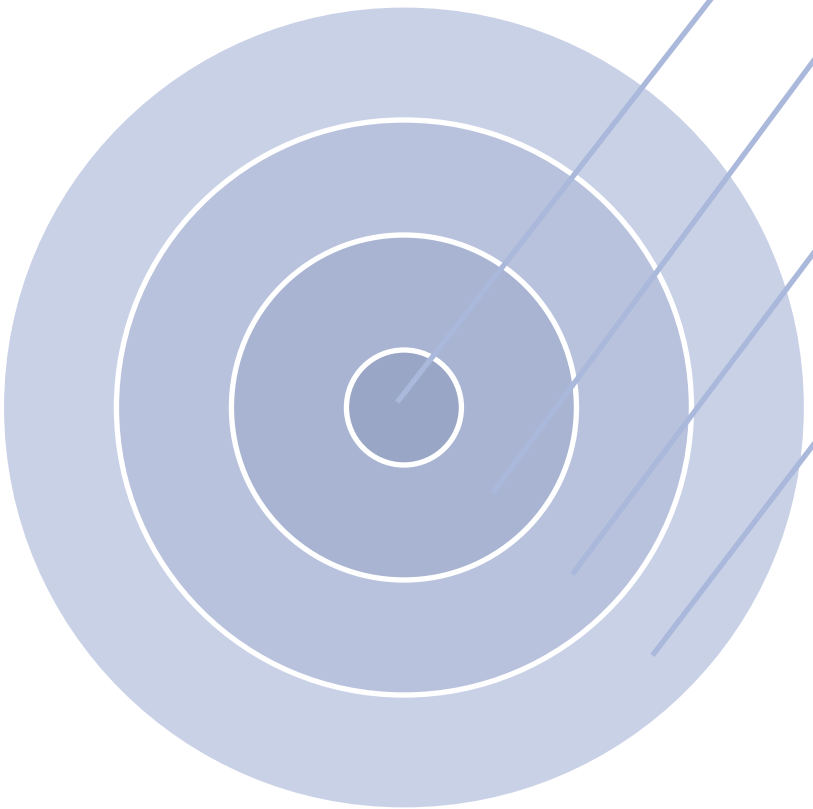
Automaton-Generator



```
#include "os.h"
#include "poti.h"
#include "servo.h"
#include "C1.h"

/*
 * The initialize The OS of ECU "MPC5554".
 */
void StartupHook() {
    /* Initialize hardware components. */
    poti_init();
    servo_init();

    /* Initialize logical components. */
    initEnvironment_C1();
}


/*
 * The environment read-function of logical component "C1".
 */
void readEnvironment_C1() {
    /* Read sensor values and write to logical components. */
    set_INPUT_PORT_C1_Input(poti_read());
}


/*
 * The environment write-function of logical component "C1".
 */
void writeEnvironment_C1() {
    /* Read outputs from logical components and write to actuators. */
    servo_set_duty(get_OUTPUT_PORT_C1_Output());
}

/* The OSEK-Task declaration. */
TASK(MPC5554_Task) {
    /* Run the logical components. */
    readEnvironment_C1();
    doStep_C1();
    writeEnvironment_C1();

    TerminateTask();
}
```

Code-Generation Beyond C

Many important artifacts can be generated:



State Automata C-Code

Middleware Code, Configuration Files and Tracing-Information

Catalogs and other Meta-Information

Makefiles, Rollout- and Flash-Automation Scripts

Tool Dev: Lessons Learned

- Trend to declarative descriptions, e.g.
  - GMF for Editors
  - Xpand for Generator-Templates
  - oAWcheck for constraint checking
  - xTend for model transformation
  - …

→ Problem: Huge mixture of languages!

  - Learning efforts
  - Composability problematic
  - Often too weak expressiveness (escape mechanisms)

  → Not every language that seems to be elegant is an enrichment for the project

Tool Dev: Lessons Learned

- Generators need a good understanding of the target code architecture
- Build Generators in a bottom-up manner
  1. Write 3-5 instances of the program
  2. Factor out their commonalities/redundancies on the code level
  3. Build the generator templates

Language Design: Lessons Learned

- Trade-off: The more abstract the concepts,
  - the smaller the language
  - the more complex the generator
- Separate orthogonal views

Literature

- „Domain Specific Languages: An Annotated Bibliography“: A.van Deursen, P. Klint, J. Visser, 2000
- „When and How to Develop Domain-Specific Languages?“: M. Mernik, J. Heering, A. Sloane, 2003
- Overview of Generative Software Development, C. Czarnecki, 2005
- „Language Workbenches – The Killer-App for Domain Specific Languages?“, Martin Fowler, 2005

42