How Domain Requirements shape Languages

Language Design Experience from 10 Years of Building DSLs

Markus Völter
<table>
<thead>
<tr>
<th>DOMAINS</th>
<th>Software Engineering</th>
<th>Language Engineering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Health &amp; Medical</td>
<td>Requirements Engineering</td>
<td>Mixed Notations and End User Programming</td>
</tr>
<tr>
<td></td>
<td>Specification and Verification</td>
<td>Informal $\rightarrow$ Semiformal $\rightarrow$ Formal</td>
</tr>
<tr>
<td>Automotive</td>
<td>Architecture</td>
<td>Languages + Verification</td>
</tr>
<tr>
<td>Aerospace</td>
<td>Safety and Security</td>
<td>Optimizations, Performance and Concurrency</td>
</tr>
<tr>
<td>Robotics</td>
<td>Implementation and Test</td>
<td>Fundamentals: Editors, Type Systems, Trafos</td>
</tr>
<tr>
<td>Finance</td>
<td></td>
<td>User-Friendly IDEs, Tools</td>
</tr>
<tr>
<td>Embedded</td>
<td></td>
<td>Methodology</td>
</tr>
<tr>
<td>Science</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Government</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

Semantics

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

Execution Engine

Tech Infrastructure

- Technical Platform for correct, efficient and scalable execution
Metamodel for Business Logic

Semantics

- generate code, deploy
- transfer data, interpret

Well-defined meaning of this data structure

Technical Platform for correct, efficient and scalable execution

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

generate code, deploy

transfer data, interpret

+ Code Inspection
+ Debugging
+ Performance & Optimization
+ Platform Conformance

+ Turn around Time
+ Runtime Change
Metamodel for Business Logic

Semantics

- generate code, deploy
- transfer data, interpret

Tech Infrastructure
Metamodel for Business Logic

Syntax

Semantics

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!

Metamodel for Business Logic

Syntax

Semantics

Language

Tech Infrastructure

generate code, deploy

transfer data, interpret
Metamodel for Business Logic

Syntax

IDE

Semantics

IDE is critically important for acceptance

generate code, deploy

transfer data, interpret

Tech Infrastructure
Metamodel for Business Logic

Syntax

IDE

Semantics

Language

IDE is critically important for acceptance

Learning
Exploration
Productivity
Tool Integration

generate code, deploy
transfer data, interpret

Tech Infrastructure
Language Workbenches let you efficiently implement languages and IDEs. They are essential for the language implementor.
What is Language Design
Requirements from Domain

Other Requirements

Metamodel for Business Logic

Syntax

IDE

Semantics

Implementation

Language Implementation

Language Design

Iterate!
Language Design

\{ Forces
Constraints
Tradeoffs
Communication
Compromises
Validation \}

Done by people. Supported by Tools.
Language Design::Influences

- Domain Structure
- Non Functionals: Permissions, IP, Sharing
- User Skills
- Model Purpose: Analyze, Generate
- Tool Capabilities: Notations, Editing, Scale
- Software Engineering Practices

Sep. of Concerns
Different Views

Educate, Put results in context

Get a better tool :-)

Refactor towards Structure
Language Design::Influences

Domain Structure

Non Functionals
Permissions, IP, Security

User Skills

Sep. of Concerns
Different Views

Model Purpose
Analyze, Generate

Tool Capabilities
Notations, Editing, Scale

Software Engineering Practices

Style!

Refactor towards Structure

Get a better tool :-)
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

Domain-Specific Data Structures

Domain-Specific Behaviors
based on existing paradigms such as imperative, functional, declarative, data flow, state-based

Functional Expressions

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
- Abstractions
- Notations

Great IDE
- Syntax Coloring
- Code Completion
- Goto Definition

Analyses
- Relevant
- Good Errors

Refactorings
- Aligned with Processes

Testing
- Write Tests
- Run them
- Report Back

GREAT

Debuggers
- Animate Execution
- Simulators

GOOD
# Language Design::Cross Influences

<table>
<thead>
<tr>
<th>Language</th>
<th>Great IDE</th>
<th>Analyses</th>
<th>Refactorings</th>
<th>Testing</th>
<th>Debuggers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Some Syntaxes can be better supported by IDEs</td>
<td>More first-class abstractions make analyses simpler</td>
<td>Can only refactor what can be analyzed</td>
<td>Good test support may limit need for debugging</td>
<td>Some abstractions are easier to debug</td>
</tr>
</tbody>
</table>

- Syntax to express tests
- Some abstractions are easier to debug
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
<table>
<thead>
<tr>
<th>Mixed Notations and End User Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Informal -&gt; Semiformal -&gt; Formal</td>
</tr>
<tr>
<td>Languages + Verification</td>
</tr>
<tr>
<td>Optimizations, Performance and Concurrency</td>
</tr>
<tr>
<td>Fundamentals: Editors, Type Systems, Trafos</td>
</tr>
<tr>
<td>User-Friendly IDEs, Tools</td>
</tr>
<tr>
<td>Simulation, Execution, Test</td>
</tr>
<tr>
<td>Language vs. Library, Code Gen</td>
</tr>
<tr>
<td>Consistent Derived Documents</td>
</tr>
</tbody>
</table>

**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!
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 variability
Write formal code in a DSL mixed with tables and text

No tool support whatsoever

No testing (except inspection)

No reuse

No modularity

No variability
Formale Beschreibung

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

verwendete Attribute:

<table>
<thead>
<tr>
<th>Name</th>
<th>Verw.</th>
<th>Entität</th>
</tr>
</thead>
<tbody>
<tr>
<td>fo_beta_satz</td>
<td>E</td>
<td>Kosten-Regeln</td>
</tr>
<tr>
<td>beta_satz</td>
<td>E</td>
<td>Rechnungsgrundlagen-KK</td>
</tr>
<tr>
<td>ko_ra_id</td>
<td>E</td>
<td>KOSTEN-RABATT</td>
</tr>
<tr>
<td>zmt_param</td>
<td>E</td>
<td>PARAMETER</td>
</tr>
<tr>
<td>kz_zus_gar</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>zm</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>beta_satz_fakt</td>
<td>E</td>
<td>VORGABEDATEN-KOSTENRABATT</td>
</tr>
<tr>
<td>zw</td>
<td>E</td>
<td>TVDKONTO_G</td>
</tr>
<tr>
<td>vtrak_zb</td>
<td>E</td>
<td>VTRK_BTG</td>
</tr>
<tr>
<td>kz_mandant</td>
<td>E</td>
<td>T_KK</td>
</tr>
<tr>
<td>satz_beta</td>
<td>A</td>
<td>Rückgabewert</td>
</tr>
</tbody>
</table>

Aufgerufene Funktionen: rg_kk_beta_bp_satzTF

In dieser Funktion wird der Kostensatz β ermittelt.

Verarbeitung

<table>
<thead>
<tr>
<th>fo_beta_satz</th>
<th>Berechnung satz_beta</th>
<th>Bemerkung</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>satz_beta = beta_satz</td>
<td>Standard</td>
</tr>
<tr>
<td></td>
<td>Falls zmt_param &lt;= 120 und kz_zus_gar = JA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>satz_beta = satz_beta * min(0,01 * max(zmt_param - 12, 0); 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ende (Falls zmt_param &lt;= 120 und kz_zus_gar = JA)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>satz_beta = beta_satz * min(0,01 * max(zm - 12; 0); 1)</td>
<td>PF</td>
</tr>
<tr>
<td>2</td>
<td>grenze = vtrak_zb * zw</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Falls grenze &lt; 10000.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>satz_beta = beta_satz</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sonst satz_beta = 0,074</td>
<td>GULPP</td>
</tr>
</tbody>
</table>

Insurance Programs

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 variability
Insurance Programs

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 variability

„Pixelcrap“

Printed, PDF
Specify/Program

Insurance Programs

„Pixelcrap“

C Code

Write formal code in a DSL mixed with tables and text

Printed, PDF

Developer reads „spec“

Very idiomatic implementation

No tool support whatsoever

No testing (except inspection)

Dev acts as a human compiler and implements it in C

No reuse

No modularity

No variability
Write formal code in a DSL mixed with tables and text

Specify/Program

Insurance Programs

„Pixelcrap“

Implement

C Code

Printed, PDF

Debug

Debugging directly in C
Search-for-use by text search
Don’t trust the documents – may be outdated!

Developer reads „spec“
Very idiomatic implementation

Dev acts as a human compiler and implements it in C

No tool support whatsoever
No testing (except inspection)

No reuse
No modularity
No variability
Write formal code in a DSL mixed with tables and text
Now with IDE support and executable tests

The same notation!
Funktionenmodell berbwvekFF

Formale Beschreibung

Funktion: berbwvekFF
Programmquelle: wmsctfal.c
Produkt-Typ: Ponds PK-Typ: Kapital-Konto
Status: 18.1

Parameter-Attribute

lk_reihe_param
lk_faell_param
ber_zweck_param
kz_rzw_param

Verwendete VADM-Attribute
Keine verwendeten VADM-Attribute, werden automatisch hinzugefügt

Rückgabe-Attribut

bwvek

aufgerufene Funktionen
VTRKeramgtfaellFF (a)
berbwveinzelFF (a; b; c)

Beschreibung
Die Funktion liefert den Barwert per @lk_reihe_param des vorschüssigen Zahlungsströms der Höhe 1 von Monat @lk_reihe_param bis @lk_faell_param jeweils einschließlich. Zahlungszeitpunkte sind jeweils die Monatsbeginne, also @lk_reihe_param = 1# bis #lk_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 Beitragsfälligkeit) und 12 (monatliche Zahlungsweise).

Hilfsvariablen
kz_rf_hilf

Verarbeitungen
Schleife über lk_reihe_hilf = lk_reihe_param bis lk_reiell_param
Falls kz_rzw_param = 12
kz_rf_hilf = 1
sonst
kz_rf_hilf = VTRKeramgtfaellFF (lk_reiell_hilf)
Ende Falls kz_rzw_param = 12
bwvek = bwvek + kz_rf_hilf * berbwveinzelFF (lk_rei_param; lk_reiell_hilf - 1; ber_zweck_param)
Ende Schleife über lk_rei_param bis lk_reiell_param

return bwvek
**Funktionenmodell berbwekFF**

**Formale Beschreibung**

- **Funktion:** berbwekFF
- **Programmquelle:** vmcsfa1.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**

- bvwek

**aufgerufene Funktionen**

- VTRKermbtgfaellFF (a)
- berweinzeFF (a; b; c)

**Beschreibung**
Die Funktion liefert den Barwert per lkm akt_param des vorschüssigen Zahlungsstroms @lkm akt_param bis @lkm faell param - jeweils einschließlich. Zahlungszeitpunkte @lkm akt_param - 1 bis @lkm faell param - 1, der Parameter @kz rzw param steuert die Abzählung der Zahlungsströme. Möglich sind zur Zeit nur die Ausprägungen 0 (z.B. 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 Schleife

bwek = bvwek + kz bf hilf * berweinzeFF (lkm akt_param, lkm faell hilf - 1, ber zweck param)

Ende Schleife

return bvwek
Formale Beschreibung

Funktion: rg_kk_beta_satzTF
Programmquelle: vns CFO2.c
Produkt-Typ: Fonds, RSR
PK-Typ: Kapital-Konto
Status: 18.1

Parameter-Attribute
- zmt_param

Verwendete VADM-Attribute
- rg_kk.beta_satz
- rg_kk.beta_satz
- rg_kk.kz_zus_gar
- rg_kk.zm
- rg_kk.wtrk_zb
- rg_kk.zw
- rg_kk.ko_ra_id
- rg_kk.kz_mandant
- rg_kk.beta_satz_fakt

Rückgabe-Attribut
- satz_beta

aufgerufene Funktionen
- Kuznanzahl_MIN (Kommzahl_a; Kommzahl_b)
- Kuznanzahl_MAX (Kommzahl_a; Kommzahl_b)
- rg_kk_beta_bp_satzTF()
- rg_kk_beta_ap_satzTF()

Beschreibung
In dieser Funktion wird der Kostensatz $\beta$ ermittelt.

Hilfsvariablen
- grenze
- fak_beta
- beta_bp_satz_hilf
- beta_ap_satz_hilf

Verarbeitungen

<table>
<thead>
<tr>
<th>rg_kk.beta_satz</th>
<th>Berechnung</th>
<th>satz_beta = rg_kk.beta_satz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Standard</td>
<td>Falls zmt_param &lt;= 120 und rg_kk.kz_zus_gar = JA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>satz_beta = satz_beta + MIN (0, 0,1 + MAX (zmt_param - 12; 0); 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sonst</td>
</tr>
<tr>
<td>1</td>
<td>PF</td>
<td>Falls Grenze &lt;= 10000,0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>satz_beta = rg_kk.beta_satz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sonst</td>
</tr>
<tr>
<td></td>
<td></td>
<td>satz_beta = 0,074</td>
</tr>
<tr>
<td>2</td>
<td>GULPP</td>
<td>Falls Grenze &lt;= 10000,0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>satz_beta = rg_kk.beta_satz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sonst</td>
</tr>
<tr>
<td></td>
<td></td>
<td>satz_beta = 0,074</td>
</tr>
<tr>
<td>3</td>
<td>FV Standard</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Falls zmt_param &gt;= 156 und rg_kk.kz_zus_gar = JA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>satz_beta = 0,05 + MIN (zmt_param / 12; 1) +</td>
</tr>
</tbody>
</table>

The same notation!
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

\[
result = \sqrt{\frac{a^2 + b^2}{2}}
\]

\[
result = \frac{a + b}{2}
\]

return Parameter (t.c.z.v.m.f.demo.sprint2.GeometrischesMittel)
Insurance Programs

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.
Insurance Programs

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

```haskell
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) =
    \[ \sum_{i: \text{int} = 0}^{\text{values.size}} i \times \text{weight} \] / values.size
```
Small-scale syntactic decisions

Tool support via palettes is important for things that „cannot be typed“.
Structure/Guid. + Mixed
Notation Mixed
Views * Clean
IDE/Tool Clean
Learn/Effective Powerful

Text 1
Powerful 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
An extensible set of integrated languages for embedded software engineering.
Composable extensions, Diverse notations
Different kinds of languages, as illustrated by the different distributions of aspect code.
Relying on modular language extension and extension composition.

Embedding

\[ L_{Host} + L_{Adapt} + L_{Emb} = \]

Extension

\[ L_{Base} + L_{Ext} = \]

Extension Composition

\[ L_{Base} + L_{Ext1} + L_{Ext2} = \]
Typical OO patterns to support extensibility
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.
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!
Integrated Verifications

Bounded Model Checking for C using the CBMC tool.

```c
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
}
```
Formal Verification is simplified if the language has the right abstractions.

... because no semantic recovery is required.
AEROSPACE

C-based satellite on-board software
Standardized Architecture
Lots of infrastructure code
Lots of documentation
Further domain-specific extensions to C. Developed by end-user lang engineer.
# constant TEMP_BUFFER_SIZE = 10;

TACQA = Instance of TemperatureAcquisition with mnemonic tail A and the Numeric Id 350
[SSENSOR = SensorA]
TACQB = Instance of TemperatureAcquisition with mnemonic tail B and the Numeric Id 351
[SSENSOR = 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 = convert[sum \forall idx = 0^{TEMP_BUFFER_SIZE - 1} MEASURED[idx]] / TEMP_BUFFER_SIZE -> °C;
    }
  }
}

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

Activity stopAcquisition with Numeric Id 2
... { TCSACQ.setMode(OFF); }

} Component TemperatureAcquisition
Infrastructure Specific in C

Activity: enableTcs with Numeric 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 hysteresis can be configured.

Constraints:
0: TCSCONTR.inMode(OFF) // switching on is possible only if the TCS is off

In-Parameter:

int16/°C/ upperThreshold: constrained // upper switching threshold
int16/°C/ lowerThreshold: constrained // lower switching threshold
component<TemperatureAcquisition> acq: constrained // acquisition 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 : int132/°C/ = FUS150.AVTEMP // initial temperature when starting thermal control ]
}

Activity: disableTcs with Numeric 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
TCSCONTR
thermal control

OFF
thermal control is inactive

ON
thermal control is active

trigger tcsControl
periodically triggered for altering the heater power state according to the measured values
exit
disable all heaters

TC(150,1) enableTcs
TC(150,2) disableTcs

TCSCONTR

OFF
TC(150,1) enableTcs
TC(150,2) disableTcs

ON

PUS128
TelemetryService

enableTcs

TACQ_startAcquisition

DELAY 10 s
TCSCONTR->ON

TM(150,11); queue=10

ProcessTelemetry

TCSACQ->ON
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).

```plaintext
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
}
```
Establish refinement relationships between data items.

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
}
Only then do you start defining types and value constraints.

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]
Then decompose hierarchical system structures.

```java
public functional component DriveTrain {
    produces SpeedFromEngine
    produces EngineStatus
    produces Gear where it < gearsCount
    consumes RoadConditions
    param int gearsCount
    consumes DrivingCommands
}
```
functional component HeadUnit {
  consumes EngineStatus
  consumes Position
  consumes GenericSpeed
  consumes Gear
}

functional component ActuatorBox {
  produces DrivingCommands
}

functional component Car {
  ActuatorBox
  LocationServices
  RoadSensors
  DriveTrain
    gearsCount = 7
  HeadUnit
}

... recursively.
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 accommodate this.
HEALTH/MEDICINE

Software Medical Devices
Accessible to Doctors
Robustness/Correctness Required
To be FDA-certified
Semi-Graphical Expressions

val c2: int = split three
\[
\begin{bmatrix}
< 0 & \Rightarrow & 0 \\
0..3 & \Rightarrow & 42 \\
> 3 & \Rightarrow & 44
\end{bmatrix}
\]

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

region == EUROPE \&\& region.in[USCAN, ASIA]

<table>
<thead>
<tr>
<th>time.range[0..5]</th>
<th>12</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>time.range[7..17]</td>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td>time.range[18..24]</td>
<td>17</td>
<td>20</td>
</tr>
</tbody>
</table>

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

<> default: LOW

[gender == MALE]

[age < 40] LOW
[age > 40] [overweight(weight)]

[gender == FEMALE]

[age > 50 \&\& overweight(weight)] HIGH

MEDIUM
Prose-like call syntax

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

<table>
<thead>
<tr>
<th></th>
<th>last &lt; 100</th>
<th>last &gt;= 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>this.age in [0, 10]</td>
<td>split previous [&lt; 10 =&gt; LOW]</td>
<td>LOW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;= 10 =&gt; MED</td>
</tr>
<tr>
<td>this.age in [11, 18]</td>
<td>LOW</td>
<td>MED</td>
</tr>
<tr>
<td>this.age &gt; 18</td>
<td>LOW</td>
<td>HIGH</td>
</tr>
</tbody>
</table>

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

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 accommodate 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.
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.

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

<table>
<thead>
<tr>
<th>Language</th>
<th>Great IDE</th>
<th>Analyses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>More first-class abstractions make analyses simpler</td>
<td></td>
</tr>
</tbody>
</table>
End Users vs. Semantics

Established Notations
End-user Happyness

- Ask user for X
- otherwise
- reply after 2 minutes

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

- S1
- on Reply [data.isValid]
- S2
- Ask
- after 2 min
- S3

State Machine

Generating „nice“ reports

Start with this ...

... derive this.

Editable. Analyzable.

Read-only. Traceback. Configurable.
Generating „nice“ reports

Value got from foo?

Value got from foo
h = high

Value got from HB?

Value got from HB
h = high

h?

send alert(“Let’s check this fever”)

send info(“Everything seems fine”)

send alert(“Let’s check the heart”)

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

User sophistication over time.
Semantics First Language Design!

Domain-Specific Data Structures

Domain-Specific Behaviors
based on existing paradigms such as imperative, functional, declarative, data flow, state-based

Functional Expressions

Completely Custom
Established paradigms and semantics, but custom language

Robust, reusable, embeddable.

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

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>42 + 33</td>
<td>75</td>
<td>int</td>
</tr>
<tr>
<td>42 + 2 * 3</td>
<td>48</td>
<td>int</td>
</tr>
<tr>
<td>aReal + anInt</td>
<td>75.33</td>
<td>real</td>
</tr>
<tr>
<td>aBool &amp;&amp; true</td>
<td>true</td>
<td>boolean</td>
</tr>
<tr>
<td>&quot;Hello, &quot; + &quot;World&quot;</td>
<td>&quot;Hello, World&quot;</td>
<td>string</td>
</tr>
<tr>
<td>if [ aBool then 42 else 33 ]</td>
<td>42</td>
<td>int</td>
</tr>
</tbody>
</table>

Option Types

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

```scala
if some?(maybe) then maybe else 0 ==> 42
```
What is a **good** way to do option types?

```plaintext
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 ...

```plaintext
fun f(x: option<int>) = with some x => val none 10
```
Problem: what about nesting?

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

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

Solution: two styles for optional names

```kotlin
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
```

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

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

This is ugly!

Better: support comma-separated multiple tests

```kotlin
fun f(x: int, y: int) =
    with some maybe(x) as xval, maybe(y) as yval
    => xval + yval none 0
```
Nesting: one can now write this

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

This is ugly!

Better: support comma-separated multiple tests

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

But: why not just use an if?

Safe
Checkable
Concise
Non-Surprising
Implementable
Why not just use an `if`?

```haskell
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
```

Problem: how to analyze this?

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

Solution: enforce `some` to be topmost expression

```haskell
fun f(x: option<int>) = if some(x) || g(x) then val else 10
```
Use an option as `bool`?

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

**Discarded:** too much magic.

Use the expression in the `then` part?

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

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

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

Solution: allow && but not other operators

```haskell
fun f(x: int, y: option<int>) =
  if some(maybe(x)) as xval && some(y)
  then xval + y else θ
```
Final, adopted solution:

```haskell
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
```

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

```typescript
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

```scala
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

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

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

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

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

```c
// 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 %f\n", iter, norm_r);
  }

  // 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
```
Metamodel for Business Logic

Syntax

IDE

Semantics

Language

Different Data Layouts
Optimizing f. Registers
Minimizing Mem Accs

Optimizing Generators.

CPU

GPU
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 scientists.
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.
Declarative type mappings
Type Guards
Concise syntax, especially for recursive calls
Composable (just as languages)
Abstractions for environments and stacks
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
}
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
Summing up
All languages shown in this talk are built with the open source JetBrains MPS language workbench.
Lots of further reading is available.
NotaFon 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

A personal note.

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!