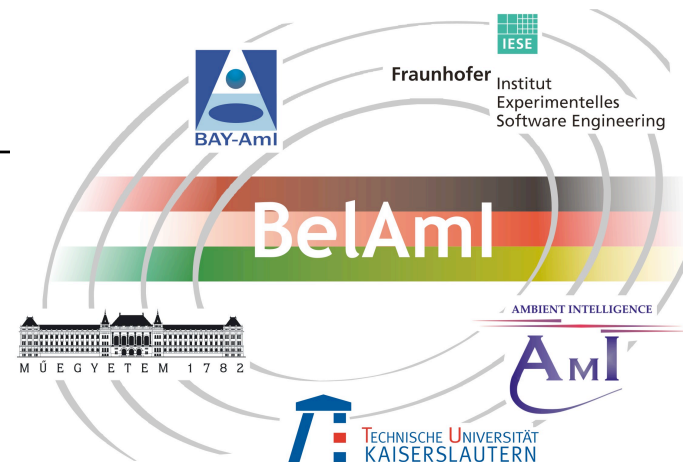
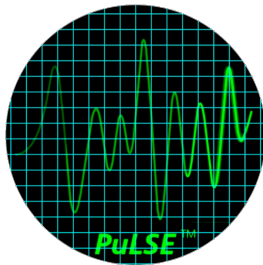

A Method for Reducing Arbitrary Source Code Complexity in Reusable Embedded Systems Code

The Frame Technology Idiom

Thomas Patzke

Thomas.Patzke@IESE.Fraunhofer.de
Fraunhofer IESE, Kaiserslautern, Germany
Product Line Architectures Department



Joseph von Fraunhofer (1787 - 1826)



Researcher

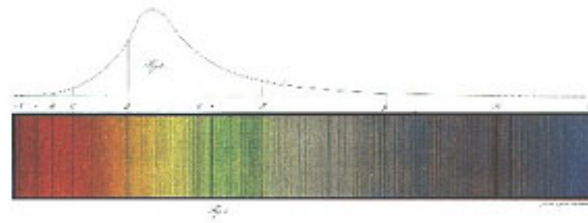
discovery of “Fraunhofer Lines”
in the sun’s spectrum

Inventor

new methods of lens processing

Entrepreneur

head of royal glass factory



Institute for Experimental Software Engineering

Mission & Role



- Advance the state-of-the art in software & system engineering
- Promote the importance of empirically based software and system engineering
- Provide innovative and value-adding customer solutions ***with measurable effects***

“Much of the complexity that [the software developer] must master is arbitrary complexity” [Brooks]

Outline

- Problem
- Approach
- Variability Mechanisms
 - Frame Technology
- Outlook

Problem

- Implementations produced for reuse in practice often fail to be reused in the long term
 - Increasing effort for adding new features (Bosch, Ricoh)
 - Reconfiguration takes too long (POSCO)
- Goals with product line engineering
 - Improved reuse across the entire SE lifecycle
 - Cost-effective software construction
 - Code reuse in many family members (space dimension)
 - Reusability over long periods (time dimension)
- Effect: higher effort than expected in all areas
 - Application engineering
 - PL Maintenance (defect removal)
 - PL Evolution (scope changes)

page 5/20

Reasons for High Implementation Effort

- Lack of architectural compliance
- Fear of efficiency penalties
- Unnecessary complexity in evolving PL code
 - Violations of SW evolution laws [Lehman]
 - ž Continuing change vs. lack of continuous adaptation
 - ž Increasing complexity vs. insensibility to simplicity
 - ž Feedback system vs. lack of feedback
 - Factors in Embedded Systems
 - ž High technical complexity [Royce]
 - ž Long-living
 - Effort = f(Code Complexity due to variability mgmt.)
 - ž Maintainability Index MI = f (Cycl. Cplx., LOC) [SEI]
 - ž Product Complexity CPLX [Boehm]
 - ž Industrial studies: effort = f (LOC³) [PutnamMyers92]

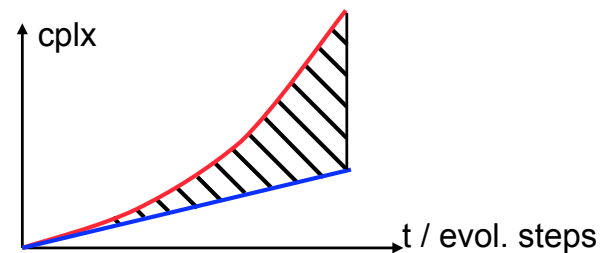
page 6/20

Complexity Triggers in Practice

- Short-term convenience
 - Obsolete features are not removed
 - Large-scale code duplication
 - Re-development instead of reuse
- Change-resistance
 - Establishment of variability management monocultures
- Mistakes are not undone properly
 - Because they are detected too late
 - E.g. inappropriate variability mechanism selection

Approach

- I provide a new complexity-aware PL-impl. method
 - Instead of describing the artifact (PL code), it guides the developer in the construction process (product/process duality, backtrack-minimizing sequences) [Alexander, Leyton, Simon]
 - Aim: code complexity optimization by balanced variability mechanism selection
- Expected benefit: 20% complexity growth reduction compared to an adhoc approach



page 8/20

Similar Work

- **Variability Mechanisms** [CzarneckiEisenecker00,SvahnbergBosch05]
 - They just catalog what works theoretically and generally, I focus on practical mechanisms for Embedded Systems development
 - I discuss mechanism consequences and interdependencies
- **Practical PL Method** [Coplien98, Krueger07, Bassett97]
 - They propose design/implementation mappings only; I consider a wider context: existing code, mechanisms, tools; developer experience, organizational issues, ...
 - I propose to gradually optimize existing variability mechanisms, if the need arises
- **Continuous Evolution** [Alexander02, Fowler98]
 - I developed an extensive catalog of elementary PL evolution scenarios
 - Artifacts are never seen as 'finished'
 - I consider more than refactorings: construction-time behavior-extending transformations
- **Complexity/Simplicity** [Alexander02, Simon62]
 - I explicitly focus on maintaining simplicity
 - Goal: ease-of-reuse with 'just enough' complexity
- **Measurement** [SEI, McCabe, Levenshtein, Kolmogorov]
 - My focus: construction behavior, not execution behavior

Copyright © Fraunhofer IESE 2008

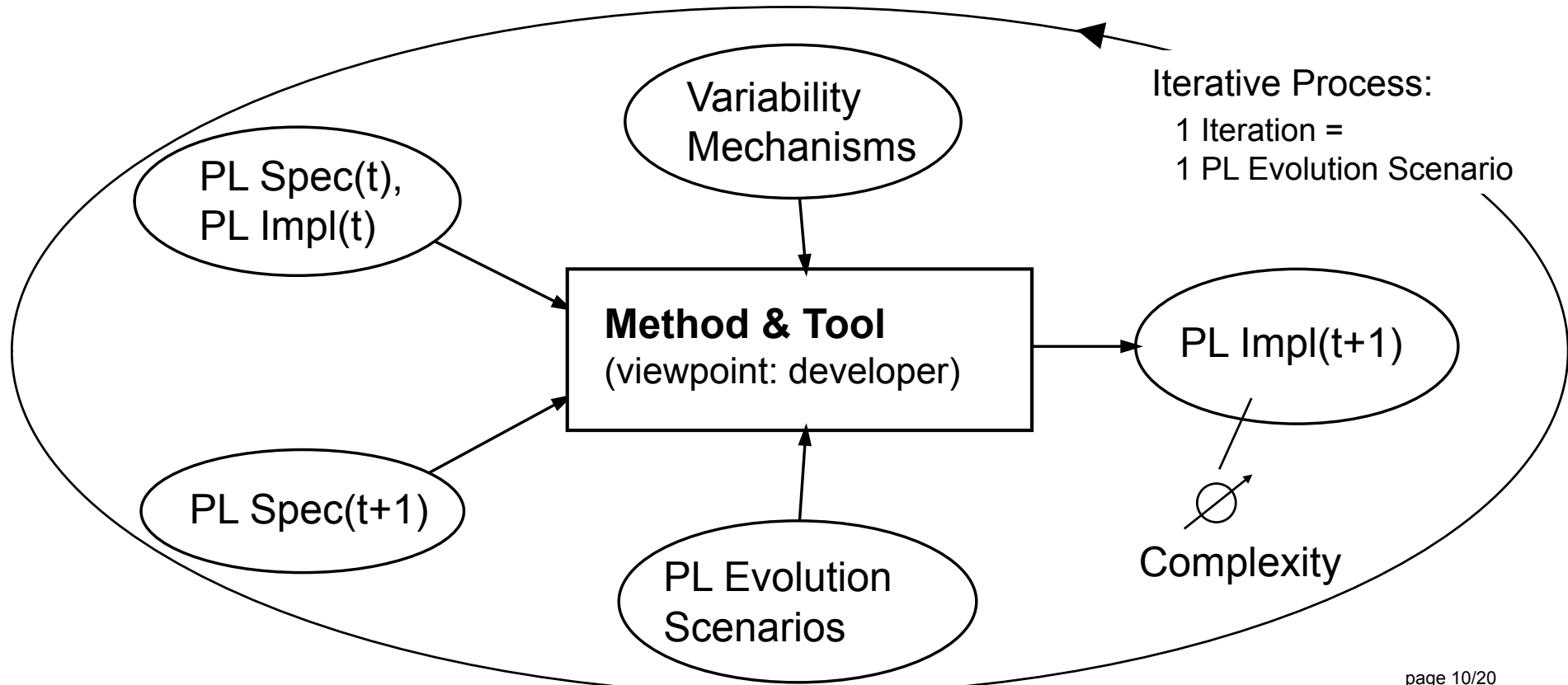
- Relevant to PL testing



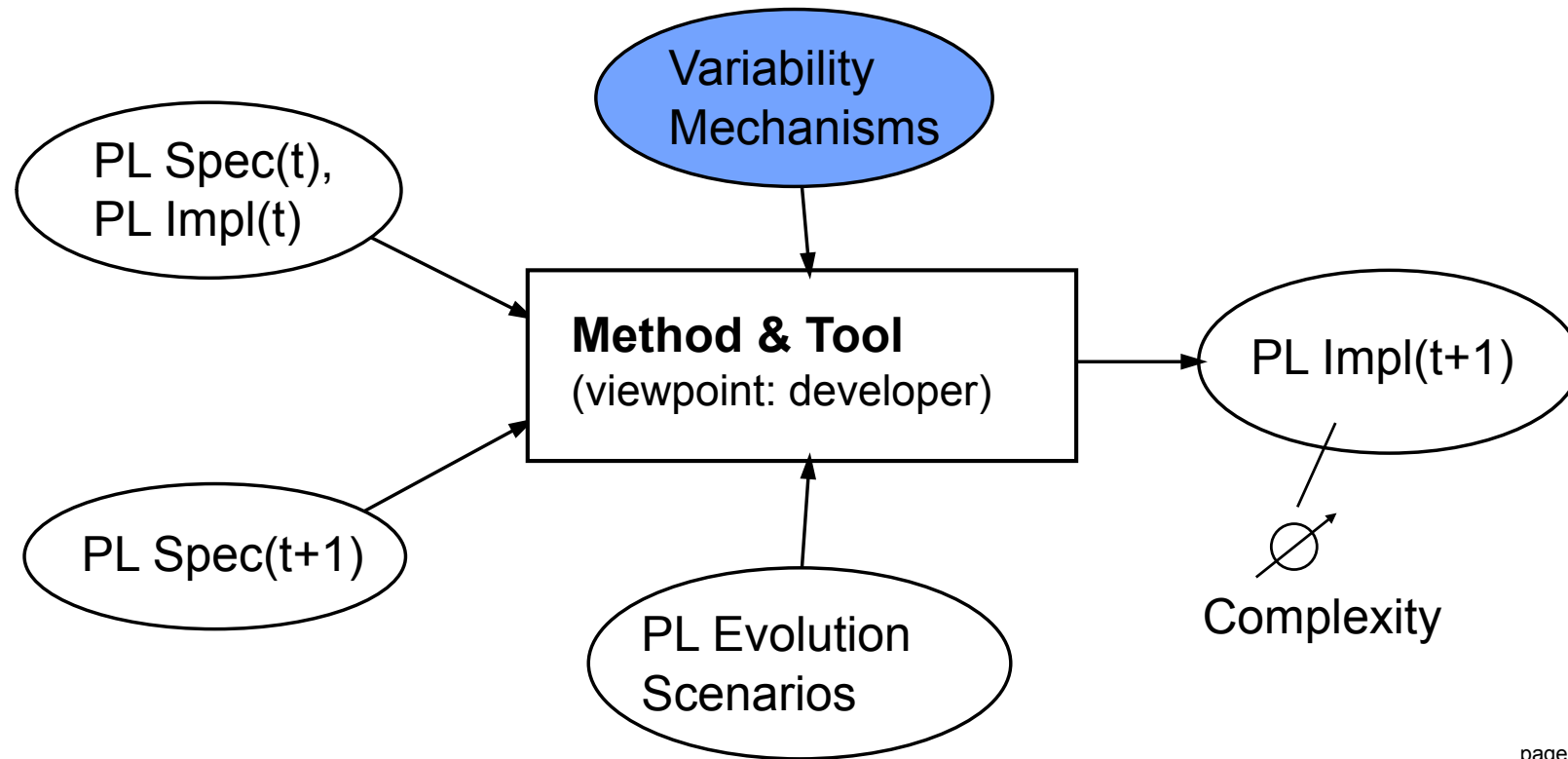
Fraunhofer Institut
Experimentelles
Software Engineering

SVPP'08, Brussels
August 9, 2008

Approach



Building Block #1: Variability Mechanisms



page 11/20

Variability Mechanisms in Embedded Systems (C/C++)

Pattern-language-like description of 5 actively used mechanisms,
1 (over-) hyped and 1 advanced one:

- Cloning!
- Conditional Execution
- Polymorphism
- Late Module Binding
- Conditional Compilation
- Aspect-Orientation
- Frame Technology

Variability Mechanism Pattern Template

Name	short name, for reference
Intent	concise description of purpose
Motivation	example scenario from emb.sys.PLI
Applicability	context in which pattern helps most
Structure	structural view of code organization
Participants	explanation of structure elements
Dynamics	dynamic view of code organization
Consequences	positive and negative effects
Implementation	specific details and variants
Sample Code	code fragments from real projects
Known Uses	independent applications
Related Mechanisms	similar alternatives

Example: Frame Technology (FT) – Origin and Concepts

Invented by Paul Bassett for Cobol code reuse in the late 70s

Aggressive reuse mechanism (typ. 90% reuse, empirically shown)

Reuse \neq Use (-as-is) => duality

properties of use: functionality, efficiency, ease-of-use

reuse properties: generality, compactness, adaptability

Separation of

construction semantics (for reuse) from

execution semantics (for use; includes compilation etc.)

First-class elements: defaults (negative variabilities)

primary mechanism: default text overriding

optimizes the number of variable parts

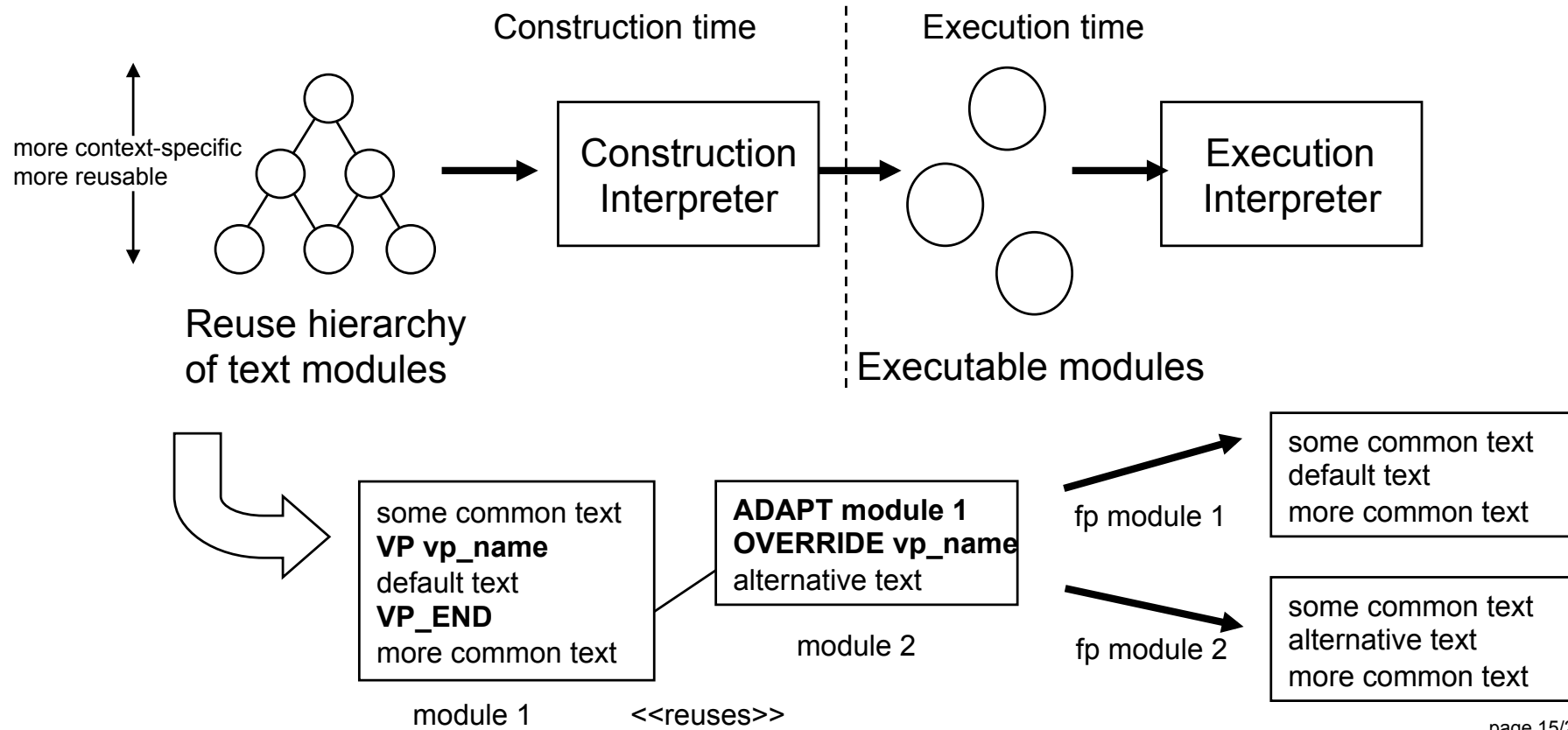
Reuse layers make different degrees of similarity explicit

Support for open & closed parameters of variation

open ones facilitate unanticipated evolution

page 14/20

FT - Structure



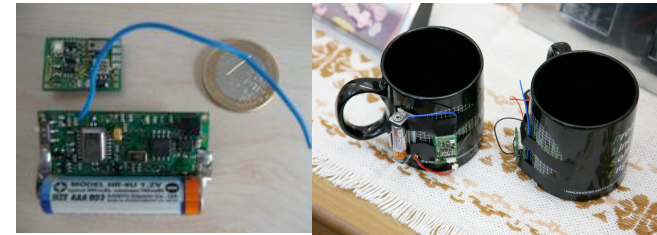
FT - Intent

Decompose textual information according to its stability over time, so that modules which need to change less frequently become nearly independent of modules that evolve more often.

Frame Technology facilitates to keep source code localized which shares the same change rate, especially in cases where otherwise the programming language syntax would enforce this code to crosscut several modules.

FT - Motivation

Wireless sensor node product line
for ambient assisted living,
implemented in C & Assembler



- Variability across language boundaries, different C dialects
- Variability in Makefiles and documentation
- Variability in two dimensions
 - across space (different products)
 - across time (evolution)

FT - Applicability

- to modularize co-evolving code parts which cannot be easily extracted into a single module using C/C++ programming language mechanisms
- to manage alternative variabilities independently
- to manage variability in multi-language source code or other textual product line artifacts
- to provide a global point of modification and configuration of variabilities
- to highlight product-specifics and hide sameness
- to provide variability at several levels of scale

FT - Consequences

- + Module organization in reuse hierarchies, according to stability over time
- + Resource efficiency-invariance
- + Feature addition and removal are equally supported
- + Support for unpredicted changes (through open parameters)
- + Minimal refactoring overhead
- The code must be available (gray-box reuse)

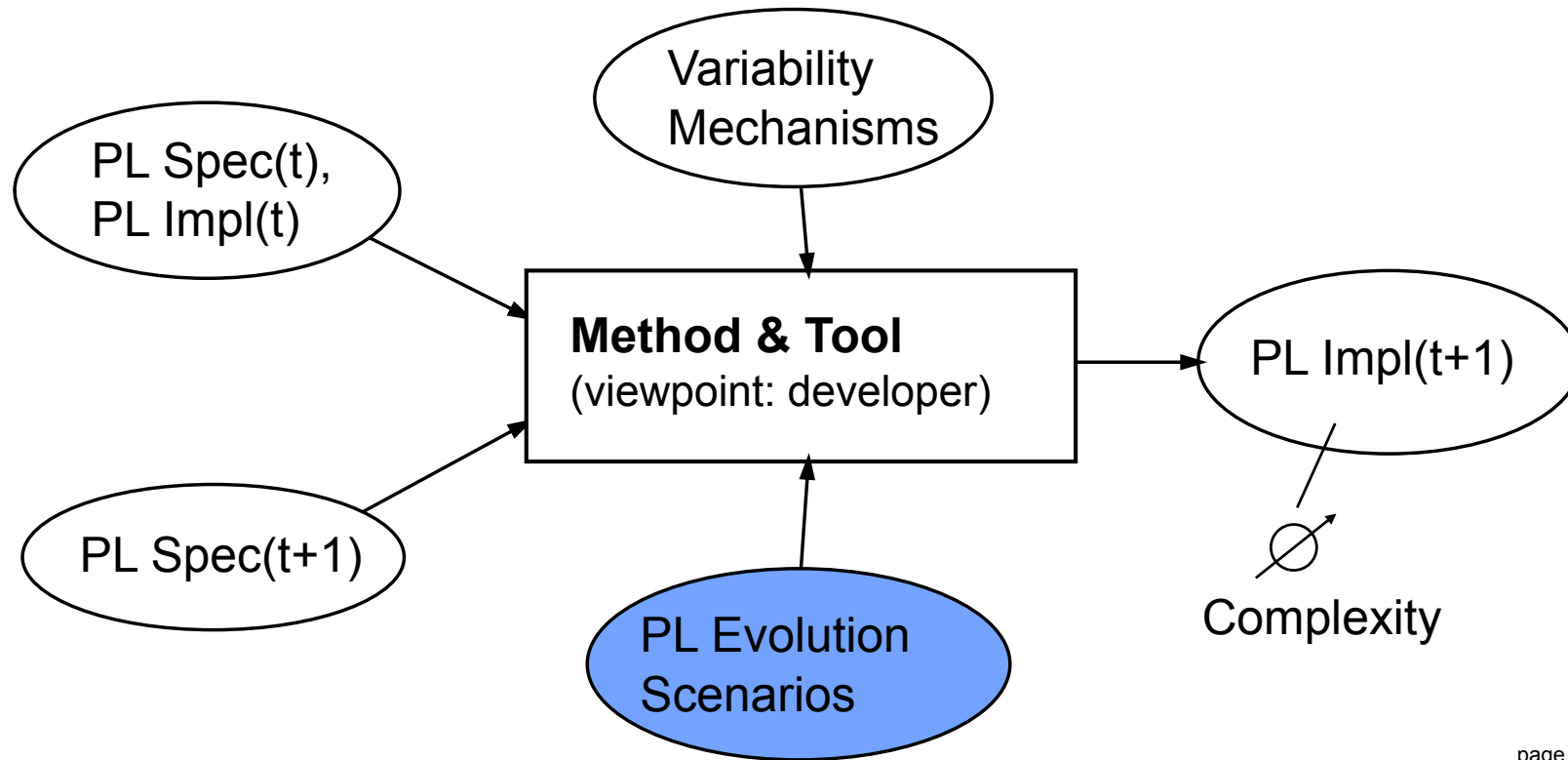
Conclusion & Outlook

- From a developer's perspective, the practical problem is to manage code complexity in evolving PL-implementations
- My approach aims at repeatable and cost-effective complexity management
- Frame technology concepts reduce variability management complexity
- Ongoing work: method refinement
 - Evolution scenarios
 - Complexity measurement
 - Complexity growth prediction

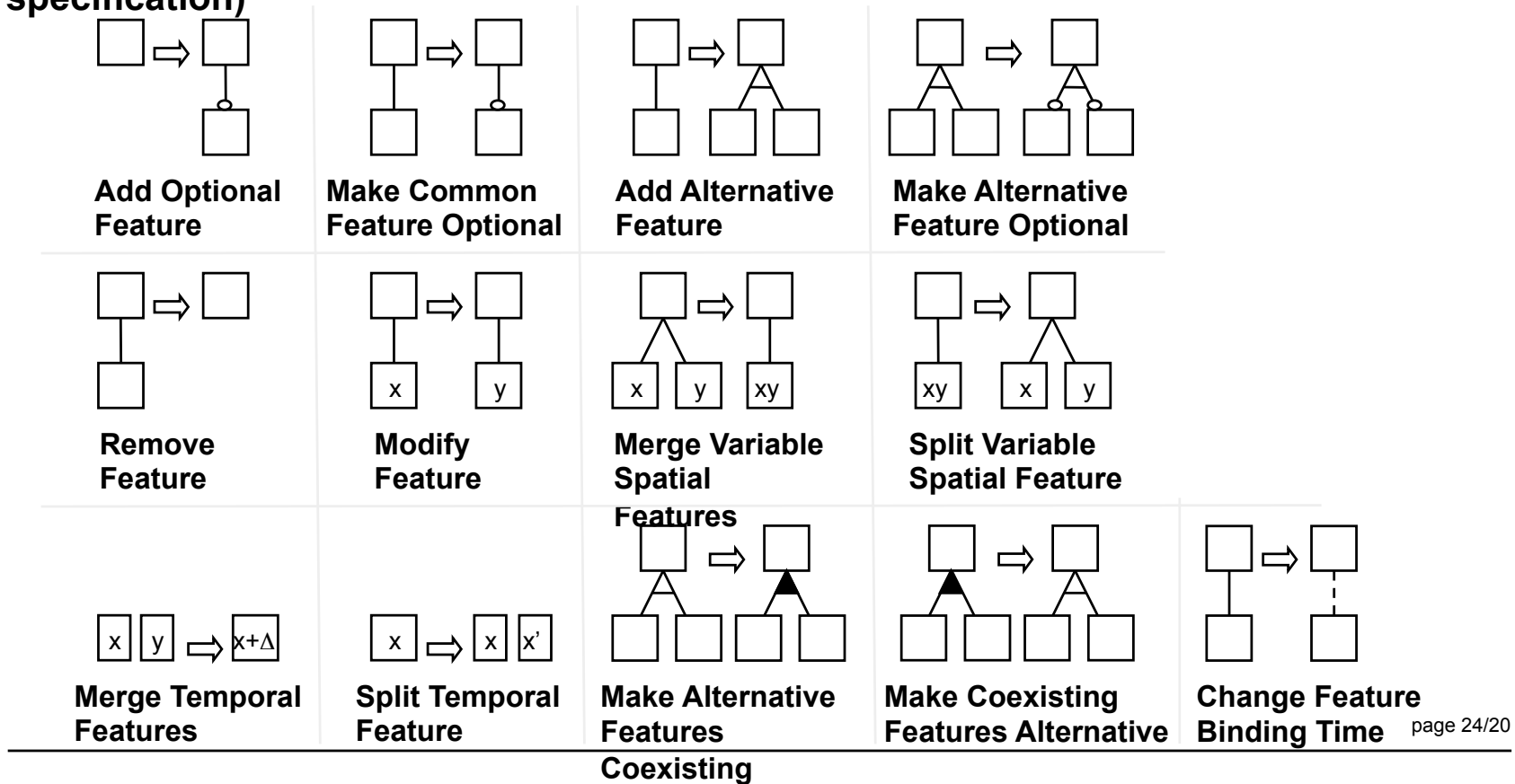
Extra Slides

Example: Conditional Compilation






1	Intent	Decouple common from variable source code, so that the variable code is highlighted, and can be automatically included or excluded from compilation. CC allows you to manage optional or alternative variable code next to common code, without introducing new modules.
2	Motivation	Sensor node transceiver pseudocode before and after adding an optional variability
3	Applicability	1) small crosscutting f., 2) no effect on existing prod., 3) developm. & production code coupling
4	Structure	Dependencies among common & variable source code, configuration etc.
5	Participants (& Responsibilities)	1) Source Code Modules (transceiver.h/.c): a) contain common & var. parts, b) serve as PP input 2) Config. Module (Makefile): a) configures product, b) make conf. persistent, c) ... 3) Macro Definition (-DHAS_ACKNOWLEDGE): ... 4) - 8) ...
6	Dynamics	Behavior of structural artifacts
7	Consequences	+ emphasizes var. parts; + allows var. parts to crosscut at arbitrary boundaries; + no efficiency penalties; o limited support for defaults; - couples common & var. parts; - closed parameters only; - no compiler error handling; - exponential growth of possibilities; - no inconsistency check; - no black-box component support
8	Implementation	1) Macro Definition: (Makefile: -D/-U vs. config. headers), 2) Macro Usage (#ifdef/#if/...), 3) Defaults/ neg. variabilities, 4) Evolution (versioning idiom), 5) Optimizations, 6) Macro Naming (conventions), 7) Tools (ifnames, diff, ifdef-mode, autotools, M4)
9	Sample Cod.	Versioning idiom for different SDCC compiler versions
10	Known Uses	Coplien (#ifdef DEBUG), Labrosse (µC/OS-II conventions), open-source-idiom (config.h), ...
11	Related P.	Cloning (ch.2):..., Cond. Execution (ch.3):..., Late Module Binding (ch.5):..., FT (ch.8):...



Basic Product Line Evolution Scenarios (covered by PL specification)



Legend

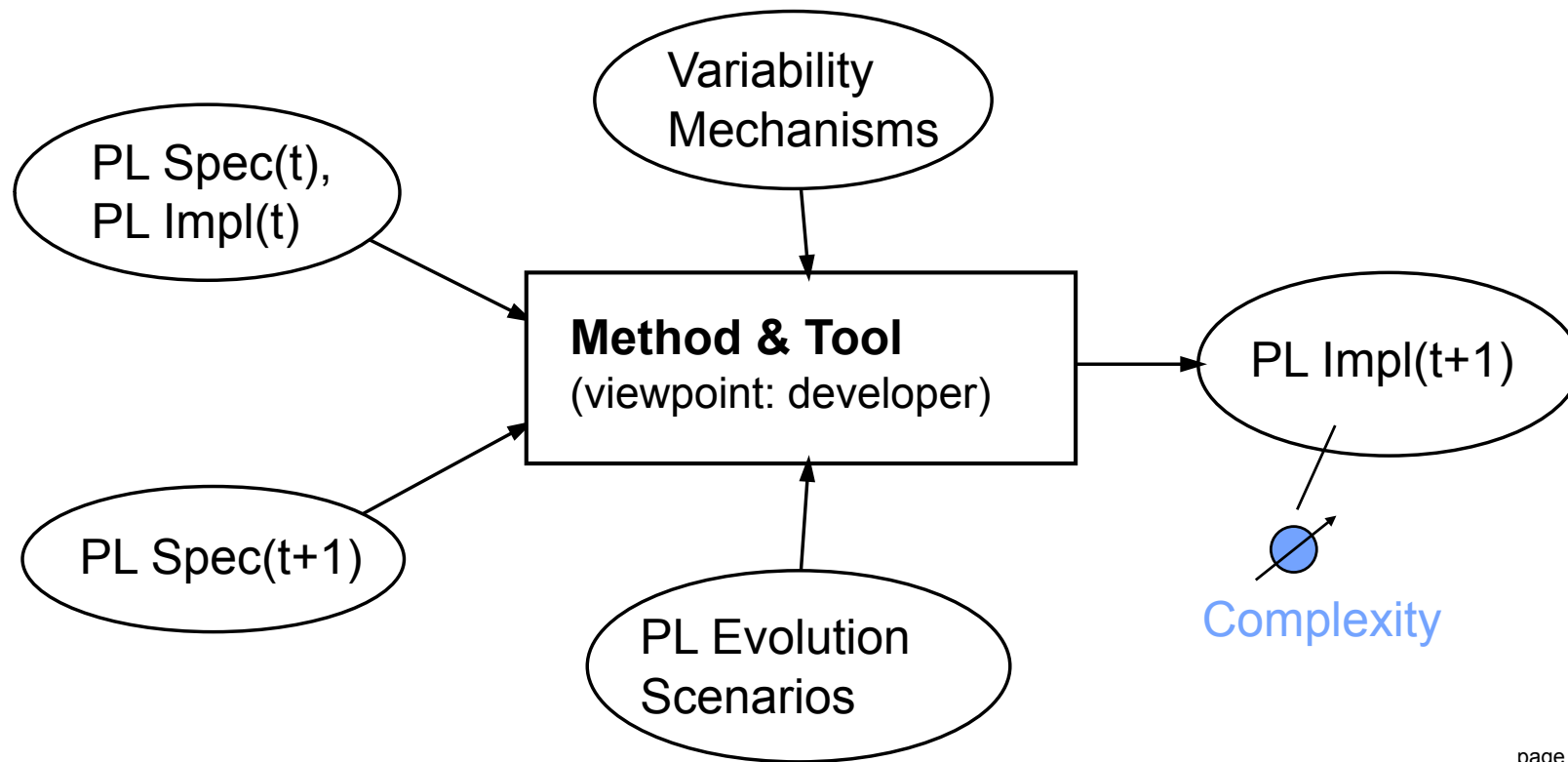
-  Feature
-  Composed-of
-  Optional
-  Alternatives
-  Coexisting Poss.

Scenario Example: Add Optional Feature

Developer sub-activities:

0. **Verify Specification Assumptions**
(Does the code already implement a similar feature?)
1. **Create New Feature** (yet without considering variability management => treat feature as commonality; unit-test)
2. **Identify Variation Points** (Do they have procedural boundaries? Degree of crosscutting?)
3. **Select Variability Mechanism**
(without refactoring the existing code too much)
 - Measure effects (verify simplicity)
 - Optimize (e.g. introduce defaults, merge variation points)
4. **Create Configuration** (focus: automated product creation)

page 25/20



Measurement Goal Categories

- **Economy:** evolving reusable code with modest effort (not just applying techniques)
- **Simplicity:** consciously implementing just what is necessary, but not more (e.g. leaving out not-yet-needed variation points)
- **Correctness:** not realizing less than specified (e.g. omitting a required variability)
- **Evolvability:** ease-of-reuse over time (continuous, not just one-time reuse)
- **Configurability:** ease-of product instantiation (e.g. optimizations by defaults or VP reductions)

Complexity Measurement (Ongoing Case Study)

Task: Perform the same PL evolution scenario

1. Use 'cup' reference product, add optional energy mgmt. feature
2. Add 'stick' product
3. Add 'presence sensor' product
4. Add optional time awareness feature
5. Change energy management feature

for these mechanism selections:

- a. Cloning only, b. Conditional Compilation only, etc., vs.
- z. Varying, dependent on current state

Complexity quantification:

Product-based: $cplx_1(t) = f(impl(t-1), spec(t))$
(idea: optimize the 3 reuse properties, measuring LOC, $v_{CT}(g)$,
#mechanisms, #VPs, mechanism appropriateness for scenario,...)

Process-based: $cplx_2 \sim \min. \#steps$
(idea: find a sufficiently small sequence of operations to transform
one PL impl. to another, using the basic change operations
addition, removal, substitution -> Levenshtein Distance)

28/20

Tool Support

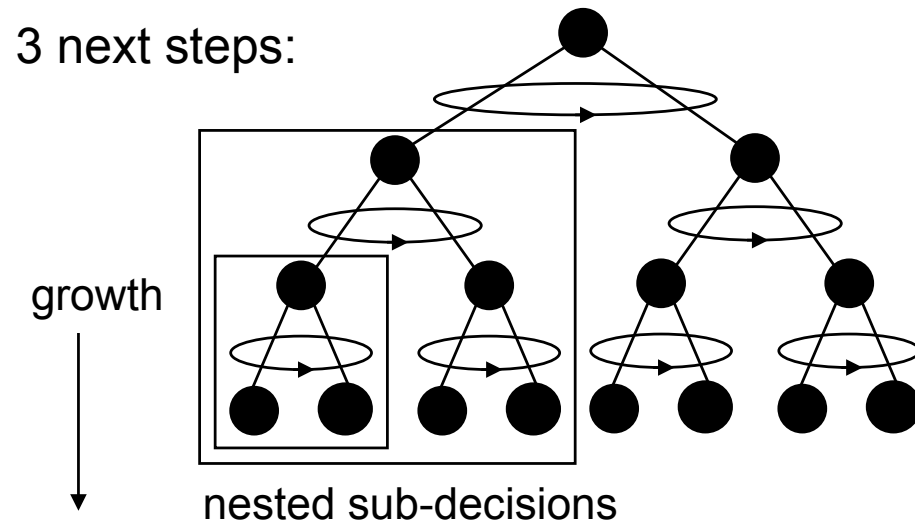
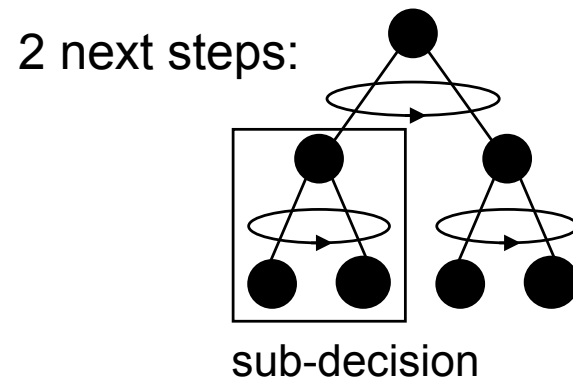
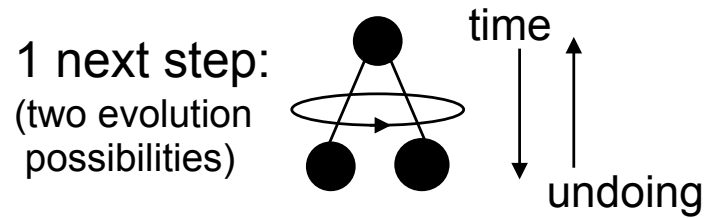
Features

- Assistance for performing the method sub-steps in the right order (dialog-based), includes generic PL evolution scenarios
- Variability mechanism browser for traversing the pattern language (HTML- or PDF-based)
- Complexity calculator (semi-automatic, with simple parser (line ctr.,ifnames), similarity tester)
- Logging facility for activities, temporary states & complexities
- Time support (conceptual backtracking & what-if-evaluation)

Technology: Eclipse, Jython, PLY

page 29/20

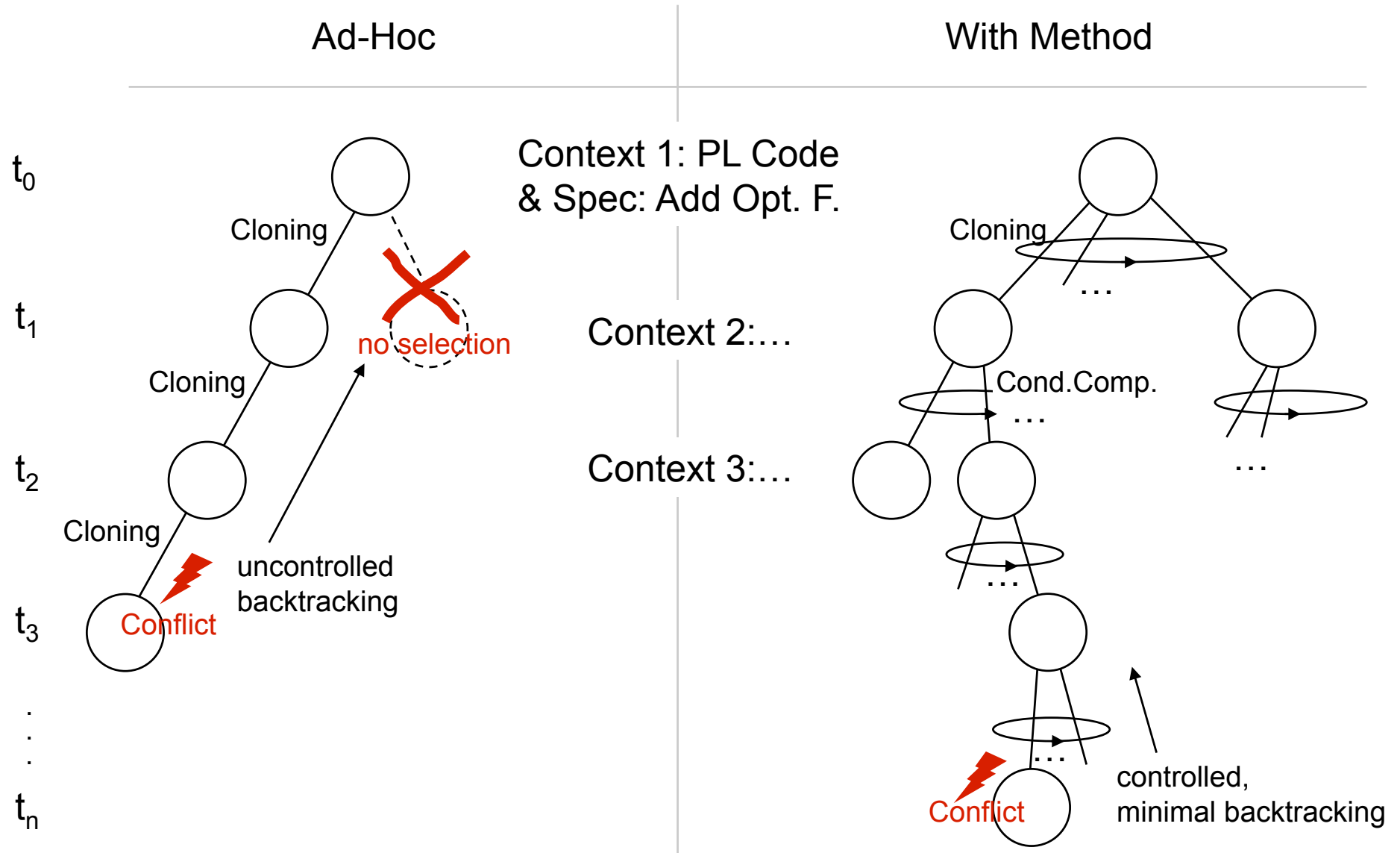
Generic Selection Model



Legend:

- Method Inputs
- / Decision
- Selection Range

Selection Model Example



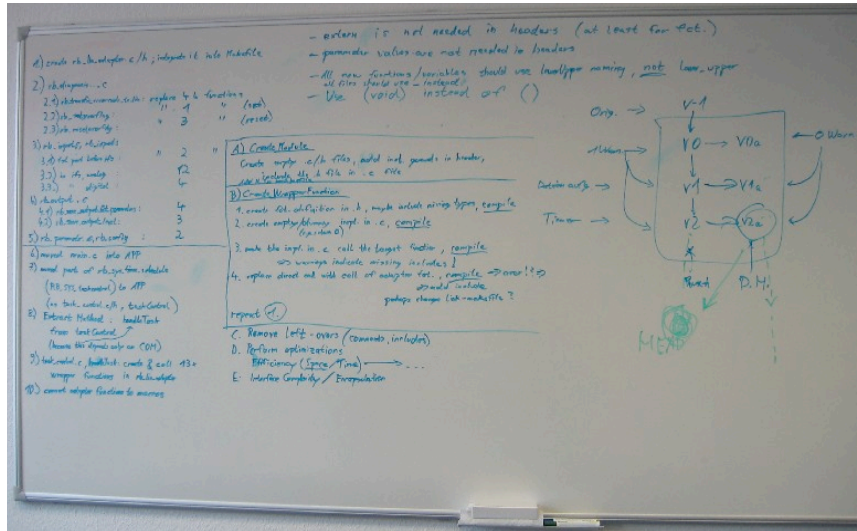
Measurement Goal Example

Analyze the **implementation of SW systems / PLs**
for the purpose of **improving**
with respect to the **simplicity**
from the viewpoint of the **SW developer as producer**

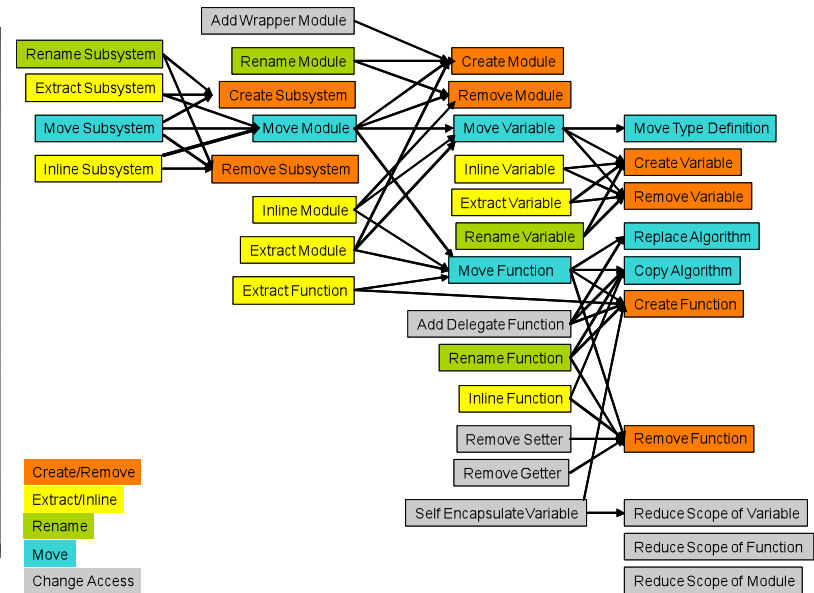
Question sub-categories: encapsulation, coupling,
binding times, specification- and code-focus

Uniform metrics: $X=1-A/B$, $0 \leq X \leq 1$, 1 better
e.g.: $A=\#(\text{unnecessary late bindings})$, $B=\#\text{bindings}$

Implementation Steps We Controlled in Practice



Bosch-EB



Ricoh-ICS

[back](#)



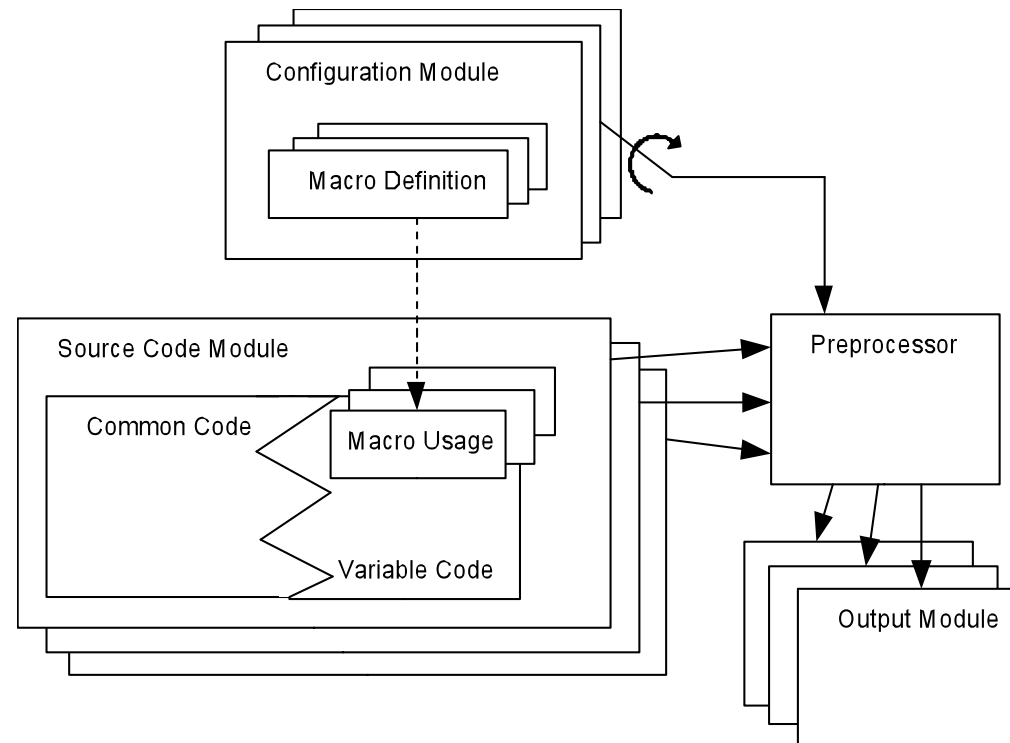
CC Motivation: Example Pseudocode

file name	before	after
transceiver.h	<pre>... extern bool acknowledged; void send(char*); ...</pre>	<pre>#if HAS_ACKNOWLEDGE==1 extern bool acknowledged; #endif void send(char*);</pre>
transceiver.c	<pre>void send(char* msg) { initialize transmission acknowledged=false; for n iterations { send message if acknowledge received ... }</pre>	<pre>void send(char* msg) { initialize transmission #if HAS_ACKNOWLEDGE==1 acknowledged=false; #endif for n iterations { send message #if HAS_ACKNOWLEDGE==1 if acknowledge received ... }</pre>
Makefile	<pre>...</pre>	<pre>... CFLAGS+= -DHAS_ACKNOWLEDGE=1 ...</pre>

[back](#)

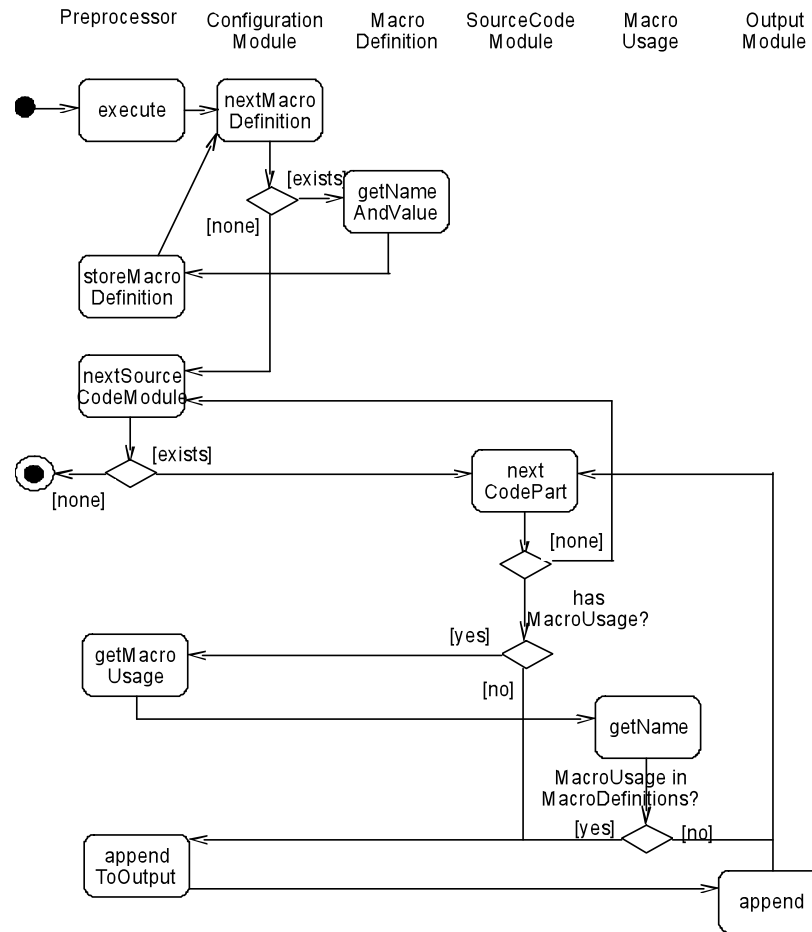


CC Structure



[back](#)

CC Behavior



back