Lessons Learned about Language Engineering from the Development of mbeddr
First:
It’s not just me!
**DOMAINS**

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

**Software Engineering**

- Requirements Engineering
- Specification and Verification
- Architecture
- Safety and Security
- Implementation and Test

**Language Engineering**

- Mixed Notations and End User Programming
- Informal -> Semiformal -> Formal
- Languages + Verification
- Optimizations, Performance and Concurrency
- Fundamentals: Editors, Type Systems, Trafos
- User-Friendly IDEs, Tools
- Methodology
Background: MPS
Language Workbench
(Martin Fowler, 2004)

Freely define languages and integrate them.
Language Workbench
(Martin Fowler, 2004)

powerful editing testing refactoring debugging groupware

language definition implies IDE definition
Language Workbench
(Martin Fowler, 2004)

support for "classical" programming and "classical" modeling

There's no difference!
A Language Workbench – a tool for defining, composing and using ecosystems of languages.
[Language Workbench]

Comprehensive Support for many aspects of Language Definition.

+ Refactorings, Find Usages, Syntax Coloring, Debugging, ...

Diagram:

- Structure: Concepts, Properties, Inheritance, Relationships
  - Provides editors for
  - Structure provides:
    - Editor: Projection Rules, Side Transformations, Intentions
    - Type System: Typing Rules, Type Checks, Other Validations
      - Defines execution semantics for
        - Transforms: Reduction Rules, Weaving Rules, Transformation Priorities
          - Specifies priority 0..*

Language extends 0..* Generates to Language
[Projectional Editing]

Parsing

1. Concrete Syntax
2. Abstract Syntax Tree

Projectional Editing

Concrete Syntax

Abstract Syntax Tree
Projectional Editing
Syntactic Flexibility

Regular Code/Text

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Mathematical

\[ \sum \]

Tables

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Graphical
**Projectional Editing**

**Syntactic Flexibility**

### Regular Code/Text

```c
// A documentation comment with references
// to @arg(data) and @arg(dataLen)
void aSummingFunction(int8[] data, int8 dataLen) {
    int16 sum;
    for (int8 i = 0; i < dataLen; i++) {
        sum += data[i];
    }
}
}
```

### Mathematical

```c
double midnight2(int32 a, int32 b, int32 c) {
    return \(-b + \sqrt{b^2 - \sum_{i=1}^{4} a_i c_i}\); 
}
```

### Tables

```c
int16 decide(int8 spd, int8 alt) {
    return spd > 0 spd > 100 otherwise 0;
    
    | alt < 0 | 1 | 1 |
    | alt == 0 | 10 | 20 |
    | alt > 0 | 30 | 40 |
    | alt > 100 | 50 | 60 |
}
```

### Graphical

```
Cst.Customer
  --------
|        |
| cust 1  |
|         |
```
```
Contract
  -------
| starts: date |
| ends: date   |
```
```
Tariff
  -------
| attributes |
| trf 1      |
```
```
[Projectional Editing] Language Composition

Separate Files

Type System
Transformation
Constraints

In One File

Type System
Transformation
Constraints
Syntax
IDE
[Projectional Editing]
Language Composition

Embedding

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

Extension

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

Extension Composition

\[ L_{\text{Base}} + L_{\text{Ext1}} + L_{\text{Ext2}} = \]
Lessons Learned about Language Engineering from the Development of mbeddr

Background: mbeddr
An extensible set of integrated languages for embedded software engineering.

<table>
<thead>
<tr>
<th>User Extensions</th>
<th>C99</th>
</tr>
</thead>
<tbody>
<tr>
<td>Languages shipped with mbeddr</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td>Logging</td>
</tr>
<tr>
<td>Visualizations</td>
<td>PLE Variability</td>
</tr>
<tr>
<td>State Machine Verification</td>
<td>Component Contracts Verification</td>
</tr>
<tr>
<td>Libraries for web server, node navigation, additional notations, pattern matching, palettes, XML processing, debugging...</td>
<td></td>
</tr>
<tr>
<td>Syntax Highlighting, Code Completion, Goto Definition, Find Usages, Type Checking, Data Flow Analysis, Refactoring, Versioning, Debugging</td>
<td></td>
</tr>
<tr>
<td>C Compiler &amp; Debugger</td>
<td>PlantUML</td>
</tr>
<tr>
<td>Implementation</td>
<td>Process</td>
</tr>
</tbody>
</table>
Composable extensions, Diverse notations
An Industrial Case Study

Smart Meter

Measures Voltage and Current
Computes Derived Values
Shows Data on LCD Display
Communicates through Networks

Precision is critical for Certification.
Evolvability is critical for it to be a viable business.

Developed with `embedddr`, a set of domain-Specific extensions to C, plus an IDE.
Hardware Architecture

<table>
<thead>
<tr>
<th>Application Logic</th>
<th>Metrology</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MSP430 F67791</strong></td>
<td><strong>MSP430 F6736</strong></td>
</tr>
<tr>
<td>25 MHz</td>
<td>25 MHz</td>
</tr>
<tr>
<td>256K Flash ROM</td>
<td>128K Flash ROM</td>
</tr>
<tr>
<td>32K RAM</td>
<td>8K RAM</td>
</tr>
</tbody>
</table>

- RS485
- IrDA
- DLMS/COSEM
- MQTT
- UART
Software Architecture

No RTOS
Interrupt-Driven
One-Threaded Programming

Required Precision leads to 4096 Hz Sampling Rate

Interrupt-Triggered: Measurement

Foreground Tasks:
App Logic, RTC
## Size of the System

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Common</th>
<th>Metro</th>
<th>App</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Files</td>
<td>134</td>
<td>101</td>
<td>105</td>
<td>340</td>
</tr>
<tr>
<td>Total LOC</td>
<td>8,209</td>
<td>10,447</td>
<td>10,908</td>
<td>29,564</td>
</tr>
<tr>
<td>Code LOC</td>
<td>4,397</td>
<td>5,900</td>
<td>5,510</td>
<td>15,807</td>
</tr>
<tr>
<td>Comment LOC</td>
<td>950</td>
<td>2,402</td>
<td>2,620</td>
<td>5,972</td>
</tr>
<tr>
<td>Whitespace LOC</td>
<td>2,852</td>
<td>2,145</td>
<td>2,778</td>
<td>7,775</td>
</tr>
</tbody>
</table>

*Common* code runs on both processors, *Metro* runs on the metrology processor and *App* runs on the application / communication processor.

+ roughly the same amount again for tests.

ca. 30,000 SLOC
## Use of Extensions

<table>
<thead>
<tr>
<th>Category</th>
<th>Concept</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chunks (≈ Files)</td>
<td>Implementation Modules</td>
<td>382</td>
</tr>
<tr>
<td></td>
<td>Other (Req, Units, etc.)</td>
<td>46</td>
</tr>
<tr>
<td>C Constructs</td>
<td>Functions</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td>Structs / Members</td>
<td>144 / 270</td>
</tr>
<tr>
<td></td>
<td>Enums / Literals</td>
<td>150 / 1,211</td>
</tr>
<tr>
<td></td>
<td>Global Variables</td>
<td>334</td>
</tr>
<tr>
<td></td>
<td>Constants</td>
<td>8,500</td>
</tr>
<tr>
<td>Components</td>
<td>Interfaces / Operations</td>
<td>80 / 197</td>
</tr>
<tr>
<td></td>
<td>Atomic Components</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td>Ports / Runnables</td>
<td>630 / 640</td>
</tr>
<tr>
<td></td>
<td>Parameters / Values</td>
<td>84 / 324</td>
</tr>
<tr>
<td></td>
<td>Composite Components</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>Component Config Code</td>
<td>1,222</td>
</tr>
<tr>
<td>State Machines</td>
<td>Machines</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>States/Transitions/Actions</td>
<td>14 / 17 / 23</td>
</tr>
<tr>
<td>Physical Units</td>
<td>Unit Declarations</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td>Conversion Rules</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>Types / Literals with Units</td>
<td>593 / 1,294</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
<th>Concept</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Line Variability</td>
<td>Feature Models / Features</td>
<td>4 / 18</td>
</tr>
<tr>
<td></td>
<td>Configuration Models</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Presence Condition</td>
<td>117</td>
</tr>
<tr>
<td>Custom Extensions</td>
<td>Register Definition</td>
<td>387</td>
</tr>
<tr>
<td></td>
<td>Interrupt Definitions</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Protocol Messages</td>
<td>42</td>
</tr>
<tr>
<td>Statements</td>
<td>Statements total</td>
<td>16,840</td>
</tr>
<tr>
<td></td>
<td>Statements in components</td>
<td>6,812</td>
</tr>
<tr>
<td></td>
<td>Statements in test cases</td>
<td>5,802</td>
</tr>
<tr>
<td></td>
<td>Statements in functions</td>
<td>3,636</td>
</tr>
<tr>
<td>Testing</td>
<td>Test Cases / Suites</td>
<td>107 / 35</td>
</tr>
<tr>
<td></td>
<td>Test-Specific Components</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>Stub / Mock Components</td>
<td>9 / 8</td>
</tr>
<tr>
<td></td>
<td>assert Statements</td>
<td>2,408</td>
</tr>
</tbody>
</table>

All mbeddr C extensions used a lot.
Some extensions built specifically for SM.
A 5-minute MPS tutorial
C Extension to dequeue a value

```c
shared int64Q data;

cyclic task sumUp {
    dequeue if available from data -> val {
        sum += val;
    }
}
```

Generated to:

```c
generated to

uint64 ___val = 0;
boolean ___taken = false;
atomic <data/readWrite> {
    if (!data.isEmpty) {
        ___val = data.take;
        ___taken = true;
    }
}
if (___taken) {
    sum += ___val;
}
```
**Structure**

**concept** ValExpr **extends** Expression

**alias** val

**concept** DequeueStatement **extends** Statement

**implements** IAtomic

**alias:** dequeue

**children:**
- `queue : GlobalVarRef [1]`
- `body : StatementList [1]`
Editor actions to implement editor behavior. Previously manually written, now generated from Grammar Cells. See SLE Paper :-)}
Constraints

Tree Constraints:

```java
can be child
    (childConcept, node, parentNode) -> boolean {
        parentNode.ancestor<DequeueStatement> != null;
    }
```

Scopes:

```java
concept GlobalVarRef extends Expression
    references:
        var : GlobalVarDecl[1]

link var scope:
    (enclosingNode, pos) -> nsequence<GlobalVarDecl> {
        enclosingNode.ancestor<Module>.contents.
        ofConcept<GlobalVarDecl>;
    }
```
Type System

Checking Rules:

```plaintext
checking rule for DequeueStatement {
    ensure node.queue.var.type.isInstanceOf(QueueType)
    else error "global variable not a queue"
    on dqs.queue;
    ensure node.queue.var.@shared != null
    else error "queue must be shared"
    on dqs.queue;
}
```

Typing Rules:

```plaintext
typing rule typeof_QueueValExpr for ValExpr {
    node<DequeueStatement> dqs =
        node.ancestor<DequeueStatement>;
    node<QueueType> qt = dqs.queue.type : QueueType;
    typeof(node) ::= typeof(qt.queue.elementType);
}
```
**Behavior**

```java
concept behavior IAtomic {
    public abstract boolean getsReadLockFor(
        node<GlobalVarRef> r);
    public abstract boolean getsWriteLockFor(
        node<GlobalVarRef> r);
}

public boolean getsReadLockFor(node<GlobalVarRef> r) {
    return r.var == this.queue.var;
}
```

```java
checking rule check_Lock for GlobalVarRef {
    node<IAtomic> a = node.ancestor<IAtomic>;
    if (a == null || !a.providesReadLockFor(node))
        error "global variables must be locked" -> gvr;
}
```
Generators/Transformations

```
node.queue.var.type : QueueType.queue.elementType;

com.mbeddr.ext.concurrency. pthreads: main
  <> com.mbeddr.core.utils: main
```
And many more...

Intentions/Quick Fixes
Refactoring
Text Generators
Language Test Cases
Data Flow Graphs/Analyses
Debuggers
...
...
Modularity Fundamentally OO
4

Lessons Learned about Language Engineering from the Development of mbeddr

Project Stats
Total Effort

10 person years
<table>
<thead>
<tr>
<th>Part</th>
<th>Language</th>
<th>#L</th>
<th>#LC</th>
<th>#S</th>
<th>#C</th>
<th>#LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>core</td>
<td>base</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>98</td>
<td>6,163</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>8</td>
<td>-</td>
<td>7</td>
<td>354</td>
<td>20,114</td>
</tr>
<tr>
<td></td>
<td>unittest</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>26</td>
<td>1,014</td>
</tr>
<tr>
<td></td>
<td>utils</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>116</td>
<td>6,306</td>
</tr>
<tr>
<td></td>
<td>build</td>
<td>2</td>
<td>-</td>
<td>1</td>
<td>48</td>
<td>2,080</td>
</tr>
<tr>
<td>ext</td>
<td>components</td>
<td>9</td>
<td>9</td>
<td>1</td>
<td>160</td>
<td>11,173</td>
</tr>
<tr>
<td></td>
<td>state machines</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>48</td>
<td>3,194</td>
</tr>
<tr>
<td></td>
<td>concurrency</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>65</td>
<td>3,078</td>
</tr>
<tr>
<td></td>
<td>math</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>446</td>
</tr>
<tr>
<td></td>
<td>messaging</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>60</td>
<td>2,151</td>
</tr>
<tr>
<td></td>
<td>units</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>30</td>
<td>1,884</td>
</tr>
<tr>
<td>cc</td>
<td>variability</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>87</td>
<td>3,638</td>
</tr>
<tr>
<td></td>
<td>reqmts/tracing</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>171</td>
<td>5,563</td>
</tr>
<tr>
<td>doc</td>
<td>doc</td>
<td>10</td>
<td>-</td>
<td>1</td>
<td>153</td>
<td>6,355</td>
</tr>
<tr>
<td>analyses</td>
<td>analysis</td>
<td>18</td>
<td>10</td>
<td>18</td>
<td>170</td>
<td>15,235</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>81</td>
<td>34</td>
<td>38</td>
<td>1,597</td>
<td><strong>88,394</strong></td>
</tr>
</tbody>
</table>

ca. 90,000 SLOC
## Tests vs. Implementation

<table>
<thead>
<tr>
<th></th>
<th>#Test Cases</th>
<th>#Assertions</th>
<th>#LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type System Tests</td>
<td>324</td>
<td>1,381</td>
<td>8,243</td>
</tr>
<tr>
<td>Executable Tests</td>
<td>310</td>
<td>990</td>
<td>14,999</td>
</tr>
<tr>
<td>Other Tests</td>
<td>784</td>
<td>1,275</td>
<td>22,453</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1,418</strong></td>
<td><strong>3,646</strong></td>
<td><strong>45,695</strong></td>
</tr>
</tbody>
</table>
Timeline of the Project

Activity
- Research Project
- Smart Meter
- Siemens ESD

Developer FTEs for mbedd Project + Accumulated Commits
- 3.2
- 4.2
- 3.4
- 2.6
- 3.8
- 4.0
- 3.5
- 2.5

Developers Joining
- A, B, C, D
- E
- F, G, H
- I

Language Development Start Date
- ext.units
- core.base, cc.requirements, analyses
- ext.statemachines
- ext.components, cc.variability
- core.c, core.build, core.unittest, core.utils

Migration to...
- github
- MPS 2.5
- MPS 3.0
- MPS 3.2
- MPS 3.3
Lessons Learned about Language Engineering from the Development of mbeddr

What can we learn?
1. Language Modularity
2. Projectional Editing
3. Managing Complexity
4. Tool Scalability
5. Development Process
For a scientific treatment with all caveats and data and details, check out the paper:

Lessons Learned about Language Engineering from the Development of mbeddr

What can we learn?

RQ 1

Language Modularity
Booom-Up Development

User Extensions

Languages shipped with mbeddr

Testing | Logging | Utilities | Messaging | Components | Physical Units | State Machines | Concurrency | Importer

C99

Plattform

Libraries for web server, node navigation, additional

MPS

Syntax Highlighting, Code Completion, Goto Definition,

Foundation

C Compiler & Debugger

Implementation
Some Hooks are known a priori
Some had to be added a posteriori.

IAssignmentLike: =, +=, *=, etc.

IFunctionLike: function, method

IBinaryLike: +, -, *, / but also . ?: !

IVoidTypeContainer: function, ftype

Conceptually this breaks modularity.
But it’s like OO programming/frameworks.
So we deem it acceptable.
Overmodularization!

Too much modularization is also not helpful.

Refactoring tough because of MPS bugs in Refactorings.
We used mostly Extension (+EC)

Embedding

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

Extension

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

Extension Composition

\[ L_{\text{Base}} + L_{\text{Ext1}} + L_{\text{Ext2}} = \]

Fundamentally it works and scales.
Is Extension Composition modular?

Multiple Extensions in C

No syntactic or IDE issues. Type systems compose well.

BUT: Semantic Interactions cannot be statically guaranteed to be correct.
Is Extension Composition modular?

Nesting in C/in each other

No syntactic or IDE issues. Type systems compose well.

BUT: Semantic Interactions cannot be statically guaranteed to be correct.
Is Extension Composition modular?

Nesting in each other

Adapter language Needed
Type system adapters needed

Adapter Language also deals with semantic interactions.

Specifically design integrations!
No modularity!
Is Extension Composition modular?

Yes, to the degree it can reasonably be expected to be (third case!)

But with the limitations that semantics cannot be statically checked to be compatible.

Might be solved in the (far) future when all transformations are declarative and analyzable.
Composing Type Systems

L1

L2

Declarative
Composing Type Systems

L1 + L2

Declarative
Order Irrelevant
Composing Transformations

Declarative
Order Relevant

User Extensions

Languages shipped with mbeddr

Testing  Logging  Utilities  Messaging  Components  Physical Units  State Machines  Concurrency  Importer

C99

Platform

Libraries for web server, node navigation, additional

MPS

Syntax Highlighting, Code Completion, Goto Definition,

Foundation

C Compiler & Debugger

Implementation
Composing Transformations

Declarative
Order Relevant

Simple extension: D extends I

A → D \overset{T_D}{\rightarrow} I \overset{T_i}{\rightarrow} \ldots

P(T_D) >> P(T_i)
Composing Transformations

Independent extension:
D1 and D2 extend I
Composing Transformations

Declarative
Order Relevant

Stacked extension:
D2 extends D1 and extends I
Composing Transformations

Declarative Order Relevant

Indep extension with gen dep:
We did not encounter this.
Composing Transformations

Getting the ordering right can be a challenge. We discuss debugging later.

Once correct, the system works well.
mbeddr’s 34 C extensions indicate that MPS’ language modularity works. It is useful for language understanding, testing and reuse.

In rare cases, modularity is compromised by necessary changes to the base language and unwanted dependencies between independent extensions.

Currently there is no way to detect (unwanted) semantic interactions between independent language extensions through analysis of their transformations.
Lessons Learned about Language Engineering from the Development of mbeddr

What can we learn?

RQ 2

Projectional Editing
Projectional Editing

Language Composition
Syntactic Flexibility is actually useful for end users – see Smart Meter.

```c
void aSummingFunction(int8 data[], int8 dataLen) {
    int16 sum;
    for (int8 i = 0; i < dataLen; i++) {
        sum += data[i];
    }
}
```

```c
int16 decide(int8 spd, int8 alt) {
    return
            spd > 0 spd > 100 otherwise 0;
    | alt < 0 | 1 1 |
    | alt == 0| 10 20|
    | alt > 0 | 30 40|
    | alt > 100| 50 60|
}
```

```c
double midnight2(int32 a, int32 b, int32 c) {
    return
        -b + \sqrt{b^2 - \sum_{i=1}^{4} a_i \cdot c_i}
    
    \sum_{i=1}^{4} a_i \cdot c_i
    
    \sum_{i=1}^{2} a_i
    
    \sum_{i=1}^{2} a_i
}
```

```
Cst.Customer

<table>
<thead>
<tr>
<th>Contract</th>
</tr>
</thead>
<tbody>
<tr>
<td>starts: date</td>
</tr>
<tr>
<td>ends: date</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tariff attributes</th>
</tr>
</thead>
</table>
```
Different Notations for one Abstraction.

State Machines

```plaintext
statemachine PedestrianButton initial = initial {
  in event requestButton() <no binding>
  in event timeout() <no binding>
  out event cars(Color carColor) <no binding>
  out event pedestrians(Color carColor) <no binding>
  state initial {
    entry {
      send cars(red);
      send pedestrians(red);
    }
  }
  state initial
  state driving {
    entry {
      send cars(green);
      send pedestrians(red);
    }
    on requestButton [ ] -> walking
  }
  state walking {
    entry {
      send cars(green);
      send pedestrians(green);
    }
    on timeout [ ] -> driving
  }
  state walking
}
```
Different Notations for one Abstraction.

State Machines

But: type checks and some actions based on AST, not on what is shown.

This can be confusing to users.
Optionally projected syntax.

Requirements Traces

Node: VERY_HIGH_SPEED [GlobalConstantDeclaration]
Kind: implements
1st Target: For each trackpoint where you go more than 200 mps, you get 20 points
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Praesent feugiat enim arcu, ut egestas velit. Suspendisse potenti. Etiam risus ante, bibendum ut mattis eget, convallis sit amet nunc.

#constant HIGH_SPEED = 10; [T]
#constant VERY_HIGH_SPEED = 20; [T]
#constant LANDING = 100; [T]
Decoupled Notational Primitives

Sum & Trees

Math in C and KernelsF

Trees for decisions and feature models
Developing new notations. MPS is Bootstrapped.
People prefer MPS over conventional IDEs
MPS more is more efficient than normal IDEs
MPS more is more productive than normal IDEs
MPS makes it easier to create correct programs
MPS enforces a structurally correct AST
People benefit from language modularity
People benefit from the flexible notations

The experience with learning MPS is mixed.
It takes some time to get used to MPS
Significantly Improved Editor Usability

Grammar Cells (see SLE Paper)

Top-Down Progr. via Target Creation

Palettes and other input support utils.

Generic Commenting
But still issues/differences in Editor Usability

Selecting code is structural, not linear

Copy/paste of similar syntax but different structure requires customization

Grammar cells go a (very) long way, but not *every* editing action is possible.

Pasting from text requires integration of a parser (hooks exist), not automatic

Very few complaints since we did the improvements.
We consider the problem mostly solved.
Version Control Integration

Fundamentally File-based
We used git – worked well
diff/merge in MPS
Can‘t use outside tools (gerrit)
Two main benefits of projectional editing – language modularity and a range of combinable notations – have been used extensively in mbeddr. The anticipated benefits have been observed.

Editor can be flexibly extended with new notational styles with acceptable effort, as exemplified by math, tables and diagrams.

The ability to use multiple and partial projections must be further improved by integrating with other language aspects, in particular, editor actions and type checks.
5 Lessons Learned about Language Engineering from the Development of mbeddr

What can we learn?

RQ 3 Managing Complexity
Projectional Editing

Reduced Complexity

Language modularization
No disambiguation code
[Language Aspects]

- Refactorings, Find Usages, Syntax Coloring, Debugging, ...

Diagram:
- Language
  - Structure: Concepts, Properties, Inheritance, Relationships
  - Editor: Projection Rules, Side Transformations, Intentions
  - Type System: Typing Rules, Type Checks, Other Validations
  - Transformations: Reduction Rules, Weaving Rules, Transformation Prios

Constraints:
- Scopes, Usage Restrictions, Property Value Limitations
[Language Aspects]

A separate DSL for each aspect

+ Refactorings, Find Usages, Syntax Coloring, Debugging...

DeclaraOve Metamodell
DeclaraOve Editor Cells
DeclaraOve Typing Rules
ImperaOve ValitaOons
Declarative Templates
Declarative Priorities
[Language Aspects]

A separate DSL for each aspect

Each language optimized for its task.

In our experience, separate aspect DSL increase productivity – as DSLs should.
A separate DSL for each aspect, but:

- Declarative Metamodel
- Declarative Editor Cells
- Declarative Typing Rules
- Imperative Validations
- Declarative Templates
- Declarative Priorities
- Imperative Constraints

MPS Base Language
essentially Java + X. Mostly Imperative
[Language Aspects]

A separate DSL for each aspect, but:

- Declarative Metamodel
- Declarative Editor Cells
- Declarative Typing Rules
- Imperative Validations
- Declarative Templates
- Declarative Priorities
- Imperative Constraints

MPS Base Language
essentially Java + X. Mostly Imperative

Declarative? { Analysis? Optimization?}
A separate DSL for each aspect, but:

\[\text{Declarative?} \quad \{ \begin{align*} &\text{Analysis?} \\ &\text{Optimization?} \end{align*} \]

The non-declarativeness leads to problems in analysis (impact, debugging, verifications) or optimization (caching).
[Language Aspects]

A separate DSL for each aspect. Debugging?

```c
int8 add(int8 x, int8 y) {
    int8[3] a = {1, 2, 3};
    int8[3] b;
    b = a;
    return x + y;
} add (function)
```

```c
void normalizePosition(Position* p) {
    if (p->x > 100) {
```

Cell Explorer

- EditorCell_Collection[refNodeList_contents] (15, 75, 386, 532, baseLine = 90, ascent = 15, descent = 517)
  - Node PlainCDemo (ImplementationModule) (>Vertical)
    - Actions
      - EditorCell_Collection[Collection_cznxc_a] (15, 75, 136, 76, baseLine = 90, ascent = 15, descent = 61)
      - EditorCell_Constant[Constant_yww9py_a] (15, 151, 2, 19, baseLine = 166, ascent = 15, descent = 4)
      - EditorCell_Collection[Collection_y1p9lm_a] (15, 170, 208, 114, baseLine = 170, ascent = 0, descent = 114)
- EditorCell_Collection[refNodeList_contents] (15, 75, 0, 0, baseLine = 0, ascent = 0, descent = 0)
  - Node add (Function) (>Vertical)
    - Actions
      - EditorCell_Collection[Collection_y1p9lm_b0] (15, 170, 208, 114, baseLine = 185, ascent = 15, descent = 99)
      - EditorCell_Constant[Constant_yww9py_a] (15, 284, 2, 19, baseLine = 299, ascent = 15, descent = 4)
      - EditorCell_Collection[Collection_y1p9lm_a] (15, 303, 296, 95, baseLine = 303, ascent = 0, descent = 95)
      - EditorCell_Constant[Constant_yww9py_a] (15, 398, 2, 19, baseLine = 413, ascent = 15, descent = 4)
      - EditorCell_Collection[Collection_y1p9lm_a] (15, 417, 386, 171, baseLine = 417, ascent = 0, descent = 171)
Language Aspects

A separate DSL for each aspect. Debugging?
[Language Aspects]

A separate DSL for each aspect. Debugging?
[Language Aspects]

A separate DSL for each aspect. Debugging?

System.out.println(...);

All in all, the debugging experience is not where it should be. Needs improvement.
[Language Aspects]

A separate DSL for each aspect, but missing aspects:

Declarative Metamodel
Declarative Editor Cells
Declarative Typing Rules
Imperative Validations
Declarative Templates
Declarative Priorities
Imperative Constraints

MPS BaseLanguage
essntially Java + X. Mostly Imperative

Documentation
Debugger
Interpreter
...

Since MPS 3.3, one can define custom aspects.
MPS is bootstrapped.

meaning: it is built with itself.

it can be extended with itself.
MPS is bootstrapped. meaning: it is built with itself. it can be extended with itself.

Declarative type mappings
Type Guards
Concise syntax, especially for recursive calls
Composable (just as languages)
Abstractions for environments and stacks
MPS is bootstrapped. meaning: it is built with itself. it can be extended with itself.

```python
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 String("equals")

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

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

Restricted, Declarative Language Transformed to Patterns Executed incrementally Very fast!
MPS is bootstrapped.

meaning: it is built with itself.

it can be extended with itself.

New primitive editor cells for math, diagram, tables

More query-oriented than default MPS notations

Notation independent from Language
Using a DSL for each language aspect works well based on our experience, even though some aspects are missing and some are not declarative enough to support meaningful analyses.

The support for debugging is spotty: it works well for transformations, but debugging generator macros, behaviours and type system rules is very tedious.

The ability to extend MPS’ language definition DSLs with MPS itself is a powerful approach for managing complexity, and we have used it extensively, even though it has some limitations.
What can we learn?

RQ 4

Tool Scalability
Editor Responsiveness

The bigger one root gets, the slower the editor gets because more “stuff” must be rendered:

- incrementally rendered, but still ...
- depends on the notation
- 3000 “lines” of C code

Split into several roots.
Impacts Language Design.
Code completion is extremely critical: you can only enter what has been “put into” the code completion menu. => Tree Constraints Scopes

```plaintext
statemachine TrafficLights initial = initial {
  in event pedestrian() <no binding>
  in event timeout() <no binding>
  state initial {
    on timeout [] -> cars
  }
  state initial {
    on timeout [] -> cars
  }
  state cars {
    on pedestrian !
    state cars {
      state pedestrian
      on timeout []
    }
  }
  state cars {
    on pedestrian
    state cars {
      state pedestrian
      on timeout []
    }
  }
  void normalizePosition()
  if (p->x > 100) ((..)....)
  p->x = 100; (*
  }
} normalizePosition (function)
```
Type Checking and Constraints

Type checking is incremental, and happens in the background.

Checks may run long – but are interruptable.

But both need locks on the repo: they can lead to performance issues.

=> Cancellable Checks
Assessments
Selective Enabling
“Energy Saver Mode”
The model with 128 roots corresponded to 68,400 LoC of generated C code (for Plain C), 202,000 LoC (C + State Machines) and 198,000 LoC (C + State Machines + Units + Components).
Transformation

Time ~ Size & & Time ~ Intermediate Model Volume
With attention to root size and the distribution of code over models, then **systems of significant size can be built** with MPS.

The performance of the **type system** (as it is evaluated in realtime in the editor) and **support for cross-model generation** are the two most critical ways of **improving MPS performance**.

During the **development of languages** we have not run into any problems regarding performance or scalability (of editor, type system or generator definitions).
Lessons Learned about Language Engineering from the Development of mbeddr

What can we learn?

RQ5

Development Process
Testing the editor

No need to test if “parsing” works.
But you have to check if you can enter what you need to enter, where you think it belongs.

```c
for ( int8 i = 0; i < 10

int8 add(int8 x, int8 y) {
    int8[3] a = {1, 2, 3};
    int8[3] b;
    struct
    return x + y;
} add (function)
```
Testing the editor

No need to test if “parsing” works.
And you have to test if the tree is constructed correctly when linear input is used.
Testing the type system

Do typing rules and constraints work? Essentially check if the red squiggles show up where you expect them.

```plaintext
Test case testingTypeChecksForLocalVariables
nodes

(  AnImplementationModule
  constraints
  imports
  nothing )

void f() {
  int8 anInt = 42;
  int8 aFloat = <check 3.3 has error>;
  int8 aString = <check "hello" has error>;
} f (function)
```
Whenever possible, test semantics by execution and test cases, and not by checking (the structure of) the generated code.
Testing semantics

Test case generation:

• Select a set of language extensions
• Randomly generate structurally sound programs
• If the type system finds errors in the program, discard the program (because errors are correctly reported)
• If the program has no errors, try to generate and compile it. If either generation or compilation fails, flag this particular generated program as a failed test.

Helps finding inconsistencies in the multi-stage transformation process.
Version Control

Works. As long as you are willing to use MPS for diff/merge.
CI Server Integration

We successfully run all our builds, tests and packaging on

However, the effort to get the MPS builds to run there is too high

- the usual build dependency hell
- the MPS build language is not flexible enough
- maybe initially not enough experience with “build stuff”

Also, builds tend to take (too) long.

- a full mbeddr build takes 12 minutes on the server and ca. 30 minutes on a fast developer PC

Mixed bag!
Migration of Models

Languages have version numbers

Automated refactorings increment that number; so can developers.

Migration scripts bring models up to date based on the language versions.

This is already more than in most other tools we know about. But it is clumsy and sometimes slow.

Mixed bag!
Except for the missing test support for model migrations and single-step transformations, language testing works well; we have achieved good coverage as demonstrated by a stable code base.

We have successfully integrated mbeddr’s build, test and packaging with the Teamcity CI server, but the effort was too high, partially because of the inadequacy of MPS’ build language.

Migrating models as languages change incompatibly is feasible with manually scripted migrations and their automatic execution based on implicitly-maintained language version numbers.
Concluding Thoughts
C is a strange language

The details were harder to build than we initially thought. But MPS can do it!
Language is not enough.

IDE, Debugger, Analyses, Interpreter.
Tool Chrome hurts.

MPS „looks“ too complicated for the prospective end user.
Onboarding devs.

It took 4 – 8 weeks to get competent new devs on board.
Optimizing Generators is hard :-}
Modularity vs. Optimization

Global optimization are in conflict with modularity and extensibility.

Perhaps two generators?
Open vs. Closed World

How do you build trust into a system that is inherently extensible?
More Declarativeness

More declarariveness and improved ability to analyze language definition code would help!
Shadow Models

We need incremental, real-time, multi-step, in-IDE transformations for advanced analyses.
Change is hard.

Prospective users were hard to convince to adopt mbeddr.
7 Lessons Learned about Language Engineering from the Development of mbeddr

Wrap Up
MPS is a capable LWB. Overall we are happy and continue to use it.

It is definitely ready for real-world use mbeddr – and this evaluation – is testament to that.

Many architectural decisions are good...
Projection, Aspect DSLs, Bootstrapping.

... but it has some weak points as well.
Internal complexity, imperativeness, build, fat Java client.

A good choice for LE projects today.
Great starting point for next gen LWBs :-}