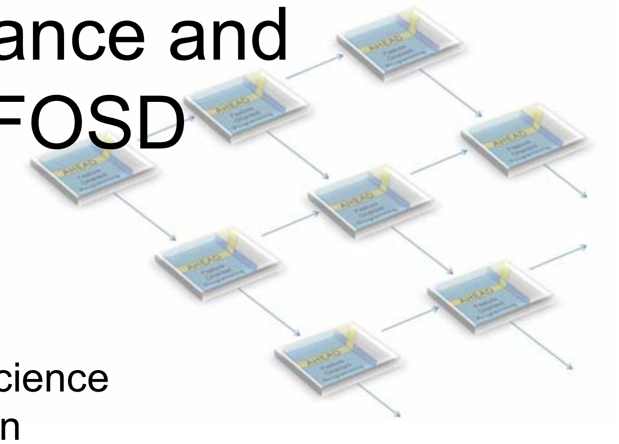
On The Importance and Challenges of FOSD



Don Batory
Department of Computer Science
University of Texas at Austin

October 6, 2009

Introduction

- I’ve been building programs by composing features for 25 years
  - called them “legos” back then (no term for ‘features’)
  - prior to work on software architectures, components, mixins
  - at the time when step-wise refinement was abandoned (back then it didn’t scale)
  - my background was databases, not software engineering
  - my ignorance in software engineering was a blessing; then current issues were irrelevant to what is now FOSD

Initial Work: Genesis

- Created family of database systems from features
    - I didn’t know about product lines back then (not sure term existed)
    - “legos” that I could snap together to build different DBMSs
        - » never been done before
    - DBMS community reaction was interesting
        - » they were interested in DB technology, not software technology
    - Remember visiting Digital Equipment’s Database Program in Colorado

“Our software is too complicated to be built [redacted]!”

- Key issue was taming software complexity
    - $$ expired and Genesis wasn’t finished
    - if it wasn’t for a feature-based structure, I could not have finished it
    - not a DBMS problem, it was a core problem of software engineering

---

Initial Work Continued

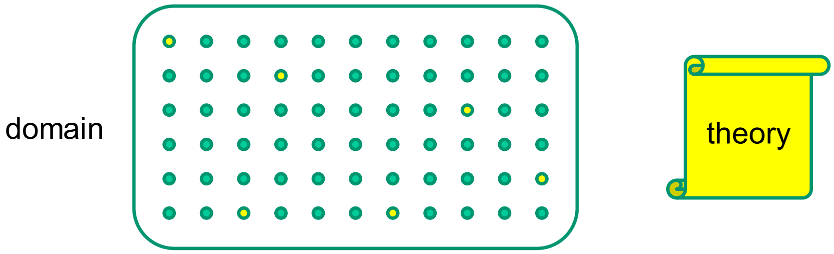
- Little to go on:
    - find a domain and understand it
    - identify features and their meaningful combinations
    - create tools to define feature modules and compose them
- After a few domains, you see similarities and patterns

“We’ve done this so often,
we’ve got it down to a science…”

- Start of a Science of (Automated) Design…

Scientific Part from Physics

- Disparate phenomena, want a theory to both explain them and predict others

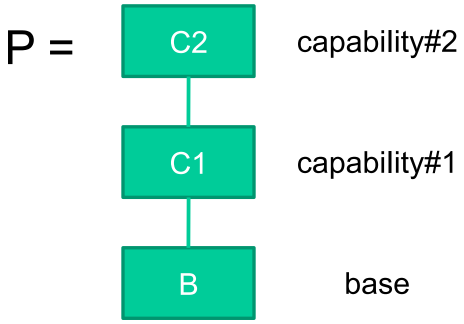


- Programs that do similar things are existing phenomena
  - theory is one of atomic construction
  - features are the atoms; feature models define legal compositions
  - the ability to create new programs that had never been written before

---

First Meta-Level Challenge

- I needed a language in which to express my ideas
  - system development = layered development
  - adding one layer at a time
  - layer was an increment in functionality
- Mid-1980s I realized that elementary mathematics provided a language
- Based on an ancient idea
  - programs were values
  - "features" are functions
  - clean model of composition



P = C2 • C1 • B

where • denotes function composition

Distinguishing Characteristics of FOSD

- Start from practice and work towards a theory

- Using mathematics as a ***language*** to define and illustrate its founding principles

    - give precise description to vague concepts

    - mathematics brings **simplicity**, **clarity**, **principled foundation** for automated software design & tools

    - imposes architectural abstractions upon tools, implementations; other contemporary approaches have no such constraints

---

This Talk

- A (reasonably) fresh view on all of this

Feature Oriented Software Development

- And the challenges ahead, as identified by polling members of the Software Engineering Community

- Am grateful to those who responded to my survey
    - “Quotes”
    - “(paraphrasing)”

- Regrets…
    - see my web site (soon)

Before I move on

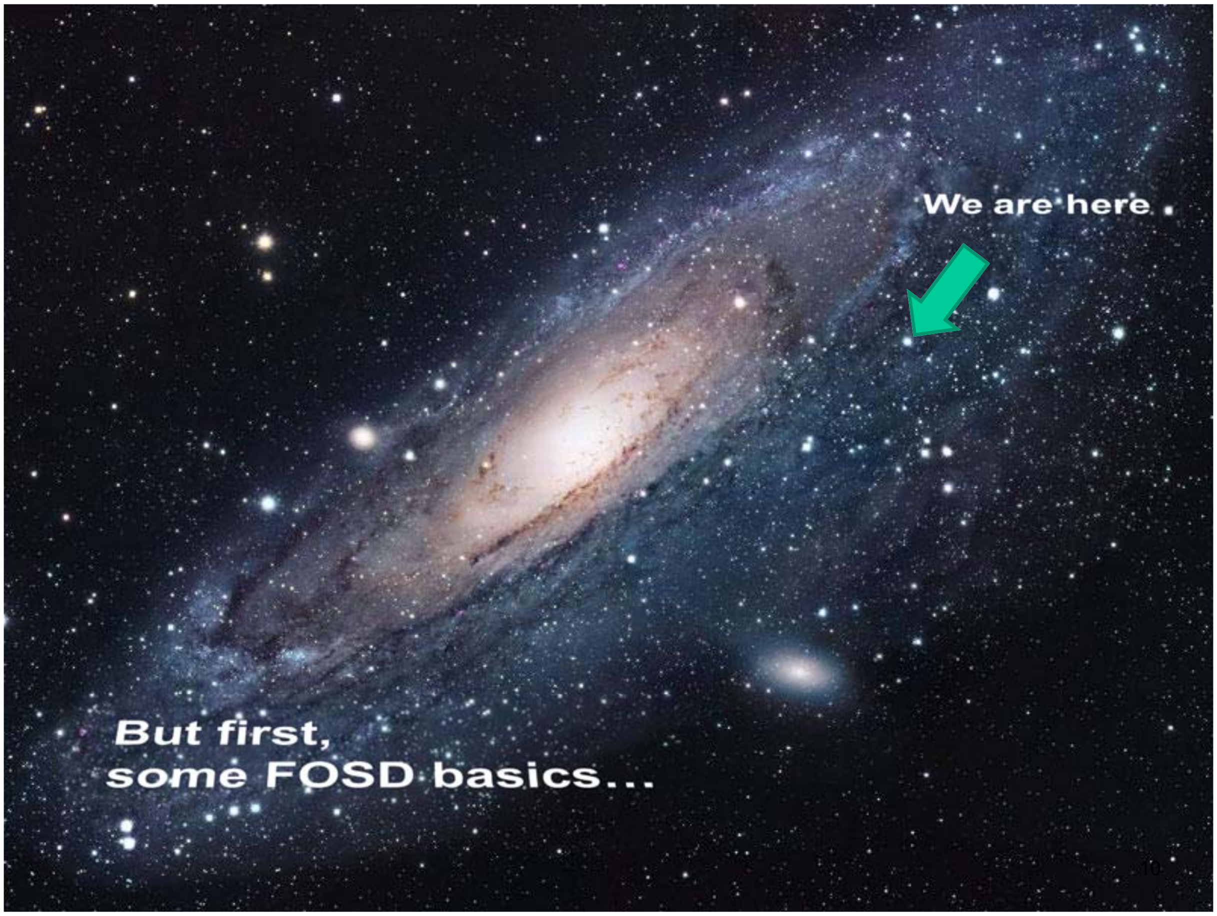
- Key: Stepwise Refinement bridges practice & theory
- The immediate challenges today for FOSD are:

Tools

Case Studies

Simple Theory

Repository of Papers



We are here

But first,
some FOSD basics…

Basic FOSD

- Domain is defined by a set features
  - **feature** is an increment in program functionality
  - build programs by composing features

F1, F2, F3, …. Fn

- Symmetric description (inspired by work of Apel & Lengauer 2008)
  - only one base program (empty program) ∅
  - all features are unary functions F(x)
  - start with nothing, add this, add this, add this, to produce program P

$$P = F7 \bullet F6 \bullet F5 \bullet F2 \bullet F1 \bullet \varnothing$$

Basic FOSD

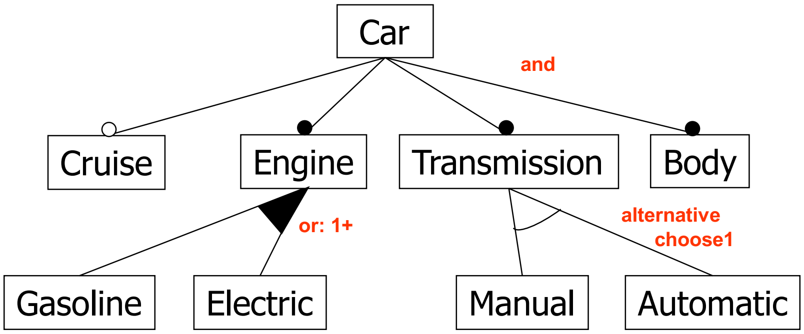
- Since ∅ is always present, drop it to simplify notation

$$P = F7 \bullet F6 \bullet F5 \bullet F2 \bullet F1 \bullet \varnothing$$

- Enables us to express fundamental concepts cleanly…
- ***Design*** of a program is an expression
- Expressions can be optimized → ***designs can be optimized***
  - no greater example than relational query optimization
  - basis of automatic programming
- ***Synthesis*** is expression evaluation

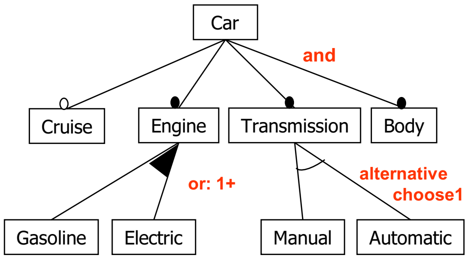
Feature Models

- Not all combinations of features are meaningful
- Role of **feature models** (Kang 1990)
  - declarative graphical DSL for specifying legal combinations of features
  - set of all legal combinations yields a **product line**



Mathematical Foundations

- A feature model is a representation of a propositional formula
  - each feature maps to a Boolean variable
  - variable is true if feature is selected, false otherwise

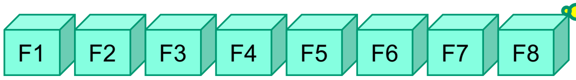


Car⇔CB ^ Car⇔Cr ^
Car⇔Eng ^ Pt⇒Car ^
Tr⇔choose1(Auto,Man) ^
Eng ⇔ (Ele v Gas) ^
Car ⇔ true

  - program = set of selected features that satisfy the formula
  - each solution is a program, set of all solutions is a product line
  - see D. Benavides (SPLC*), K. Czarnecki (GPCE*), T. Thüm (ICSE09) on recent advances in editing and analyzing feature models

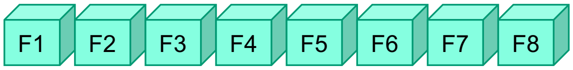
Conjectured Foundations of FOSD

- Express features of a domain not as a set but **as an ordered array**
  - model of a product line is a 1D array of features



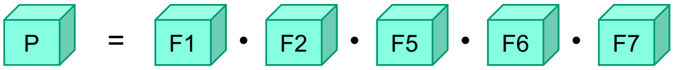
in composition order

- **Projection** – eliminate unneeded features
  - feature models define legal projections



Program synthesis is projecting and contracting 1D arrays of features

- **Contraction** – compose remaining features to produce a scalar



Basic Mathematics

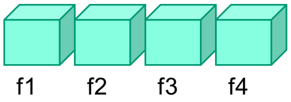
- A function (feature, transformation) can be decomposed into a composition of simpler functions (features, transformations)

$$F_1 = F_{11} \bullet F_{12} \bullet F_{13} \bullet F_{14}$$

- **Principle of Uniformity**
  - use features to build any kind of document (ICSE 2003, TSE 2004) (code, makefiles, documentation, formal models)
- **Multi-Dimensional Models**
  - formalization of Multi-Dimensional Separation of Concerns (ASE 2002, SIGSOFT 2003)
- **Product Lines of Product Lines**
  - evolution of product lines by features (ASE 2002, SIGSOFT 2003, Völter SPLC 07)

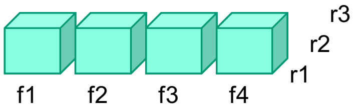
Starting Point

- A product line is an array of composable features (transformations)



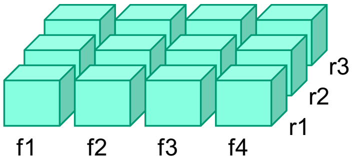
Principle of Uniformity

- All documents (code, makefiles, documentation, models) are refined by features
- Each document type is another dimension



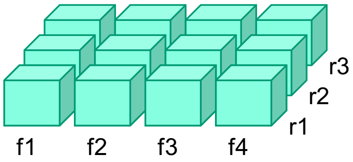
Principle of Uniformity

- All documents (code, makefiles, documentation, models) are refined by features
- Each document type is another dimension



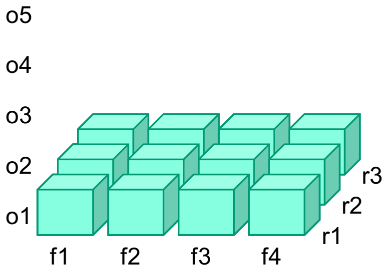
Principle of Uniformity

- All documents (code, makefiles, documentation, models) are refined by features
- Each document type is another dimension



Multi-Dimensional Models

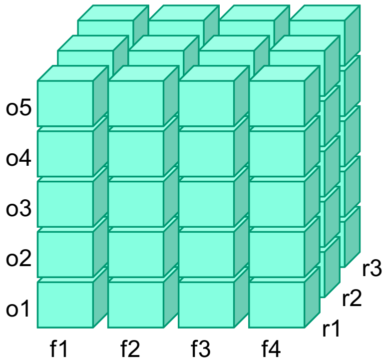
- "Expression Problem" (Reynolds 76, Cook 90, Wadler 98)
  - dimension for data types or data structure
  - dimension for operations



Models of Product Lines are n-dimensional arrays of xforms are Kubes

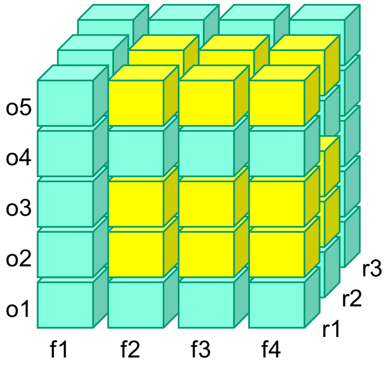
Product Lines are Kubes

- Program P is a subkube:
  - operations (o2, o3, o5)
  - structures (f2, f3, f4)
  - representations (r1, r2)



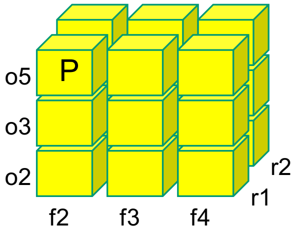
Kube Projection

- Eliminates unnecessary elements
  - legal projections are defined by a feature model



Kube Contraction

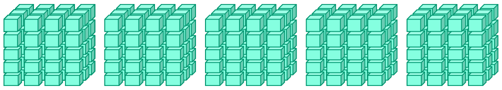
- Composes elements by dimension (in any order) to produce a scalar (expression) – which synthesizes P



Projection & contraction of Kubes seem to underlie mathematical models of product lines

Kube Scalability

- Product line
- Product line of product lines
- Product line of product lines of product lines



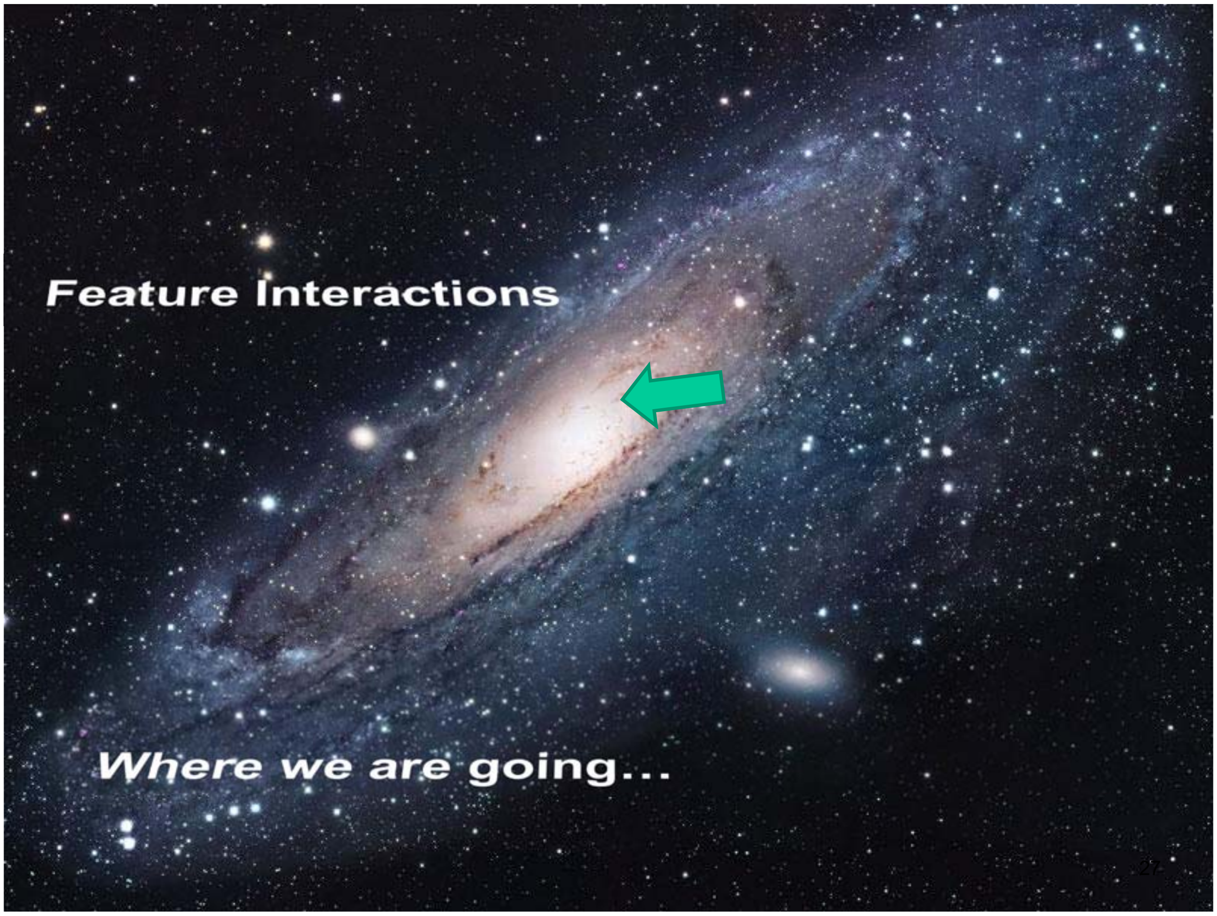
See:

Völter SPLC 07
ASE 2002
SIGSOFT 2003

Recap

- Mathematics is a language that unifies and expresses fundamental principles in FOSD and in software design in a simple way

- Against this backdrop, tremendous challenges lie ahead
    - use Kubes to illustrate some of these challenges



Feature Interactions

Where we are going...

Observations

- “The major problem I come across is Feature Interaction because either due to the feature itself or the state of the code base features interact with each other at the specification or implementation level.”
  – D. Thomas
- “Recovering feature descriptions and interactions within a single legacy system (is very important).”
  – J. Gray
- “Feature implementations that behave correctly in isolation may lead to undesired behavior in combination (feature interaction problem)”
  – S. Apel

Feature Interactions

- For almost 20 years, I *never* noticed interactions among features
  - reason: domains that I studied allowed interactions to be hidden
  - when I added a feature X, I knew all the features Y that had to be present for X to work
  - I extended "hook" methods of Y appropriately
  - packaged the X with its core introductions AND its interactions (how it changed other features) as the features in Y were not optional
  - no need to distinguish core introductions from interactions
- **Lesson: don't see or recognize everything in decades of work, so be careful about claims…**

To Credit Others

- Lots of others did notice, esp. in telecommunications
  - bi-annual "Feature Interaction" conference
  - C. Prehofer (ECOOP 1997) 1st to propose usable definition features & interactions
- Current state: Early work by Liu, recent work by Kästner reinforces the need to separate features from their interactions
  - I now see feature interactions everywhere
- **Lesson: Only when you experience 1st-hand a problem will you appreciate research on it**
  - human nature

“Structural” Feature Interactions

- Easy to recognize
- A feature introduces new classes, and adds new fields and methods to existing classes – core of a feature
- Feature F interacts with feature G (F|G) when feature F modifies the introductions of G
- **Interactions are unavoidable** – when you introduce new classes and members, you must integrate their functionality into an existing program by modifying existing introductions

---

4-Program SPL with 3 Features

```
class calculator {
    float result;
    void add( float x ) { result=+x; }
    void sub( float x ) { result=-x; }
}

class gui {
   JButton format = new JButton(“format”);
   JButton add    = new JButton(“+”);
   JButton sub    = new JButton(“-”);

   void initGui() {
     ContentPane.add( format );
     ContentPane.add( add );
     ContentPane.add( sub );
   }

   void initListeners() {
     format.addActionListener(...);
     add.addActionListener(...);
     sub.addActionListener(...);
   }

   void formatResultString() {...}
}
```

new methods

new fields

new fields

extend existing methods

extend existing methods

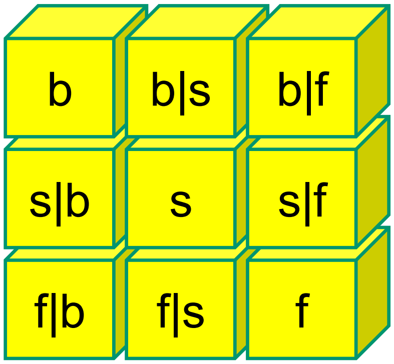
new methods

2.500

format • sub • base = p_4

Visualize 2-Way Interactions

- As a 3×3 kube:



b	b|s	b|f
s|b	s	s|f
f|b	f|s	f

base

sub

format

4-Program SPL with 3 Features

```
class calculator {
    float result;
    void add( float x ) { result=+x; }
    void sub( float x ) { result=-x; }
}
class gui {
   JButton format = new JButton(“format”);
   JButton add    = new JButton(“+”);
   JButton sub    = new JButton(“-”);

   void initGui() {
     ContentPane.add( format );
     ContentPane.add( add );
     ContentPane.add( sub );
   }
   void initListeners() {

     add.addActionListener(...);
     sub.addActionListener(...);
   }
   void formatResultString() {...}
}
```

new methods

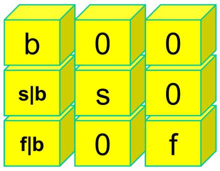
new fields

new fields

extend existing methods

extend existing methods

new methods



b	0	0
s|b	s	0
f|b	0	f

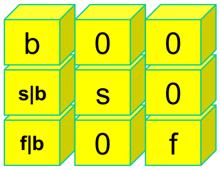
format • sub • base = p_4

4-Program SPL with 3 Features

```
class calculator {
    float result;
    void add( float x ) { result=+x; }
    void sub( float x ) { result=-x; }
}
class gui {
   JButton format = new JButton(“format”);
   JButton add    = new JButton(“+”);
   JButton sub    = new JButton(“-”);

   void initGui() {
     ContentPane.add( format );
     ContentPane.add( add );
     ContentPane.add( sub );
   }
   void initListeners() {

     add.addActionListener(...);
     sub.addActionListener(...);
   }
   void formatResultString() {...}
}
```



b	0	0
s|b	s	0
f|b	0	f

$\text{format} \bullet \ \text{sub} \bullet \ \text{base} = \ p_4$

Degrees of Interactions

- Matrix captures 2-way interactions
  - 3-way interactions require 3D kube
  - n-way interactions require nD kube (ouch…)
- Know: 3+ way interactions are rare in telecommunications
  - don’t seem particularly common in software either
- **Q: what tools do we need to visualize and recognize structural interactions?**
- See:
  - Kästner’s Colored IDE (CIDE)
  - Czarnecki’s template models

An Interesting Observation

– “I think a major open problem in (FOSD) and related approaches is the reconciliation with modularity. This may sound contradictory, since the goal of (FOSD) is to increase the modularity, but on the other hand (FOSD) is also often anti-modular in that it presupposes a global view on the software or software domain.”
– K. Ostermann

- Features interact – no way to avoid it
- If you build programs by composing features (but ignore their interactions), your program will likely not work
- Period

Does this Mean That…

- Features are anti-modular?
- This is a fundamental problem:
    - features are building blocks of programs
    - they are also building blocks of modules
- Modules (compound features) interact
    - see Sullivan 1994 on mediators
- If we want declarative specifications where users select the features that they want, a domain engineer MUST understand how features interact if synthesized programs are correct – this requires a global view

Evolution of Product Lines

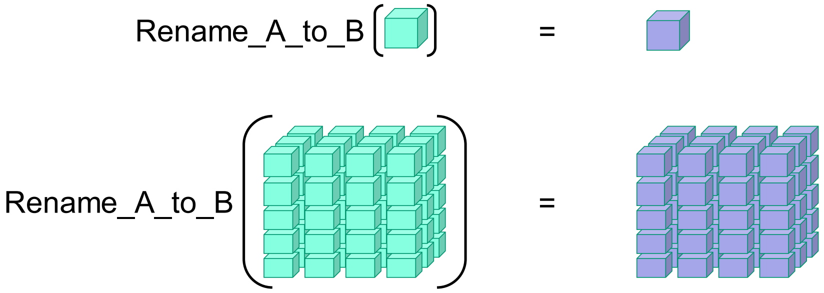
Where we are going…

Evolution of Product Lines

- “How (do we) ensure consistency of feature selections and correctness of generated software in a way that supports evolution and maintenance?”
  – P. Sestof
- “(Evolution is mostly a manual process; can more be automated?)”
  – S. Jarzabek
- “New customer requirements, technology changes, and internal enhancements lead to the continuous evolution of a product line… PLE should thus treat evolution as the normal case and not the exception.”
  – P. Grünbacher

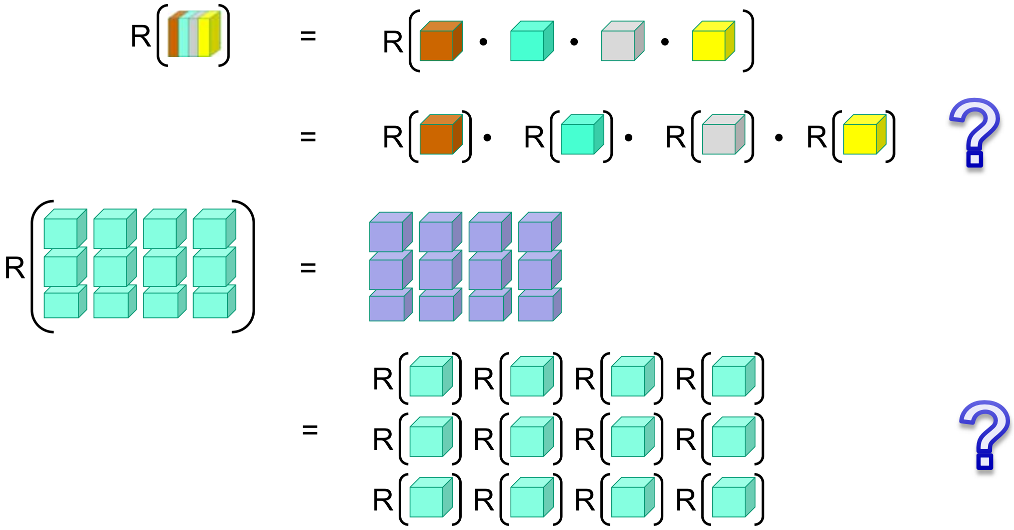
Evolution

- Parnas noted that successful programs evolve
- **Refactorings** are devoted to automating tedious and repetitive tasks of software evolution
- Challenge: how can product lines be refactored?



Conjecture

- Distributivity laws that simplify (parallelize) refactorings of kubes of arbitrary size, compositions



Engineering Challenge

- Most (all?) refactoring engines work on *entire* program, not *fragments* of a program
  - it may be difficult to build tools

- Remember: refactorings affect not just code, but other representations as well (xml)
  - must change **all** representations consistently

- Important area
  - see M. Kuhlemann's work (GPCE 09), R. Johnson & students as steps in this direction

---

Correctness

Where we are going…

Correctness

- Today we can produce 10s, …, 1Ms of customized programs quickly
- Can say nothing on the semantic correctness of these programs
    - correctness (verifying selected properties) in general is hard
    - 2007 grand challenge of Hoare, Misra, Shankar "verifying compiler"
- Hope: we are dealing with very specialized subproblems
    - units of modularity (features) are increments in semantic functionality
    - not arbitrary pieces of code, but refinements
    - features should be based on compatible assumptions, single consistent vocabulary s.t. reasonable analyses and tests are possible
    - this is our biggest advantage – we have already structured the problem
- 3 topics: Formal Methods, Feature Refinement of Theorems, Testing

Formal Methods

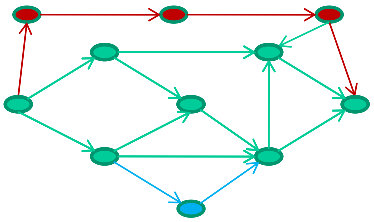
– "How do we verify compositions of features to ensure that the requirements of each feature are satisfied, and there are no conflicts?"
– D. Hutchins

– How can we ensure that our feature implementations (e.g., feature modules) behave as expected both in isolation and in all possible combinations?
– S. Apel

– "How can we be sure of – or at least aware of – the unintended consequences of adding a feature? … Something akin to a spec for a feature is needed."
– S. Nedunuri

Challenge Problem

- If we know property T holds in state machine S
- How can we characterize a refinement of S such that T always holds?



S

property T holds

T should
hold in all refinements

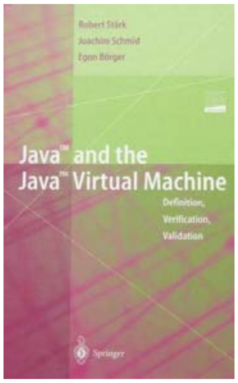
- **Large amount of work on formal modeling**
  - **connect to existing research results**
  - **see: Krishnamurthi & Fisler (circa 2004)**

Feature-Refinement of Theorems

– “Currently, research has focused too much on syntactic issues.”
– P. Heymans

- True
  - but… I see this as general syntactic structure that applies to *all* documents, including semantic documents like theorems

- Revealing example of is E. Börger’s 2001 JBook that uses **Abstract State Machines (ASMs)**
  - ASMs – abstract language based on state machines that makes program development and proofs easier
  - 350 pages of ASM definitions, manual proofs, etc.



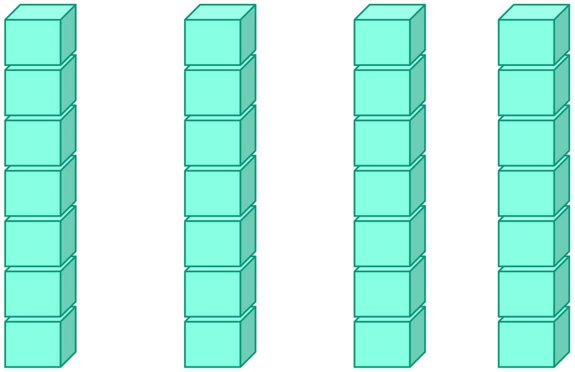
Robert Stärk
Joachim Schmid
Egon Börger

Java™ and the
Java™ Virtual Machine
Definition,
Verification,
Validation

Springer

Overview of JBook

- JBook presents structured way using ASMs to incrementally develop the Java 1.0 grammar, interpreter, compiler, and bytecode (JVM) interpreter (that includes a bytecode verifier)
  - incrementally refine sublanguage of imperative expressions



grammar interpreter compiler JVM

ExpI

StmI

ExpC

StmC

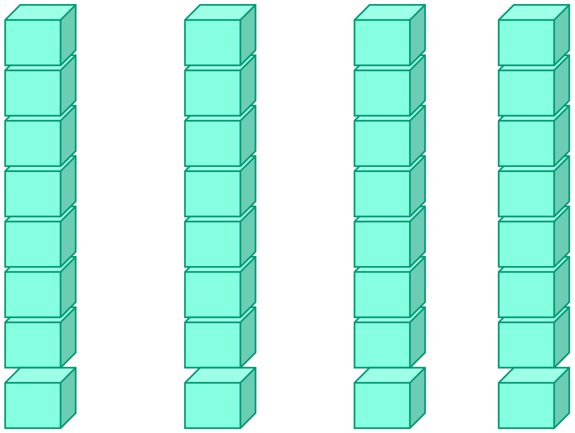
ExpO

ExpE

StmE

Overview of JBook

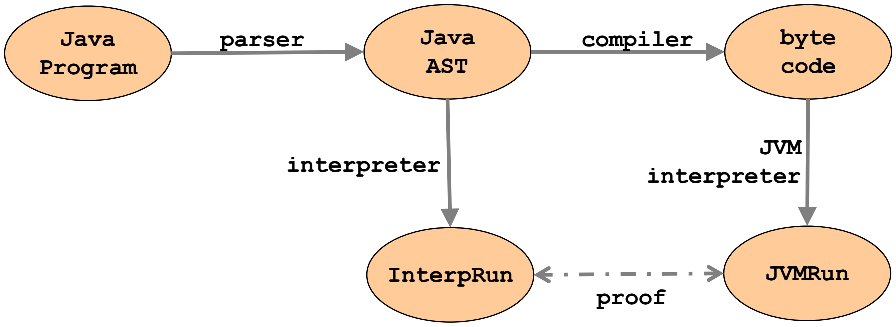
- JBook presents structured way using ASMs to incrementally develop the Java 1.0 grammar, interpreter, compiler, and bytecode (JVM) interpreter (that includes a bytecode verifier)
  - incrementally refine sublanguage of imperative expressions



grammar interpreter compiler JVM

ExpI
•
StmI
•
ExpC
•
StmC
•
ExpO
•
ExpE
•
StmE
=
Java1.0

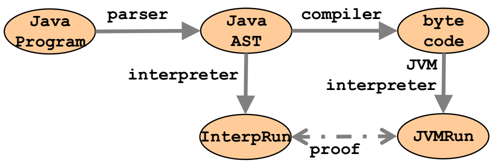
Structure of JBook

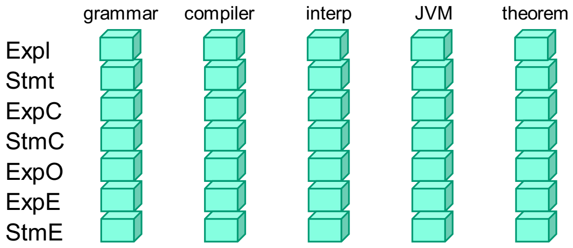


- When entire Java 1.0 is created, various properties are considered
  - ex: correctness of the compiler
  - equivalence of interpreter execution of program and the JVM execution of compiled program
- Here's what we realized…

Apply Principle of Uniformity

- Modularize and refine theorems like other program representations

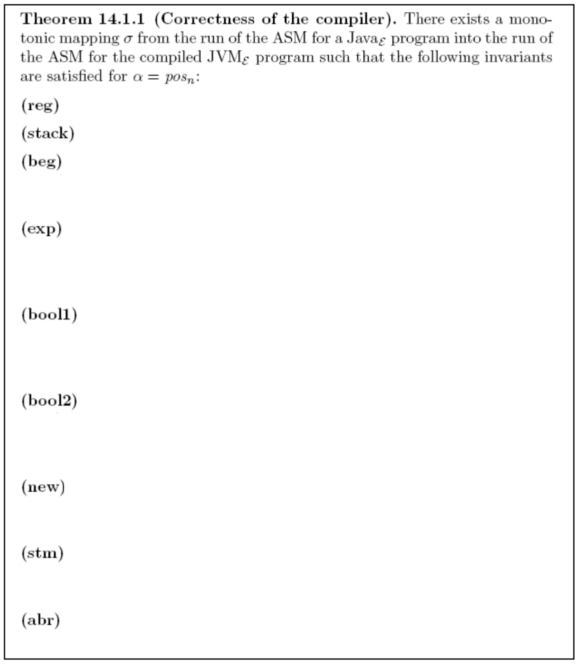




- What did we learn?
  - Theorem of Correctness: Statement
  - Theorem of Correctness: Proof

Theorem T is Correctness of Compiler

- Statement of T is a list of invariants
- 14 invariants in all
- Don’t need to know the specifics of the invariants for this presentation



Theorem 14.1.1 (Correctness of the compiler). There exists a monotonic mapping σ from the run of the ASM for a Java$_{\mathcal{E}}$ program into the run of the ASM for the compiled JVM$_{\mathcal{E}}$ program such that the following invariants are satisfied for $\alpha = pos_n$:

(reg)

(stack)

(beg)

(exp)

(bool1)

(bool2)

(new)

(stm)

(abr)

Statement of Correctness

Theorem 14.1.1 (Correctness of the Compiler). There exists a monotonic mapping σ from the run of the ASM for a Java program into the run of the ASM for the compiled JVM program such that the following invariants are satisfied:

(reg) **(begS)** **(exc-clinit)**

As features are composed, the theorem statement is refined by adding new invariants

there’s more...

(begE) **(stack)**

Java1.0 = StmE • ExpE •ExpO •StmC •ExpC •StmI • ExpI

Invariants Can Be Refined Too!

- Stml feature defines **(abr)** invariant:
  - conditions_1 do not apply to exceptions

```
if restbody_n/A=abr

     then <conditions_1>
```

As features are composed, the theorem statement is refined by the addition of new invariants and the refinement of existing invariants.
(Refinement of existing invariants are feature interactions)

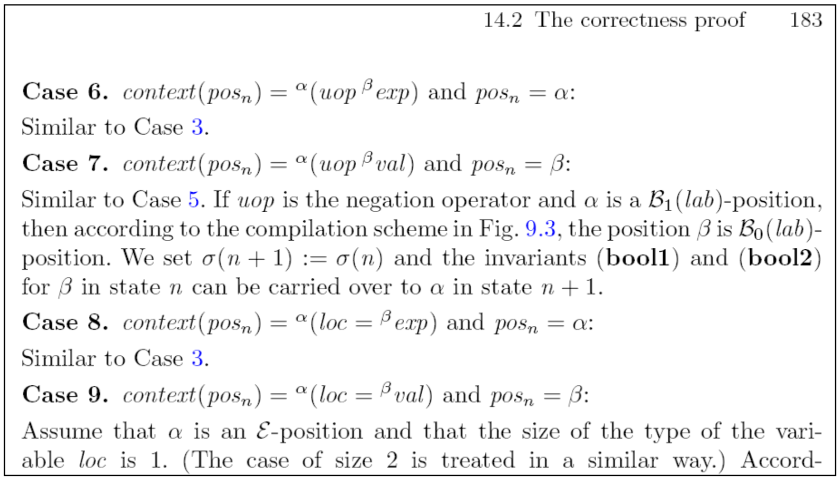
there's more…

- ExpE
  and r
  by ac
- And introduces **(exc)** invariant to cover case when abruption is an exception

```
if restbody_n/A=abr
   and abr is an exception ...
     then <conditions_2>
```

Proof of Correctness

- Is a case analysis using structural induction to show correctness of compiling each kind of expression
  - Proof is a **list of cases** that show invariants hold
  - 83 cases in all



Case 6. $context(pos_n) = {}^{\alpha}(uop\,{}^{\beta}exp)$ and $pos_n = \alpha$:

Similar to Case 3.

Case 7. $context(pos_n) = {}^{\alpha}(uop\,{}^{\beta}val)$ and $pos_n = \beta$:

Similar to Case 5. If uop is the negation operator and α is a $\mathcal{B}_1(lab)$-position, then according to the compilation scheme in Fig. 9.3, the position β is $\mathcal{B}_0(lab)$-position. We set $\sigma(n+1) := \sigma(n)$ and the invariants (**bool1**) and (**bool2**) for β in state n can be carried over to α in state $n+1$.

Case 8. $context(pos_n) = {}^{\alpha}(loc = {}^{\beta}exp)$ and $pos_n = \alpha$:

Similar to Case 3.

Case 9. $context(pos_n) = {}^{\alpha}(loc = {}^{\beta}val)$ and $pos_n = \beta$:

Assume that α is an $\mathcal{E}$-position and that the size of the type of the variable loc is 1. (The case of size 2 is treated in a similar way.) Accord-

As Features are Added…

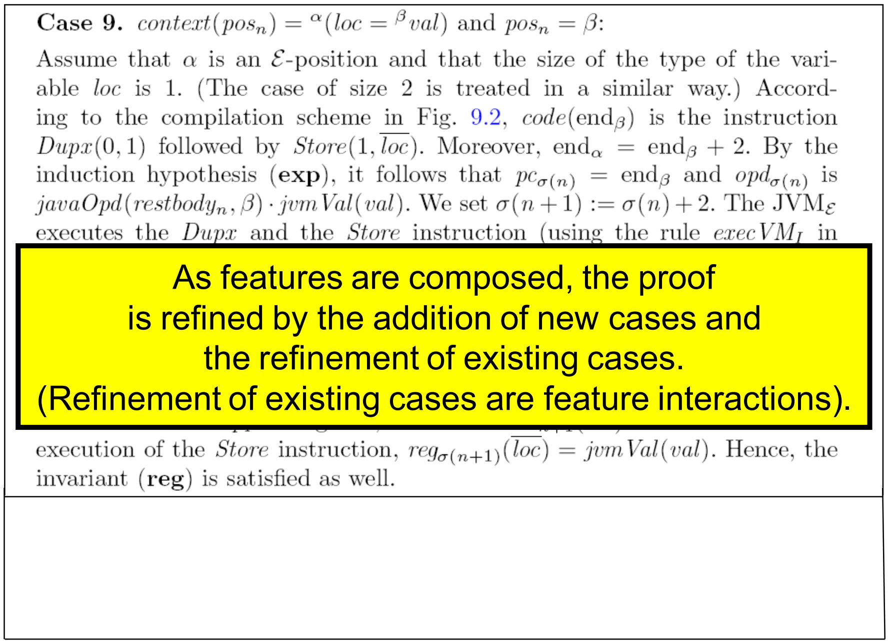
- New proof cases appear
- Theorem gets understandably longer in a very structured and controlled manner

Composition	total # of cases in Proof of Theorem `T`
`j1 = ExpI`	`13`

we introduce more proof cases, but there's more…

Proof Cases Can Be Refined Too!

original proof case of ExpI



Case 9. $context(pos_n) = {}^{\alpha}(loc = {}^{\beta}val)$ and $pos_n = \beta$:

Assume that α is an $\mathcal{E}$-position and that the size of the type of the variable loc is 1. (The case of size 2 is treated in a similar way.) According to the compilation scheme in Fig. 9.2, $code(\mathrm{end}_\beta)$ is the instruction $Dupx(0,1)$ followed by $Store(1, \overline{loc})$. Moreover, $\mathrm{end}_\alpha = \mathrm{end}_\beta + 2$. By the induction hypothesis (**exp**), it follows that $pc_{\sigma(n)} = \mathrm{end}_\beta$ and $opd_{\sigma(n)}$ is $javaOpd(restbody_n, \beta) \cdot jvmVal(val)$. We set $\sigma(n+1) := \sigma(n) + 2$. The JVM$_\mathcal{E}$ executes the $Dupx$ and the $Store$ instruction (using the rule $execVM_I$ in

As features are composed, the proof
is refined by the addition of new cases and
the refinement of existing cases.
(Refinement of existing cases are feature interactions).

execution of the $Store$ instruction, $reg_{\sigma(n+1)}(\overline{loc}) = jvmVal(val)$. Hence, the invariant (**reg**) is satisfied as well.

part that is added by Stml

Observations on Semantic Documents

- Proofs, like code and grammars, have a similar syntactic structure when given feature representations
- Just one case study – how does it generalize, if at all??
- **Challenge: How can we modularize, compose, and verify proofs by composing features? And do so efficiently?**
- **Lesson: others are working with features that we don’t know about and that don’t know about us. When you find an opportunity to work with them, pursue it vigorously – it will pay off!**

Challenge Problem for PL Types…

- Type systems for languages have soundness theorems
- Feature-extensible languages require a feature-extensible type system, which in turn requires feature-extensible soundness theorems…
- How could such a type system & its theorems be defined?
- Could they be refined incrementally as in JBook?
- See Hutchins09 thesis, Delaware SIGSOFT’09, …

Testing

- Although formal methods can be enormously helpful, we can't prove correctness of everything
  - may be able to prove abstract algorithms correct
  - but not our hand-written implementation
- Eventually, we will have to test…

Testing Product Lines

- "(How do we test a particular product in an SPL?)"
  – M. Grechanik
- "With an SPL of possibly millions of potential products, can we test them all (possibly in parallel)?"
  – C. Kästner
- "The deeper challenge is to understand how to specify the interaction of features within SPL members. It is unlikely that poor humans would be able to properly specify these interactions, which is why testing may solely be focused on the accumulative test cases simply to ensure that feature F, when present, is operating properly."
  – G. Heineman

- **Challenge: feature interactions are the key. Demonstrating that a particular feature F in isolation is "correct" in some sense isn't the problem. How F is altered (structurally, behaviorally) by other features should be our focus.**
  We know these alterations – we should be able to say something useful. Stay tuned…

Testing Product Lines

- “(How does model-based testing fit in?)”
  – B. Cheng, A. Schurr
- “(A key problem) is generating test suites specific to chosen features … and specialize unit tests for a particular specialization of (features)”
  – P. Sestoft
- “Wouldn't it be great to have the ability to generate all possible combinations of features and pass these configurations into a tool that would identify 'detectable interactions' and generate appropriate test cases to ensure proper behavior was being managed? I feel nervous about trusting any auto-generated test cases, but I keep coming across papers whose titles suggest they have techniques to do this, so perhaps this is not as far-fetched as one might think.”
  – G. Heineman

- Little that I can point you to (see Uzuncaova 2008)

Scaling the Number of Features

- “The high number of features and components in real-world systems means that modelers need strategies and mechanisms to organize the modeling space. A particular challenge lies in understanding and modeling the dependencies among multiple related product lines.”
  – P. Grünbacher

- Many domains have a rather small number of features
    - database systems [50…250]
    - fire support systems [50…150]
    - AHEAD [100]
- Automobile companies claim to have 10K features!
    - Windoze [200…1000] easily (treat services as features)

Scaling the Number of Features

- Nature of FOSD and its tool support will change
- Large #s of features do exist
- Central problem: where are the examples???
  what are the domains???
- **Lesson: if you have such a domain, you’re set for years!**
    - some thoughts…

Time to Look Elsewhere…

- “It would be helpful to identify some "killer apps" – it would help the rest of the community better understand and appreciate the value of features.”
  – B. Cheng

- Lot of work on customizing Linux

- Linux itself has lots of features – may not be in the exact form that we want, but so what?

Time to Look Elsewhere…

- Another source: Eclipse (2M+ LOC)
  - plug-ins are large-scale features
  - feature dependencies too!

- Open Universe
  - clearly large #s of features
  - requires different implementation of feature composition

  Observation: as features become “conceptually larger” standard notions of frameworks, plug-ins and standard interfaces become a natural way to implement features

- **Challenge:**
  What are large-scale features?
  How are they implemented?
  How does the theory change (if at all) ?
  How do features compose dynamically?

Where we are going...

Language Support

Observations

- “This area won't really be solid until it is integrated in mainstream tools and programming languages, i.e., not macros, or obscure meta-programming, but a change in paradigm with full language support. (We must emphasize) modular type safety, so that configurable components have a meaning on their own without having to consider the entire program they participate in.”
  – Y. Smaragdakis
- “We would like to have modular type checking and separate compilation of features for certain specific languages, like Java, instead of a generic tool which pre-processes source files.”
  – D. Hutchins
- “I believe that FOSD must be an integral part of the underlying programming language, rather than being implemented via external tool chains.”
  – K. Ostermann

Personally…

- I've been disappointed in the general PL community
    - basic ideas of feature modules circa 1990 with mixins, nested mixins
    - I went route of preprocessors waiting for better languages to arise
- What I've learned (to quote Bracha) : costs of learning, tooling and interoperability argue for the status quo (on programming languages)
    - ex: if we add a "feature" construct to an existing language, do you honestly think that people will rewrite their refactoring engines to accommodate this?
    - maybe if you're Sun (Oracle), but not us…

Fundamental Question

  – "Is (feature-based development) a language problem, a design problem, or a tool problem?" – C. Kästner

- Yes, it is obviously a design problem – we are creating new design and synthesis methods
- Yes, it is a language problem, but we can't wait for PL community to help
    - work on languages MUST continue (see S-S. Huang PLDI'09)
- No, in interim, main thrust must be through the development of non-invasive tools that don't require language support
    - see C. Kästner (ASE'08), B. Delaware (SIGSOFT'09)
- Advice: press on with both tool & language support

There is much, much more

but I've run out of time...

Closing Thoughts

- “I am living in the reality where beautiful theories are murdered by gangs of ugly facts” – J. Bosch (knowingly misquoting T. Huxley)

- True. We all are in this same reality.

- Been this way since the beginnings of classical Science
  - Copernicus 1500s
  - Weight of evidence was against him
  - His only evidence was that his heliocentric theory was simpler
  - He didn’t have all the answers: If the Earth was moving, why didn’t we feel it?
  - Heliocentric theory meant that stars could be different distances from earth, and as the earth moved in its orbit, we should see movement of these stars (called parallax). But no one ever saw this (because movements were so small). Copernicus could only contend that all stars were too distant to see parallaxes.
  - http://www.friesian.com/hist-2.htm

I Don't Have All The Answers, Either…

- Immediate Challenges

Tools

Case Studies

Integrative Theories

Repository of Papers

- Long Term Challenges

Feature Interactions

Evolution of Product Lines

Correctness

Scaling Case Studies

Languages vs. Tools

Our Job is to Look to the Future…

- Find the scientific principles behind software design and program synthesis
    - can't possibly be ad hoc – there is a simple mathematics behind it
- There's evidence that features (increments in functionality) can help
- FOSD is tied to incremental development (refinements)
    - key to controlling complexity
    - **that will never grow old**
- Astonishing results are waiting to be found
    - if you don't look, you won't find them
    - if you do, you will

Special Thanks To

First	Last
M	Aksit
S	Apel
E	Börger
J	Bosch
B	Cheng
W	Cook
E	Denny
U	Eisenecker
C	Ghezzi
J	Gray
M	Grechanik
P	Gruenbacher
G	Heineman
P	Heymans

First	Last
D	Hutchins
S	Jarzabek
J-M	Jezequel
C	Kaestner
R	Lopez-Herrejon
J	McGregor
S	Nedunuri
K	Ostermann
A	Schurr
P	Sestoft
Y	Smaragdakis
D	Thomas
S	Trujillo
F	van der Linden
M	Völter

- To see their ideas, see my web site (to be updated shortly)