Moderne Programmierung durch Spracherweiterung und DSLs

Markus Völter

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

- 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
- Optimizations, Performance and Concurrency
- Fundamentals: Editors, Type Systems, Trafos
- User-Friendly IDEs, Tools
- Methodology
Why should you care?

Bottom-Line Up Front
Domain -> Code takes too long. Stakeholders are insufficiently integrated.

Code and Business Logic are mixed. Evolving one without the other is hard; Legacy problem.

DSLs are proven solution to the problem. Abstractions and Notations optimized for domain.

LWB make language dev efficient. Very much different from lex/yacc/antlr days.

All tools are open source software! So you can try it yourself in your own organization.
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 (e.g., 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

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

generate code, deploy
transfer data, interpret

Technical Platform for correct, efficient and scalable execution
Tech Infrastructure

- generate code, deploy
- transfer data, interpret

<table>
<thead>
<tr>
<th>Generation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Code Inspection</td>
<td>+ Turn around Time</td>
</tr>
<tr>
<td>+ Debugging</td>
<td>+ Runtime Change</td>
</tr>
<tr>
<td>+ Performance &amp;</td>
<td></td>
</tr>
<tr>
<td>Optimization</td>
<td></td>
</tr>
<tr>
<td>+ Platform Con-</td>
<td></td>
</tr>
<tr>
<td>formance</td>
<td></td>
</tr>
</tbody>
</table>
Metamodel for Business Logic

Semantics

Tech Infrastructure

generate code, deploy
transfer data, interpret
Metamodel for Business Logic

Syntax

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!
Tech Infrastructure

IDE is critically important for acceptance

Language

Syntax

IDE

Semantics

generate code, deploy

transfer data, interpret
Meta-model for Business Logic

Syntax
IDE
Semantics

IDE is critically important for acceptance

Learning
Exploration
Productivity
Tool Integration

generate code, deploy
transfer data, interpret

Tech Infrastructure
Metamodel for Business Logic

Syntax

IDE

Semantics

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 Workbench

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

Sep. of Concerns
Different Views

Model Purpose
Analyze, Generate

Tool Capabilities
Notations, Editing, Scale

Educate, Put results in context

Get a better tool :-)

Refactor towards Structure

Software Engineering Practices
Language Design::Influences

Domain Structure

Non Functionals
- Permissions, IP, Sharing

User Skills

Sep. of Concerns
- Different Views

Model Purpose
- Analyze, Generate

Tool Capabilities
- Notations, Editing, Scale

Software Engineering Practices

Style!

Refactor towards Structure

Create, in context

Get a better tool :-)

Educate, Put results in context
Good Language Design?

Fitness for purpose
Evaluation/Acceptance/Productivity

Balancing the Forces
Discussion & 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

- Great IDE
  - Syntax Coloring
  - Code Completion
  - Goto Definition

- Analyses
  - Relevant Errors

- Refactorings
  - Aligned with Processes

- Testing
  - Write Tests
  - Run them
  - Report Back

- GREAT
  - Debuggers
  - Animate Execution
  - Simulators

- GOOD
  - Language Design
  - More than Lang
Some Syntaxes can be better supported by IDEs

More first-class abstractions make analyses simpler

Can only refactor what can be analyzed

Good test support may limit need for debugging

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
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:** vmsco2.c  
**Produkt-Typ:** FONDS, RSR  
**PK-Typ:** Kapital-Konto

<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>vtrk_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 = vtrk_zb * zw</td>
<td>GULPP</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</td>
<td></td>
</tr>
<tr>
<td></td>
<td>satz_beta = 0,074</td>
<td></td>
</tr>
</tbody>
</table>
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

Written, 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
Now with IDE support and executable tests

The same notation!
Funktionenmodell berbwvekFF

Formale Beschreibung

Funktion: berbwvekFF
Programmquelle: vmsctfal.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
   VTRKernabtfgaellFF(a)
   berbwveinzellFF(a; b; c)

Beschreibung
Die Funktion liefert den Barwert per @lkm_akt_param des vorschüssigen Zahlungstroms 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 Zahlungstroms. 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 = VTRKernabtfgaellFF(lkm_faell_hilf)
      Ende Falls kz_rzw_param = 12
      bwvek = bwvek + kz_bf_hilf * berbwveinzellFF(lkm_akt_param; lkm_faell_hilf - 1; ber_zweck_param)
      Ende Schleife über lkm_akt_param bis lkm_faell_param
   return bwvek
Funktionenmodell berbwekeFF

Formale Beschreibung

Funktion: berbwekeFF  
Programmquelle: vmscfaf1.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

Rückgabet-Attribut
bwvek

aufgerufene Funktionen
VTRKermbtgaellFF (a)
BerweinzelfFF (a; b; c)

Beschreibung
Die Funktion liefert den Barwert per lkm_akt_param des vorschüssigen Zahlungssatzes. 
lkm_akt_param ist der vorschüssige Zahlungssatz. Die Zeitpunkte liegen in der Zeitperiode 
@lkm_akt_param bis @lkm_faell_param = jeweils einschließlich. Zahlungszeitpunkte sind je-weils die Monatsbeginne, also lkm_akt_param - 1 bis lkm_faell_param - 1.

Hilfsvariablen
kz_bf_hilf

Verarbeitungen
Schleife über lkm_faell_hilf = lkm_akt_param bis lkm_faell_param

<table>
<thead>
<tr>
<th>Falls kz_rzw_param = 12</th>
<th>kz_bf_hilf = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sonst</td>
<td>kz_bf_hilf = VTRKermbtgaellFF (lkm_faell_hilf)</td>
</tr>
</tbody>
</table>

Ende Falls kz_rzw_param = 12
bwvek = bwvek + kz_bf_hilf * berweinzelfFF (lkm_akt_param; lkm_faell_hilf - 1; ber_zweck_param)  

Ende Schleife über lkm_akt_param bis lkm_faell_param

return bwvek
Formale Beschreibung

Funktion: rg_kk_beta_satzTF  
Programmquelle: vmscf02.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.zm          | 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

- Konnzahl MDN (Konnzahl a; Konnzahl b)
- Konnzahl MAX (Konnzahl a; Konnzahl b)

rg_kk.beta_dp_satzTF ()
rg_kk.beta_ap_satzTF ()

Beschreibung

In dieser Funktion wird der Kostensatz β ermittelt.

Hilfsvariablen

| grenze   |
| fak_beta |
| beta_dp_satz_hilf |
| beta_ap_satz_hilf |

Verarbeitungen

<table>
<thead>
<tr>
<th>_rg_kk.fo_beta_satz</th>
<th>Berechnung @satz_beta</th>
<th>Bemerkung</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>satz_beta = rg_kk.beta_satz</td>
<td>Standard</td>
</tr>
<tr>
<td></td>
<td>Falls zmt_param &lt;= 120 und rg_kk.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>sonst</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Verarbeitung hinzufügen]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ende Falls zmt_param &lt;= 120 und rg_kk.kz_zus_gar = JA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>satz_beta = rg_kk.beta_satz + MIN (0,01 * MAX (rg_kk.zm - 12; 0); 1)</td>
<td>PF</td>
</tr>
<tr>
<td>2</td>
<td>grenze = rg_kk.vtrk_ab + rg_kk.zw</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Falls grenze &lt; 10000,0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>satz_beta = rg_kk.beta_satz</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sonst</td>
<td></td>
</tr>
<tr>
<td></td>
<td>satz_beta = 0,074</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>satz_beta = rg_kk.beta_satz</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Falls zmt_param &lt;= 156 und rg_kk.kz_zus_gar = JA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fak_beta = 0,05 + MIN (zmt_param / 12; 1)</td>
<td></td>
</tr>
</tbody>
</table>

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!
Funktionenmodell GeometrischesMittel

Formale Beschreibung

Funktion: GeometrischesMittel
Programmlinie: Programmlinie 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

Rückgabeargument
result

aufgerufene Funktionen
Keine aufgerufenen Funktionen, werden automatisch hinzugefügt

Beschreibung
Berechnet das geometrische Mittel der Parameter

Hilfsvariable
Hilfsvariable hinzufügen

Error: Quadratwurzel ist nur für positive Zahlen erlaubt
Write formal code in a DSL mixed with tables and text
Now with IDE support and executable tests
Exactly the same C code.
The same notation!
Incremental Refinement/Refactoring of languages:

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

Still exactly the same C code, or improved as needed.
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) =
    \[
    \sum_{i=0}^{values.size} i \cdot \text{weight} \over \text{values.size}
    \]
```
Small-scale syntactic decisions

Tool support via palettes is important for things that „cannot be typed“.
Structure/Guid.  +  -
Notation  Mixed  Text
Views  *
IDE/Tool  Clean  1
Learn/Effective  L  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.

<table>
<thead>
<tr>
<th>User Extensions</th>
<th>User-defined Layer</th>
<th>Languages shipped with mbeddr</th>
<th>Platform</th>
<th>MPS</th>
<th>Foundation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing</td>
<td>Visualizations</td>
<td>Libraries for web server, node navigation, additional notations, pattern matching, palettes, XML processing, debugging...</td>
<td>Syntax Highlighting, Code Completion, Goto Definition, Find Usages, Type Checking, Data Flow Analysis, Refactoring, Versioning, Debugging</td>
<td>C Compiler &amp; Debugger</td>
<td>PlantUML, Latex, HTML</td>
</tr>
<tr>
<td>Logging</td>
<td>PLE Variability</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Utilities</td>
<td>Requirements &amp; Tracing</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Messaging</td>
<td>Documentation</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Components</td>
<td>Reports &amp; Assessments</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Physical Units</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>State Machines</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Concurrency</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Importer</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C99</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
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  
Delft University of Technology, The Netherlands  
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
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 reverse engineering is required.
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.

<table>
<thead>
<tr>
<th>User Extensions</th>
<th>User-defined Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Languages shipped with mbeddr</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td>Logging</td>
</tr>
<tr>
<td>C99</td>
<td></td>
</tr>
<tr>
<td>Platform</td>
<td>Libraries for web server, node navigation, additional notations, pattern matching, palettes, XML processing, debugging...</td>
</tr>
<tr>
<td>MPS</td>
<td>Syntax Highlighting, Code Completion, Goto Definition, Find Usages, Type Checking, Data Flow Analysis, Refactoring, Versioning, Debugging</td>
</tr>
<tr>
<td>Foundation</td>
<td>C Compiler &amp; Debugger</td>
</tr>
<tr>
<td>Implementation</td>
<td>Process</td>
</tr>
</tbody>
</table>
Infrastructure Specifics in C

```c
#define 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
        \[ \text{AVTEMP} = \text{convert}\left(\sum_{\text{idx} = 0}^{\text{TEMP_BUFFER_SIZE} - 1} \frac{\text{MEASURED[idx]}}{} / \text{TEMP_BUFFER_SIZE} \rightarrow ^\circ\text{C} \right) \]
      }
    }
  }

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

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

Component TemperatureAcquisition
Infrastructure Specifics 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`: <no constraint> // upper switching threshold
- `int16/°C/ lowerThreshold: constrained`: lowerThreshold < upperThreshold // lower switching threshold

- `component<TemperatureAcquisition> acq: constrained`: <no constraint> // 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 : int32/°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

ON

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

PUS128
TelemetryService

enableTcs

TACQ_startAcquisition

DELAY 10 s

TCSCONTR->ON

TM(150,1); 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
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.

```haskell
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
}
```
components Step11Car

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
}
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

„Program“ accessible to Doctors

Robustness/Correctness Required

To be FDA-certified
Semi-Graphical Expressions

val c2: int = split three
<table>
<thead>
<tr>
<th></th>
<th>=&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>0</td>
</tr>
<tr>
<td>0..3</td>
<td>42</td>
</tr>
<tr>
<td>&gt; 3</td>
<td>44</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>region == EUROPE</th>
<th>region.in[USCAN, ASIA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>time.range[0..5]</td>
<td>12</td>
</tr>
<tr>
<td>time.range[7..17]</td>
<td>20</td>
</tr>
<tr>
<td>time.range[18..24]</td>
<td>17</td>
</tr>
</tbody>
</table>

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

\[\text{default: LOW}\]

\([\text{gender == MALE}]\):

\([\text{age < 40}]\):

LOW

\([\text{age > 40}]\):

[overweight(weight)]

MEDIUM

\([\text{gender == FEMALE}]\):

\([\text{age > 50} \&\& \text{overweight(weight)}]\):

HIGH
Prose-like call syntax

```kotlin
ext fun calculateRisk(this: Person, last: int, previous: int) =
    last < 100
    split previous
    < 10 => LOW
    >= 10 => MED
    LOW
    MED
    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

```kotlin
@syntax{stroke risk for last @[last] and but-last @[previous] blood sugar}

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</td>
<td></td>
</tr>
<tr>
<td></td>
<td>previous</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; 10   =&gt;</td>
<td>LOW</td>
</tr>
<tr>
<td></td>
<td>&gt;= 10 =&gt;  MED</td>
<td></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.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.
Prose-like business rules

Person ilse is parent of harald.
Person harald is parent of markus.
Person harald is parent of peter.

Person markus works at itemis.
Person harald works at siemens.
Person bernd works at itemis.

Person Ancestor is ancestor if Descendant if
   [Person Ancestor is parent of Descendant].
Person Ancestor is ancestor if Descendant if
   (with variables I)
   [Person Ancestor is parent of I]
   [Person I is ancestor if Descendant].

Persons P1 and P2 are colleagues at company C if
   [Person P1 works at C]
   [Person P2 works at C].

Prolog-style, deductive execution semantics,
but human-readable prose-sentence notation.
Prose-like business rules II

```
val db = datalog<SimpleDatalog>

val worksAtItemis = db.query[P|worksAt(P, itemis)].P.distinct

test case TestcaseDatalog [incomplete] {
  assert db.allAncestors.V.distinct

  assert worksAtItemis

  assert db.query[P, Q|worksAt(P, Q)].asTuples

  assert {
    val work = db.query[P, Q|worksAt(P, Q)]
    val haraldSWork = work.asTuples.where(|it[0] == harald|)
    haraldSWork.map(|it[1]|)
  }
}
```

Direct integration with functional language for testing purposes. Live execution of test cases.
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.
Established approaches for developing reliable and safe systems applied to languages.
Established approaches for developing reliable and safe systems applied to languages.
Established approaches for developing reliable and safe systems applied to languages.
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.
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 Happiness

  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
  
  /entry
  
  Ask
  
  on Reply [data.isValid]
  
  after 2 min

  S2

  S3
  
  State Machine

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

```scala
case class Expr(value: Int)
case class Op(left: Expr, right: Expr, operation: String)

val expr = Op(Expr(42), Op(Expr(33), Op(Expr(2), Expr(3), "*")), "+")
val result = expr.
```

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

```javascript
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

```javascript
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
```

But: why not just use an `if`?
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
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 language 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 0
```

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 0
```
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
```

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

```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).

```
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
```
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
Performance counts!
Math computation with sliding windows

// Set up the input pollution
norm_b = 0.0;

array operation with dim ([0,0][0,N + 1][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][1,N][1,N]) and linear access {
            val temp = 6 * u - \begin{bmatrix} 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) \end{bmatrix}
            norm_r = norm_r + temp * temp
        }
        norm_r = sqrt(norm_r) / norm_b;
        printf("Iteration %d : residual %lf\n", iter, norm_r);
    }

    // Compute new grid points
    array operation with dim ([1,N][1,N][1,N]) and linear access {
        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][1,N][1,N]) and linear access {
    u = S^{-1,-1,-1}(u_{new})
}
}
Metamodel for Business Logic

Language

Syntax
IDE
Semantics

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.
Metamodel for Business Logic

Language

Syntax
IDE
Semantics

Abstractions that contain enough semantics to enable generators!

Notations that make „the power“ accessible to scientists.

Different Data Layouts
Optimizing f. Registers
Minimizing Mem Accs

CPU
GPU
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 Editor Notations

New primitive editor cells for math, diagram, tables
More query-oriented than default MPS notations
Notation independent from Language
unit concept Class extends IProgramContent {

    concept ThisExpression extends Expr {
    }

abstract concept IClassContent {
    property boolean public
}

concept Attribute extends IClassContent, ITyped {
    sig name ident
    identity name
    child opt<Expr> init
}

reference concept AttrRef extends Expr -> Attribute [local]

unit concept Method extends IFunctionLike, IClassContent {
}

concept Constructor extends Method {
    virtual concept ConstructorArg extends Arg
        where it.isUnder<Constructor>
}

virtual concept QueryMethod extends Method
    where it.args.isEmpty
virtual concept IPublicContent extends IClassContent
    where it.public

sig[all<IPublicContent>] child list<IClassContent> contents unique
ref opt<Class> parent [unit]
}
Navigating ASTs

test case TestNavigation [fail] {
    assert program.descendants<PrintStatement>.size equals 1
    assert program.descendants<Statement>.size equals 5
    assert program.descendants<Any>.size equals 5
    assert program.descendants<BlockStatement>.size equals 2
    assert thePrint.ancestors++.size equals 3
    assert thePrint.ancestors<BlockStatement>.size equals 2
    assert thePrint.ancestors<Any>.size equals 3
    assert program.descendants++.ofConcept<PrintStatement>.size equals 1
    assert program.descendants++.ofConcept<BlockStatement>.size equals 2
    assert program.descendants++.ofConcept<Statement>.size equals 5
    assert program.descendants++.ofConcept<Any>.size equals 5
    assert if some?(thePrint.ancestor++<PrintStatement>) then true else false equals true
    assert if some?(thePrint.ancestor<PrintStatement>) then true else false equals false
    assert program.descendants++.size equals 5
    assert program.descendants++.size equals 6
    assert program.descendants++.Program.size equals 1
    assert program.descendants<Program>.size equals 0
    assert thePrint.ancestors<PrintStatement>.size equals 0
    assert thePrint.ancestors++.PrintStatement.size equals 1
}
fun aLoop(core: Statement) = AST

  Program dummy {
    statements Label begin [name:/begin]
    statements = core.copy
    statements GotoWrapperStatement {
      goto Goto { label -> begin }
    }
  }

fun loop2() = AST

  Program dummy {
    statements Label begin [name:/begin]
    statements PrintStatement [text = "Hello"]
    statements GotoWrapperStatement {
      goto Goto { label = node.container<Program>.statements.first.as<Label> }
    }
  }
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.
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

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!