Architecting
Domain-Specific Languages

Markus Völter
Introduction
<table>
<thead>
<tr>
<th></th>
<th>more in GPLs</th>
<th>more in DSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain Size</td>
<td>large and complex</td>
<td>smaller and well-defined</td>
</tr>
<tr>
<td>Designed by</td>
<td>guru or committee</td>
<td>a few engineers and domain experts</td>
</tr>
<tr>
<td>Language Size</td>
<td>large</td>
<td>small</td>
</tr>
<tr>
<td>Turing-completeness</td>
<td>almost always</td>
<td>often not</td>
</tr>
<tr>
<td>User Community</td>
<td>large, anonymous and widespread</td>
<td>small, accessible and local</td>
</tr>
<tr>
<td>In-language abstraction</td>
<td>sophisticated</td>
<td>limited</td>
</tr>
<tr>
<td>Lifespan</td>
<td>years to decades</td>
<td>months to years (driven by context)</td>
</tr>
<tr>
<td>Evolution</td>
<td>slow, often standardized</td>
<td>fast-paced</td>
</tr>
<tr>
<td>Incompatible Changes</td>
<td>almost impossible</td>
<td>feasible</td>
</tr>
</tbody>
</table>
Case Study: mbeddr
An extensible set of integrated languages for embedded software engineering.
#constant TAKEOFF = 100; \(\Rightarrow\) implements PointsForTakeoff
#constant HIGH_SPEED = 10; \(\Rightarrow\) implements FasterThan100
#constant VERY_HIGH_SPEED = 20; \(\Rightarrow\) implements FasterThan200
#constant LANDING = 100; \(\Rightarrow\) implements FullStop

[verifiable]
exported statemachine FlightAnalyzer initial = beforeFlight {
    in event next(Trackpoint* tp) <no binding>
    in event reset() <no binding>
    out event crashNotification() \(\Rightarrow\) raiseAlarm
    readable var int16 points = 0
    state beforeFlight {
        // [ Here is a comment on a transition. ]
        on next [tp->alt == 0 m] \(\Rightarrow\) airborne
        exit { points := TAKEOFF; } \(\Rightarrow\) implements PointsForTakeoff
    } state beforeFlight {
        Error: type int16 [m/s] is not comparable with (uint8 || int8)
    } state airborne {
        on next [tp->alt == 0 m & tp->speed == 0] \(\Rightarrow\) crashed
    } state airborne {
        on next [tp->speed > 200 mps] \(\Rightarrow\) crashed
        on next [tp->speed > 100 mps] \(\Rightarrow\) crashed
        on reset [ ] \(\Rightarrow\) beforeFlight
    } state airborne {
        on next [tp->speed == 0 mps] \(\Rightarrow\) landed
    } state landing {
        on next [tp->speed == 0 mps] \(\Rightarrow\) landed
    } state landing {
        on next [tp->speed == 0 mps] \(\Rightarrow\) landing { points--; } \(\Rightarrow\) ImplementShutdown
    }
}

next(Trackpoint* tp) {
    beforeFlight {
        // [ Here is a comment on a transition. ]
        [tp->alt == 0 m] \(\Rightarrow\) airborne
        [tp->alt == 0 m & tp->speed == 0] \(\Rightarrow\) crashed
        [tp->alt == 0 m & tp->speed > 0 mps] \(\Rightarrow\) lan
        [tp->speed > 200 mps & tp->speed > 0 mps] \(\Rightarrow\) lan
        [tp->speed > 100 mps & tp->speed <= 200 mps] \(\Rightarrow\) lan
    } airborne {
        [tp->speed == 0 mps] \(\Rightarrow\) landed
    } landing {
        [tp->speed == 0 mps] \(\Rightarrow\) landed
    } landed {
        [tp->speed == 0 mps] \(\Rightarrow\) landed
    }
}

^DataStructures.Trackpoint.alt (Member)
- crashNotification ^StateMachines.FlightAnalyzer.crashNotification (OutEvent)
- id
- speed
- time
- x
- y
- ^DataStructures.Trackpoint.x (Member)
- ^DataStructures.Trackpoint.y (Member)
Open Source @ eclipse.org
Eclipse Public License 1.0
http://mbeddr.com
itemis France: Smart Meter

First significant mbeddr project
ca. 100,000 LoC
about to be finished
great modularity due to components
uses physical units extensively
great test coverage due to special extensions
LMS INTERNATIONAL
Researchpark Haasrode 1237 | Interleuvenlaan 68 | B-3001 Leuven [Belgium]
T +32 16 384 200 | F +32 16 384 350 | info@lmsintl.com | www.lmsintl.com

Worldwide
For the address of your local representative, please visit www.lmsintl.com/lmsworldwide

LMS is a leading provider of test and mechatronic simulation software and engineering services in the automotive, aerospace and other advanced manufacturing industries. As a business segment within Siemens PLM Software, LMS provides a unique portfolio of products and services for manufacturing companies to manage the complexities of tomorrow’s product development by incorporating model-based mechatronic simulation and advanced testing in the product development process. LMS tunes into mission-critical engineering attributes, ranging from system dynamics, structural integrity and sound quality to durability, safety and power consumption. With multi-domain and mechatronic simulation solutions, LMS addresses the complex engineering challenges associated with intelligent system design and model-based systems engineering. Thanks to its technology and more than 1250 dedicated people, LMS has become the partner of choice of more than 5000 manufacturing companies worldwide. LMS operates in more than 30 key locations around the world.
20+ Projects in various stages by various “Big Name” companies.

Branching into other domains insurance, financial, tax
Open Source
Apache 2.0
http://jetbrains.com/mps
Language Workbench

- 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 Priors
- Constraints: Scopes, Usage Restrictions, Property Value Limitations
- Generates to: Structure
- Extends 0..*
[Projectional Editing]

 Parsing

 Concrete Syntax

 Abstract Syntax Tree

 Projectional Editing

 Concrete Syntax

 Abstract Syntax Tree
[Projectional Editing]

Syntactic Flexibility

Regular Code/Text

Mathematical

Tables

Graphical
Projectional Editing
Language Composition

Separate Files
Type System
Transformation
Constraints

In One File
Type System
Transformation
Constraints
Syntax
IDE
Expressivity
Shorter Programs

More Accessible Semantics
For a limited Domain!

Domain Knowledge encapsulated in language
Smaller Domain

More Specialized Language

Shorter Programs
The do-what-I-want language
Single Program vs. Class/Domain

No Variability!
Domain Hierarchy

more specialized domains
more specialized languages
Reification

\[ D_{n+1} \]

\[ D_n \]
Reification

Transformation/Generation

Language Definition

\[ \text{Transformation/Generation} \]

\[ \text{Language Definition} \]
C Extensions
Requirements
Linguistic Abstraction

In-Language Abstraction

Libraries
Classes
Frameworks
Linguistic Abstraction
Analyzable
Better IDE Support

In-Language Abstraction
User-Definable
Simpler Language
Linguistic Abstraction

In-Language Abstraction

User-Definable Language

Simpler Language

Analyzable

Better IDE Support

Special Treatment!
Linguistic Abstraction

Std Lib

In-Language Abstraction
Block Library
Unique State Names
Unreachable States
Dead End States
Guard Decidability
Reachability

Exhaustive Search, Proof!
Component Verification
SM Model Checking
Notation
UI for the language!
Important for acceptance by users!

Textual
Prose
Symbolic/Math
Tabular
Graphical
Reuse existing syntax of domain, if any!

Tools let you freely combine all kinds.
Reuse existing syntax of domain, if any!
State Machine Tables
Math
Component Wiring
Prose
Traces
Type System
Static Semantics

Execution Semantics
Static Semantics

Execution Semantics
Static Semantics

Constraints

Type Systems
Unique State Names
Unreachable States
Dead End States

Example Extended C
Unique State Names
Unreachable States
Dead End States

... Easier to do on a declarative Level!
Unique State Names
Unreachable States
Dead End States

Easier to do on a declarative Level!

Thinking of all constraints is a coverage problem!
var int x = 2 * someFunction(sqrt(2));

Assign fixed types

Derive Types

Calculate Common Types

Check Type Consistency

What does a type system do?
Intent + Check

var int x = 2 * someFunction(sqrt(2));

- More code
- Better error messages
- Better Performance

Derive

var x = 2 * some Function(sqrt(2));

- More convenient
- More complex checkers
- Harder to understand for users
Classical C Types

Closures
Execution
Def: Semantics

... via mapping to lower level

\[
\text{semantics}(p_{L_{D}}) := q_{L_{D-1}}
\]

where \( OB(p_{L_{D}}) == OB(q_{L_{D-1}}) \)

OB: Observable Behaviour (Test Cases)
Def: Semantics

... via mapping to lower level

\[ \text{semantics}(p_{L_D}) := q_{L_{D-1}} \]

where \( \text{OB}(p_{L_D}) = \text{OB}(q_{L_{D-1}}) \)
Transformation

$D_{n+1}$

$D_n$
Transformation

$\text{Transformation} \downarrow$

$\text{Known Semantics!}$
Transformation

Correct!?  Transformation

$L_D$  $L_{D-1}$
Run tests on both levels; all pass.
Coverage Problem!
mbeddr Tests
Multi-Stage

\[ L_3 \]
\[ L_2 \]
\[ L_1 \]
\[ L_0 \]

Modularization
Multi-Stage: Reuse

Reusing Later Stages Optimizations!
Multi-Stage: Reuse

Robot Control
State Machine
Components

C (MPS tree)

C Text
Multi-Stage: Reuse

Robot Control State Machine
- Consistency
- Model Checking

Components
- Efficient Mappings

C Type System
- Syntactic Correctness
- Headers

C (MPS tree)

C Text
Multi-Stage: Reuse

Reusing Early Stages

Portability
Multi-Stage: Reuse

$L_3 \rightarrow L_2 \rightarrow L_1 \rightarrow L_0$

Java

C#
Mock Components
Multi-Stage: Preprocess

Adding an optional, modular emergency stop feature
Composite Blocks Transformation
Platform

Generated Application
- Domain Frameworks
- Libraries
- Middleware
- Drivers
- Operating System
No Platform
A program at $D_0$ that acts on the structure of an input program at $D_{>0}$
Transformation

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

Interpretation
Essentially Everything :-}
Transformation

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

Interpretation

+ Turnaround Time
+ Runtime Change
Business Rules in Requirements
Def: Semantics

... via mapping to lower level

\[ L_D \]

Transformation

\[ L_{D-1} \]
Multiple Mappings
... at the same time

Similar Semantics?

all green!
Multiple Mappings

... at the same time

$L_D \circ T$

$L_x \circ T$

$L_y \circ T$

$L_z \circ T$

all green!

Similar Semantics?
Multiple Mappings
... alternatively, selectably

\[ L_D \]

Extend \( L_D \) to include explicit data that determines transformation

\[ L_X \quad L_Y \quad L_Z \]
Multiple Mappings

... alternatively, selectably

External Data:
- Switches
- Annotation Model
Build Config
Comp Static Wiring
Separation of Concerns
Several Concerns

... in one domain
Several Concerns
... in one domain

integrated into
one fragment

separated into
several fragments
Components + Instances
Viewpoints

\[ \forall r \in \text{Refs}_f \mid \text{fo}(r.to) = \text{fo}(r.from) = f \]

independent

\[ \forall e \in E_f \mid \text{lo}(\text{co}(e)) = l \]

dependent
Viewpoints: Why?

- Sufficiency
- Different Stakeholders
- Different Steps in Process - VCS unit!
Separate Requirements
Viewpoints

independent

sufficient?

contains all the data for running a meaningful transformation
Viewpoints: Why?

1:n Relationships
Viewpoints

Well-defined Dependencies

No Cycles!

Avoid Synchronization!
(unless you use a projectional editor)
Sync in Comp + Interfaces
Completeness
Can you generate 100% of the code from the DSL program?

More generally: all of $D_{-1}$
Incomplete: What to do?

\[ \text{OB}(F_D) \neq F_{D-1} \]
Incomplete: What to do?

Manually written code!

\[ \text{OB}(F_D) = F_{D-1} + F_{D-1, \text{man}} \]
Manually written code?

Call "black box" code
(foreign functions)
State Machine
Event Functions
Manually written code?

Call "black box" code
(foreign functions)

Embed $L_{D-1}$ code in $L_D$ program
All of mbeddr
Manually written code?

Call "black box" code (foreign functions)

Embed $L_{D-1}$ code in $L_D$ program

Use composition mechanisms of $L_{D-1}$ (inheritance, patterns, aspects, ...)
Manually written code?

Call "black box" code (foreign functions)

Embed $L_{D-1}$ code in $L_D$ program

Use composition mechanisms of $L_{D-1}$ (inheritance, patterns, aspects, ...)

Use protected regions (if you really have to...)
Manually written code?

Call "black box" code (foreign functions)

Embed $L_{D-1}$ code in $L_D$ program

Use composition mechanisms of $L_{D-1}$ (inheritance, patterns, aspects, ...)

Use protected regions (if you really have to...) DON'T!
Roundtripping

$L_D \downarrow \quad L'_{D-1} \quad \ldots \quad L'_{D-1}$
Roundtripping – Don’t!

Semantic Recovery!
8

Fundamental Paradigms
Structure

Modularization, Visibility

Namespaces, public/private importing
Structure

Modularization, Visibility

Namespaces,
public/private
importing

divide & conquer
reuse
stakeholder integration
mbeddr Chunks
Structure

Partitioning (Files)

VCS Unit
Unit of sharing
Unit of IP

!= logical modules
may influence language design
MPS models
Spec vs. Implementation

plug in different Impl's
different stakeholders
Interfaces + Components
Specialization

Liskov substitution property leaving holes ("abstract")
Structure

Specialization

Liskov substitution P
leaving holes ("abstract")

variants (in space)
evolution (over time)
Components
Polymorphism
Behavior

Not all DSLs specify behavior
Some just declare behavior

This section is not for those!
Behavior

Imperative

sequence of statements changes program state

<table>
<thead>
<tr>
<th>write</th>
<th>understand</th>
<th>debug</th>
<th>analyze</th>
<th>performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>simple -</td>
<td>simple (step)</td>
<td>hard</td>
<td>good</td>
</tr>
</tbody>
</table>
Behavior

Functional

functions call other functions. No state. No aliasing.

<table>
<thead>
<tr>
<th>write</th>
<th>understand</th>
<th>debug</th>
<th>analyze</th>
<th>performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>simple</td>
<td>simple (tree)</td>
<td>good</td>
<td>good -</td>
</tr>
</tbody>
</table>
ACCENT Blocks
Behavior

Functional

pure expressions are a subset of functional (operators hard-wired)
guards
preconditions
derived attributes
Business Rules – Debugging
Behavior

Declarative
only facts and goals.
no control flow.
eval engine, solver (several)

<table>
<thead>
<tr>
<th>write</th>
<th>understand</th>
<th>debug</th>
<th>analyze</th>
<th>performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>simple -</td>
<td>hard</td>
<td>depends</td>
<td>often bad</td>
</tr>
</tbody>
</table>
Behavior

Declarative

concurrency

constraint programming

solving

logic programming
Behavior

Data Flow

chained blocks consume continuous data that flows from block to block

<table>
<thead>
<tr>
<th>write</th>
<th>understand</th>
<th>debug</th>
<th>analyze</th>
<th>performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>simple/hard</td>
<td>hard</td>
<td>simple</td>
<td>can be good</td>
</tr>
</tbody>
</table>
Behavior

Data Flow

continuous, calc on change
quantized, calc on new data
time triggered, calc every x
Behavior

Data Flow

Embedded Programming

Enterprise ETL & CEP
ACCENT Blocks
Behavior

State Based

states, transitions, guards, reactions

event driven, timed

<table>
<thead>
<tr>
<th>write</th>
<th>understand</th>
<th>debug</th>
<th>analyze</th>
<th>performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple -</td>
<td>simple/hard</td>
<td>s/h</td>
<td>simple +</td>
<td>can be good</td>
</tr>
</tbody>
</table>
State Machines
Behavior

Combinations

Data flow uses functional, imperative or declarative language inside block.
Behavior

Combinations

state machines use expressions in guards and often an imperative lang in actions
Modularity
Language Modularity, Composition and Reuse (LMR&C) increase efficiency of DSL development.

4 ways of composition:

- Referencing
- Reuse
- Extension
- Reuse
Language Modularity, Composition and Reuse (LMR&C) increase efficiency of DSL development.

4 ways of composition:

distinguished regarding dependencies and fragment structure
Dependencies:

do we have to know about the reuse when designing the languages?

Fragment Structure:

homogeneous vs. heterogeneous („mixing languages“)
Dependencies & Fragment Structure:

<table>
<thead>
<tr>
<th>Independent Languages</th>
<th>Reuse</th>
<th>Embedding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Referencing</td>
<td></td>
<td>Extension</td>
</tr>
</tbody>
</table>

homogeneous fragment structure
Dependencies & Fragment Structure:

Referencing

Reuse

Extension

Embedding
Referencing

<table>
<thead>
<tr>
<th>Reuse</th>
<th>Embedding</th>
</tr>
</thead>
<tbody>
<tr>
<td>independent languages</td>
<td>homogeneous fragment structure</td>
</tr>
<tr>
<td>dependencies dependent</td>
<td>heterogeneous</td>
</tr>
</tbody>
</table>

Diagram:

- I$_2$ with B1
- I$_1$ with A1, A2, A3
- Dashed line from B1 to I$_1$
- Solid line from I$_2$ to I$_1$
Referencing

Dependent

No containment
Referencing

Used in Viewpoints
Containment

Dependent

Extension
more specialized domains
more specialized languages
Extension

\[ D_{n+1} \]

\[ D_n \]
Extension

Good for bottom-up (inductive) domains, and for use by technical DSLs (people)
Extension

Drawbacks

tightly bound to base
potentially hard to analyze
the combined program
Embedding

<table>
<thead>
<tr>
<th>Reuse</th>
<th>Embedding</th>
</tr>
</thead>
<tbody>
<tr>
<td>independent</td>
<td>homogeneous</td>
</tr>
<tr>
<td>languages</td>
<td>heterogeneous</td>
</tr>
<tr>
<td>dependencies</td>
<td>fragment structure</td>
</tr>
<tr>
<td>dependent</td>
<td></td>
</tr>
</tbody>
</table>
**Embedding**

- Independent
- Containment

<table>
<thead>
<tr>
<th>Reuse</th>
<th>Embedding</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Referencing</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>homogeneous</td>
<td>heterogeneous</td>
</tr>
<tr>
<td>fragment structure</td>
<td></td>
</tr>
</tbody>
</table>
Units in State Machines
Thank you!