this.splash 2016
Amsterdam Sun 30 October – Fri 4 November 2016

SPLASH-I nsight nnovation nteraction nvited ndustry

How Domain Requirements shape Languages

Language Design Experience from 10 Years of Building DSLs

Markus Völter

voelter ingenieurbüro für softwaretechnologie // itemis

voelter@acm.org
www.voelter.de
@markusvoelter

DOMAINS

- Health & Medical
- Automotive
- Aerospace
- Robotics
- Finance
- Embedded
- Science
- Government

Software Engineering

Requirements Engineering

Specification and Verification

Architecture

Safety and Security

Implementation and Test

Language Engineering

Mixed Notations and End User Programming

Informal -> Semiformal -> Formal

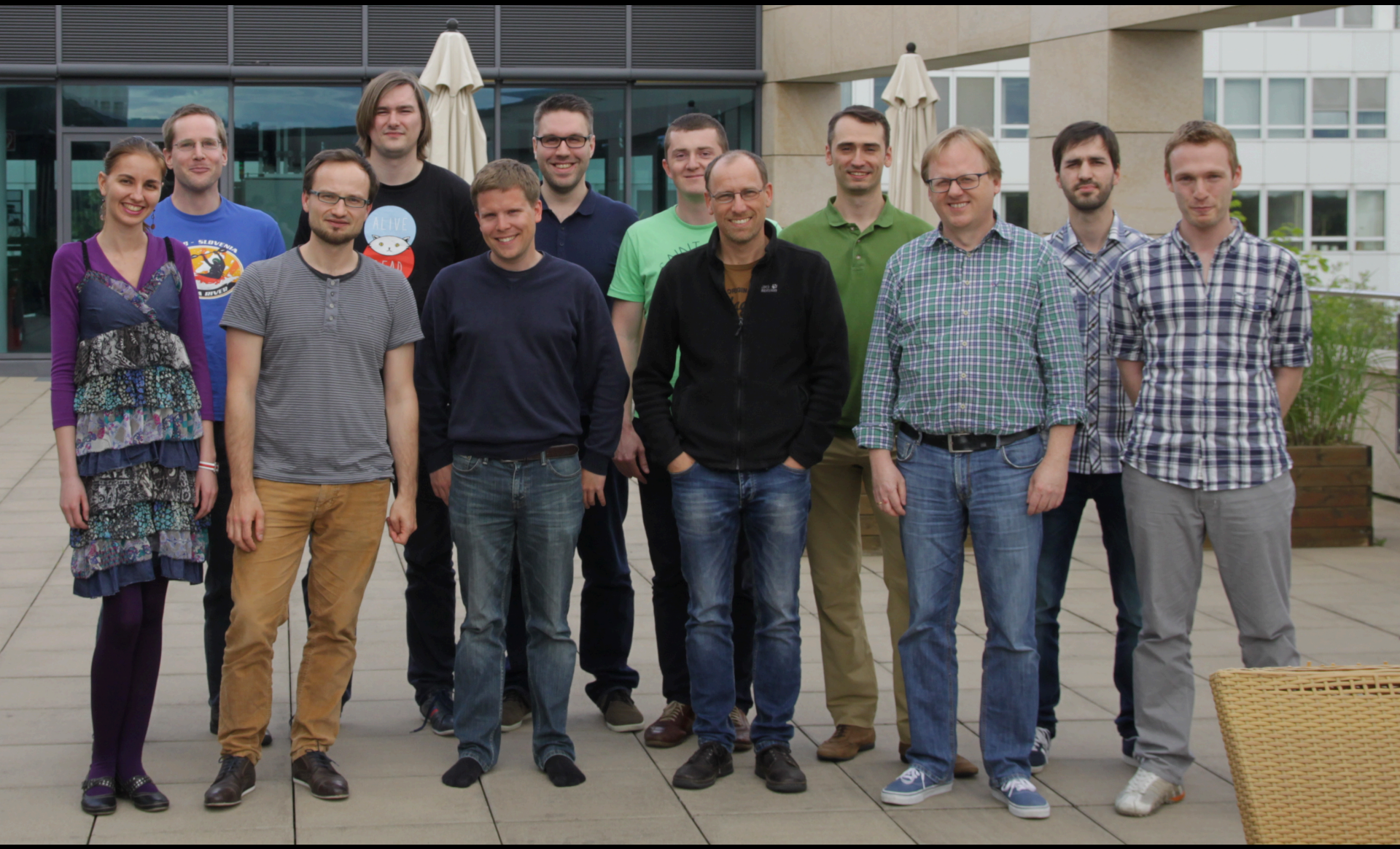
Languages + Verification

Optmizations, Performance and Concurrency

Fundamentals: Editors, Type Systems, Trafos

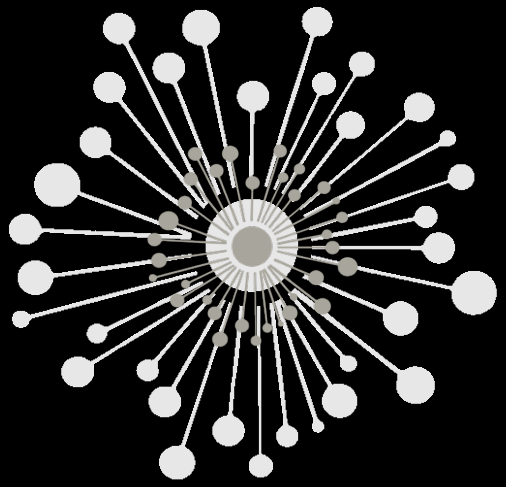
User-Friendly IDEs, Tools

Methodology



How
Domain
Requirements
Shape
Languages

1



What is a Language

Metamodel for Business Logic

Clearly defined data structure to express all business-relevant structures, behaviors and non-functional concerns.

Metamodel for Business Logic

- Data Structures
- Behavioral Rules
- Expressions
- Validations
- Special Types (eg. temporal)
- ...

Metamodel for Business Logic

Semantics

Clearly defined data structure to express all business-relevant structures, behaviors and non-functional concerns.

Well-defined meaning of this data structure

Metamodel for Business Logic

Semantics

Clearly defined data structure to express all business-relevant structures, behaviors and non-functional concerns.

Well-defined meaning of this data structure

IDE Support is possible
Evolution is possible
Portability is possible

Metamodel for Business Logic

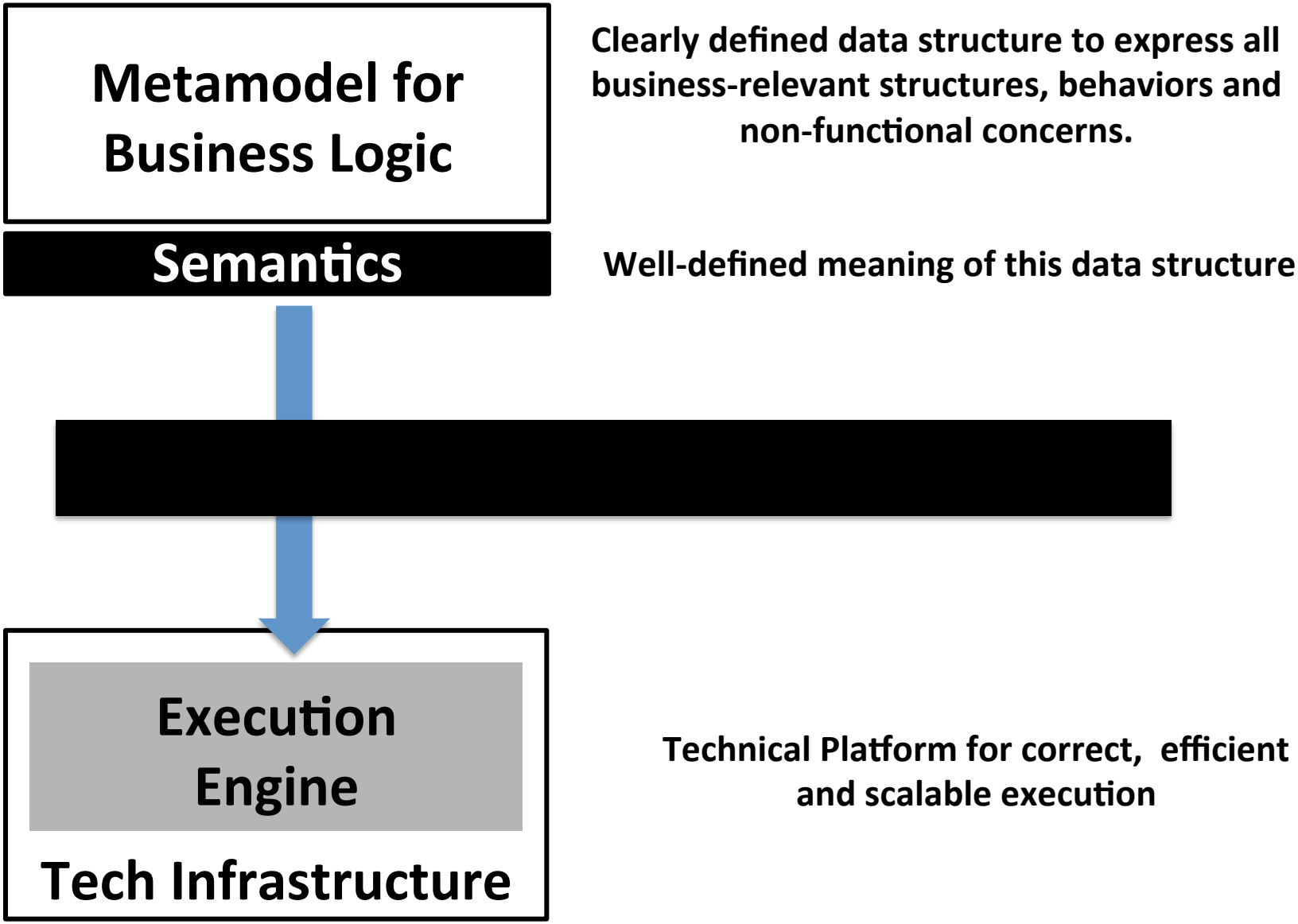
Semantics

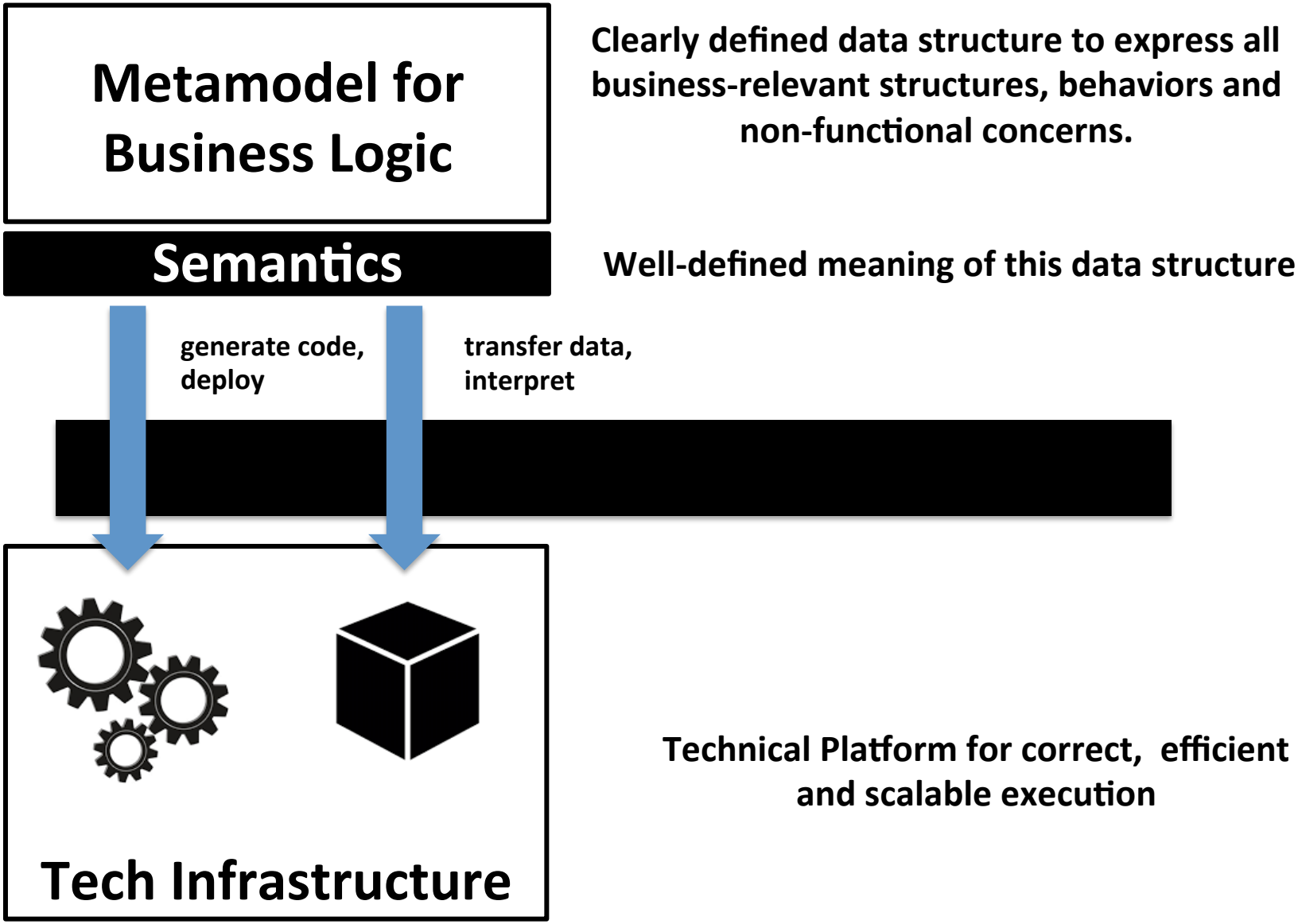
Clearly defined data structure to express all business-relevant structures, behaviors and non-functional concerns.

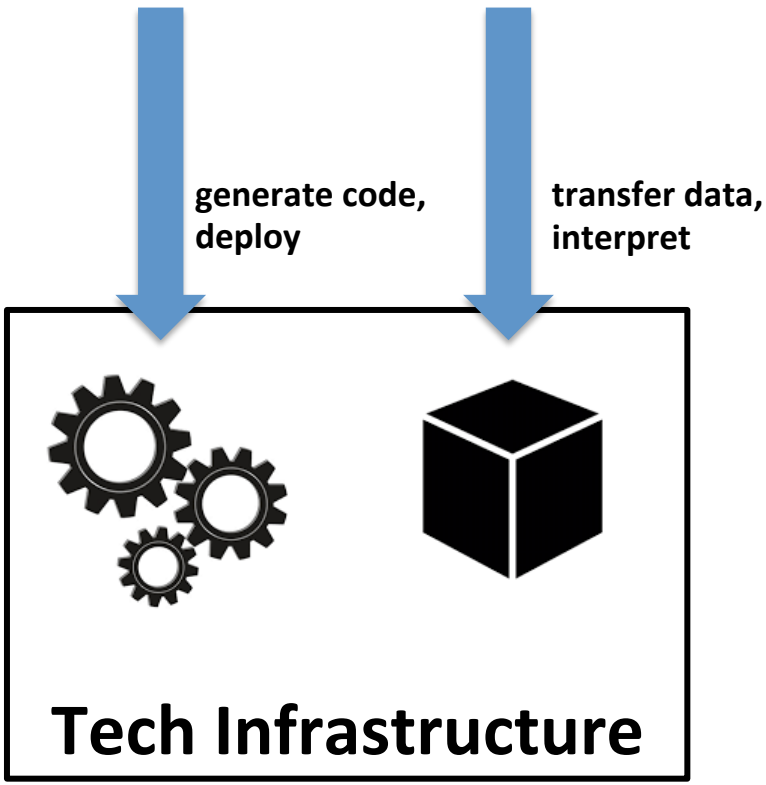
Well-defined meaning of this data structure

- IDE Support is possible
  Evolution is possible
  Portability is possible

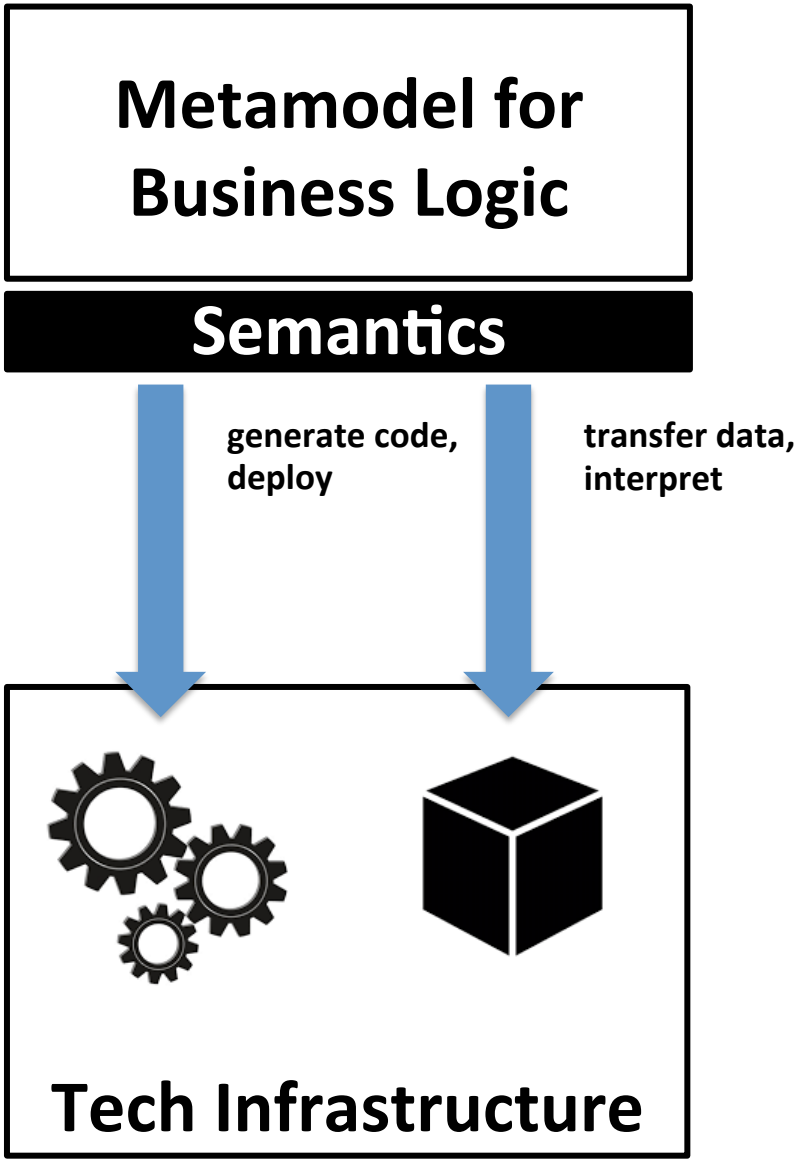
- Type Checking
  Solver-Integration
  Model Checking
  Contracts

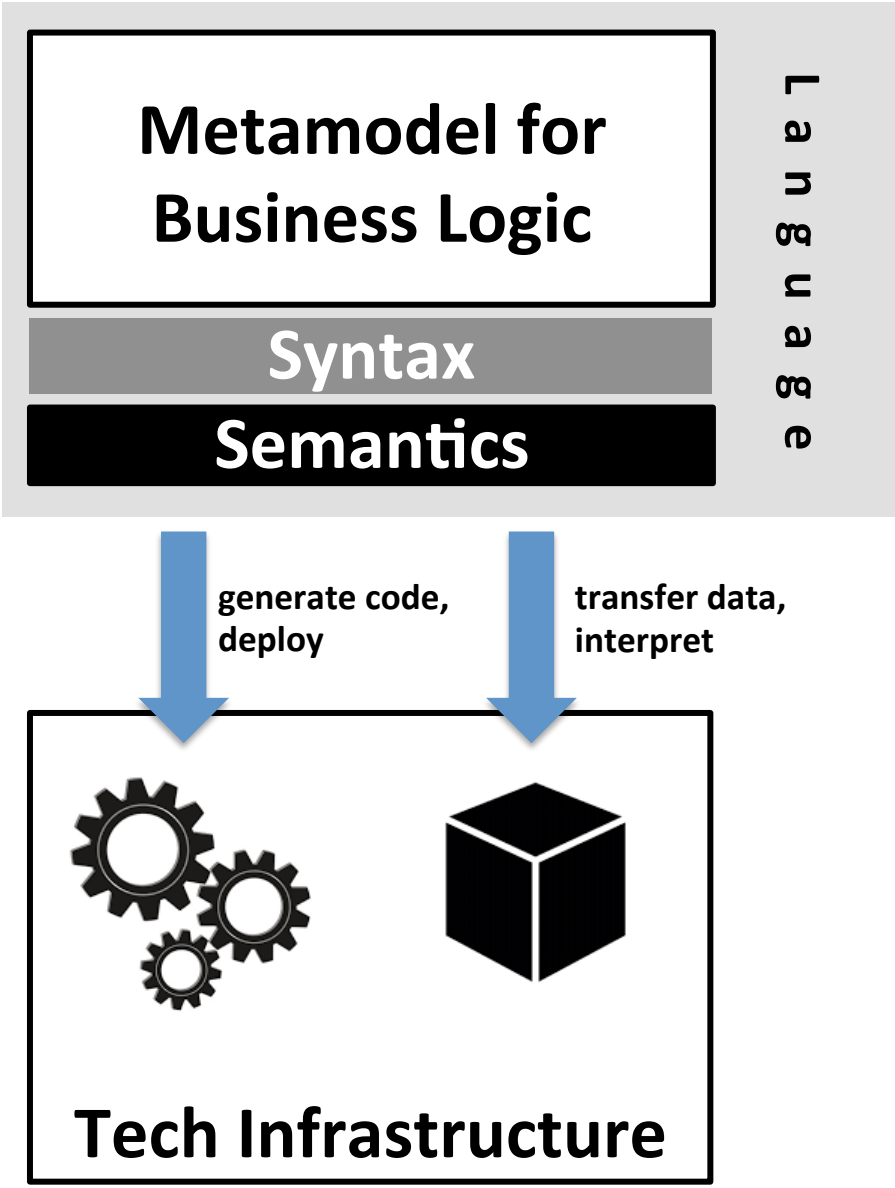

Metamodel for Business Logic
Clearly defined data structure to express all business-relevant structures, behaviors and non-functional concerns.
Semantics
Well-defined meaning of this data structure
Execution Engine
Tech Infrastructure
Technical Platform for correct, efficient and scalable execution

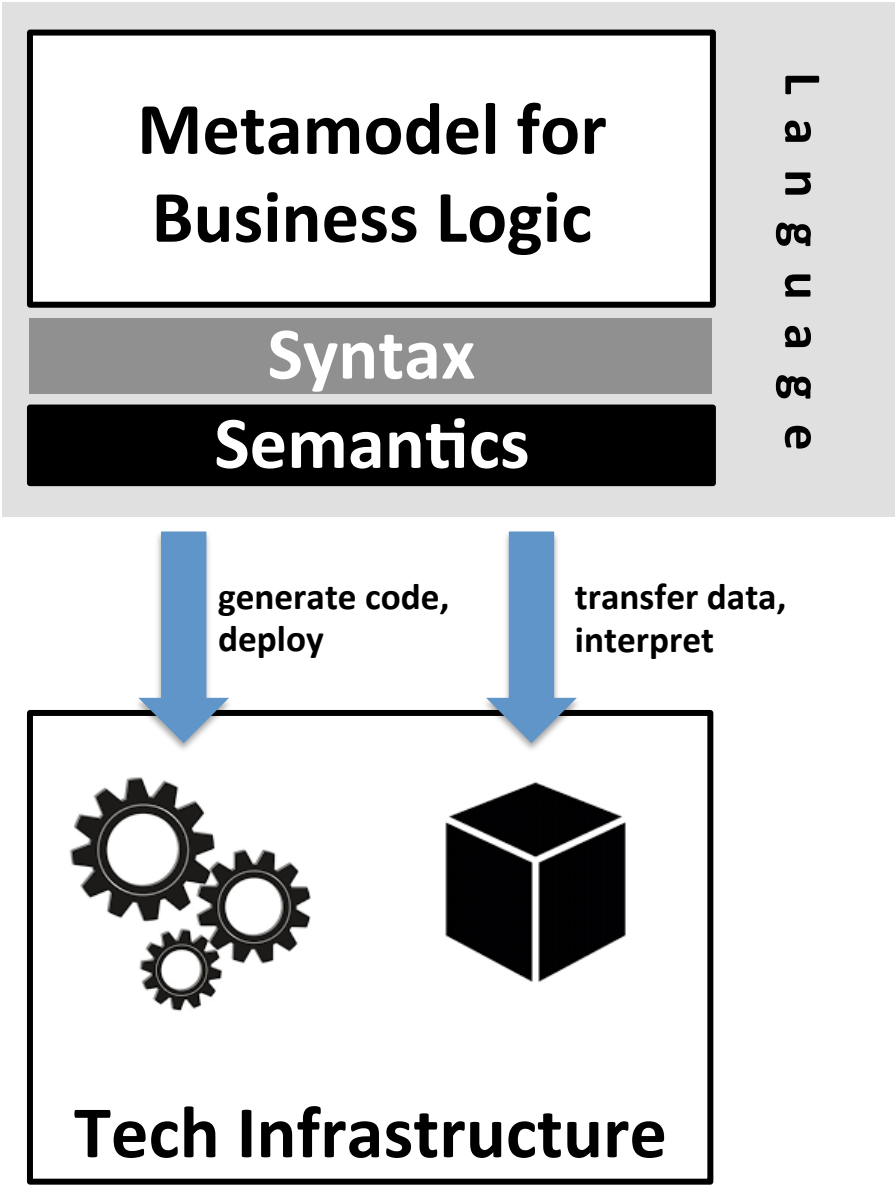

Metamodel for Business Logic
Clearly defined data structure to express all business-relevant structures, behaviors and non-functional concerns.
Semantics
Well-defined meaning of this data structure
generate code, deploy
transfer data, interpret
Tech Infrastructure
Technical Platform for correct, efficient and scalable execution



Generation	Interpretation
+ Code Inspection	+ Turn around Time
+ Debugging	+ Runtime Change
+ Performance & Optimization	
+ Platform Con-formance	


Metamodel for Business Logic
Semantics
generate code, deploy
transfer data, interpret
Tech Infrastructure

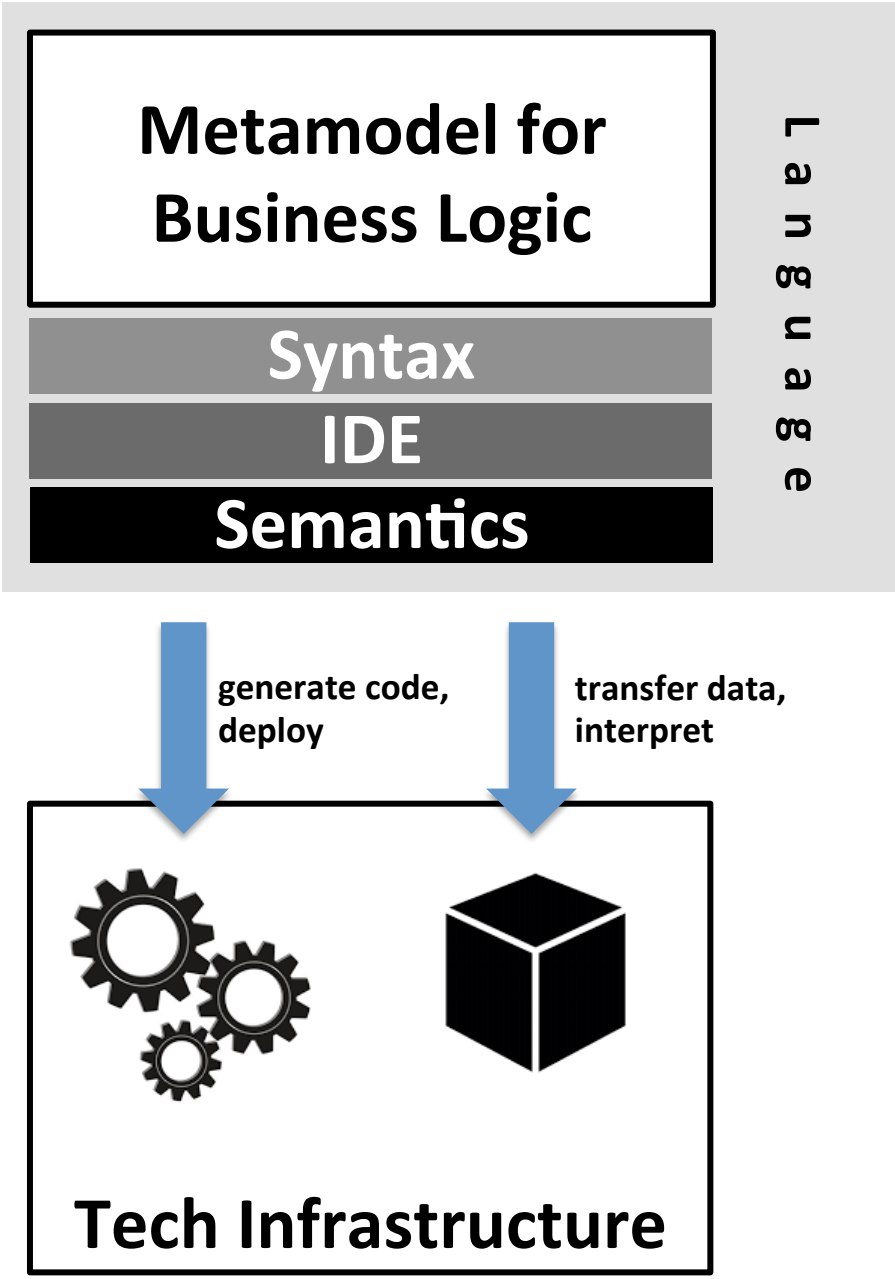

Metamodel for Business Logic
Syntax
Semantics
Language
generate code, deploy
transfer data, interpret
Tech Infrastructure



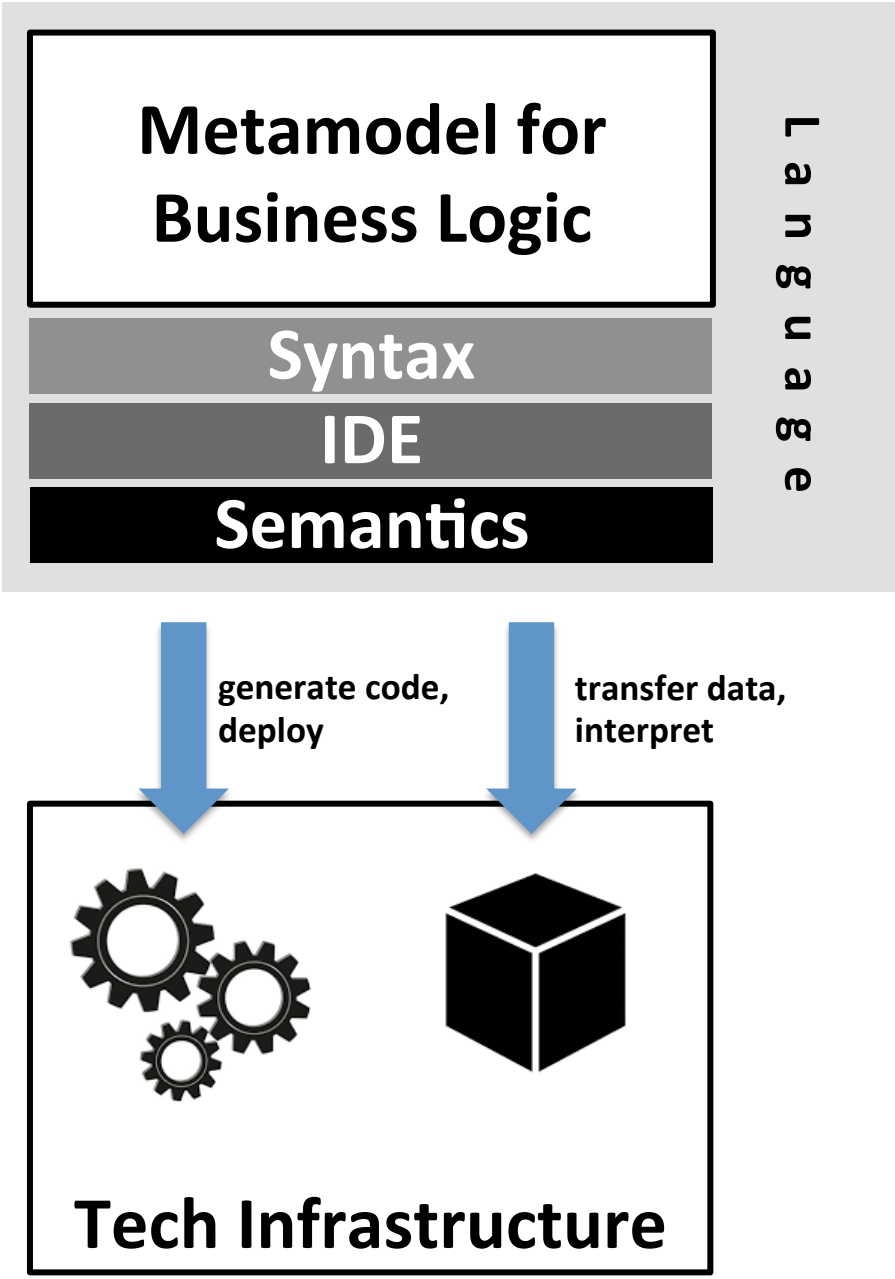
Syntax is **critically important** for

- Productivity
- Communication and Review
- Domain Expert Integration

- Only Buttons and Forms don't work!

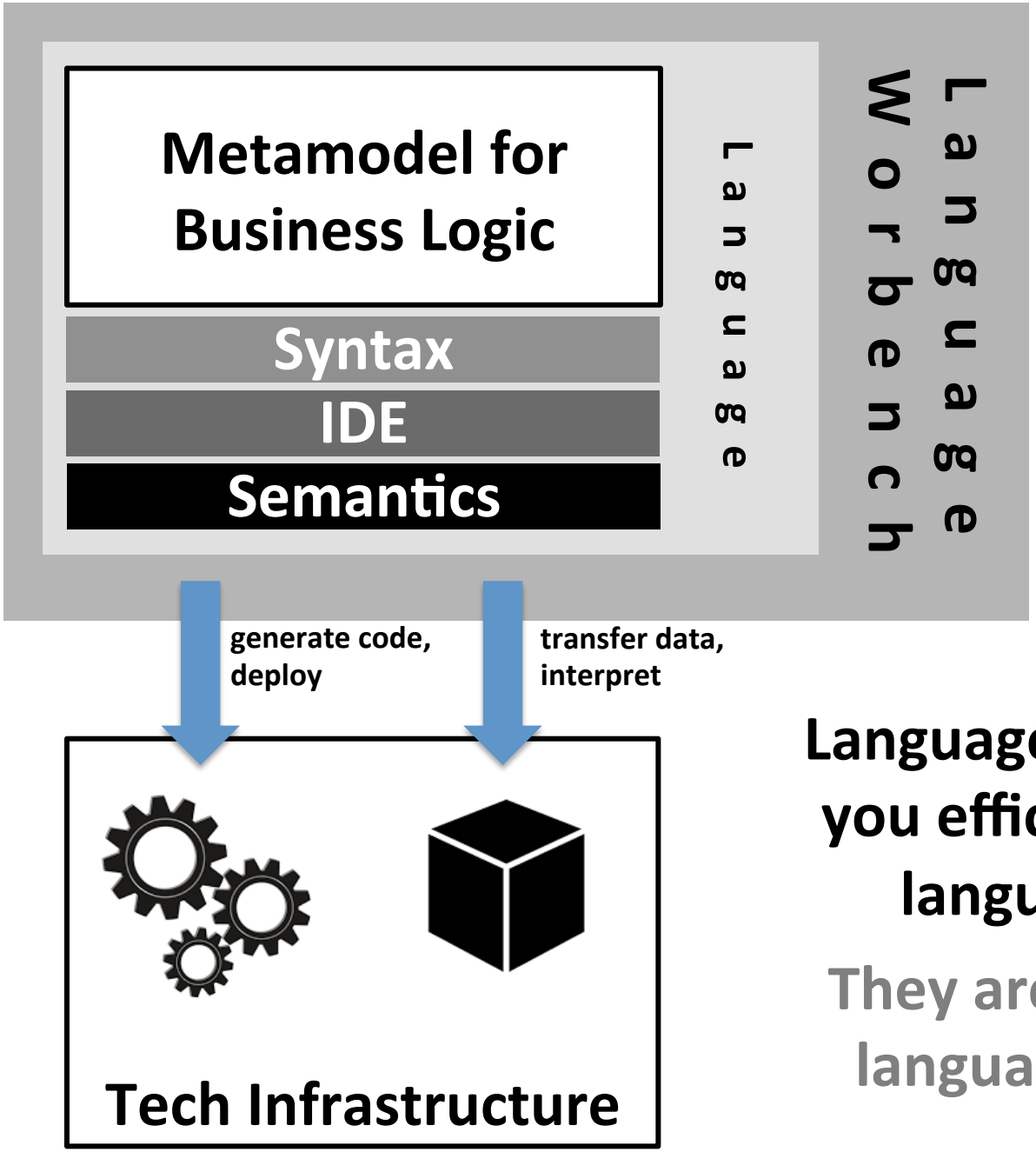


IDE is critically important for acceptance



IDE is **critically important** for acceptance

- Learning
- Exploration
- Productivity
- Tool Integration

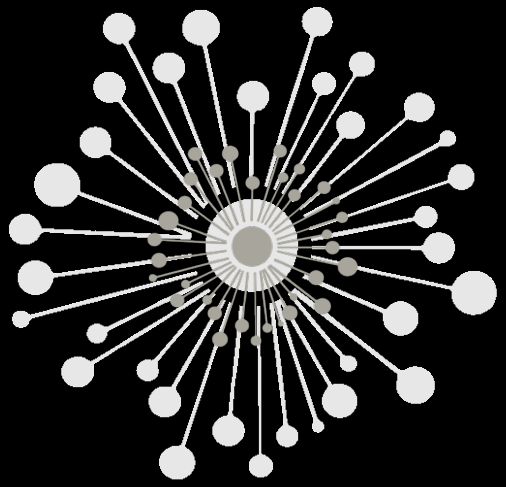


Language Workbenches let you efficiently implement languages and IDEs.

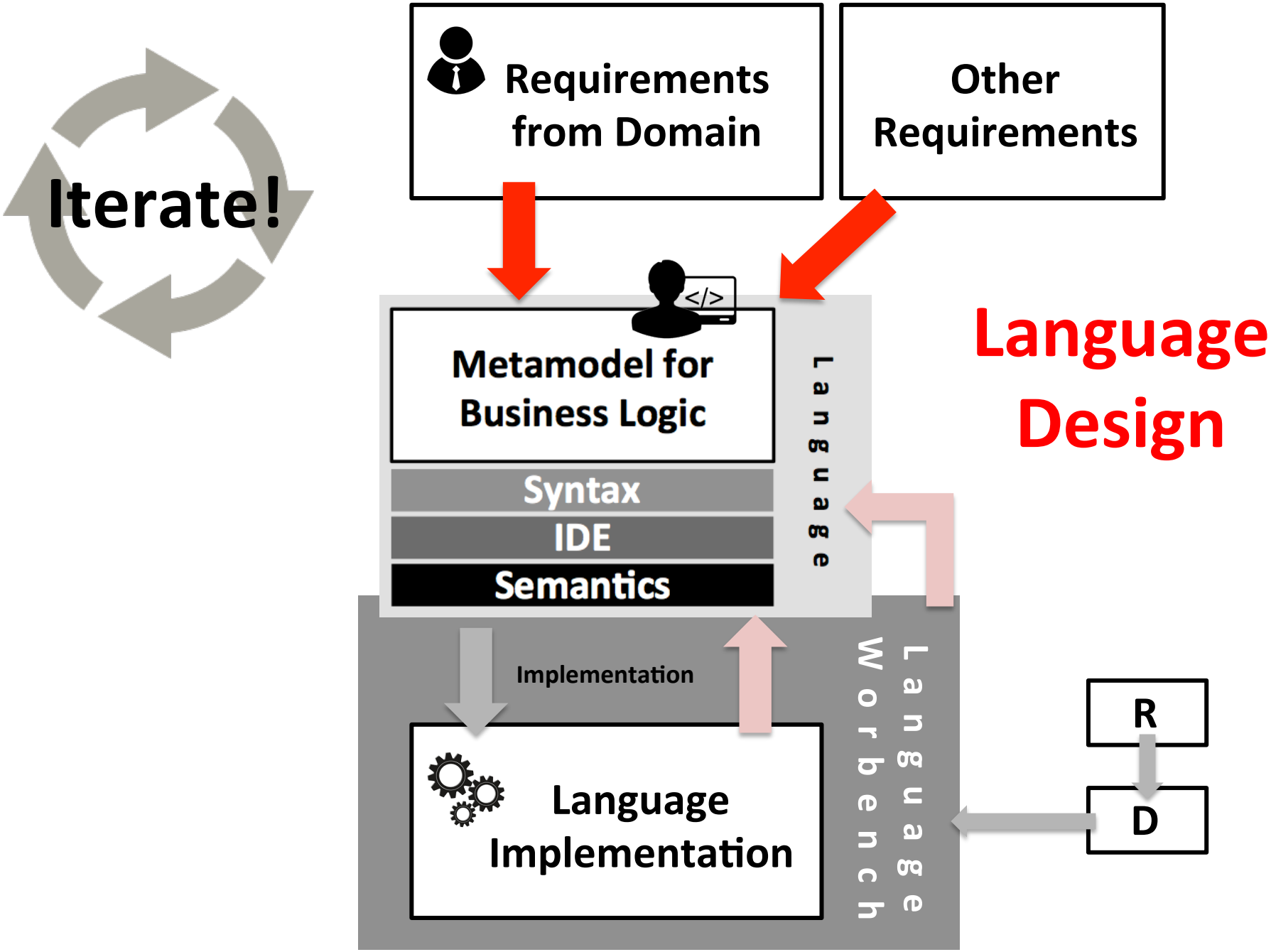
They are essential for the language implementor.

How
Domain
Requirements
Shape
Languages

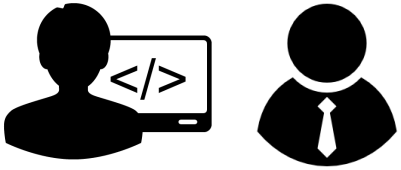
2



What is Language Design


Iterate!
Requirements from Domain
Other Requirements
Language Design
Metamodel for Business Logic
Syntax
IDE
Semantics
Language
Implementation
Language Implementation
Language Workbench
R
D

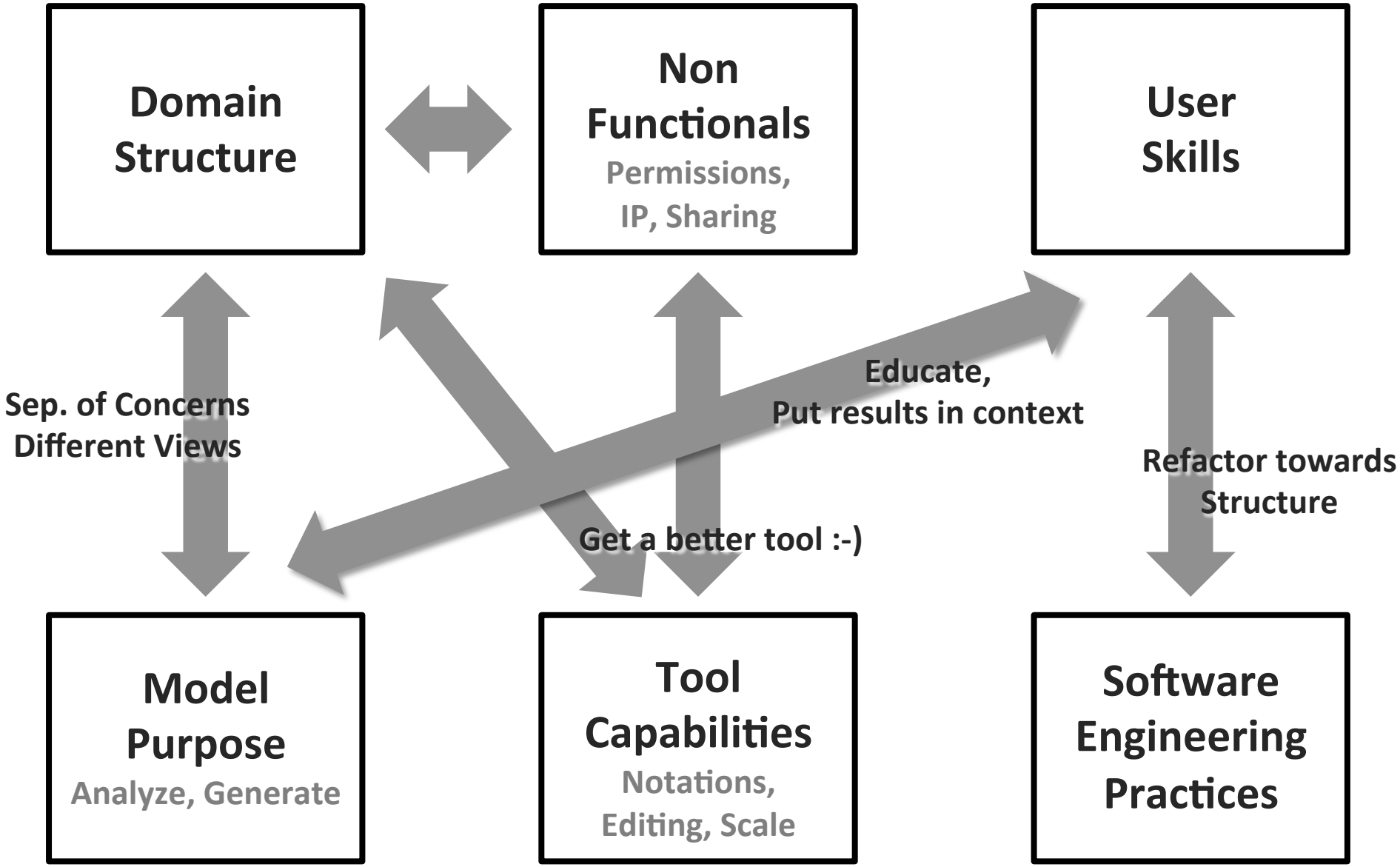
Language Design



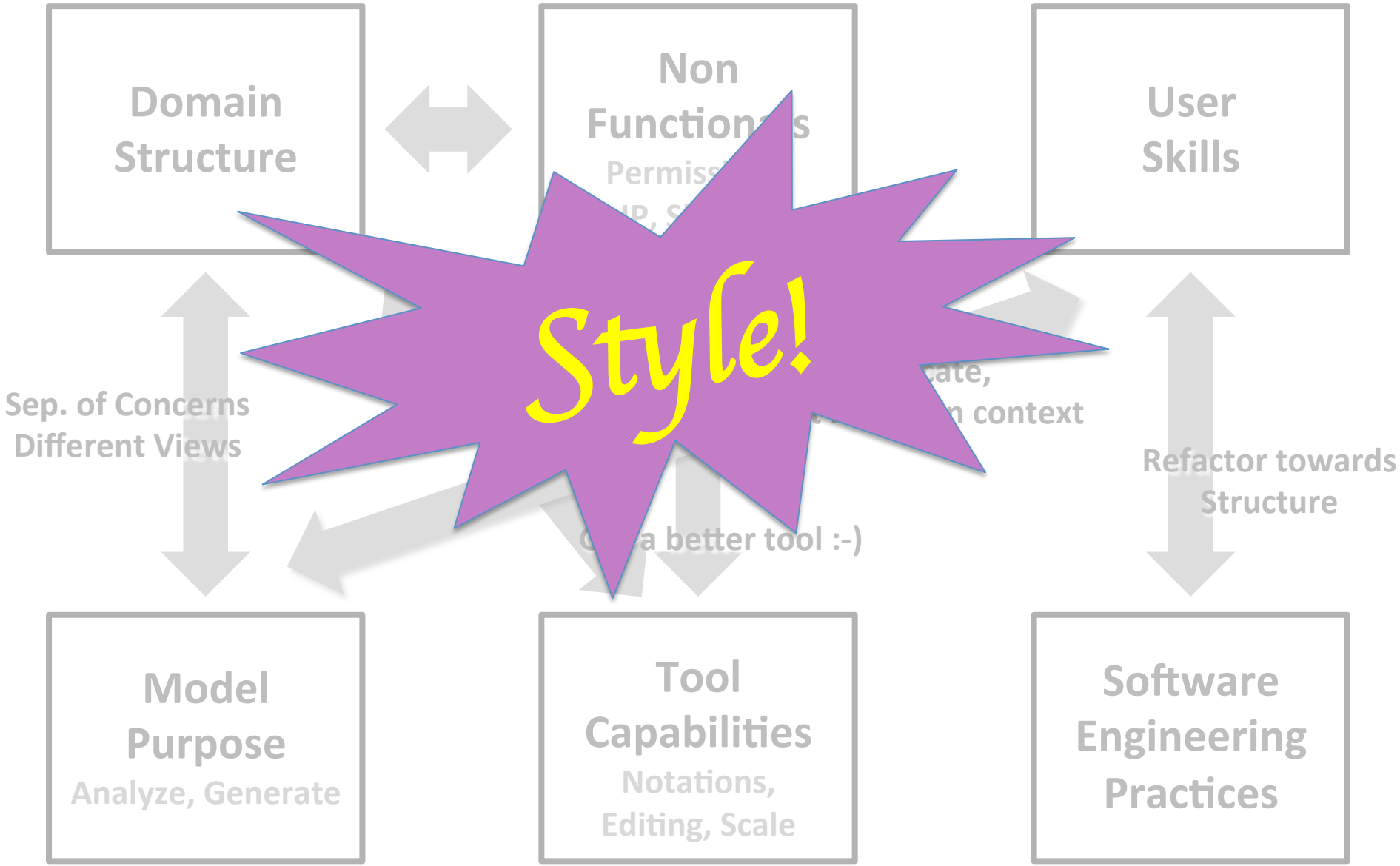
Forces
Constraints
Tradeoffs
Communication
Compromises
Validation

Done by people.
Supported by Tools.

Language Design::Influences



Language Design::Influences



Good Language Design ?

Fitness for purpose

Evaluation.

Balancing the Forces

Rationales.

Conceptual Frameworks

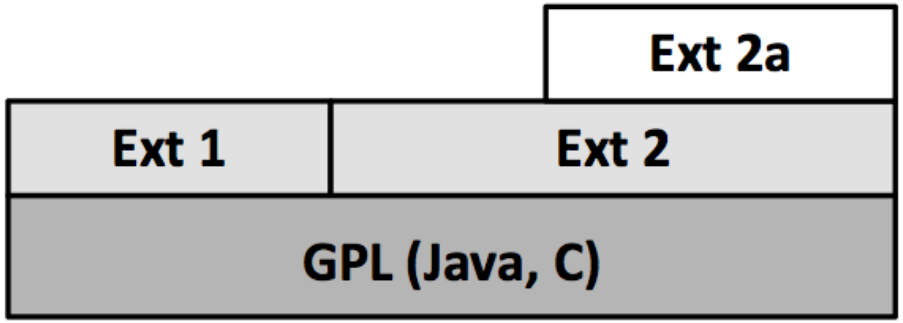
e.g. Cognitive Dimensions of Notations

Abstraction gradient • Closeness of mapping • Consistency • Diffuseness/Terseness • Error-proneness • Hard mental operations • Hidden dependencies • Juxtaposability • Premature commitment • Progressive evaluation • Role-expressiveness • Secondary notation and escape from formalism • Viscosity • Visibility

Language Design::Main Patterns

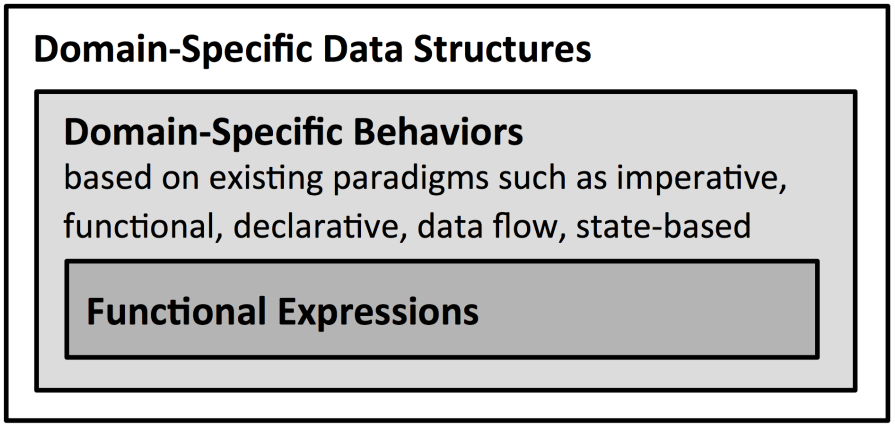
GPL Extension

Reuse GPL incl. Expressions and TS
Add/Embed DS-extensions
Compatible notational style
Reduce to GPL



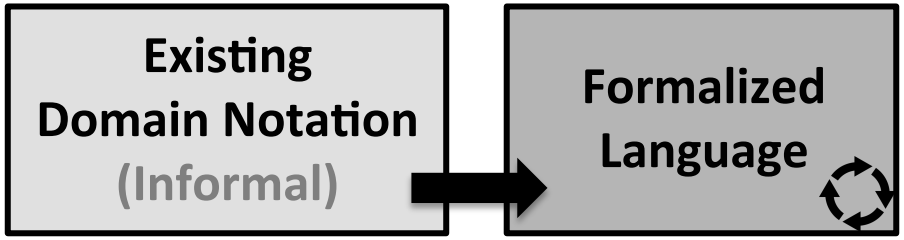
New Language

Analyze Domain to find Abstractions
Define suitable, new notations
Rely on existing behavioral paradigm
Reuse standard expression language
Interpret/Generate to one or more GPLs

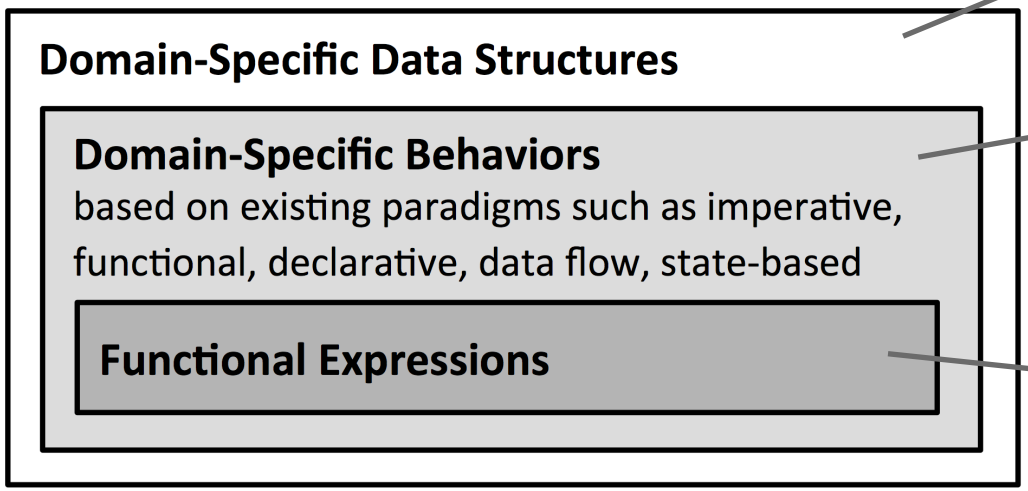


Formalization

Use existing notation from domain
Clean up and formalize
Generate/Interpret
Often import existing „models“



Language Design::Granularity



Contract Structure?
Inheritance?

State Machine?
Graphical or Textual?
Imperative?
Temporal Types?

Error Handling?
Syntax f. Option Types?

```
fun f(x: option<int>) = with some x => val none 10
```

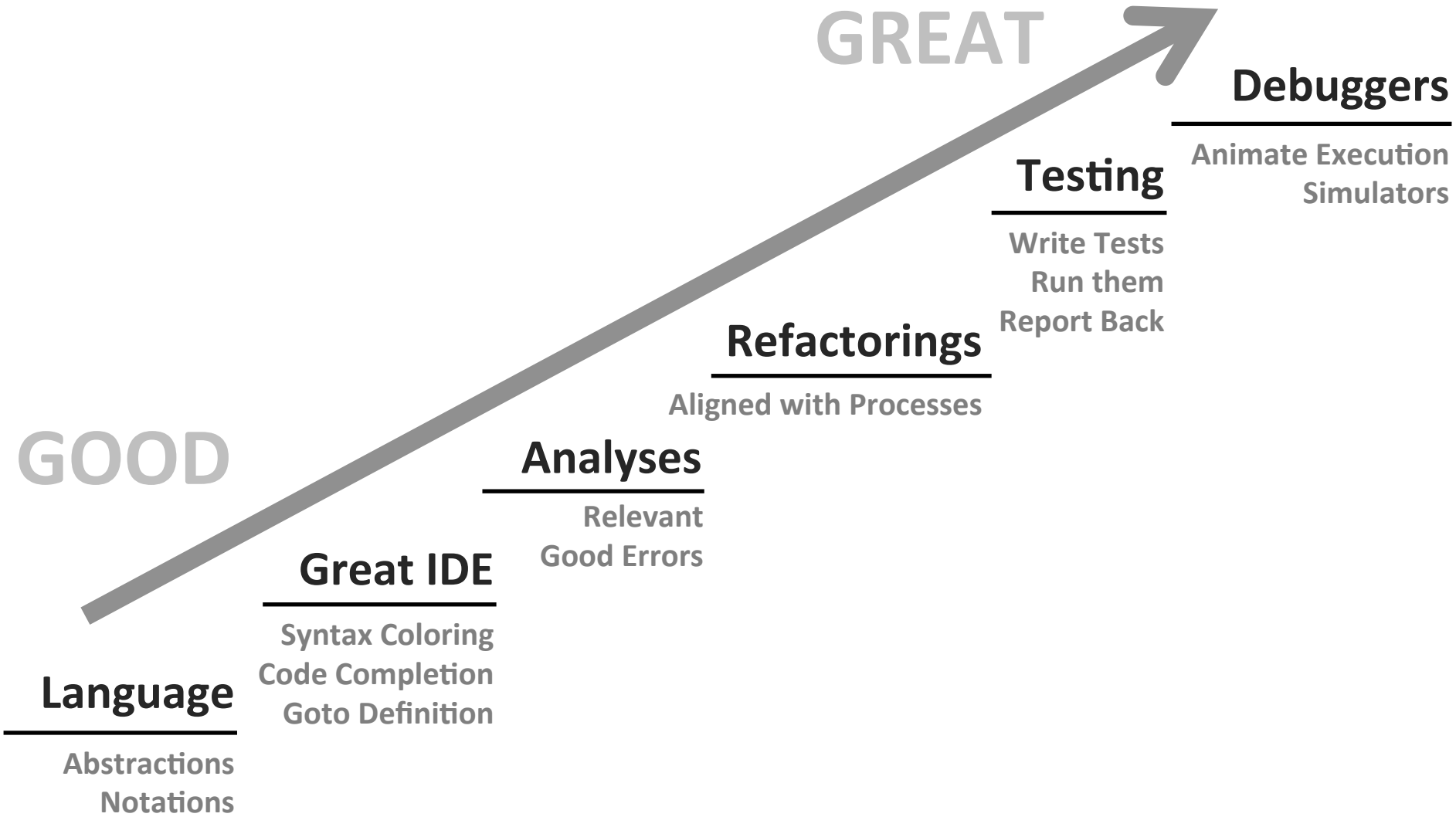
vs.

```
fun f(x: option<int>) = if some(x) then x else 10
```

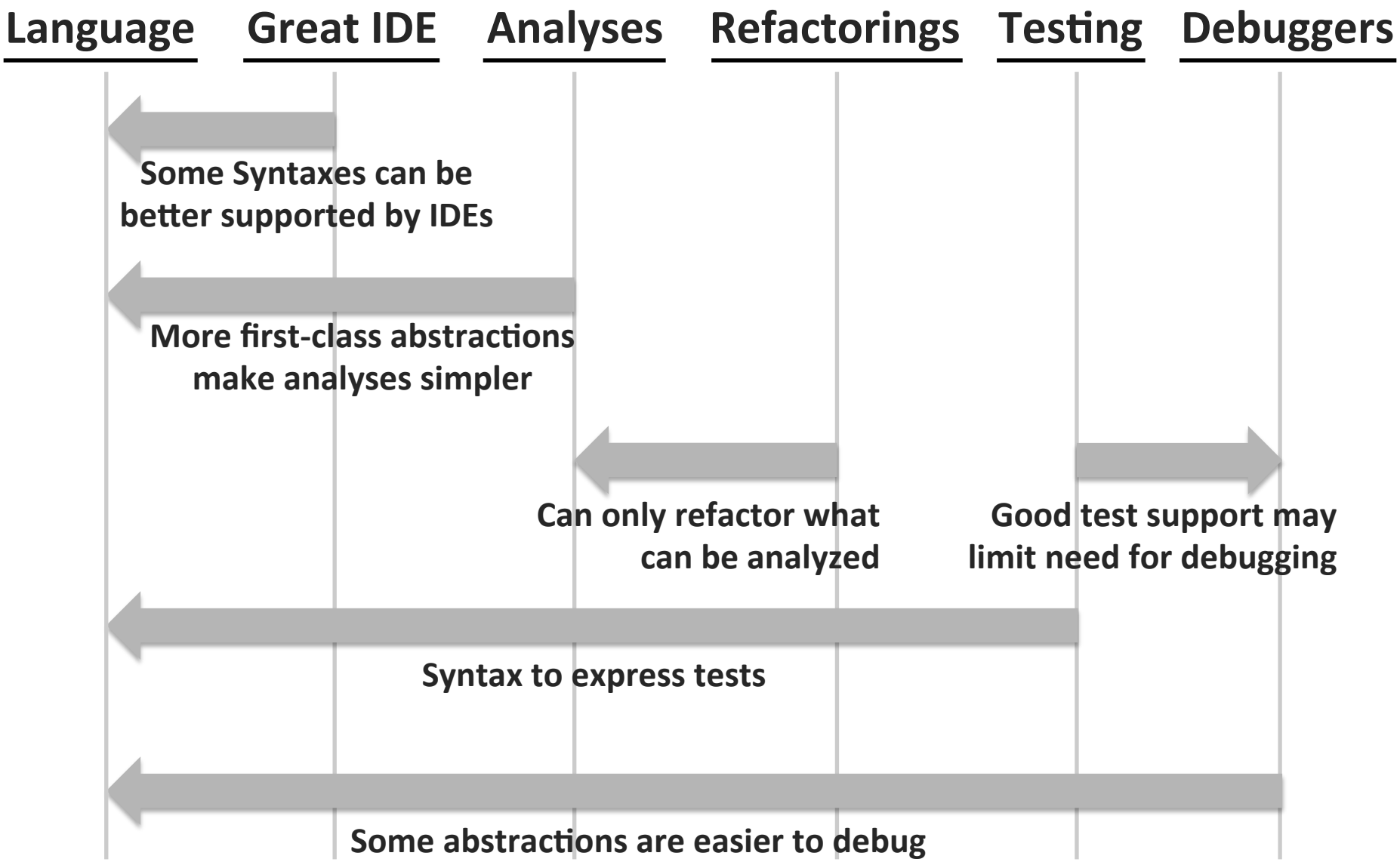
vs.

?

Language Design::More than Lang

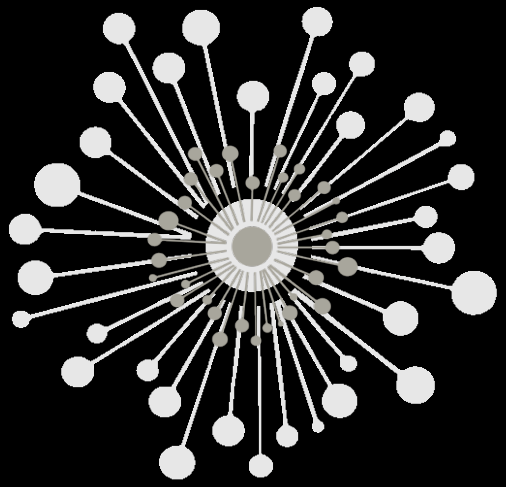


Language Design::Cross Influences



How
Domain
Requirements
Shape
Languages

3



Example Domains & Languages

Stakeholder integration, Scalable Business, Document Generation + Certification

Code Complexity, Frameworks (Autosar), Product Lines

Reduction of Accidental Complexity in Code, Process Conformance (Docs)

A powerful language and IDE for existing frameworks (Industry Robots, ROS)

Precise Specification and Implementation of Insurance Products („Rules“)

Multi-Paradigm Programming, not just Simulink and C

Consistent Derived Documents

Changing Regulations, Fast Implementation, End User Empowerment

	Mixed Notations and End User Programming	Informal -> Semiformal -> Formal	Languages + Verification	Optmizations, Performance and Concurrency	Fundamentals: Editors, Type Systems, Trafos	User-Friendly IDEs, Tools	Simulation, Execution, Test	Language vs. Library, Code Gen	Consistent Derived Documents
	✖	✖	✖		✖	✖	✖		✖
			✖	✖	✖	✖	✖	✖	
			✖		✖	✖		✖	✖
					✖	✖		✖	
	✖	✖			✖	✖	✖		✖
			✖	✖	✖	✖	✖		
			✖		✖	✖	✖		
	✖	✖			✖	✖	✖		

Observations:

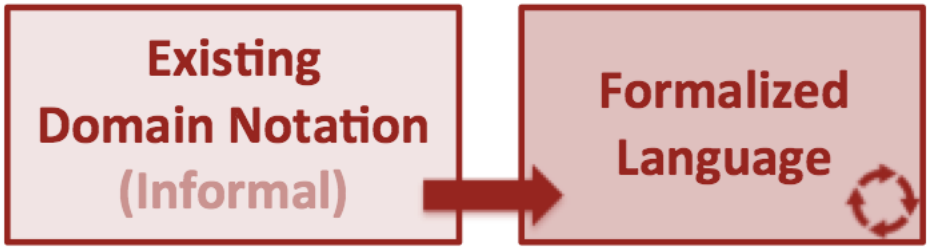
1) Every domain has **different reasons** why language engineering is used.
2) No domain has **only one reason** – it is always a combination
3) Languages, Editors, Type Systems and good IDEs are **always** important!

F I N A N C E

Customer had „coded“ in Word

Needed a real IDE

Syntax to be retained if possible


Existing
Domain Notation
(Informal)
Formalized
Language

Specify/Program

Insurance Programs

Write formal code in a DSL
mixed with tables and text

No tool support whatsoever
No testing (except inspection)

No reuse
No modularity
No varibility

Specify/Program

Write formal code in a DSL
mixed with tables and text

No tool support whatsoever
No testing (except inspection)

No reuse
No modularity
No varibility

Formale Beschreibung

Funktion:	berbwvekFF		
Programmquelle:	vmsctfa1.c		
Produkt-Typ:	FONDS	**PK-Typ:**	KAPITAL-KONTO
	Name	Verw.	Entität
verwendete Attribute:	lkm_akt_param	E	PARAMETER
	lkm_faell_param	E	PARAMETER
	ber_zweck_param	E	PARAMETER
	kz_rzw_param	E	PARAMETER
	bwvek	A	RETURN
aufgerufene Funktionen:	berbweinzelFF		
	VTRKermbtgfaellFF		
Status:	18.1		

Verarbeitungen

Die Funktion liefert den Barwert per lkm_akt_param des vorschüssigen Zahlungsstroms der Höhe 1 von Monat lkm_akt_param bis lkm_faell_param – jeweils einschließlich. Zahlungszeitpunkte sind jeweils die Monatsbeginne, also **lkm_akt_param** -1 bis **lkm_faell_param** – 1.

Der Parameter **kz_rzw_param** steuert die zu berücksichtigende Zahlweise des Zahlungsstroms. Möglich sind zur Zeit nur die Ausprägungen 0 (Zahlungen zu den Beitragsfälligkeiten) und 12 (monatliche Zahlungsweise).

Schleife über lkm_faell_hilf = **lkm_akt_param** bis **lkm_faell_param**

Falls **kz_rzw_param** = 12

kz_bf_hilf = 1

sonst

kz_bf_hilf = *VTRKermbtgfaellFF*(lkm_faell_hilf)

Ende Falls **kz_rzw_param** = 12

bwvek = **bwvek**

+ kz_bf_hilf * *berbweinzelFF*(**lkm_akt_param**, lkm_faell_hilf - 1, **ber_zweck_param**)

Ende Schleife

return **bwvek**

Specify/Program

Insurance Programs

Write formal code in a DSL mixed with tables and text

No tool support whatsoever
No testing (except inspection)

No reuse
No modularity
No varibility

Formale Beschreibung

Funktion: rg_kk_beta_satzTF
Programmquelle: vmscfo2.c
Produkt-Typ: FONDS, RSR **PK-Typ:** Kapital-Konto

verwendete Attribute:

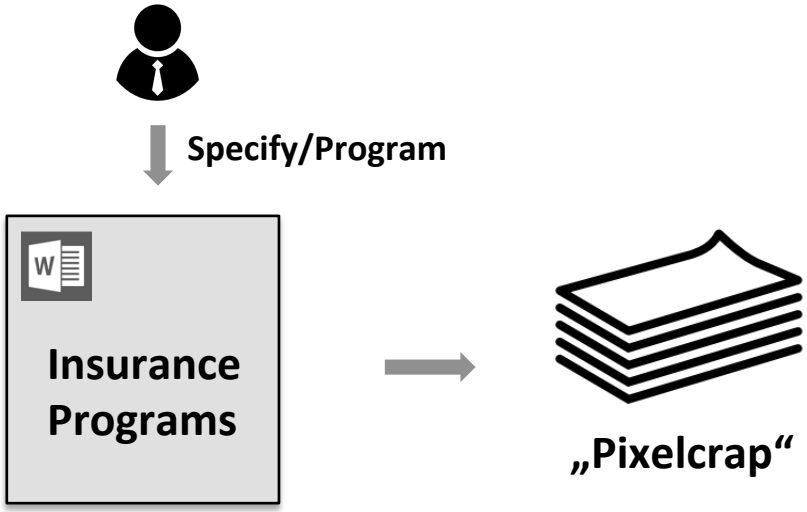
Name	Verw.	Entität
fo_beta_satz	E	Kosten-Regeln
beta_satz	E	Rechnungsgrundlagen-KK
ko_ra_id	E	KOSTEN-RABATT
zmt_param	E	PARAMETER
kz_zus_gar	E	
zm	E	
beta_satz_fakt	E	VORGABEDATEN-KOSTENRABATT
zw	E	TVDKONTO_G
vtrk_zb	E	VTRK_BTG
kz_mandant	E	T_KK
satz_beta	A	Rückgabewert

Aufgerufene Funktionen: rg_kk_beta_bp_satzTF

In dieser Funktion wird der Kostensatz β ermittelt.

Verarbeitung

fo_beta_satz	Berechnung **satz_beta**	Bemerkung
0	**satz_beta = beta_satz** Falls **zmt_param** <= 120 und **kz_zus_gar** = JA **satz_beta = satz_beta** * min(0,01 * max(**zmt_param** - 12; 0); 1) Ende (Falls **zmt_param** <= 120 und **kz_zus_gar** = JA)	Standard
1	**satz_beta = beta_satz** * min(0,01 * max(**zm** – 12; 0); 1)	PF
2	grenze = **vtrk_zb** * **zw** Falls grenze < 10000.0 satz_beta = **beta_satz** Sonst satz_beta = 0,074	GULPP



Write formal code in a DSL
mixed with tables and text

No tool support whatsoever
No testing (except inspection)

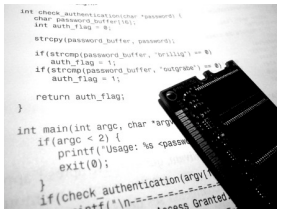
No reuse
No modularity
No varibility

Printed, PDF

Specify/Program

„Pixelcrap“

Implement



C Code

Write formal code in a DSL
mixed with tables and text

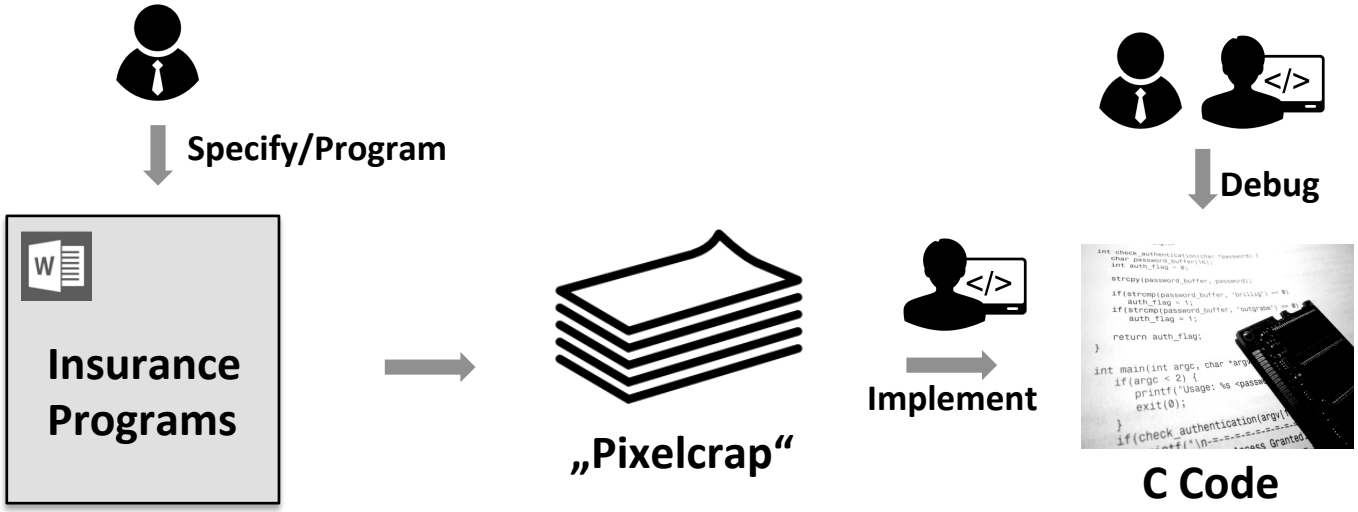
No tool support whatsoever
No testing (except inspection)

No reuse
No modularity
No varibility

Printed, PDF

Developer reads „spec“
Very idiomatic implementation

Dev acts as a human compiler and implements it in C



Debugging directly in C
Search-for-use by text search
Don‘t trust the documents – may be outdated!

Write formal code in a DSL mixed with tables and text

No tool support whatsoever
No testing (except inspection)

No reuse
No modularity
No varibility

Printed, PDF

Developer reads „spec“
Very idiomatic implementation

Dev acts as a human compiler and implements it in C

Specify/Program/Test/Debug

Write formal code in a DSL
mixed with tables and text

Now with IDE support and executable tests

The same notation!

Specify/Prog

Insurance Programs

Write formal code in a D
mixed with tables and te

Now with IDE support ar

The same nota

berbwvekFF (lkm_akt_param; lkm_faell_param; ber_zweck_param; kz_rzw_param)

Funktionenmodell berbwvekFF

Formale Beschreibung

Funktion: berbwvekFF
Programmquelle: vmsctfa1.c
Produkt-Typ: Fonds **PK-Typ:** Kapital-Konto
Status: 18.1

Parameter-Attribute

lkm_akt_param
lkm_faell_param
ber_zweck_param
kz_rzw_param

Verwendete VADM-Attribute

Keine verwendeten VADM-Attribute, werden automatisch hinzugefügt

Rückgabe-Attribut

bwvek

aufgerufene Funktionen

VTRKermbtgfaellFF (a)
berbweinzelFF (a; b; c)

Beschreibung

Die Funktion liefert den Barwert per @lkm_akt_param des vorschüssigen Zahlungsstroms der Höhe 1 von Monat @lkm_akt_param bis @lkm_faell_param – jeweils einschließlich. Zahlungszeitpunkte sind jeweils die Monatsbeginne, also #lkm_akt_param – 1# bis #lkm_faell_param – 1#. Der Parameter @kz_rzw_param steuert die zu berücksichtigende Zahlweise des Zahlungsstroms. Möglich sind zur Zeit nur die Ausprägungen 0 (Zahlungen zu den Beitragsfälligkeiten) und 12 (monatliche Zahlungsweise).

Hilfsvariablen

kz_bf_hilf

Verarbeitungen

```
Schleife über lkm_faell_hilf = lkm_akt_param bis lkm_faell_param
  Falls kz_rzw_param = 12
      kz_bf_hilf = 1
    sonst
      kz_bf_hilf = VTRKermbtgfaellFF (lkm_faell_hilf)
  Ende Falls kz_rzw_param = 12
  bwvek = bwvek + kz_bf_hilf * berbweinzelFF (lkm_akt_param; lkm_faell_hilf - 1; ber_zweck_param)
Ende Schleife über lkm_akt_param bis lkm_faell_param

return bwvek
```

berbwvekFF (lkm_akt_param; lkm_faell_param; ber_zweck_param; kz_rzw_param)

Funktionenmodell berbwvekFF

Formale Beschreibung

Funktion: berbwvekFF
Programmquelle: vmsctfa1.c
Produkt-Typ: Fonds **PK-Typ:** Kapital-Konto
Status: 18.1

Parameter-Attribute

lkm_akt_param
lkm_faell_param
ber_zweck_param
kz_rzw_param

Verwendete VADM-Attribute

Keine verwendeten VADM-Attribute, werden automatisch hinzugefügt

Rückgabe-Attribut

bwvek

aufgerufene Funktionen

VTRKermbtgfaellFF (a)
berbweinzelFF (a; b; c)

Beschreibung

Die Funktion liefert den Barwert per @lkm_akt_param des vorschüssigen Zahlungss
@lkm_akt_param bis @lkm_faell_param – jeweils einschließlich. Zahlungszeitpunkt
#lkm_akt_param – 1# bis #lkm_faell_param – 1#. Der Parameter @kz_rzw_param steu
Zahlweise des Zahlungsstroms. Möglich sind zur Zeit nur die Ausprägungen 0 (Zahl
und 12 (monatliche Zahlungsweise).

Hilfsvariablen

kz_bf_hilf

Verarbeitungen

```
Schleife über lkm_faell_hilf = lkm_akt_param bis lkm_faell_param
  Falls kz_rzw_param = 12
      kz_bf_hilf = 1
    sonst
      kz_bf_hilf = VTRKermbtgfaellFF (lkm_faell_hilf)
  Ende Falls kz_rzw_param = 12
  bwvek = bwvek + kz_bf_hilf * berbweinzelFF (lkm_akt_param; lkm_faell_hilf - 1
Ende Schleife über lkm_akt_param bis lkm_faell_param

return bwvek
```

Formale Beschreibung

Funktion:	berbwvekFF		
Programmquelle:	vmsctfa1.c		
Produkt-Typ:	FONDS	**PK-Typ:**	KAPITAL-KONTO
	Name	Verw.	Entität
verwendete Attribute:	lkm_akt_param	E	PARAMETER
	lkm_faell_param	E	PARAMETER
	ber_zweck_param	E	PARAMETER
	kz_rzw_param	E	PARAMETER
	bwvek	A	RETURN
aufgerufene Funktionen:	berbweinzelFF		
	VTRKermbtgfaellFF		
Status:	18.1		

Verarbeitungen

Die Funktion liefert den Barwert per lkm_akt_param des vorschüssigen Zahlungsstroms der Höhe 1 von Monat lkm_akt_param bis lkm_faell_param – jeweils einschließlich. Zahlungszeitpunkte sind jeweils die Monatsbeginne, also **lkm_akt_param** -1 bis **lkm_faell_param** – 1.

Der Parameter **kz_rzw_param** steuert die zu berücksichtigende Zahlweise des Zahlungsstroms. Möglich sind zur Zeit nur die Ausprägungen 0 (Zahlungen zu den Beitragsfälligkeiten) und 12 (monatliche Zahlungsweise).

Schleife über lkm_faell_hilf = **lkm_akt_param** bis **lkm_faell_param**

Falls **kz_rzw_param** = 12

kz_bf_hilf = 1

sonst

kz_bf_hilf = *VTRKermbtgfaellFF*(lkm_faell_hilf)

Ende Falls **kz_rzw_param** = 12

bwvek = **bwvek**

+ kz_bf_hilf * *berbweinzelFF*(**lkm_akt_param**, lkm_faell_hilf - 1, **ber_zweck_param**)

Ende Schleife

return **bwvek**

Specify/Program/Test

Insurance Programs

Write formal code in a DSL mixed with tables and text

Now with IDE support and execu

The same notation!

rg_kk_beta_satzTF (zmt_param)

Formale Beschreibung

Funktion: rg_kk_beta_satzTF
Programmquelle: vmscfo2.c
Produkt-Typ: Fonds, RSR **PK-Typ:** Kapital-Konto
Status: 18.1

Parameter-Attribute
zmt_param

Verwendete VADM-Attribute
rg_kk.fo_beta_satz E
rg_kk.beta_satz E
rg_kk.kz_zus_gar E
rg_kk.zm E
rg_kk.vtrk_zb E
rg_kk.zw E
rg_kk.ko_ra_id E
rg_kk.kz_mandant E
rg_kk.beta_satz_fakt E

Rückgabe-Attribut
satz_beta

aufgerufene Funktionen
Kommazahl MIN (Kommazahl a; Kommazahl b)
Kommazahl MAX (Kommazahl a; Kommazahl b)
rg_kk_beta_bp_satzTF ()
rg_kk_beta_ap_satzTF ()

Beschreibung
In dieser Funktion wird der Kostensatz β ermittelt.

Hilfsvariablen
grenze
fak_beta
beta_bp_satz_hilf
beta_ap_satz_hilf

Verarbeitungen

rg_kk.fo_beta_satz	Berechnung @satz_beta	Bemerkung
0	*satz_beta* = **rg_kk.beta_satz** Falls zmt_param <= 120 **und rg_kk.kz_zus_gar = JA** *satz_beta* = *satz_beta* * MIN (0,01 * MAX (zmt_param - 12; 0); 1) sonst Verarbeitung hinzufügen Ende Falls zmt_param <= 120 und rg_kk.kz_zus_gar = JA	Standard
1	*satz_beta* = **rg_kk.beta_satz** * MIN (0,01 * MAX (**rg_kk.zm** - 12; 0); 1)	PF
2	grenze = **rg_kk.vtrk_zb** * **rg_kk.zw** Falls grenze < 10000,0 *satz_beta* = **rg_kk.beta_satz** sonst *satz_beta* = 0,074 Ende Falls grenze < 10000,0	GULPP
3	*satz_beta* = **rg_kk.beta_satz** Falls zmt_param <= 156 **und rg_kk.kz_zus_gar = JA** fak_beta = 0,05 * MIN (zmt_param / 12 - 1; 1) +	FV Standard Z-D ab TV 8

Specify/Pro

Insurance Programs

Write formal code in a
mixed with tables and

Now with IDE support a

The same not

GeometrischesMittel () ×

Funktionenmodell GeometrischesMittel

Formale Beschreibung

```
Funktion:        GeometrischesMittel
Programmquelle:  Programmquelle auswählen
Produkt-Typ:     Produkt-Typen auswählen
PK-Typ:          PK-Typ auswählen
Status:          Status auswählen
```

Parameter-Attribute

a
b

Verwendete VADM-Attribute

Keine verwendeten VADM-Attribute, werden automatisch hinzugefügt

Rückgabe-Attribut

result

aufgerufene Funktionen

Keine aufgerufenen Funktionen, werden automatisch hinzugefügt

Beschreibung

Berechnet das geometrische Mittel der Parameter

Hilfsvariablen

Hilfsvariable hinzufügen

Error: Quadratwurzel ist nur für positive Zahlen erlaubt

```
result = √-1
result = √a * b
return re
```

```
r result                                  ^returnParameter (t.c.z.v.m.f.f.demo.sprint2.GeometrischesMittel)
I HOCHR_VK.REN_A                          ^literals (t.c.z.v.m.f.f.demo.sprint2.HOCHR_VK)
I HOCHR_VK.REN_S                          ^literals (t.c.z.v.m.f.f.demo.sprint2.HOCHR_VK)
I HOCHR_VK.REN_U                          ^literals (t.c.z.v.m.f.f.demo.sprint2.HOCHR_VK)
I PK_TYP_ID.GRUPPIERUNG_RECHNUNGSLEGUNG   ^literals (t.c.z.v.m.f.f.demo.sprint2.PK_TYP_ID)
I PK_TYP_ID.REGULIERUNG                   ^literals (t.c.z.v.m.f.f.demo.sprint2.PK_TYP_ID)
a tarif.reg.fo_aktiv                      ^members (t.c.z.v.m.f.f.demo.sprint2.tarif)
a tarif.reg.fo_bil_dir                    ^members (t.c.z.v.m.f.f.demo.sprint2.tarif)
a tarif.reg.fo_bs                         ^members (t.c.z.v.m.f.f.demo.sprint2.tarif)
a tarif.rund.rd_la2_rel                   ^members (t.c.z.v.m.f.f.demo.sprint2.tarif)
```

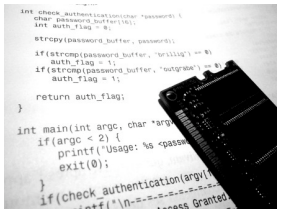
Context Actions

Attribute
- a
- b
- result

Aufzählungen
- HOCHR_VK.BUZB
- HOCHR_VK.BUZR
- HOCHR_VK.FLV
- HOCHR_VK.GFV
- HOCHR_VK.HZV_A
- HOCHR_VK.HZV_S
- HOCHR_VK.KAP
- HOCHR_VK.REN_A
- HOCHR_VK.REN_S
- HOCHR_VK.REN_U
- HOCHR_VK.RIS
- HOCHR_VK.RIZ
- HOCHR_VK.RSR
- HOCHR_VK.SBU
- HOCHR_VK.SEU
- PK_TYP_ID.AUSGANGS_KOMPONENTE
- PK_TYP_ID.BANK_KOMPONENTE
- PK_TYP_ID.DYNAMIK
- PK_TYP_ID.EXTERN_VERWALTETER_TARIF
- PK_TYP_ID.FOERDERKENNZEICHEN
- PK_TYP_ID.GRUPPIERUNG_RECHNUNGSLEGUNG
- PK_TYP_ID.KAPITAL_KONTO
- PK_TYP_ID.LV_PRODUKT_VT_PARAMETER
- PK_TYP_ID.LV_TARIF
- PK_TYP_ID.MANDANT
- PK_TYP_ID.OPTION
- PK_TYP_ID.PROZESS
- PK_TYP_ID.REGULIERUNG
- PK_TYP_ID.RISIKO_PRUEFUNG
- ECK_VERSICHERUNG
- ILAUSZAHLUNG
- RGUETUNGSGRUNDLAGE
- RTRIEBSWEG
- AEHRUNG
- AHLWEG
- AHLWEISE
- D
- KUNFT
- MY
- SERVICE.ERH

Specify/Program/Test/Debug

Generate



C Code

Write formal code in a DSL
mixed with tables and text

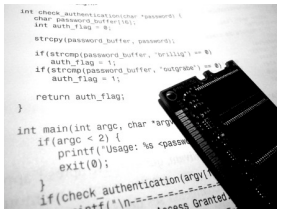
Now with IDE support and executable tests

The same notation!

Exactly the same C code.

Specify/Program/Test/Debug

Generate



C Code

Still exactly the same C code,
or improved as needed.

Incremental Refinement/Refactoring of languages:

Partially automated migration of models
Add model natural notations (insurance-specific, math)
Add Support for modularity, reuse, variants

Small-scale syntactic decisions

```
fun rebate() = switch {
                 case age > 18 && isWorking  : 0.1
                 case age > 18 && !isWorking : 0.2
                 case age < 18 && isWorking  : 1.0
               }
```

```
fun rebate() = alt [ age > 18 && isWorking  => 0.1
                     age > 18 && !isWorking => 0.2
                     age < 18 && isWorking  => 1.0 ]
```

```
fun weightedAverage(values: list<int>, weight: int) =
    frac [ sum[ i: int = 0 ; values.size ; i * weight ] / values.size ]
```

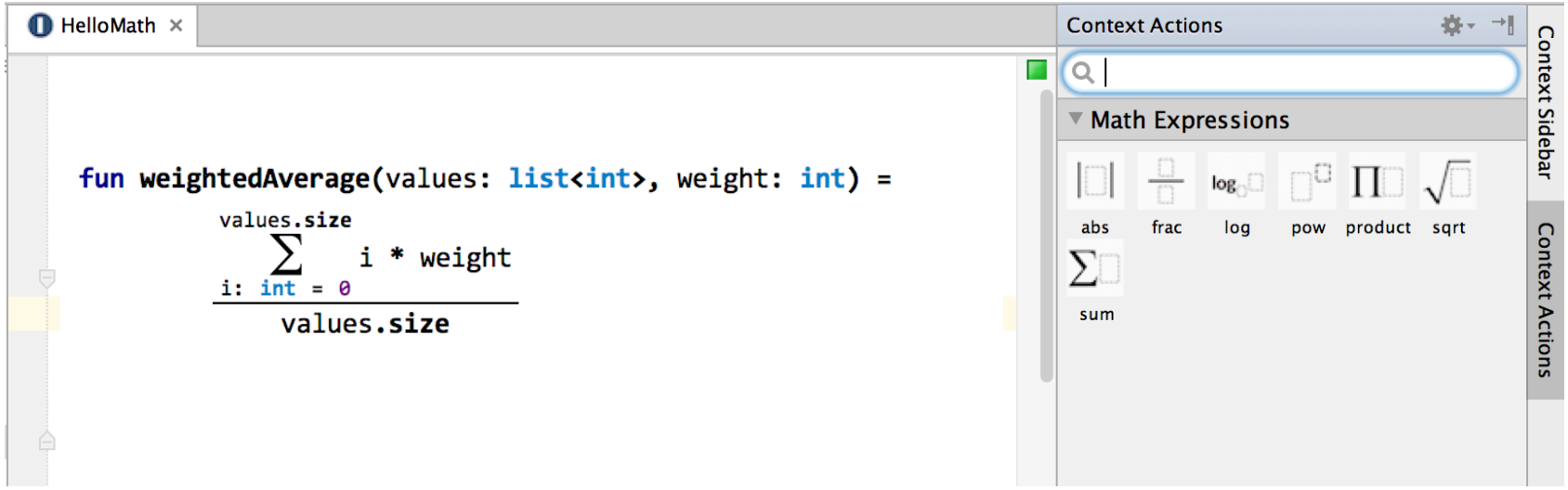
```
fun weightedAverage(values: list<int>, weight: int) =
    values.map(|it * weight|).sum / values.size
```

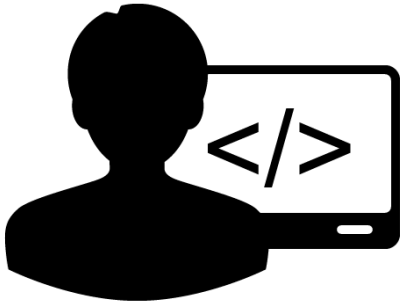
fun weightedAverage(values: list<int>, weight: int) =

$$\frac{\sum_{i:\ int\ =\ 0}^{values.size} i * weight}{values.size}$$

Small-scale syntactic decisions

Tool support via palettes is important for things that „cannot be typed“.

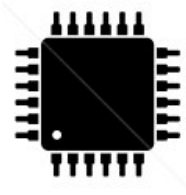




Structure/Guid.	+	-
Notation	Mixed	Text
Views	*	1
IDE/Tool	Clean	Powerful
Learn/Effective	L	E

Syntax is the „UI“ of the language.

Often perceived as the most important aspect from the customer‘s perspective.
Replicating existing syntax is often a good starting point, if your tool can do it.



EMBEDDED SOFTWARE

„General Purpose DSL“

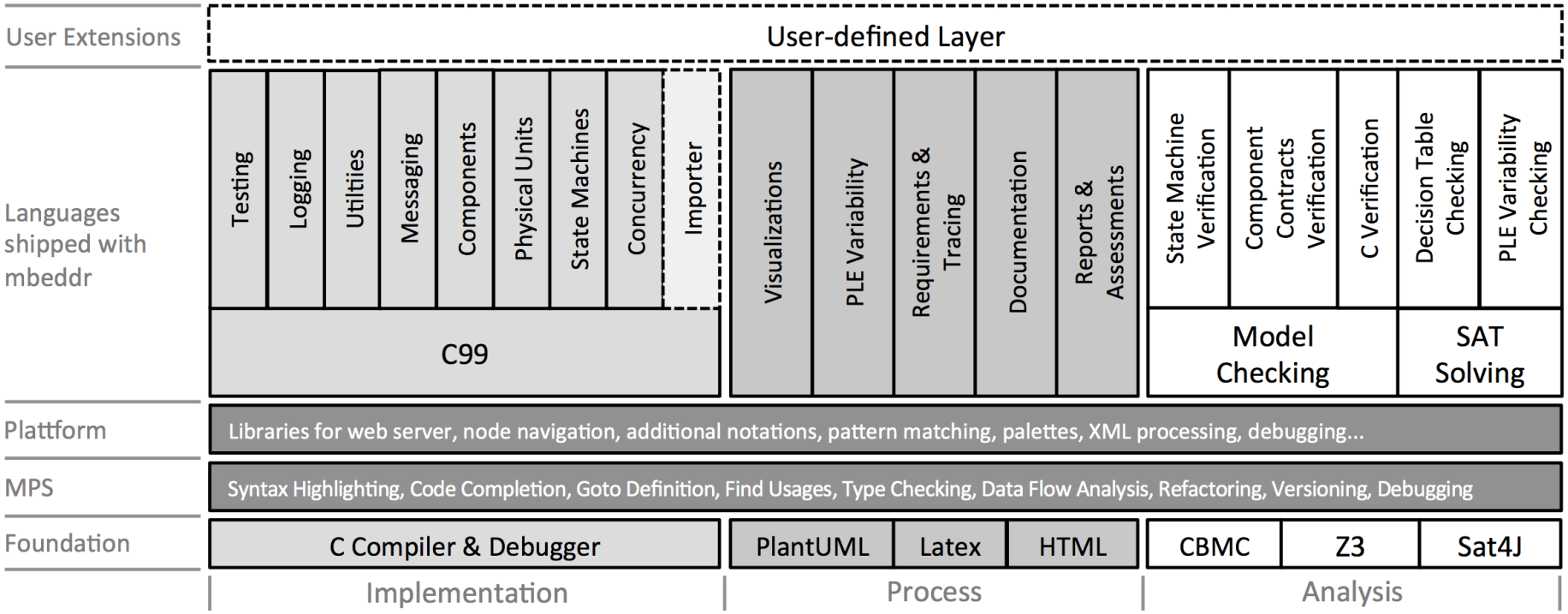
Based on C

Improved Robustness, Productivity

Designed for end-user extensibility


Ext 2a
Ext 1
Ext 2
GPL (Java, C)

An extensible set of integrated languages for embedded software engineering.



Composable extensions, Diverse notations

DLT645ProtocolImpl

Events

```
dataReceived(uint8 data)
idle        [data == DLT645_FRONT_LEADING_BYTE] -> awaking
awaking     [data == DLT645_FRAME_STARTFLAG] -> recvFrame
              {
                send startFrame(idx, data);
                idx++;
              }
recvFrame   [idx < length - 1 && idx != DLT645_HEADER_DATALENGTH_OFFSET]
              idx++;
            }
            [idx == DLT645_HEADER_DATALENGTH_OFFSET] -> rec
              {
```

States

Error: type boolean is not a subtype of uint8

idx ^DLT645ProtocolImpl.DLT645ProtocolImpl.DLT645RxFrameParser.DLT645RxFrameParser.idx (StatemachineVar
rxQueueHeadIdx ^DLT645ProtocolImpl.DLT645ProtocolImpl.rxQue
rxQueueTailIdx ^DLT645ProtocolImpl.DLT645ProtocolImpl.rxQue

UnitDeclarations

```
section AngleUnits {
  exported unit ° := <no spec> f
  exported unit rad := <no spec>
  exported unit rotation := <no

  exported #constant FULL_ROTAT

  exported eager conversion ° ->
   val as <no type> -> val/°/ *
```

```
exported cs interface IGraphPlotter {
  handle newCurve(Color color, string const label)
  void addPoint(handle hCurve, double x, double y)
  void plot(string const title, handle hCurve)
  void plotAll(string const title)
  void clear(handle hCurve)
  void clearAll()
  void deleteCurve(handle hCurve)
  void dispose()
}
```

IGraphPlotter

DCFilterTestHarness

internal instances {

dcFilterTestRunner

dcFilterTestRunnerInst: DCFilterTestRunnerImpl

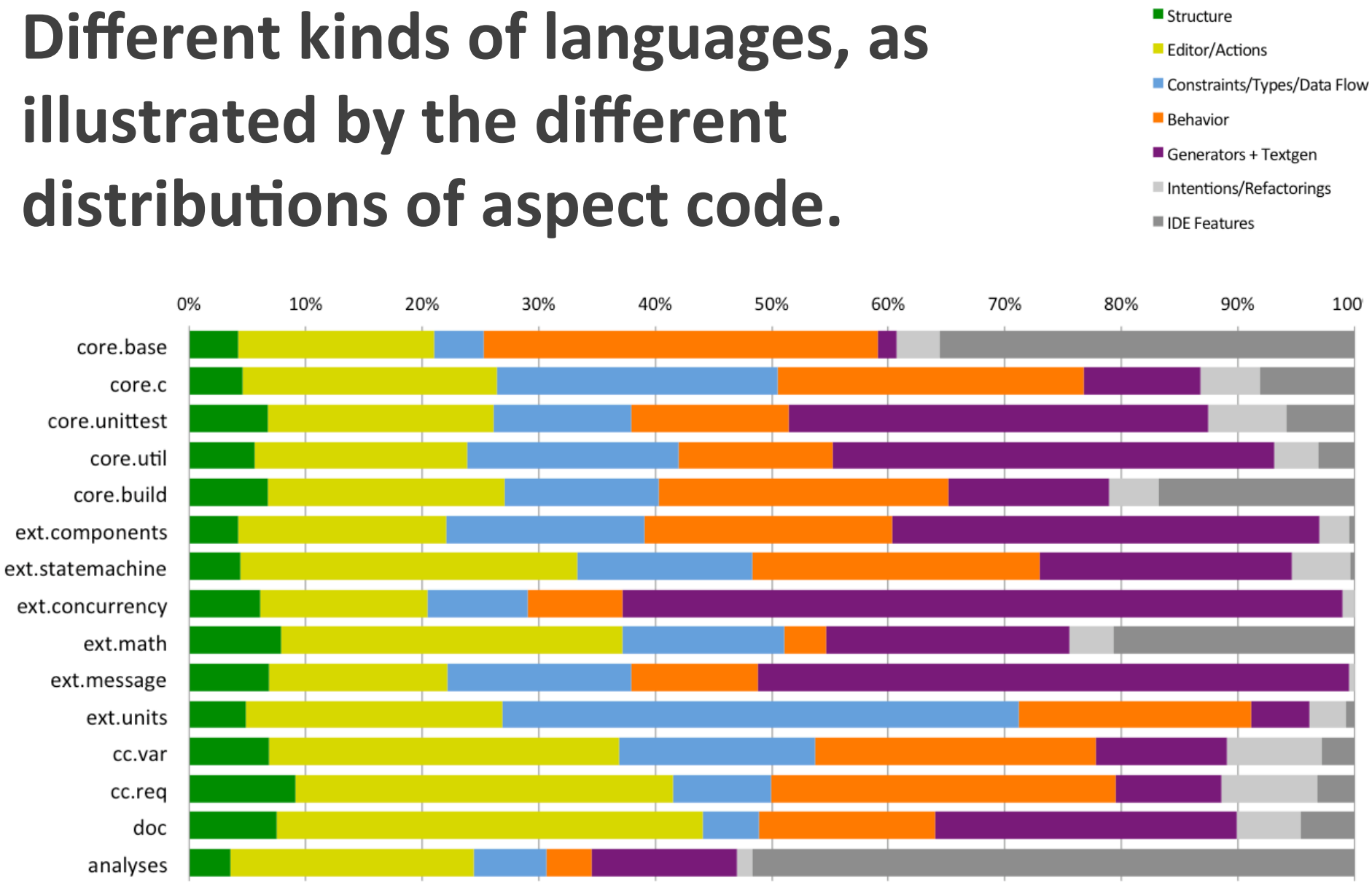
IMetrologyRawSignalSimulator

IMetrologyRawSignalData

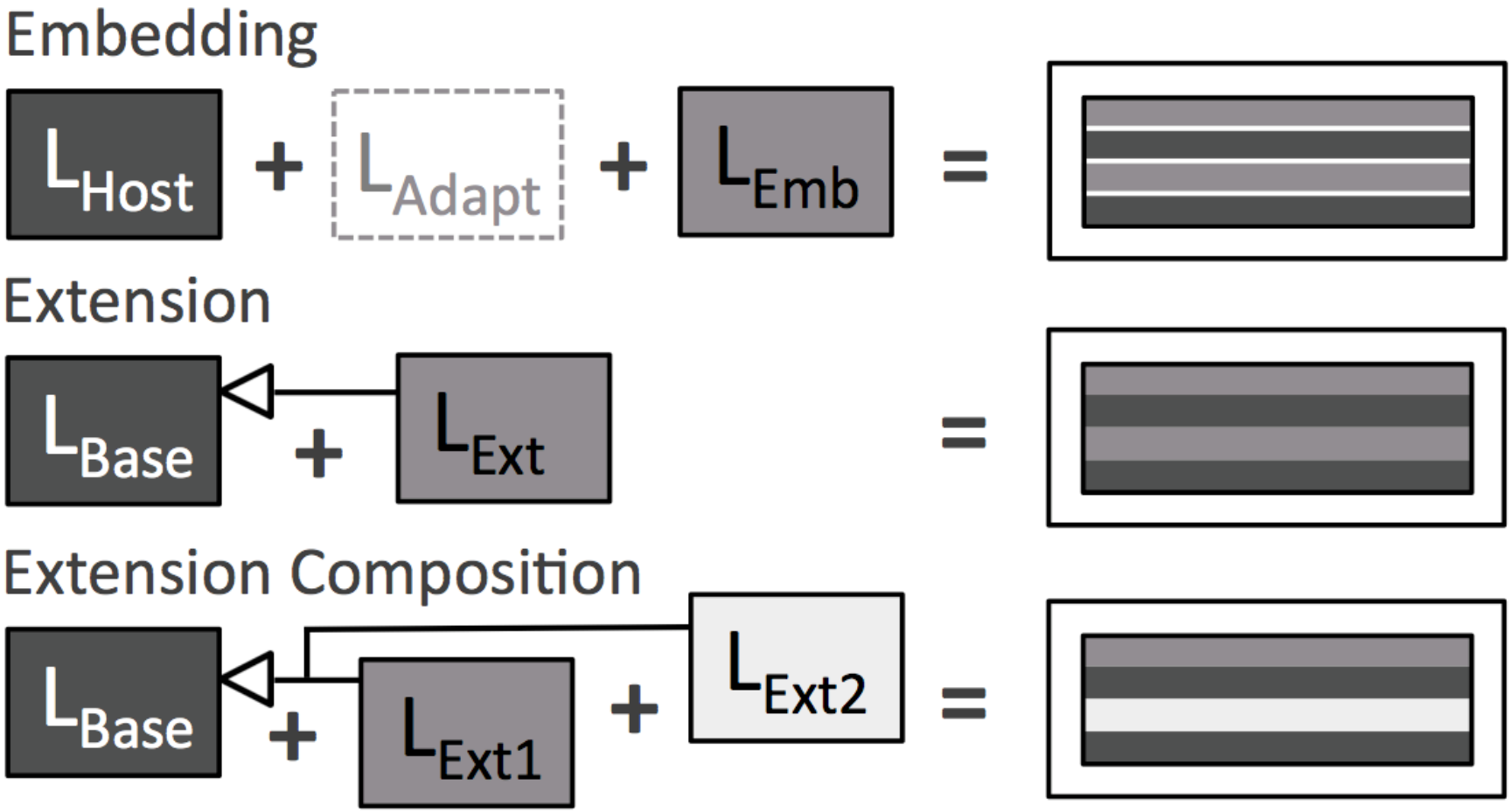
metrologyRawSignalSimulatorIn MetrologyRawSignalSimulatorIm

IMetrologyRawSignalHandler

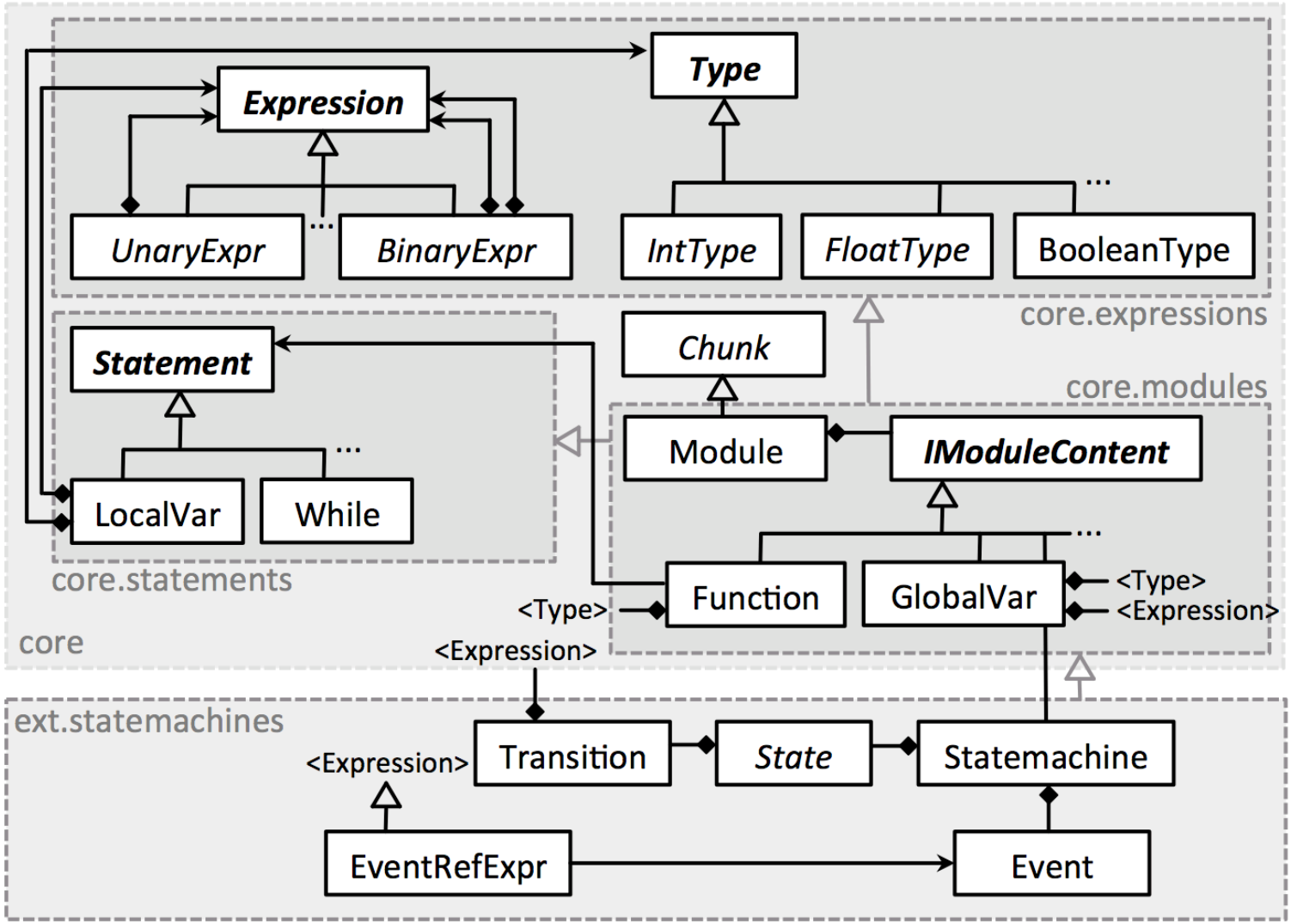
Different kinds of languages, as illustrated by the different distributions of aspect code.



Relying on modular language extension and extension composition.



Typical OO patterns to support extensibility



mbeddr

Using C Language Extensions for Developing Embedded Software: A Case Study

Markus Voelter
independent/itemis, Germany
voelter@acm.org

Arie van Deursen
Delft University of Technology, The Netherlands
Arie.vanDeursen@tudelft.nl

Bernd Kolb, Stephan Eberle
itemis AG, Germany
{kolb|eberle}@itemis.de

Abstract

We report on an industrial case study on developing the embedded software for a smart meter using the C programming language and domain-specific extensions of C such as components, physical units, state machines, registers and interrupts. We find that the extensions help significantly with managing the complexity of the software. They improve testability mainly by supporting hardware-independent testing, as illustrated by low integration efforts. The extensions also do not incur significant overhead regarding memory consumption and performance. Our case study relies on mbeddr, an extensible version of C. mbeddr, in turn, builds on the MPS language workbench which supports modular extension of languages and IDEs.

Categories and Subject Descriptors D.3.2 [*Extensible languages*]; D.3.4 [*Code Generation*]; D.2.3 [*Program Editors*]; C.3 [*Real-time and embedded systems*]

Keywords Embedded Software, Language Engineering, Language Extension, Domain-Specific Language, Case Study

an extensible version of C that comes with extensions relevant to embedded software development. At the same time, C's native constructs are available to write efficient low-level code if needed. For details on mbeddr see Section 2.

Contribution To provide empirical evidence to what extent the kinds of language extensions supported by mbeddr are useful, we report on a case study on the development of a smart meter (SMT). Our contribution is to analyze, in a real-life project, how the extensions affect the complexity, testability, and runtime overhead of embedded software, as well as the effort for its development.

SPLASH
Pittsburgh · 2015
October 25-30

Design for Extensibility

The internal design decisions may not be relevant to the end user, but they are decisive wrt. the ability to modularly extend a language.

This may in itself be a goal!

mbeddr

Integrated Verifications

Bounded Model Checking for C using the CBMC tool.

Components - tutorial - [~/Documents/mbeddr/3.3/mbeddr.core/code/applications/tutorial]

Components

store

```
exported cs interface TrackpointStore1 {
  void store(Trackpoint* tp)
    pre(0) isEmpty()
    pre(1) tp != NULL
    post(2) !isEmpty()
  Trackpoint* get()
    pre(0) !isEmpty()
  Trackpoint* take()
    pre(0) !isEmpty()
    post(1) result != NULL
    post(2) isEmpty()
  query boolean isEmpty()
}

[checked]
exported component InMemoryStorage extends nothing {
  provides TrackpointStore1 store

  Trackpoint* storedTP = NULL;

  void store_store(Trackpoint* tp) <= op store.store {
    // here is a regular statement comment.
       @arg(tp) is a reference to an argument.
    return;
  } runnable store_store

  void init() <= on init { ... }
  Trackpoint* store_get() <= op store.get { ... }
  Trackpoint* store_take() <= op store.take { ... }
  boolean store_isEmpty() <= op store.isEmpty { ... }
} component InMemoryStorage
```

CBMC Lifted Results

Idx	Property	Status	Trace Size	Time
	Components Contracts (7)	FAIL (1)		8.59s
001	pre(0) isEmpty()	SUCCESS		1.57s
002	pre(1) tp != null	SUCCESS		1.54s
003	post(2) !isEmpty()	FAIL	40	1.5s
004	pre(0) !isEmpty()	SUCCESS		1.16s
005	pre(0) !isEmpty()	SUCCESS		1.16s
006	post(1) result != null	SUCCESS		1.06s
007	post(2) isEmpty()	SUCCESS		0.6s

Filter: Regex

Mark Results in Editor Project cex.

Idx	Raw	Thread	Kind	Value
1	6	0	call	Components_verification
2	8	0	call	UnitTestMessages____testing_runn...
3	9	0	return	UnitTestMessages____testing_runn...
4	11	0	enter instance config	verificationInstances
5	17	0	call	init
6	19	0	return	init
7	20	0	leave instance config	verificationInstances
8	22	0	call	process
9	28	0	call	Components_InMemoryStorage_sto...
10	30	0	return	store_isEmpty
11	37	0	call	Components_InMemoryStorage_sto...
12	39	0	call	Components_InMemoryStorage_sto...
13	41	0	return	store_isEmpty
14	45	0	call	Components_InMemoryStorage_sto...
15	47	0	return	store_isEmpty
16	49	0	FAIL	Error Label Reached

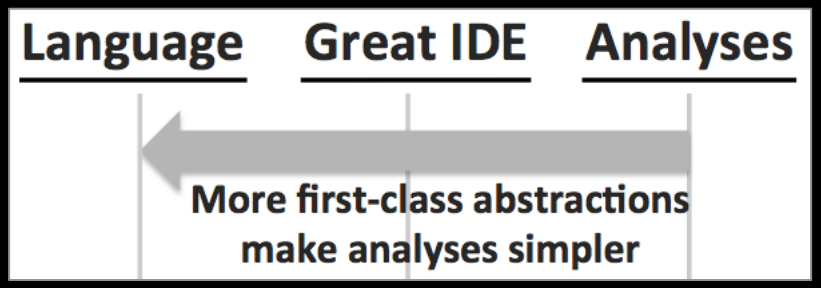
Filter: Regex

Call/Return Locations Last 100 Re-run

CBMC Results Console 3: Usages 0: Messages 9: Version Control Documentation 2 Event Log 2: Inspector

VCS Addons: This project uses Git. For better integration with MPS, it is recommended to update global VCS settings (More info). // Update Dismiss (4 minutes ago) n/a n/a Git: master :ON 883 of 3641M

Formal Verification is simplified if the language has the right abstractions.



... because no semantic recovery is required.

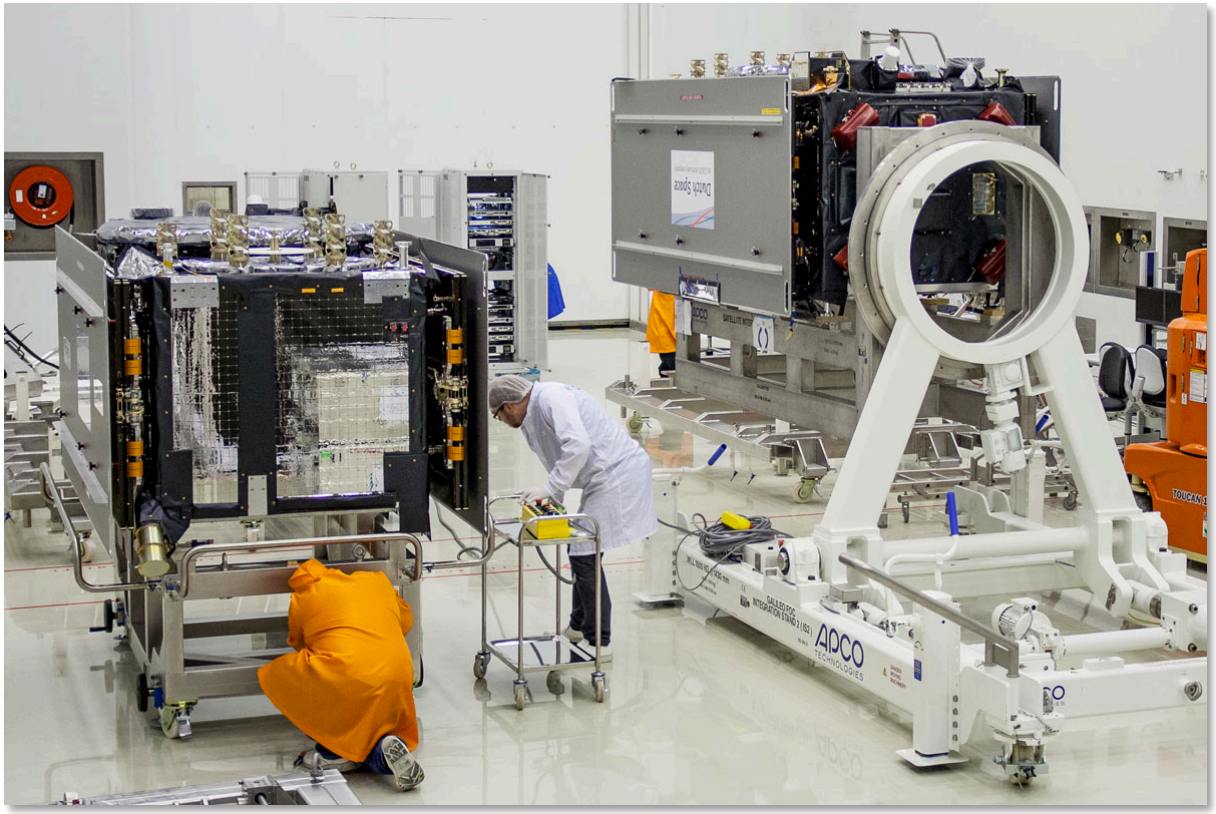
A E R O S P A C E

C-based satellite on-board software

Standardized Architecture

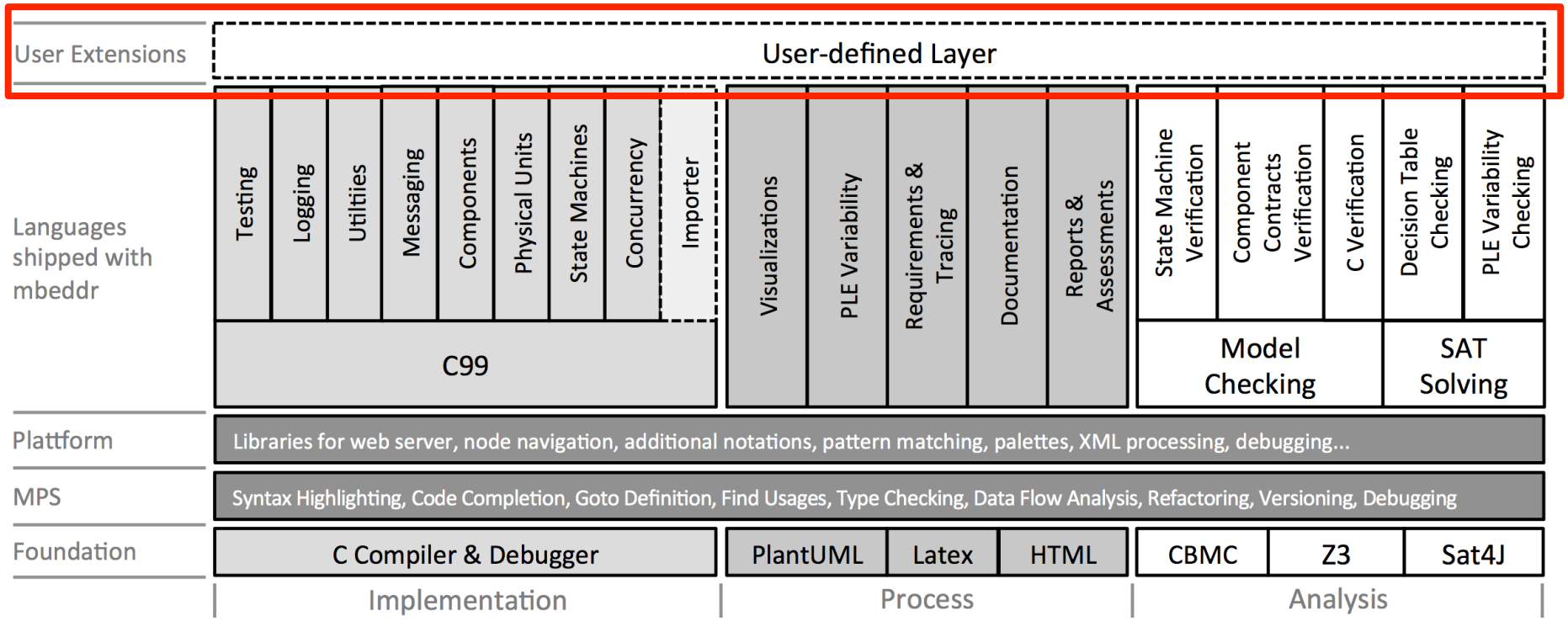
Lots of infrastructure code

Lots of documentation



OHB
SYSTEM

Further domain-specific extensions to C. Developed by end-user lang engineer.



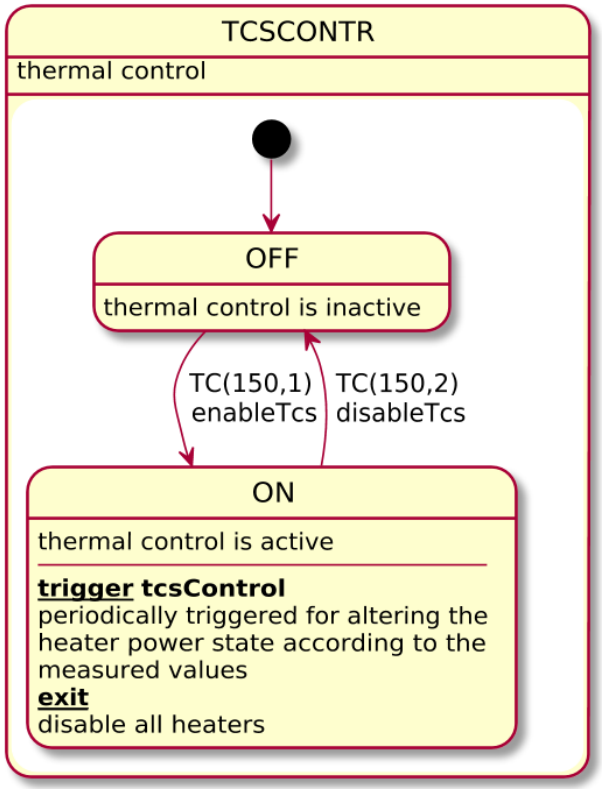
Infrastructure Specifics in C

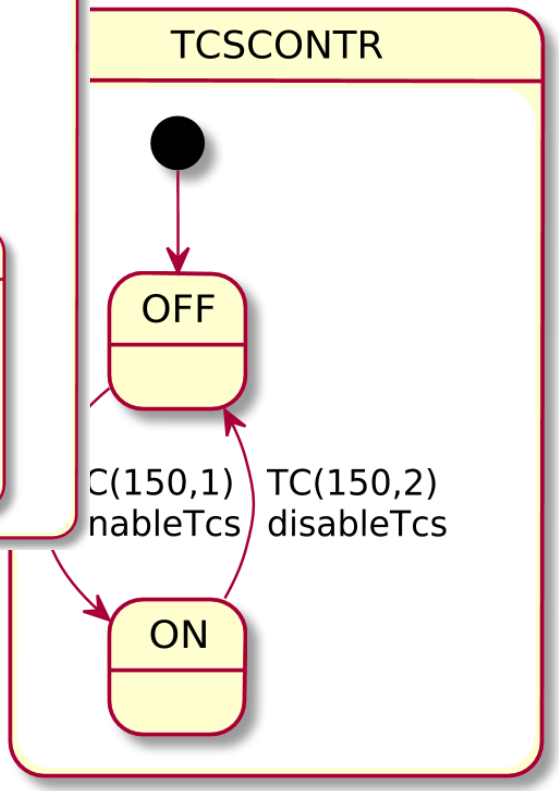
```
#constant TEMP_BUFFER_SIZE = 10;

TACQA = Instance of TemperatureAcquisition with mnemonic tail A and the Numeric Id 350
  [SENSOR = SensorA]
TACQB = Instance of TemperatureAcquisition with mnemonic tail B and the Numeric Id 351
  [SENSOR = SensorB]

Component TemperatureAcquisition with Base Mnemonic: TACQ
  Short Description: acquisition of temperatures

  Description: The components acquires the measurements of an assigned set of thermistors
{
  Attribute (hidden) int32/rawTemp/[TEMP_BUFFER_SIZE] MEASURED = <no init> ; // measured raw values

  Attribute (hidden) uint32 ACQCNT = 0 ; // index for filling data acquisition buffer

  Attribute (readwrite) tempSensor SENSOR (Id= 2 ) = <no init> ; // selected sensor for this component

  ModeChart TCSACQ (Id= 3 ) initial = OFF {
    Trigger tcsAcquisition
    Mode OFF {
      << ... >>
    }
    Mode ON {
      entry { ACQCNT = 0; }
      on trigger tcsAcquisition {

        // measure a value
        MEASURED[ACQCNT] = readTemperature(SENSOR);
        ACQCNT = (ACQCNT + 1) % TEMP_BUFFER_SIZE;

        // calculate average of the @top(TEMP_BUFFER_SIZE) latest measurements and convert to °C
                                    TEMP_BUFFER_SIZE - 1
        PUS150.AVTEMP = convert[        Σ          (MEASURED[idx]) / TEMP_BUFFER_SIZE -> °C];
                                    idx = 0
      }
    }
  }

  Activity startAcquisition with Nummeric Id 1
  ... { TCSACQ.setMode(ON); }

  Activity stopAcquisition with Nummeric Id 2
  ... { TCSACQ.setMode(OFF); }
} Component TemperatureAcquisition
```

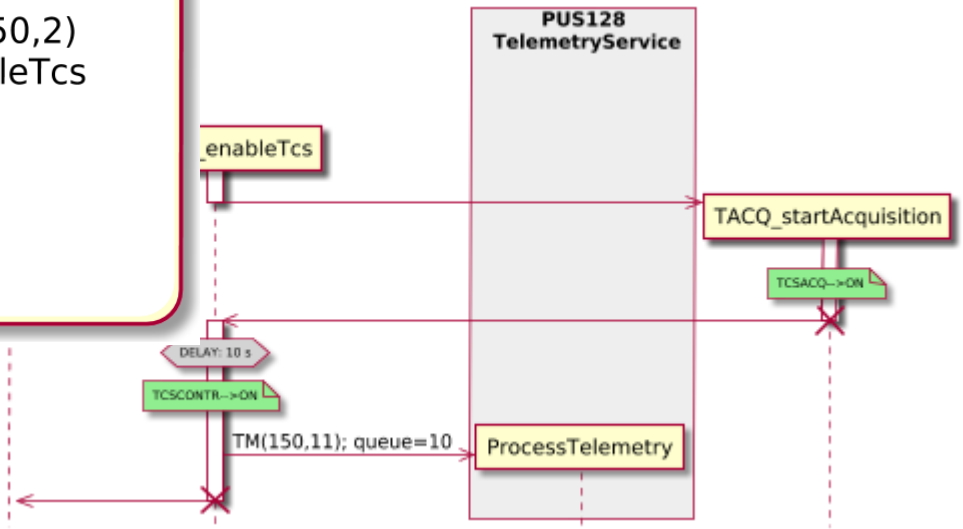
Infrastructure Specific in C

```
Activity enableTcs with Nummeric Id 1 is commandable by TC(150,1)
  Short Description: enable thermal control
  Description: The thermal control heats the system if it is too cold. The switching histeresis can be configured.
  Constraints:
    0: TCSCONTR.inMode(OFF ) // switching on is possible only if the TCS is off
  In-Parameter:
    int16/°C/ upperThreshold: constrained :  <no constraint> // upper switching threshold
    int16/°C/ lowerThreshold: constrained :  lowerThreshold < upperThreshold // lower switching threshold
    component<TemperatureAcquisition> acq: constrained :  <no constraint> // acqusition component instance to use
  {
    REQUEST acq.startAcquisition ( << ... >> ) --> ( << ... >> )
      on error do nothing special
      on error abort
    UPTH = upperThreshold;
    LOTH = lowerThreshold;
    DELAY for 10 s
    TCSCONTR.setMode(ON);
    TELEMETRY (150,11)
      Description: Report switching on in a dedicated packet that reports the initial temperature.
      [initialTemp : int32/°C/ = PUS150.AVTEMP // initial temperature when starting thermal control   ]
  }

Activity disableTcs with Nummeric Id 2 is commandable by TC(150,2)
  Short Description: disable thermal control
  Description:
  Constraints:
    0: TCSCONTR.inMode(ON ) // switching off is possible only if the TCS is on
  In-Parameter:
    << ... >>
  {
    TCSCONTR.setMode(OFF);
    REQUEST TACQA.stopAcquisition ( << ... >> ) --> ( << ... >> )
      on error do nothing special
    REQUEST TACQB.stopAcquisition ( << ... >> ) --> ( << ... >> )
      on error do nothing special

  }
} Component ThermalControlSystem
```

Generated Documentation







An existing meta model / standard is often a great starting point for a DSL.

There is more to generate/derive than just executable software.

“I am committing myself to develop the next satellite‘s on board software in one year instead of the usual 50.”

The more infrastructure you have to deal with, the more can be generated.

Remote communication, protocols, buffering/serialization, multi-platform interfaces...

AUTOMOTIVE

Specification of Functions and Interfaces
And the Data communicated betw. Them
Incrementally more specific

Start with a few ontological data definitions

```
data Speed
data Position
data EngineStatus
data RoadConditions
data Gear
```

Define components that produce or consume the data items (allocate responsibilities).

```
data Speed_RoadSensors
data Speed
data Position
data EngineStatus
data RoadConditions
data Gear

functional component DriveTrain {
  produces Speed
  produces EngineStatus
  produces Gear
}

functional component RoadSensors {
  produces Speed_RoadSensors
  produces RoadConditions
}

functional component LocationServices {
  produces Position
}
```

```
data GenericSpeed
data SpeedFromRoad extends GenericSpeed
data SpeedFromEngine extends GenericSpeed

data Position
data EngineStatus
data RoadConditions

data Gear

functional component DriveTrain {
  produces SpeedFromEngine
  produces EngineStatus
  produces Gear
}

functional component RoadSensors {
  produces SpeedFromRoad
  produces RoadConditions
}

functional component LocationServices {
  produces Position
}
```

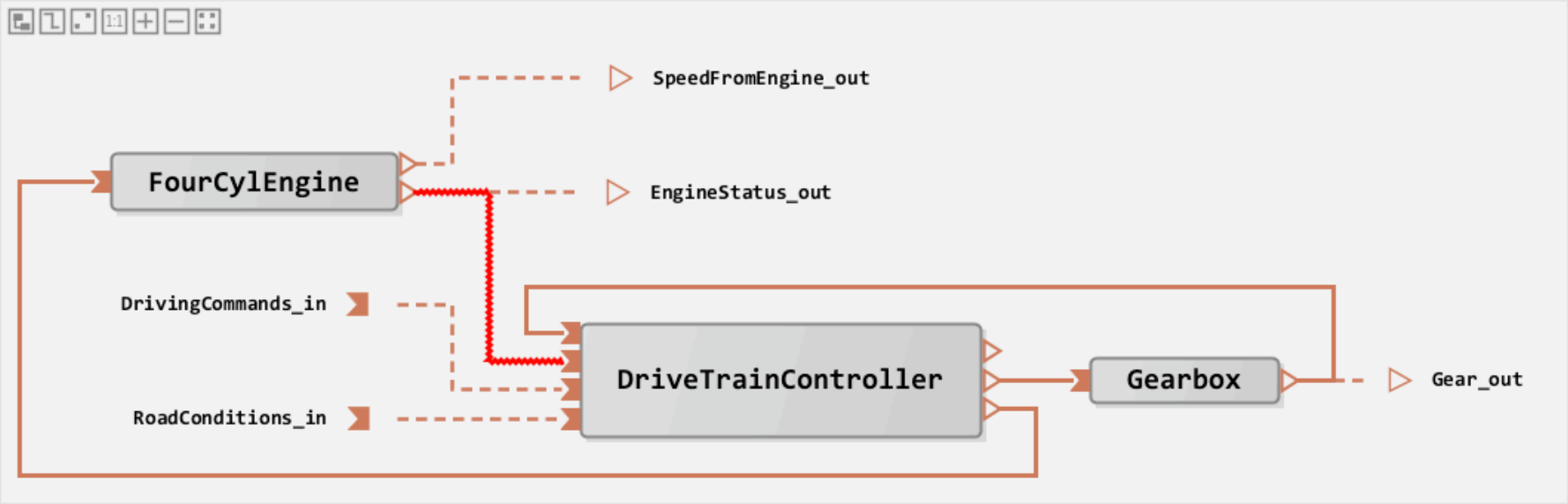
Establish refinement relationships between data items.

```
data GenericSpeed : int
data SpeedFromRoad extends GenericSpeed
data SpeedFromEngine extends GenericSpeed where it.range[0..350]

data SpeedFromRoad_Fwd extends SpeedFromEngine where it.range[0..300]

data Pos_Lat : int where it.range[-90..90]
data Pos_Long : int where it.range[-180..180]
data Position {
  Pos_Lat
  Pos_Long
}

data EngineRPM : int where it.range[0..10000]
data EngineOnOff : enum[on, off]
data EngineStatus {
  EngineOnOff
  EngineRPM
}

data Gear : int where it.in[1, 2, 3, 4, 5, 6]

data RoadConditions : enum[dry, wet, snow]
```

Only then do you start defining types and value constraints.

Then decompose hierarchical system structures.

```
public functional component DriveTrain {
  produces SpeedFromEngine
  produces EngineStatus
  produces Gear where it < gearsCount
  consumes RoadConditions
  param int gearsCount
  consumes DrivingCommands
```

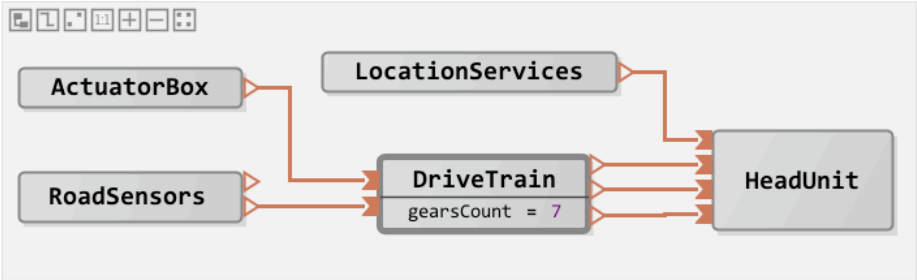



```
}
```

... recursively.

Step11Car

```
components Step11Car                    imports Step11

functional component HeadUnit {
  consumes EngineStatus
  consumes Position
  consumes GenericSpeed
  consumes Gear
}

functional component ActuatorBox {
  produces DrivingCommands
}

functional component Car {
```




```
}
```

Tree : structure for Car

structure

- Car
  - HeadUnit
  - DriveTrain
    - FourCylEngine
    - Gearbox
    - DriveTrainController
    - DriveTrainController.Req_Accelerator_out -> FourCylEngine.Req_Accelerato
    - DriveTrainController.Req_Gear_out -> Gearbox.Req_Gear_in
    - RoadConditions_in -> DriveTrainController.RoadConditions_in
    - FourCylEngine.EngineStatus_out -> EngineStatus_out
    - Gearbox.Gear_out -> Gear_out
    - FourCylEngine.SpeedFromEngine_out -> SpeedFromEngine_out
    - FourCylEngine.EngineStatus_out -> DriveTrainController.EngineRPM_in
    - Gearbox.Gear_out -> DriveTrainController.Gear_in
    - DrivingCommands_in -> DriveTrainController.DrivingCommands_in
  - RoadSensors
  - LocationServices
  - ActuatorBox
  - DriveTrain.SpeedFromEngine_out -> HeadUnit.GenericSpeed_in
  - LocationServices.Position_out -> HeadUnit.Position_in
  - DriveTrain.EngineStatus_out -> HeadUnit.EngineStatus_in
  - RoadSensors.RoadConditions_out -> DriveTrain.RoadConditions_in
  - DriveTrain.Gear_out -> HeadUnit.Gear_in
  - ActuatorBox.DrivingCommands_out -> DriveTrain.DrivingCommands_in

Workflow may impact the language.

Not defining a type for a variable is unintuitive for the software engineer. But it is the way a systems engineer works. Make sure you accomodate this.

HEALTH/MEDICINE

Software Medical Devices

Accessible to Doctors

Robustness/Correctness Required

To be FDA-certified

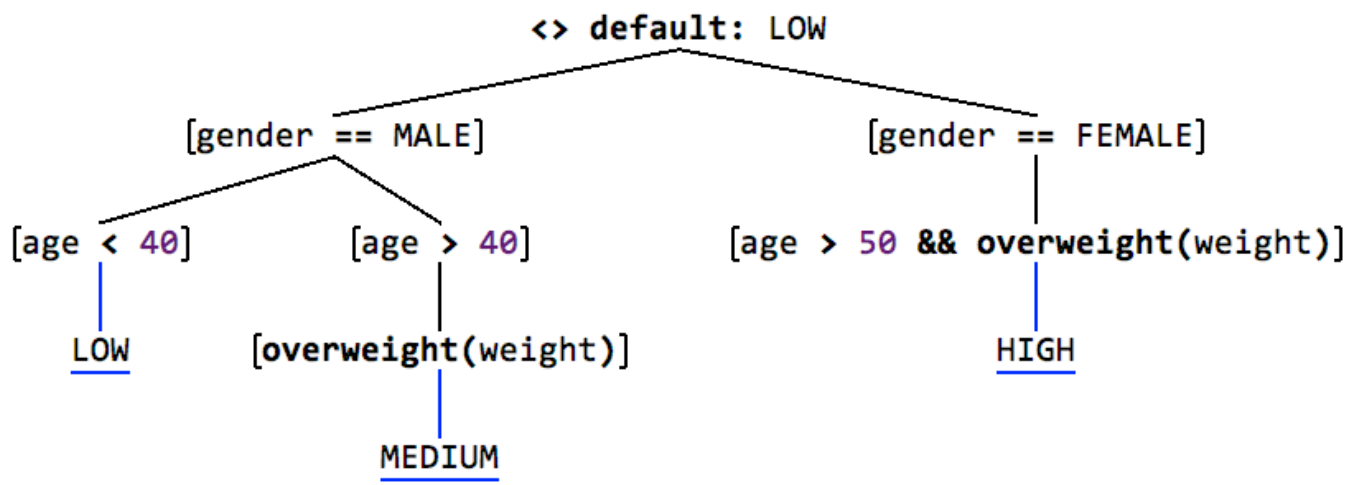
Semi-Graphical Expressions

```
val c2: int = split three [ < 0   => 0 ]
                          [ 0..3  => 42 ]
                          [ > 3   => 44 ]
```

```
fun pricePerMin(time: int, region: int) =
```

	region == EUROPE	region.in[USCAN, ASIA]
time.range[0..6]	12	10
time.range[7..17]	20	22
time.range[18..24]	17	20

```
fun riskFactor(gender: int, age: int, weight: int) =
```



Prose-like call syntax

```
ext fun calculateRisk(this: Person, last: int, previous: int) =
```

	last < 100	last >= 100
this.age.in[0, 10]	split previous [< 10 => LOW; >= 10 => MED]	LOW
this.age.in[11, 18]	LOW	MED
this.age > 18	LOW	HIGH

```
record Person { age: int }
val p = #Person{20}
```

```
p.calculateRisk(100, 60) ==> HIGH
```

Extension function can be called in dot-notation, perfectly suitable for developers.

Prose-like call syntax

```
@syntax{stroke risk for last  @[last]  and but-last  @[previous]  blood sugar}
ext fun calculateRisk(this: Person, last: int, previous: int) =
```

	last < 100	last >= 100
this.age.in[0, 10]	split previous [< 10 => LOW; >= 10 => MED]	LOW
this.age.in[11, 18]	LOW	MED
this.age > 18	LOW	HIGH

```
record Person { age: int }
val p = #Person{20}
```

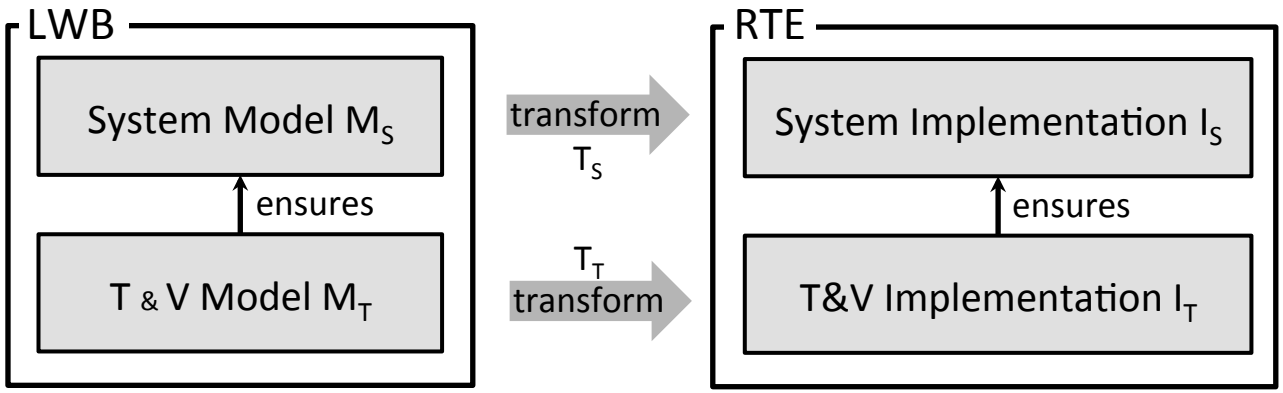
```
p.stroke risk for last 100 and but-last 60 blood sugar ==> HIGH
```

For non-programmers, a more prose-like notation is helpful. Notice the prose-call facility is a modular extension of the expression language.

Once again, be creative in the notations you use to accomodate end users.

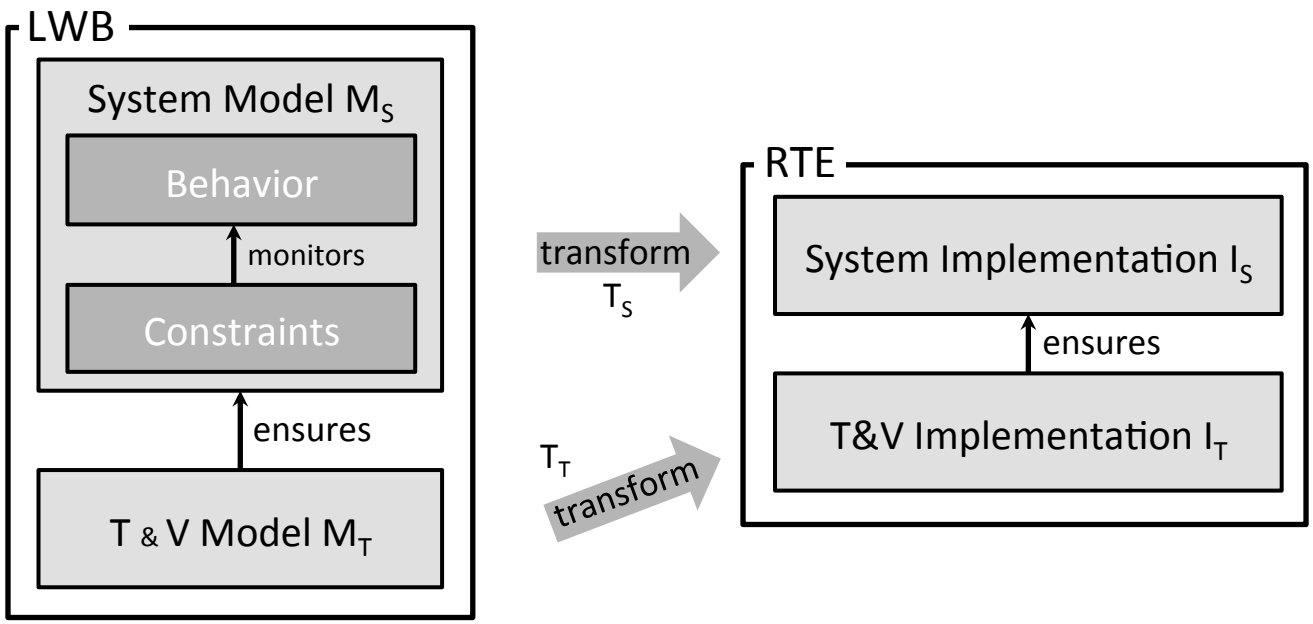
They might not like your fancy higher-order, lambda-based pure functional language.

Test + Monitoring Aspects



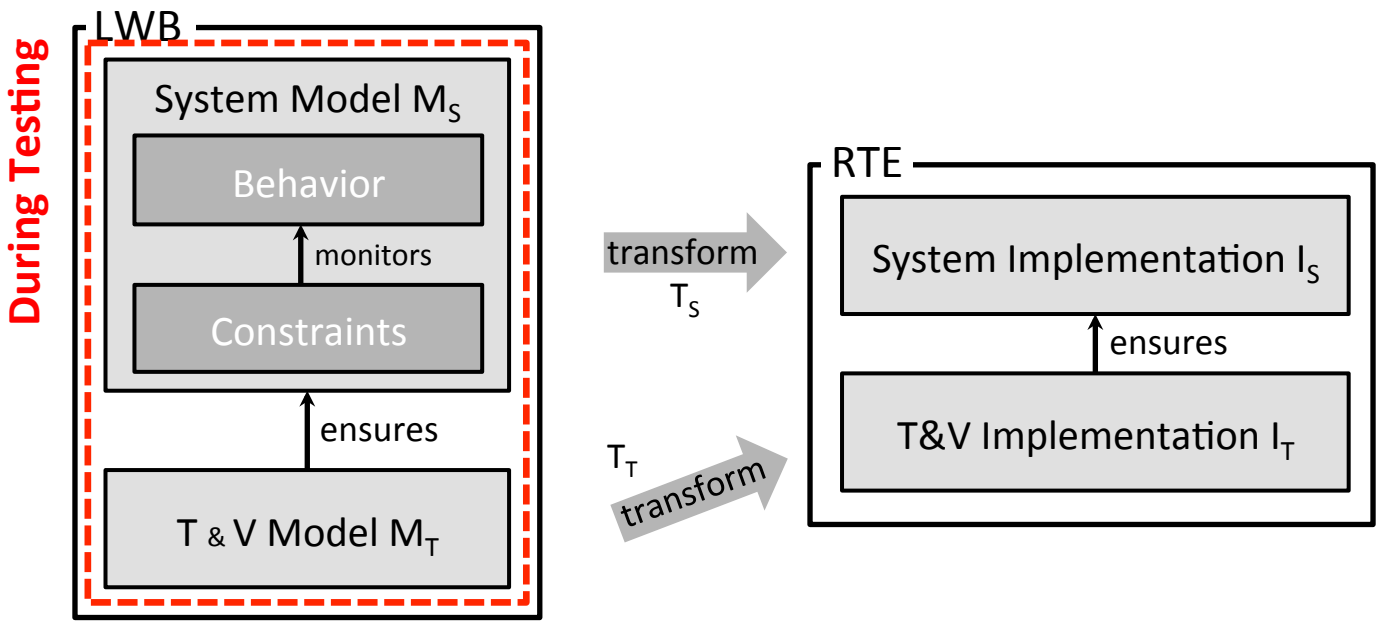
Established approaches for developing reliable and safe systems applied to languages.

Test + Monitoring Aspects



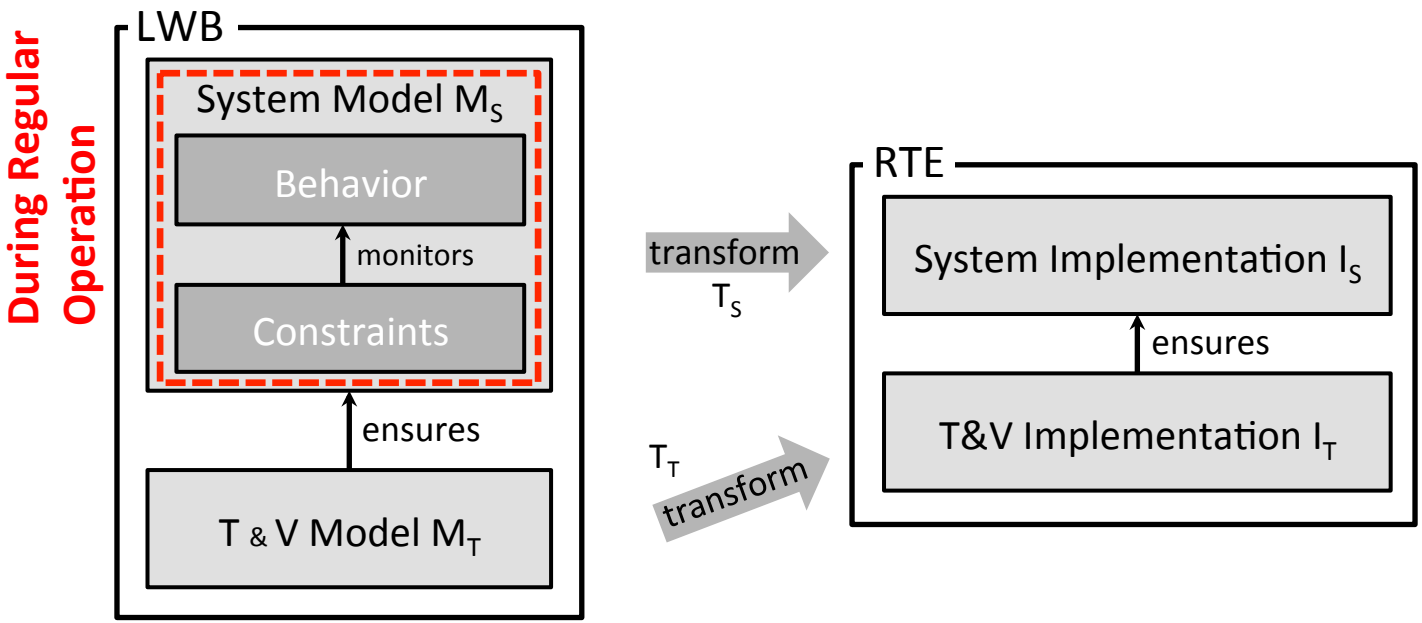
Established approaches for developing reliable and safe systems applied to languages.

Test + Monitoring Aspects



Established approaches for developing reliable and safe systems applied to languages.

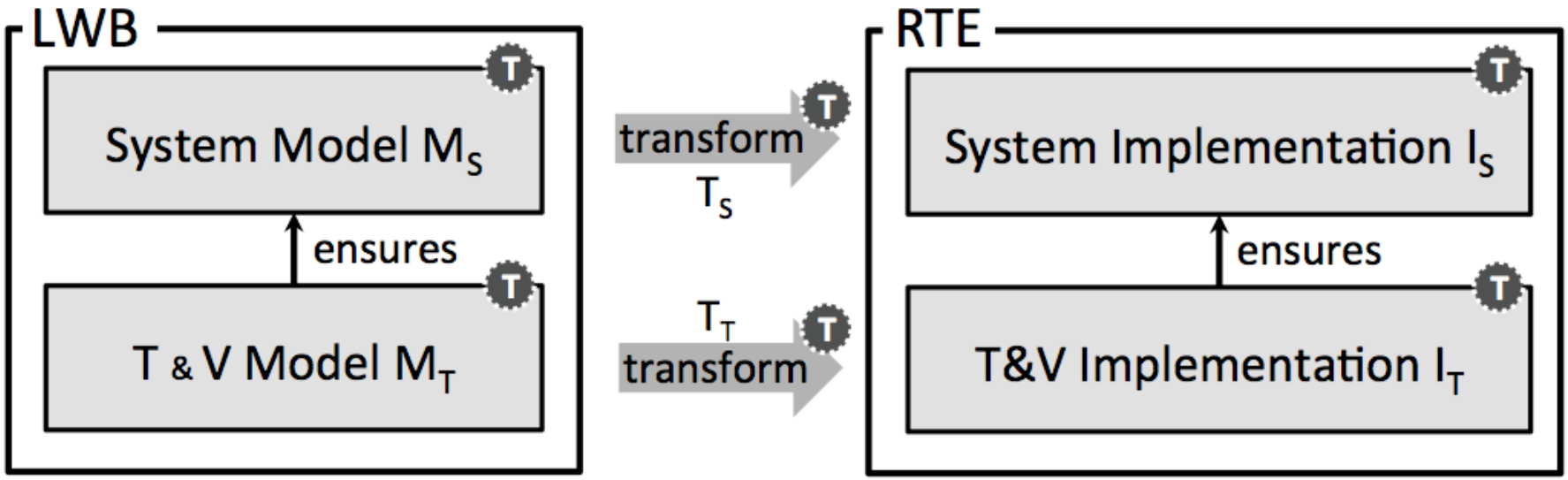
Test + Monitoring Aspects



Established approaches for developing reliable and safe systems applied to languages.

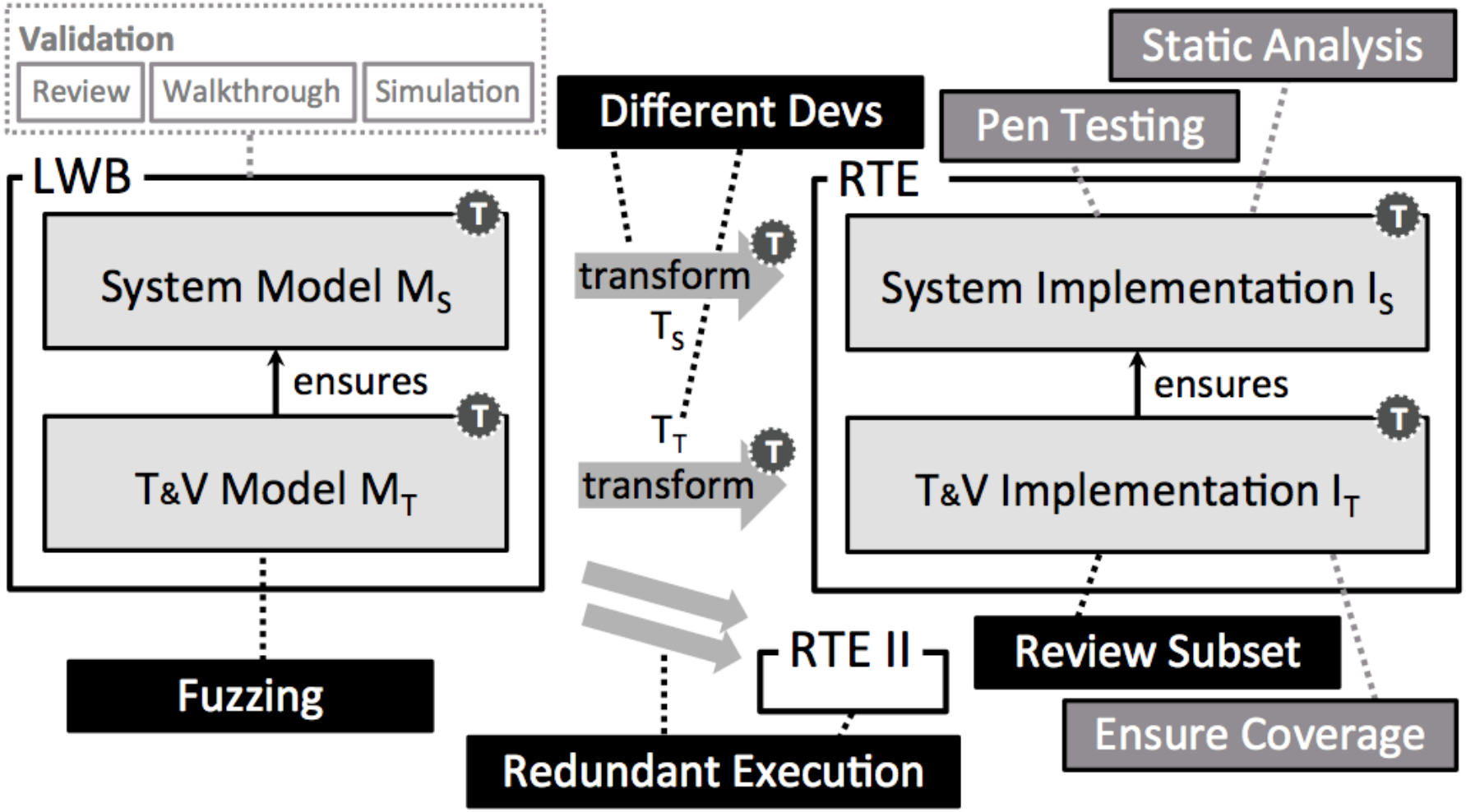
Redundancy in the Tool Chain

The basic idea is that the system model and the test model are distinct, and are transformed separately.



Redundancy in the Tool Chain

A number of additional steps must be taken to address possible error scenarios; many rely on redundancy.



Integrated Verifications

Integrated SMT solver checks for logical and arithmetic inconsistencies in decision tables.

```
type sugarLvl: int where [it.in[0, 200]]
type age: int where [it.in[0, 120]]

record Person { age: age }

ext fun calculateRisk(this: Person, last: sugarLvl, previous: sugarLvl) =
```

	last.range[0..100]	last.range[100..200]
this.age <= 10	split previous [< 10 => LOW; >= 10 => MED]	LOW
this.age.range[11..18[	LOW	MED
this.age > 18	LOW	

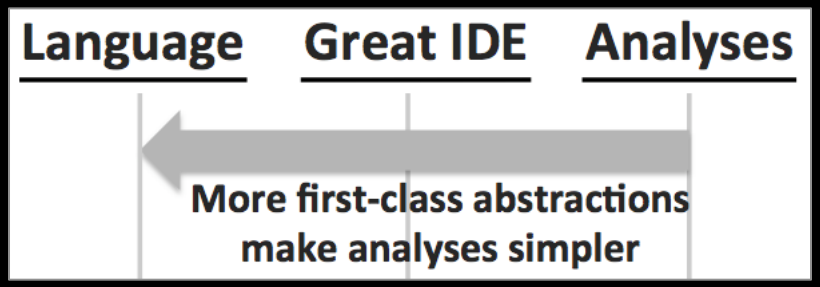
Error: incomplete: this.age == 18 not covered
Error: inconsistent: last == 100 is ambiguous

Actions

Solver also for decision trees, state machines and protocol/contract checking in other projects.

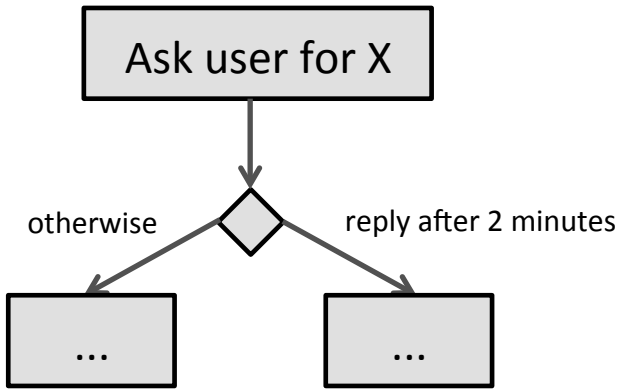
Non-Functional Concerns can have a huge impact on language design and transformation architecture.

Safety, Security, Performance



End Users vs. Semantics

Estsblished Notations
End-user Happyness

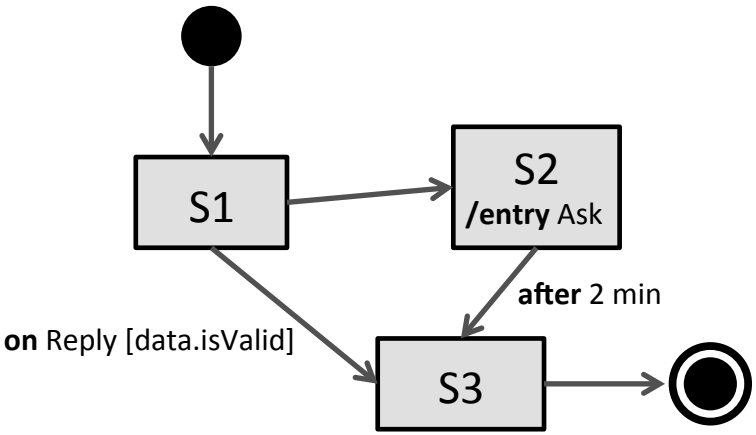


Some kind of decision tree/diagram

Asynchronicity (User Interaction)
Time-based decisions (timeouts)
Checking for completeness and other criteria (model checking)

?

Expressive Formalism
Verification feasible

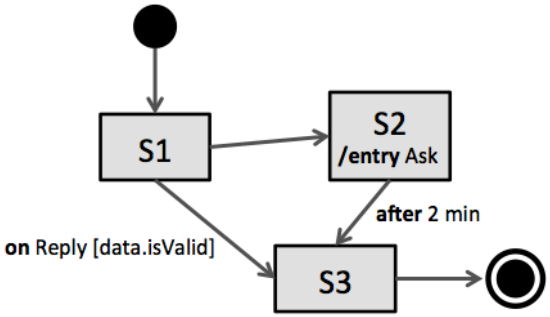


State Machine

No problem for semantic issues.
But „strange“ way of expressing things.
Hard to get accepted.

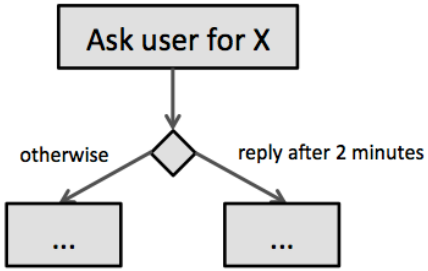
Generating „nice“ reports

Start with this ...

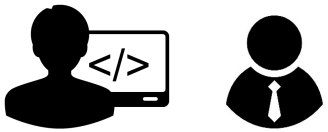


Editable. Analyzable.

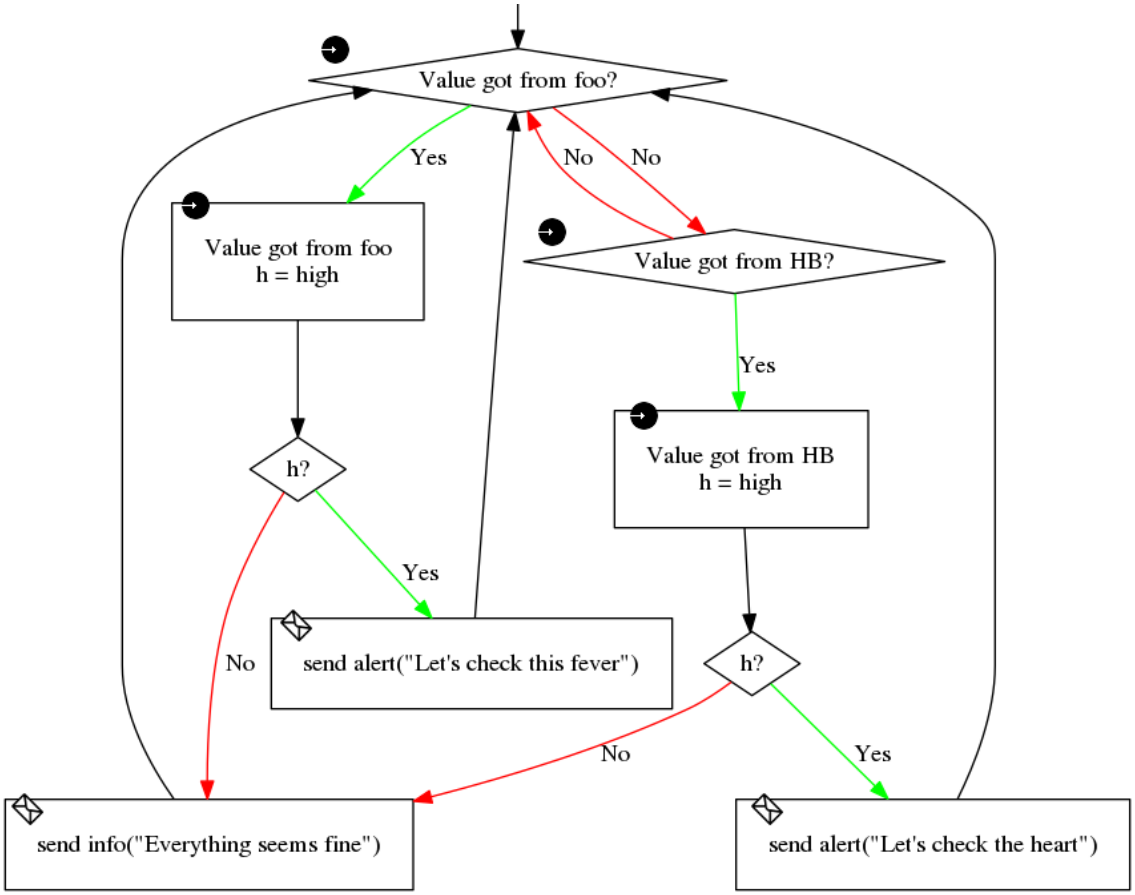
... derive this.



Read-only.
Traceback.
Configurable.



Generating „nice“ reports



Read-only.
Traceback.
Configurable.

Generate partially.
Avoid some details.
Mark Exceptional Paths.
Show only Happy Path.
Highlight Safety-Checks.
...

Levels of

Domain Expert Integration

Generate derived artifacts

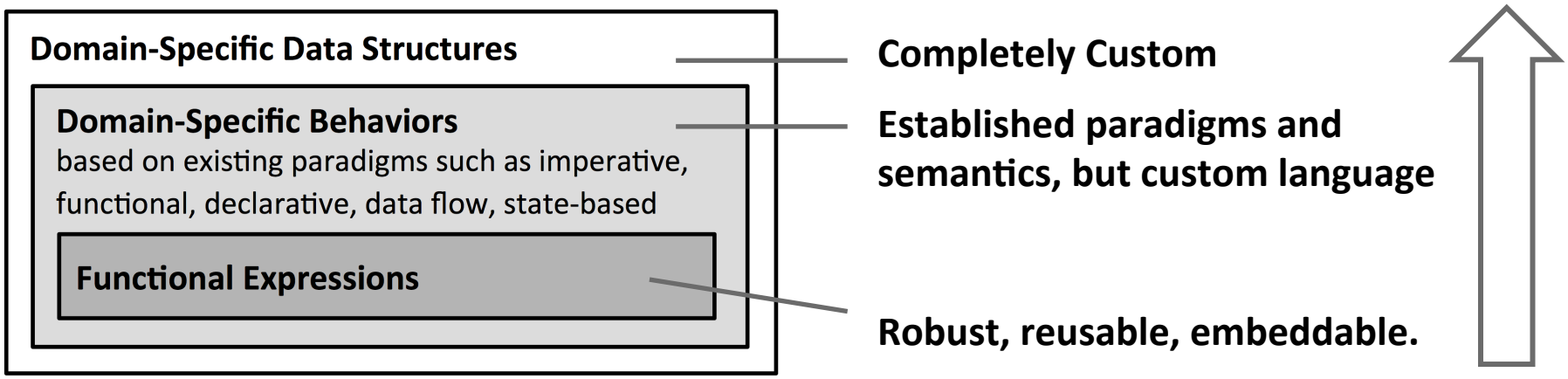
Review the DSL sources

Pair programming

Independent Development

Development over time.
User sophistication

Semantics First Language Design!



LWB must support language composition!

EXPRESSIONS

Reusable Expression Language

Embedded in > 10 DSLs

What is a good way to do option types?

Basic Expressions and types

```
val aBool   : boolean = true
val anInt   : int     = 42
val aReal   : real    = 33.33
val aString : string  = "Hello"
```

```
42 + 33                        ==> 75 [int]
42 + 2 * 3                     ==> 48 [int]
aReal + anInt                  ==> 75.33 [real]
aBool && true                  ==> true [boolean]
"Hello, " + "World"            ==> "Hello, World" [string]
if [ aBool then 42 else 33 ] ==> 42 [int]
```

Option Types

```
val maybe : option<int> = if aBool then some(42) else none
```

```
if some?(maybe) then maybe else 0 ==> 42
```

What is a good way to do option types?

```
if some?(maybe) then maybe else 0 ==> 42
```

Good == Safe

can the value only be used in the `then` part?

Checkable

does type checking work?

Concise

how writable and readable is it?

Non-Surprising

is the syntax surprising or obvious?

Implementable

is it implementable with MPS and easily extensible for language extension developers?

Starting Point: a first-class expression ...

```
fun f(x: option<int>) = with some x => val none 10
```

Safe
Checkable
Concise
Non-Surprising
Implementable

Problem: what about nesting?

one has specify names for the value
otherwise the val is ambiguous

```
fun f(x: int, y: int) = {
  val xval = with some maybe(x) => val none 10
  with some maybe(y) => val + xval none 20
}
```

Safe
Checkable
Concise
Non-Surprising
Implementable

Solution: two styles for optional names

```
fun f(x: int) = with some v = maybe(x)    => v none 10
fun f(x: int) = with some maybe(x) as v   => v none 10
```

Safe
Checkable
Concise
Non-Surprising
Implementable

We opted for the 2nd version, because

IDE support is better: write post-facto
we already had a framework for the **as** notation

Nesting: one can now write this

```
fun f(x: int, y: int) = {
  with some maybe(x) as xval
     => with some maybe(y) as yval => xval + yval
                                      none 0
     none 0
}
```

Safe
Checkable
Concise
Non-Surprising
Implementable

This is ugly!

Better: support comma-separated multiple tests

```
fun f(x: int, y: int) =
  with some maybe(x) as xval, maybe(y) as yval
     => xval + yval none 0
```

Safe
Checkable
Concise
Non-Surprising
Implementable

Nesting: one can now write this

```
fun f(x: int, y: int) = {
  with some maybe(x) as xval
     => with some maybe(y) as yval => xval + yval
                                      none 0
     none 0
}
```

Safe
Checkable
Concise
Non-Surprising
Implementable

This is ugly!

Better: support comma-separated multiple tests

```
fun f(x: int, y: int) =
  with some maybe(x) as xval, maybe(y) as yval
     => xval + yval none 0
```

Safe
Checkable
Concise
Non-Surprising
Implementable

But: why not just use an `if`?

Why not just use an `if`?

```
fun f(x: option<int>) = if some(x)      then val   else
fun f(x: option<int>) = if some(x)      then x.val else
fun f(x: int) = if some(maybe(x))       then val   else 10
fun f(x: int) = if some(maybe(x) as v)  then v     else 10
```

Safe
Checkable
Concise
Non-Surprising
Implementable

Problem: how to analyze this?

```
fun f(x: option<int>) = if some(x) || g(x) then val else 10
```

Solution: enfore `some` to be topmost expression

```
fun f(x: option<int>) = if some(x) || g(x) then val else
```

Safe
Checkable
Concise
Non-Surprising
Implementable

Use an option as `bool`?

```
fun f(x: option<int>) = if x then val else 10
```

Safe
Checkable
Concise
Non-Surprising
Implementable

Discarded: too much magic.

Use the expression in the `then` part?

```
fun f(x: option<int>) = if some(x) then x else 10
```

Safe
Checkable
Concise
Non-Surprising
Implementable

Challenging to implement with MPS because
- one cannot change the type of a node (**x**) depending on its context.
- alternatively, the **x** in the `then` part must be a different lang concept with negative consequences for scoping.

Adopted, accepting the scoping challenge.

Final issue: comma in expressions

```
fun f(x: int, y: option<int>) =
    if some(maybe(x)) as xval, some(y)
       then xval + y else 0
```

Safe
Checkable
Concise
Non-Surprising
Implementable

Solution: allow && but not other operators

```
fun f(x: int, y: option<int>) =
    if some(maybe(x)) as xval && some(y)
       then xval + y else 0
```

Safe
Checkable
Concise
Non-Surprising
Implementable

Final, adopted solution:

```
fun f(x: option<int>) = if some(x) then x else 10

fun f(x: int) = if some(maybe(x) as v) then v else 10

fun f(x: int, y: option<int>) =
    if some(maybe(x)) as xval && some(y)
       then xval + y else 0
```

Safe
Checkable
Concise
Non-Surprising
Implementable

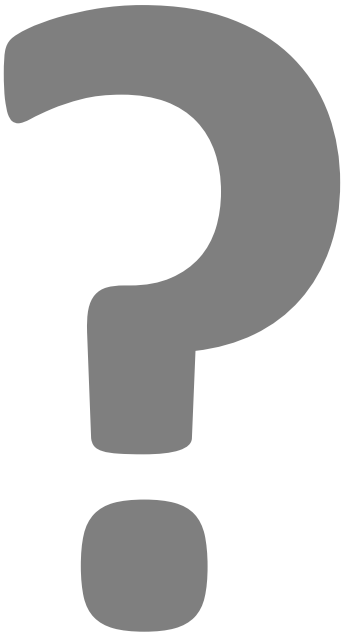
We accept the tradeoff between end-user syntax and the somewhat more complicated implementation (scoping).

Many concerns affect language design; tradeoffs are often required.

Specifically, if the language is expected to be extended or embedded, since then the customers' language engineers are also stakeholders.

How to deal with numbers?

Seems to be obvious: why think about it?
We have `int` and `real`. Why bother?



But many domains have to deal with

- **numbers bigger than int or long**
- **fixed precision (e.g., currencies)**
- **numbers with specific ranges**
- **or other constraints on the values.**

This should be part of the language!

Ranges and Constraints

```
// integer type, unlimited range
number                  => number[-inf .. inf]{0}

// integer type, range as specified
number[10..20]          => number[10 .. 20]{0}

// decimal type with 2 decimal places, unlimited range
number{2}               => number[-inf .. inf]{2}

// range as specified, precision derived from range decimals
number[3.3 .. 4.5]      => number[3.3 .. 4.5]{1}
```

Modifying Precision

```
type preciseT: number[0 .. 10]{5}
type roundedT: number[0 .. 10]{2}
type wholeT:   number[0 .. 10]{0}

val precisePI: preciseT = 3.14156
val roundedPI: roundedT = precision[round up precisePI to 2]
val wholePI    wholeT   = precision[cut precisePI to 0]

test case Precision {
  assert precisePI equals 3.14156 [number[0..10]{5}]
  assert roundedPI equals 3.15    [number[0..10]{2}]
  assert wholePI   equals 3       [number[0..10]]
}
```

Modifying Limits

```
val high = limit<wholeT>(20)
val mid  = limit<wholeT>(5)
val low  = limit<wholeT>(-1)

test case TestLimit {
  assert high equals 10 <number[0..10]>
  assert mid  equals 5  <number[0..10]>
  assert low  equals 0  <number[0..10]>
}
```

Ranges and Operators

```
42 + 33                    ==> 75      [number[75..75]{0}]
42 + 2 * 3                 ==> 48      [number[48..48]{0}]
aReal + anInt              ==> 75.33   [number[75.33..75.33]]
if aBool then 42 else 33 ==> 42        [number[33..42]{0}]

type tt: number[-10|10]
val n3, n4: tt              = 0
val n34: number[-100|100]   = n3 * n4
```

Ranges are checked statically by the TS.
Other constraints are checked at runtime.
(Later maybe with a solver).

```
record Bounds {
  min: number
  max: number
} where [min <= max]
```

Even „obvious“ issues may be worth revisiting for DSLs.

„Normal people“ don‘t really care about integer and real, they just want to work with numbers.

Science

High-Performance Computing

Optimized Math on Specialized H/W

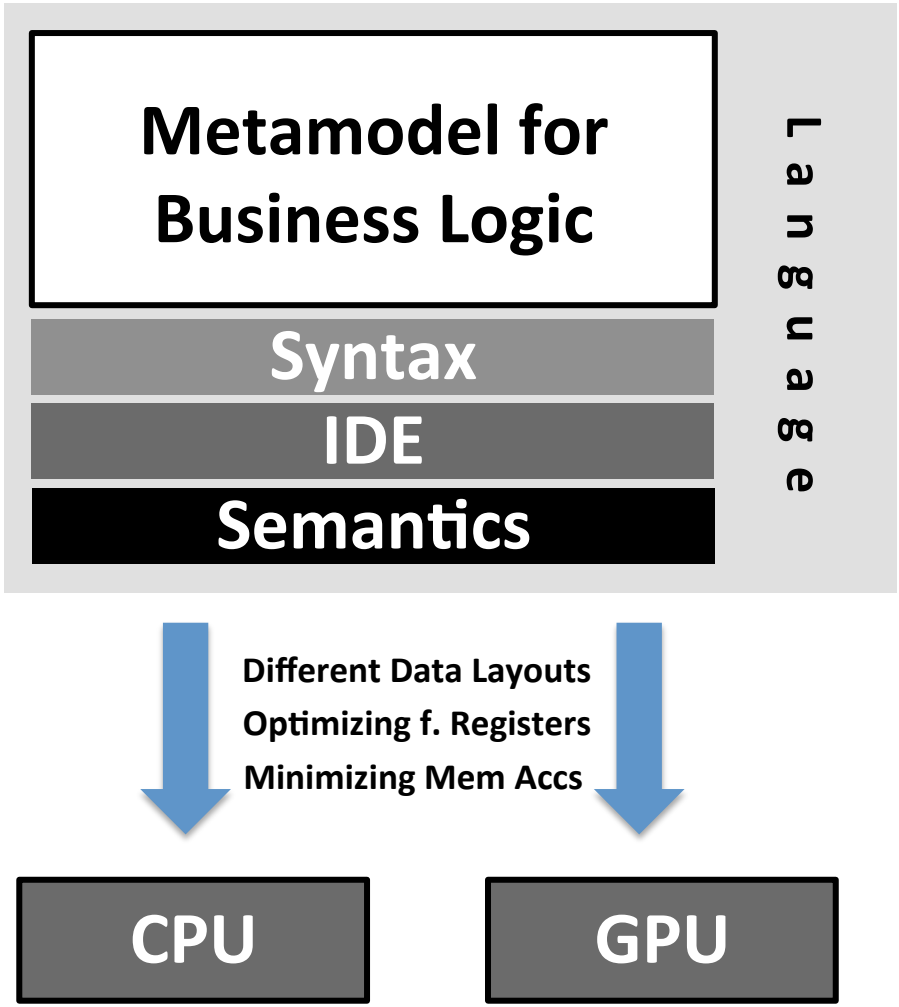
Parallel Computing

High-Throughput

Math computation with sliding windows

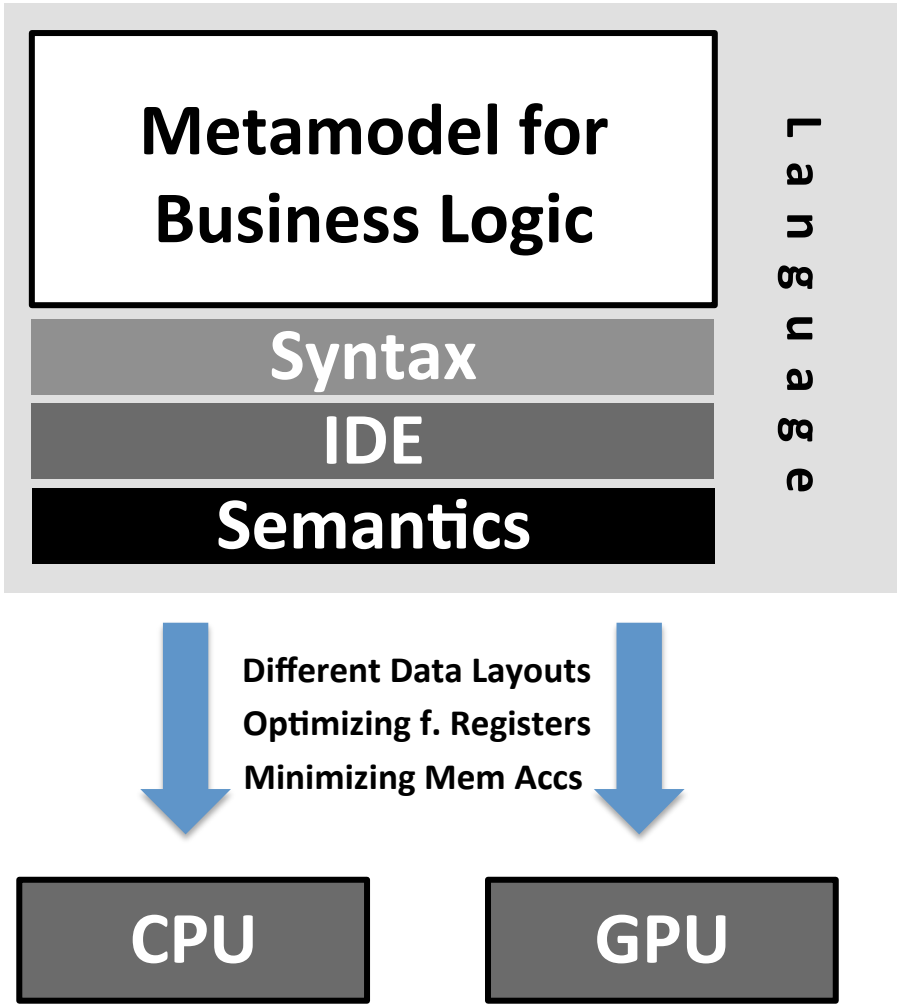
```
// Set up the input pollution
norm_b = 0.0;

array operation with dim ([0,0]x[0,N + 1]x[0,N + 1]) and linear access {
  u = exp(-1.0 * SCALE * ((0.5 - ((double) dim[1]) * h) * (0.5 - ((double) dim[1]) * h) +
      (0.5 - ((double) dim[2]) * h) * (0.5 - ((double) dim[2]) * h)))
  norm_b = norm_b + u * u
}
norm_b = sqrt(norm_b);

// Solve using Jacobi iterator
for (iter = 0;iter <= MAXITER; iter++ ) {
  if (iter % PRINTFREQ == 0 && false) {
    // Compute a residual to see our progress
    norm_r = 0.0;
    array operation with dim ([1,N]x[1,N]x[1,N]) and linear access {
      val temp = 6 * u - S^{1,0,0}(u) - S^{-1,0,0}(u) - S^{0,1,0}(u) - S^{0,-1,0}(u) - S^{0,0,1}(u) - S^{0,0,-1}(u)
      norm_r = norm_r + temp * temp
    }
    norm_r = sqrt(norm_r) / norm_b;
    printf("Iteration %d : residual %lf\n", iter, norm_r)
  } if

  // Compute new grid points
  array operation with dim ([1,N]x[1,N]x[1,N]) and linear
    S^{-1,-1,-1}(u_new) = one_sixth * (S^{1,0,0}(u) + S^{-1,0,0}(u) +
  }

  // Update grid
  array operation with dim ([1,N]x[1,N]x[1,N]) and linear
    u = S^{-1,-1,-1}(u_new)
  }
} for
```

```
vector<int32, 1x2> v2 = [1 2]^T;

vector<int32, 5x1> v3 = [1 2 3 4 5]^T;

matrix<int32, 5x1> m3 = [-2  0  0  1  0
                          0  0  1  0  0
                          0  3  0  0  0
                          0  0 -2  1  1
                          0  0  1  0  2] * v3;
```



Optimizing Generators.

“After making an algorithm efficient (for efficient execution), we cannot recognize the science or math anymore. We cannot maintain the code.”



Abstractions

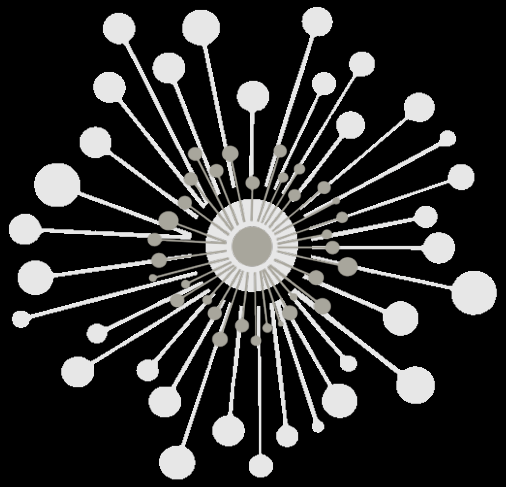
that contain enough semantics to enable generators!

Notations

that make „the power“ accessible to scientitsts.

How
Domain
Requirements
Shape
Languages

4

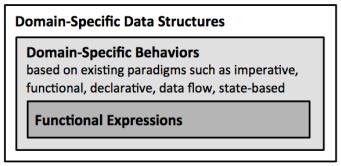


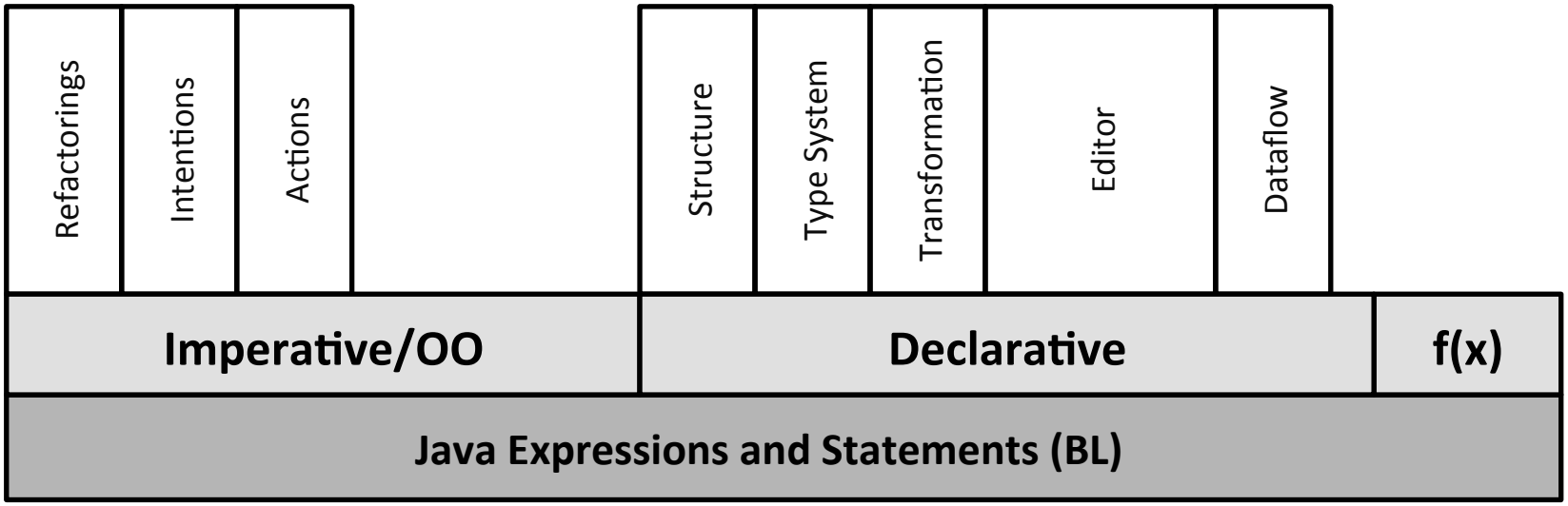
Meta

MPS is bootstrapped,

i.e. it is mostly built with itself.

The MPS team and us are eating our own dogfood.

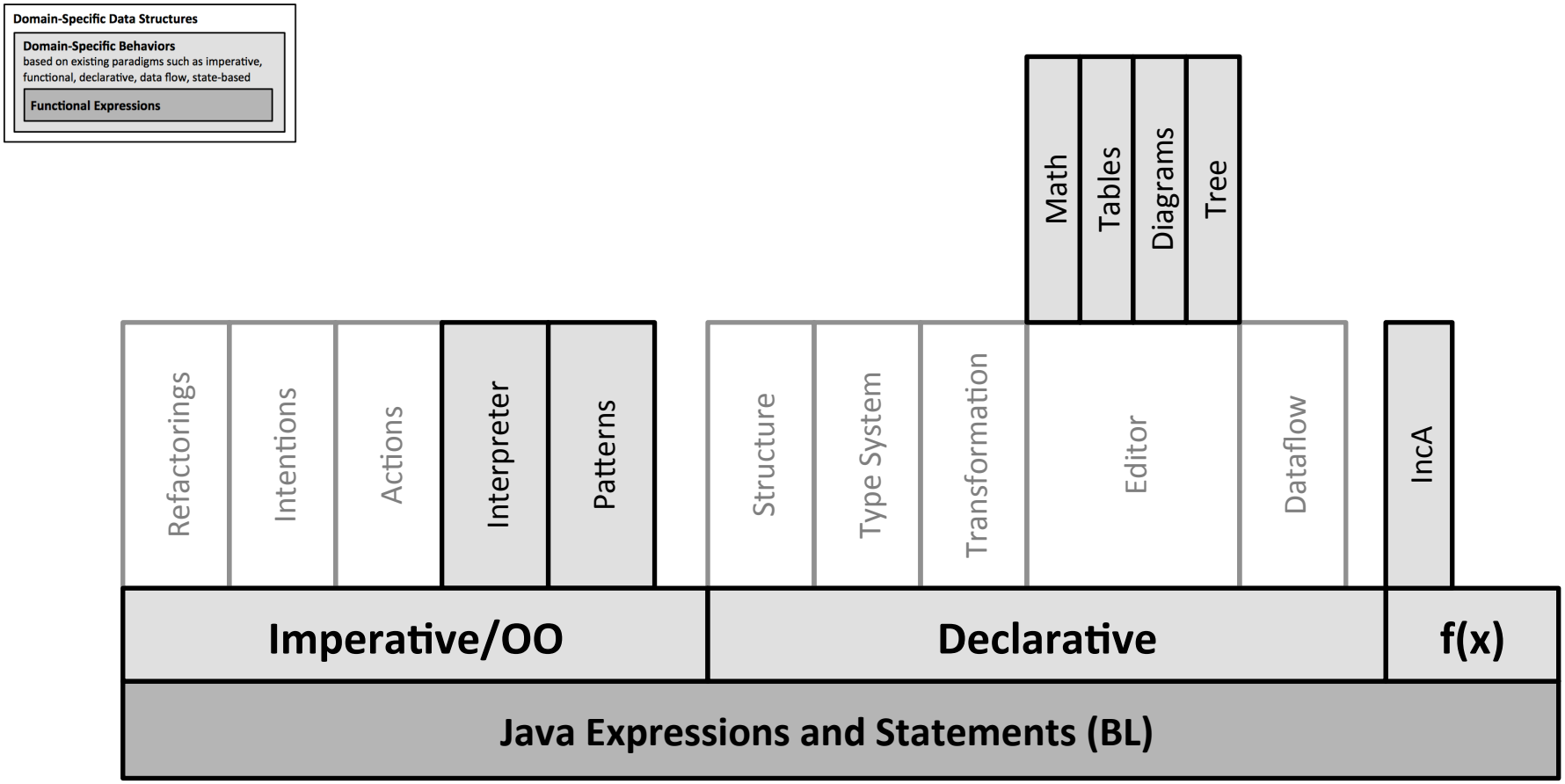




MPS is bootstrapped,

i.e. it is mostly built with itself.

The MPS team and us are eating our own dogfood.



Interpreter

```
!                    [expr <= $boolean]                    = !#expr
&&                   [left <= $boolean, right <= $boolean] = #left && #right
||                   [left <= $boolean, right <= $boolean] = #left || #right
<=>                  [left <= $boolean, right <= $boolean] {
                                                             boolean l = #left;
                                                             boolean r = #right;
                                                             if (l && r) { return true; }
                                                             if (!l && !r) { return true; }
                                                             return false;
                                                           }

IFunctionCall        []                                    = #->function(#args => node.function.args)
IFunctionLike        []                                    = node.body
ArgRef               []                                    = env[node.arg]
```

Declarative type mappings
Type Guards
Concise syntax, especially for recursive calls
Composable (just as languages)
Abstractions for environments and stacks

Program Analysis (IncA)

```
def CI_CONFUSED_INHERITANCE(class : ClassConcept) = {
  assert class.isFinal == true
  member := class.member
  assert member instanceOf FieldDeclaration
  assert member.visibility instanceOf ProtectedVisibility
}

def EQ_ABSTRACT_SELF(class : ClassConcept) = {
  method := class.member
  assert method instanceOf InstanceMethodDeclaration

  // check method name and return type
  name := method.name
  assert method.returnType instanceOf BooleanType
  assert eval(name.equals("equals"))

  // assert that the method has only one parameter
  assert count getParameters(method) == 1

  // and check that the parameter's type is the same as the class
  parameter := getParameters(method)
  parameterType := parameter.type
  assert parameterType instanceOf ClassifierType
  assert class == parameterType.classifier
}
```

Restricted, Declarative Language
Transformed to Patterns
Executed incrementally
Very fast!

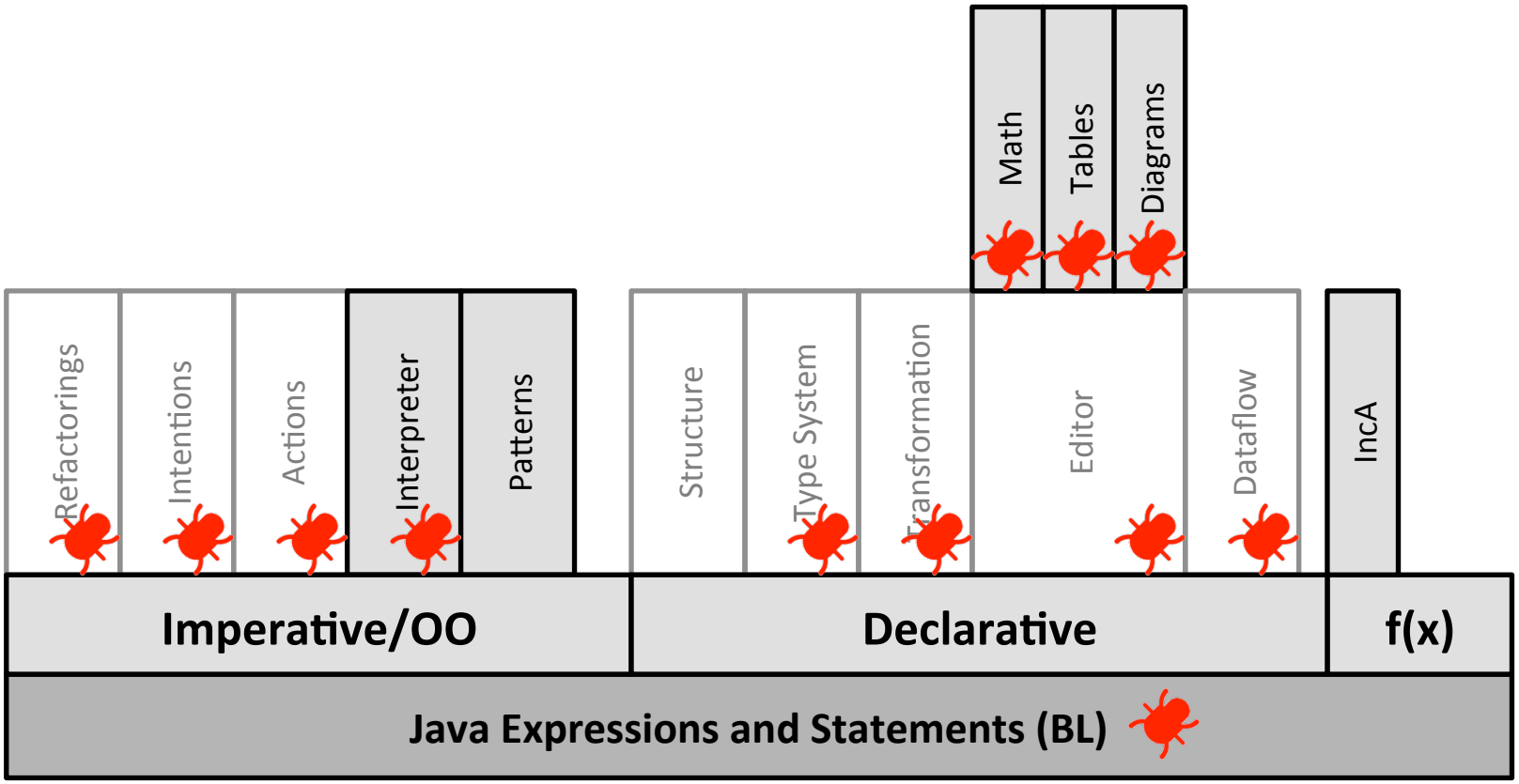
New Editor Notations

```
<default> editor for concept SumExpression
  node cell layout:
    LOOP
    lower:        [> { name } : % varType % = % lower % <]
    upper:        % upper %
    body:         % body %
    symbol:       SumSymbolSerif
    parentheses:  (node)->boolean {
                    Utils.hasFollowingExpression(node.body);
                  }
```

New primitive editor cells for math, diagram, tables
More query-oriented than default MPS notations
Notation independent from Language

Java as an imperative base language is problematic

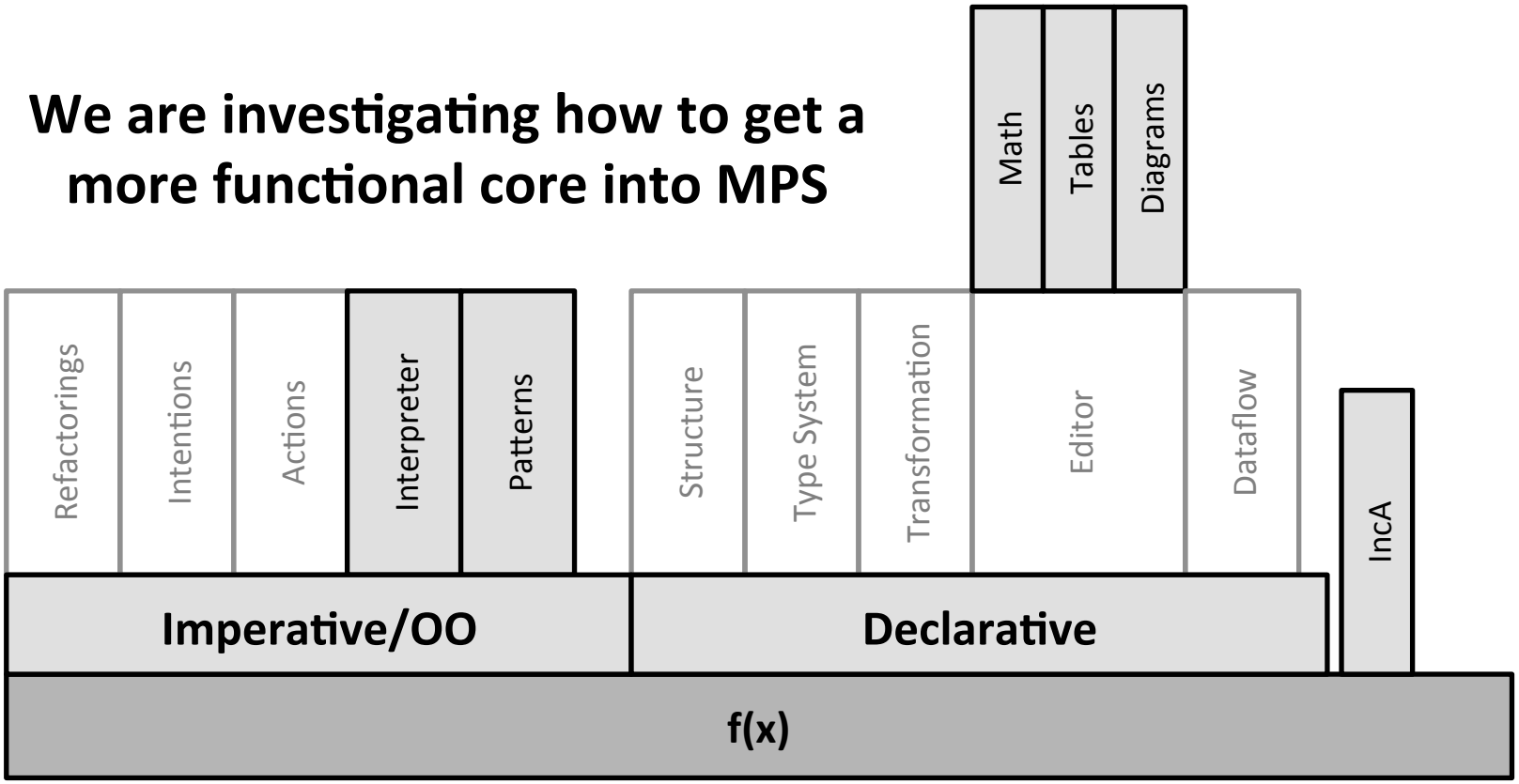
Because of side effects, it is very hard to analyze and optimize



Java as an imperative base language is problematic

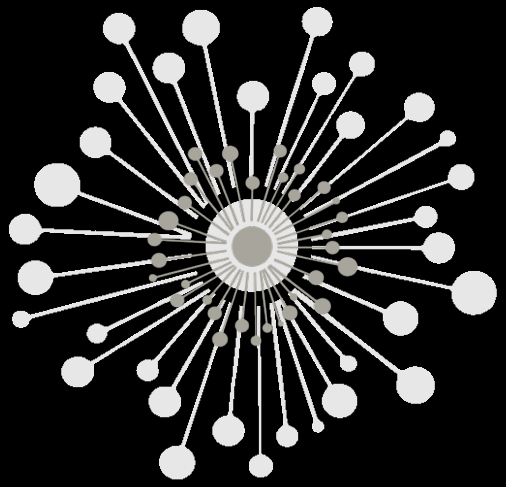
Because of side effects, it is very hard to analyze and optimize

We are investigating how to get a more functional core into MPS



How Domain Requirements Shape Languages

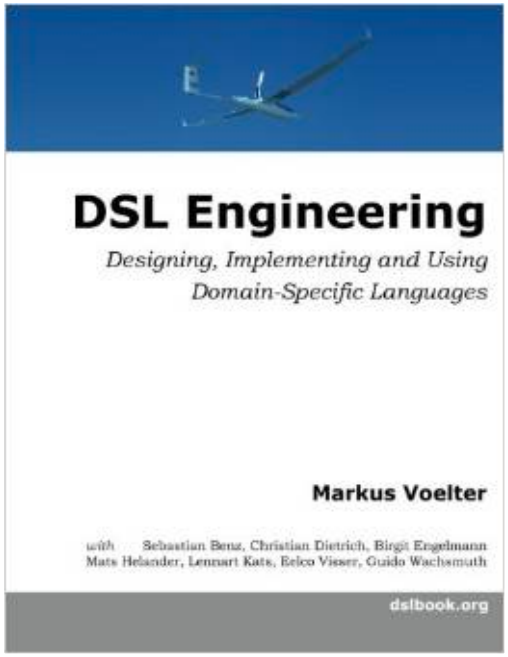
5

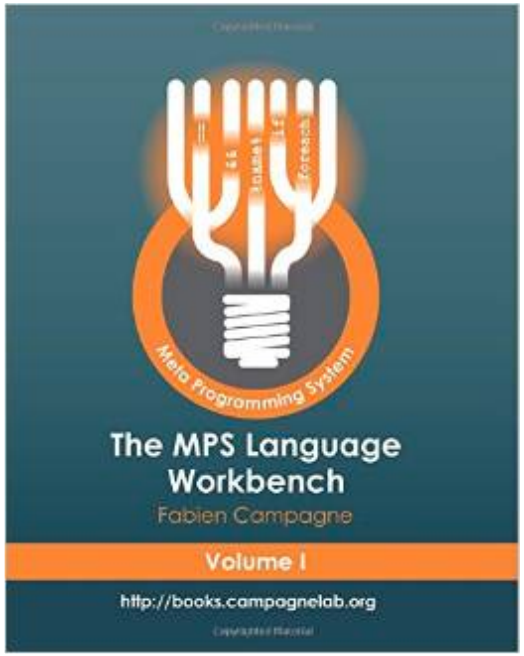


Summing up

All languages shown in this talk

are built with the open source JetBrains MPS language workbench.





mbeddr

MPS Meta Programming System

Lots of further reading is available.

Notation is crucial for users.
Be open to requirements and be creative.

Rely on a solid semantic foundation.
Otherwise you cannot deliver the promises of DSLs.

Simulation, Verification and Testing
the models is a major part of the value-add of a DSL.

Try to incorporate SE best practices
(modularity, SoC) to keep overall complexity in check.

Use a powerful language workbench
to make sure it does not unduly constrain the design.

A personal note.

1999

UML Modeling with Rational Rose and „Skeleton Code“ generation. Classes, Stereotypes, Tagged Values, Simple OCL constraints.

Rational Rose - ShoppingCart.mdl

File Edit View Format Browse Report Query Tools Add-Ins Window Help

ShoppingCart
- Use Case View
  - Main
  - Administrator
  - Client
  - Visitor
  - Create a client account
  - Manage a client account
  - Associations
- Logical View
  - Main
  - Account
  - CartItem
  - Catalog
  - Client
  - NewClass
  - OnlineStore
  - Product
  - ShoppingCart
  - Associations
- Component View
- Deployment View
- Model Properties

Class Diagram: Logical View / Main

ShoppingCart
totalCost : Double
validityPeriod : Integer

Account
login : String
password : String

+cart 0..1 — +account 1

+cart 1 — +item n

CartItem
quantity : Integer

+item n — +product 1

Product
name : String
description : String
price : Double

+product n — +catalog 0..1

Catalog
addProduct()
removeProduct()

+catalog 0..1 — +store 1

OnlineStore

+store 1 — +account n

+account 0..1 — +client n

Client
name : String
email : String
adress : String

Use Case Diagram: Use Case View ...

Client

Administrator

Visitor

Manage a client account

Create a client account

A personal note.

2016

MPS with full code generation.

Mixed notations, multiple integrated languages, type checking and inferencing, data flow checking, refactoring, version control, diff/merge, non-trivial transformations, languages for non-programmers.

„Modeling 2.0“

I cannot understand how developers still deal with modeling 1.0 – and then complain that it does not work.

Try 2.0!