DSLs for Product Lines: Approaches, Tools, Experiences

Markus Voelter
Independent/itemis
voelter@acm.org

DSLs for Product Lines: Approaches, Tools, Experiences

1. DSLs
2. Example DSLs
3. From FMs to DSLs
4. PLE Concepts & DSLs
5. Customization and Configuration
6. PLE for Languages
7. DSL Tools

SPLC 2011
DSLs

programming started close to the hardware

abstractions computing chips
abstractions
～computing

bits
abstractions
～computing?

Java

abstractions
～computing?

SQL
general purpose

domain specific
A DSL is a **focused**, **processable** language for describing a specific **concern** when building a system in a specific **domain**. The **abstractions** and **notations** used are natural/suitable for the **stakeholders** who specify that particular concern.
map

DSL Program
(aka Model)

map

automated!

GPL Program
Activities

Analysing Domains
Defining Languages
Adapting/Selecting
Building Editors
Transforming Models
Building Generators
Building Frameworks
... and using all of that to build apps

Example DSLs
Example 1: Embedded Protocol Handler

Component Specification

```c
processing digitalIn "02" moduleType DoRO hal = DigitalInHAL |

datatypes |
    SimplePointIndicationWithoutTime; 
    SimplePointIndicationWithTime; 
    DoublePointIndicationWithoutTime; 
    DoublePointIndicationWithTime; 
    BitStringType00BitStringType; 
    BitStringType11BitStringType |

parameters |
    DataTypes default | 
        moduleType.do # (integratedDeviceType == 5) SimplePointIndicationWithTime; |

    ParameterFilterTime default | 
        dtIx InitialTime[ms] == 0 ms |
        dtIx.d # (sp == 0xD) |
        dtIx.d # (sp == 0x0D) |

    / ReceiveOscillatingFrequency; |

function READDATA | | | | | ProcessData; |
function WRITEASK | | | | | Processokable; |

struct ProcessData |
    int8 channel; 
    int8 fkt(index); |

struct fkt | |
    int8 state; 
    ProcessTable entry; |

instance memory Memory |
```
Message Format Definition

```
procedure wireRegisterNumber2 requestCode 0x29 {
  request: struct request1 {
    int8 src pattern |
    2:b00;
    0:parentRequestCode;
  };
  int8 registerAddress;
};
reply: struct donCatReply {
  int8 statusType patterned statusType;
  int8 donCat patterned defaultReturn;
};
procedure request2 {
  int8 registerType pattern (4:b000;
  4:registerType);
};
int8 registerAddress;
int8 registeredData [2];
}
```

Testing

```
procedure wireRegisterNumber2 requestCode 0x29 {
  request: struct request1 {
    int8 src pattern |
    2:b00;
    0:parentRequestCode;
  };
  int8 registerAddress;
};
reply: struct donCatReply {
  int8 statusType patterned statusType;
  int8 donCat patterned defaultReturn;
};
procedure request2 {
  int8 registerType pattern (4:b000;
  4:registerType);
};
int8 registerAddress;
int8 registeredData [2];
}
```

```
tests

```
Example 2:

Pension Fund Specification
Calculation Rules and Tests

Intentional Software’s Domain Workbench
Example 3:
OSGi-based System

Component Specification

```java
subsystem the.world.economy {

public:
  immutable Type ProblemSolved {
    problem: string
    memory: int
    memorysize: long
  }

  interface Radio {
    reportAt: ProblemSolved; void
  }

  interface Printer {
    horrendous: string; string
  }

private:
  component Mobile {
    provides: object; void
    requires: phone; Phone
  }

  component PrinterPhone {
    requires: printer; Printer
  }

}

subsystem the.world.nation {

  uses the.economy

private:
  component American {
    task: portfolio scheduled oncePerStartUp
    requires home! Radio [*]
  }

}
```
import "cliamap/traversals.example"

subsystem the.world.scenario ( 
  public:
    immutable type ProblemFactory { 
      problems: string
      emergency: int
      emergency: bool
    }
  
  interface Fresh { 
    export: ProblemFactory? : void
  }
  
  interface Fresh { 
    broadcast?: string?: string
  }
  
  private:
    component model { 
      provides: Option?: Radio
      requires: input: Fresh
    }
    component FreshFactory { 
      provides: source: Fresh
    }
)

subsystem the.world.scenario ( 
  public:
    immutable type ProblemFactory { 
      problems: string
      emergency: int
      emergency: bool
    }
  
  interface Fresh { 
    export: ProblemFactory? : void
  }
  
  interface Fresh { 
    broadcast?: string?: string
  }
  
  private:
    component model { 
      provides: Option?: Radio
      requires: input: Fresh
    }
    component FreshFactory { 
      provides: source: Fresh
    }
)
Example 4:

Fountains
Hardware Structure

```
feature BasicOnePump
    pump compartment ccl
    static compressor cl
feature AtLeastOneZone extends BasicOnePump
    water compartment comp1
    pumped by cl
    compartment levelsensor ct_f1
    light l_f1
feature[f] SuperPowerCompartment
    water compartment adds to f
    superPowerMode
feature WithAlarm
    level alarm a1
fountain StdFountain extends AtLeastOneZone
```
Behaviour

```plaintext
pumping program PI for AtLeastOneSone + WithAlarm + SuperPowerCompartment[f=com1] {
    parameter defaultWaterLevel : int
    parameter superWaterLevel: int
    event superPowerTimeout

    init {
        set compl->targetHeight = defaultWaterLevel
    }

    start:
        on (compl->needsPower == true) && !(compl->isPumping) {
            do compl->pumpOn
        }
        on compl->enough {
            do compl->pumpOff
        }
        on compl.superPumping->turnedOn {
            set compl->targetHeight = superWaterLevel
            raise event superPowerTimeout after 20
        }
        on compl.superPumping->turnedOff or superPowerTimeout {
            set compl->targetHeight = defaultWaterLevel
        }
    }
```

Plus:

In-IDE Simulator
Unit Test Support
Eclipse Modeling
Eclipse Xtext

More Examples:
Miscellaneous
Hearing Aids

Refrigerators
BPEL Designer

Block Diagrams
PLC Programming

State Charts
Requirements (Eclipse)

Requirements (MPS)
From FMs to DSLs

Structure

Fountain -> Basin PUMP(rpm:int)
Basin -> ISFULLSENSOR? (ONENOZZLE | TWONOZZLES)

Behaviour?

Selection vs. Creation?
Programming in C?
Selection – DSL – Programming in C
### Structure + Behavior

**Fountain**
- Basin PUMP\((\text{rpm:} \text{int})\) Behavior

**Basin**
- ISFULLSENSOR? NOZZLE*

**Behavior**
- Rule*

**Rule**
- CONDITION CONSEQUENCE

### Structure + Behavior

**Fountain**
- Basin \(\text{id:} \text{PUMP}(\text{rpm:} \text{int})?\) Behavior

**Basin**
- \(\text{id:} \text{ISFULLSENSOR}(\text{full:} \text{boolean})?\) \(\text{id:} \text{NOZZLE} *\)

**Behavior**
- Rule *

**Rule**
- Condition Consequence

**Condition**
- Expression

**Expression**
- ATTRREFEXPRESSION | AndExpression
  - GreaterThan | INTLITERAL

**AndExpression**
- Expression Expression

**GreaterThan**
- Expression Expression

**Consequence**
- ATTRREFEXPRESSION Expression
## Structure + Behavior as Grammar

<table>
<thead>
<tr>
<th>Concrete Syntax</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fountain</td>
<td>&quot;fountain&quot; Basin Pump Behavior</td>
<td>Basin Pump Behavior as Grammar</td>
</tr>
<tr>
<td>Basin</td>
<td>&quot;basin&quot; IsFullSensor Nozzle*</td>
<td>Basin Pump Behavior as Grammar</td>
</tr>
<tr>
<td>Rule</td>
<td>Rule*</td>
<td>Rule as Grammar</td>
</tr>
<tr>
<td>Condition</td>
<td>&quot;if&quot; Condition &quot;then&quot; Consequence</td>
<td>Condition as Grammar</td>
</tr>
<tr>
<td>Expression</td>
<td>AttrRefExpression</td>
<td>Expression as Grammar</td>
</tr>
<tr>
<td>AndExpression</td>
<td>Expression &quot;&amp;&amp;&quot; Expression</td>
<td>AndExpression as Grammar</td>
</tr>
<tr>
<td>GreaterThan</td>
<td>Expression &quot;&gt;&quot; Expression</td>
<td>GreaterThan as Grammar</td>
</tr>
<tr>
<td>AttrRefExpression</td>
<td>&lt;attribute-ref-by-name&gt;</td>
<td>AttrRefExpression as Grammar</td>
</tr>
<tr>
<td>IntLiteral</td>
<td>(0..9)*</td>
<td>IntLiteral as Grammar</td>
</tr>
<tr>
<td>Consequence</td>
<td>AttrRefExpression &quot;=&quot; Expression</td>
<td>Consequence as Grammar</td>
</tr>
<tr>
<td>IsFullSensor</td>
<td>&quot;sensor&quot; ID (full:boolean)?</td>
<td>IsFullSensor as Grammar</td>
</tr>
<tr>
<td>Nozzle</td>
<td>&quot;nozzle&quot; ID</td>
<td>Nozzle as Grammar</td>
</tr>
<tr>
<td>Pump</td>
<td>&quot;pump&quot; ID (rpm:int)?</td>
<td>Pump as Grammar</td>
</tr>
</tbody>
</table>

## Structure + Behavior Example

```
fountain
  basin sensor s
  nozzle n1
  nozzle n2
pump p
  if s.full && p.rpm > 0
    then p.rpm = 0
```
Domain Specific Language

Concrete Syntax

**Definition**

**Program**

```plaintext
fountain basin sensor s
nozzle n1
nozzle n2
pump p
if s.full && p.rpm > 0
then p.rpm = 0
```

PLE Concepts & DSLs
domain engineering

... domain analysis
... classifying variability
... defining DSLs

application eng.

... using the DSL to express systems
platform

... target environment
... abstractions of solution space
... can influence DSL concepts
(Arch DSLs)

config knowledge

... transformations
... generators
... interpreters
core asset

... languages
... generators
... editors

variation point

... bind how
... kind of variability
... bind when
Bind How
Kinds of Variability
Bind When

Variability Mechanisms
Removal

... optionally take away from
Variability Mechanisms
Removal

... optionally take away from overall whole

Challenge:
overall whole can get big and unwieldy

Variability Mechanisms
Injection

... optionally add to minimal core
Variability Mechanisms

Injection

... optionally add to minimal core

Challenge:
how to point into the core and add something to it

Variability Mechanisms

Parametrization

... define values for predefined params
Variability Mechanisms

Parametrization

... define values for predefined params

Challenge:
types for parameters can be non trivial (DSLs)

4

PLE Concepts & DSLs

Bind How
Kinds of Variability
Bind When
Configuration vs. Customization

Variability

Routine Configuration vs. Creative Construction

Guidance, Efficiency vs. Complexity, Flexibility

Configuration Parameters
Feature-Model Based Configuration
Graph-Like Languages
Manual Programming

Property Files
Wizards
Tabular Configurations

Configuration Parameters
Feature-Model Based Configuration
Graph-Like Languages
Manual Programming

... selecting options
... setting param values
Configuration

... selecting options
... setting param values

Feature Models
Customization

... DSLs
... instantiation
... connections

4 PLE Concepts & DSLs

Bind How
Kinds of Variability
Bind When
# Binding Time

<table>
<thead>
<tr>
<th></th>
<th>flexibility</th>
<th>performance</th>
<th>code size</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>source time</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>compile time</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>link time</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>load time</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>run time</td>
<td>+++</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

# Generation

resulting code can be easily inspected
Interpretation vs. Generation

Generation
resulting code can be easily debugged

Generation
resulting code can be optimized and more efficient
Interpretation vs. Generation

Generation

Templates can be derived from existing code

Generation

work around limitations of target language
Interpretation vs. Generation

Generation

**no changes**

to target environment

(leaves no trace)

Generation

**reuse**

runtime infrastructure

(garbage collection, monitoring...)
Interpretation vs. Generation

**Interpretation**
- faster turnaround
  - no regeneration
  - test
  - build
  - deploy

Interpretation vs. Generation

**Interpretation**
- for platform independence
  - an interpreter might be less porting effort
Interpretation
possibly direct feedback from
in-IDE interpreter

Customization
and Configuration
Customization and Configuration

In-Language

Varying models

DSLs as Parameters

Trafo Variability

The language provides **explicit** support!

(we can learn that from GPLs)
In language

**Conditionals**
conditional execution

**Specialization**
overriding, overwriting

**Leaving Holes**
for variant to fill in

**Inject Stuff**
several places at a time?

---

In language

**Conditionals**
ifdef

**Specialization**
inheretance

**Leaving Holes**
template methods

**Inject Stuff**
AOP
Customization and Configuration

In-Language Varying models
DSLs as Parameters
Trafo Variability

Two Levels
~ problem space vs. software space

Problem Space: Configuration
Solution Space: Customization
Two Levels
~ problem space vs. software space

```c
#ifndef ACE_HAS_TLI
static ssize_t t_snd_n (ACE_HANDLE handle, const void *buf, size_t len, int flags,
ACE_Time_Value *timeout = 0, size_t *bytes_transferred = 0);
#endif /* ACE_HAS_TLI */
```

Two Levels Removal
Two Levels Removal

component DelayCalculator {
    provides default: IDelayCalculator
    requires screens[0..n]: IInfoScreen
    provides mon: IMonitoring feature monitoring
}

Two Levels Removal

System
  └── Failover
      └── Graceful Degradation
  ├── Monitoring
  └── Data Management
      ├── Centralized
      └── Distributed

Persistence
XML
Hibernate
JDO

Party
NeedsState
Multiple Addresses
Phone
International
Phone
LocalPhone

<<entity>>
Party
name: String

<<dependentOb>>
Phone
number: int
regionCode: int
countryCode: int

<<dependentOb>>
Address
city: String
state: String
zip: String
street: String

address 1

0..n

09.08.2011
Two Levels Removal

```csharp
namespace monitoringStuff
{
    feature monitoring
    {
        component MonitoringConsole
        {
            requires devices: [*]: IMonitor
        }

        instance monitor: MonitoringConsole
        {
            dynamic connect
            {
                monitor.devices
                query
                {
                    type = IMonitor
                }
            }
        }
    }
}
```
Two Levels Injection

```cpp
namespace monitoring {

  component MonitoringConsole ...
  instance monitor: ...
  dynamic connect monitor.devices ...

  aspect (*) component {
    provides mon: IMonitoring
  }
}
```

Two Levels Injection

```cpp
component DelayCalculator {
  ...
}
component AircraftModule {
  ...
}
component InfoScreen {
  ...
}

aspect (*) component {
  provides mon: IMonitoring
}
component DelayCalculator {
  ...
  provides mon: IMonitoring
}
component AircraftModule {
  ...
  provides mon: IMonitoring
}
component InfoScreen {
  ...
  provides mon: IMonitoring
}
Two Levels Removal Injection

namespace monitoring { feature monitoring {
  component MonitoringConsole ...
  instance monitor: ...
  dynamic connect monitor.devices ...

  aspect (*) component {
    provides mon: IMonitoring
  }
}

Two Levels ~ varying models, not text

... fewer var points
... semantically meaningful
... more manageable
Two Levels

~ varying models, not text

... simplifies traceability
you can „point“ easily to any element ->
generic traceability approach

... simplifies traceability
Customization and Configuration

In-Language Varying models
DSLs as Parameters
Trafo Variability

Feature Model w/ Parameters

... parameters with simple types
... complex types: DSL/metamodel
Customization and Configuration

In-Language Varying models
DSLs as Parameters
Trafo Variability

Model-Based Implementation
Model-Based Implementation

Problem Space

<table>
<thead>
<tr>
<th>Domain Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formal Domain Model</td>
</tr>
<tr>
<td>Formal Solution Space Model</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Assets</td>
</tr>
</tbody>
</table>

Domain Engineering

Application Engineering

Problem Space

Model Requirements

Product Requirements

Software Space

Formal Domain Model

Formal Solution Space Model

Transformation Variability

create System transformFs2Cbd{ Building building }:

hasFeature("burglarAlarm") ? (handleBurglarAlarm() -> this) : this;

handleBurglarAlarm{ System this }:

let conf = createBurglarConfig(): {

configurations.add( conf ) ->

...conf.connectors.add( connectSimToPanel( createSimulatorInstance(),
createControlPanelInstance() ) ) ->

hasFeature("siren") ? conf.addAlarmDevice("AlarmSiren") : null ->

hasFeature("bell") ? conf.addAlarmDevice("AlarmBell") : null ->

hasFeature("light") ? conf.addAlarmDevice("AlarmLight") : null
};
Transformation Variability

Generator Variability
We don’t want to model, we want to program!

... at different levels of abstraction
... from different viewpoints
... integrated!
We don’t want to model, we want to program!

... with different degrees of domain-specificity
... with suitable notations
... with suitable expressiveness

And always: precise and tool processable
We don’t want to model, we want to program!

**PLE** for DSLs suitable to **related domains**

Big Language?

with many first class concepts!
Small Language?

with a few, orthogonal and powerful concepts

Modular Language

with many optional, composable concepts
Viewpoints

suitable abstractions and notations for each
Viewpoints

Integrated
via symbolic references and seamless transitions

Viewpoints
General Purpose

predefined library
configure
Viewpoints
Domain Specific

custom
purpose-built
create/include

Custom Notations
real business expert integration

Viewpoints

General Purpose

Domain Specific

LEGO Robot Control

C

Components

State Machines

Sensor Access

LEGO Robot Control

Viewpoints

General Purpose

Domain Specific

LEGO Robot Control

C

Components

State Machines

Sensor Access

LEGO Robot Control
A DSL is a **focussed, processable language** for describing a specific concern when building a system in a specific **domain**. The **abstractions** and **notations** used are natural/suitable for the **stakeholders** who specify that particular concern.

### What's the Problem here?

```java
// A
int[] arr = ... for (int i=0; i<arr.size(); i++) {
  sum += arr[i];
}

// B
List<Integer> l = ... for (int i=0; i<arr.size(); i++) {
  l.add(arr[i]);
}
```

### Linguistic Abstraction
What’s the Problem here?

```java
// A
int[] arr = ...
for (int i=0; i<arr.size(); i++) {
    sum += arr[i]:
}
```

```java
// B
List<Integer> l = ...
for (int i=0; i<l.size(); i++) {
    l.add(arr[i]);
}
```

Much better with new **linguistic** abstraction

```java
// A'
for (int i in arr) {
    sum += i;
}
```

```java
// B'
seqfor (int i in arr) {
    l.add(arr[i]);
}
```

No sophisticated analysis required to understand the semantics of a construct and treat it “correctly”.

What’s this? And what’s the Problem?

```java
var linfollower_states_event linfollower_currentstate = linfollower_states_event::STATEunning;
enum linfollower_events_event { EVENT_INITIALIZED, EVENT_BLOCKED, EVENT_CLOSED, EVENT_UNLOCKED };
enum linfollower_states_event { STATE_INITIALIZING, STATE_PAUSED, STATE_RUNNING, STATE_PAUSED };
void linfollower_states_event linfollower_states_event event ) { 
    if ( linfollower_currentstate == linfollower_states_event::STATE_INITIALIZING ) { 
        if ( event == linfollower_events_event::EVENT_INITIALIZED ) { 
            linfollower_currentstate = linfollower_states_event::STATE_RUNNING;
            return;
        }
        if ( linfollower_currentstate == linfollower_states_event::STATE_RUNNING ) { 
            if ( true ) { 
                linfollower_currentstate = linfollower_states_event::STATE_DEAD;
                return;
            }
            if ( event == linfollower_events_event::EVENT_BLOCKED ) { 
                if ( true ) { 
                    linfollower_currentstate = linfollower_states_event::STATE_PAUSED;
                    state i = 5;
                    return;
                }
            }
        }
    }
}
```
Much better!

```latex
state_machine linefollower {
  event initialized;
  event bumped;
  event blocked;
  event unblocked;
  initial state initializing {
    initialized [true] -> running
  }
  state paused {
    entry int16 i = 1;
    unblocked [true] -> running
  }
  state running {
    blocked [true] -> paused
    bumped [true] -> crash
  }
  state crash {
    <<transitions>>
  }
}
```

<table>
<thead>
<tr>
<th>linefollower</th>
<th>initializing</th>
<th>paused</th>
<th>running</th>
<th>crash</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialized</td>
<td>true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bumped</td>
<td></td>
<td>true</td>
<td>crash</td>
<td></td>
</tr>
<tr>
<td>blocked</td>
<td></td>
<td>true</td>
<td>paused</td>
<td></td>
</tr>
<tr>
<td>unblocked</td>
<td></td>
<td>true</td>
<td>running</td>
<td></td>
</tr>
</tbody>
</table>
Domains are Hierarchical

Creating Linguistic Abstractions
A DSL is a language at D that provides linguistic abstractions for common patterns and idioms of a language at D-1 when used within the domain D.

A good DSL does not require the use of patterns and idioms to express semantically interesting concepts in D. Processing tools do not have to do “semantic recovery” on D programs.
Several Concerns in a Domain

Viewpoints
A: mixed
B: separate Viewpoints

When DSL (for several concerns) are developed from scratch, as a group, then dependencies between the concerns can be materialized as dependencies between the languages and the language concepts.
A language B extends another language A if B contains additional language concepts. This means that for programs written in B, all concepts from A are available, plus those defined in B.

A language has been developed to be used in contexts not known at the time of development. No dependencies allowed! The reusable language has to be extended so it can reference concepts from languages in that context.
Composition is a special case of reuse, where the reused language is syntactically embedded into languages from the context.
```c
exported interface MotorControl {
    void stop();
    void setLeftSpeed(int8 speed);
    void setRightSpeed(int8 speed);
}

exported component Motors {
    provides motorControl: MotorControl;
}
exported component implementation MotorsMXT: Motors {
    procedure void motorControl.stop() {
        nxt_motor_set_speed(MOTOR_PORT_B: NXT_PORT_B, 0, 1);
        nxt_motor_set_speed(MOTOR_PORT_C: NXT_PORT_C, 0, 1);
    }
    procedure void motorControl.setLeftSpeed(int8 speed) {
        nxt_motor_set_speed(MOTOR_PORT_B: NXT_PORT_B, speed, 1);
    }
    procedure void motorControl.setRightSpeed(int8 speed) {
        nxt_motor_set_speed(MOTOR_PORT_C: NXT_PORT_C, speed, 1);
    }
}
```

---

**SOLUTION**

### Function Block Fundamental Stuff

uses Aircraft, Environment

The dynamic pressure $p_{\text{dyn}}$ is calculated from the current air density $\varrho$ and the square of the flight speed $w$.

$$ p_{\text{dyn}} = \frac{1}{2} \cdot \varrho \cdot w^2 \quad \text{(N/m²)} $$

- $p_{\text{dyn}} = 0 \varrho < 0.125 \rightarrow 0$
- $p_{\text{dyn}} = 0.125 \rightarrow 0.3$
- $p_{\text{dyn}} = 0.25 \rightarrow 0.6$

### Function Block Stuff on the Rings

uses Environment, Aircraft, Fundamental Stuff

Aus dem Druck $p_{\text{dyn}}$ lässt sich dann der aktuelle Auftrieb $F_A$ berechnen; die Form wird $c_A$ beschrieben und die Fläche durch $A$.

$$ F_A = \text{double} \left( \frac{p_{\text{dyn}} \cdot A}{c_A} \right) \quad \text{(N)} $$

- $c_A = 0.3 \quad p_{\text{dyn}} = 0.125 \rightarrow 163.75$
- $p_{\text{dyn}} = 0.25 \rightarrow 367.5$

Auch der Widerstand $F_W$ berechnet sich entsprechend mit Hilfe des Beiwertes $c_W$:

$$ F_W = \text{double} \left( \frac{p_{\text{dyn}} \cdot A}{c_W} \right) \quad \text{(N)} $$

Angenommen wir haben mehrere Flügel $n_{\text{wings}}$ an Flugzeug, dann berechnet sich der Auftrieb

$$ F_A = n_{\text{wings}} \cdot F_A $$

Angenommen wir haben mehrere Flügel $n_{\text{wings}}$ an Flugzeug, dann berechnet sich der Widerstand

$$ F_W = n_{\text{wings}} \cdot F_W $$
Incremental Extension of C with DSLs for Embedded Systems, integrated with Formal Methods and support for PLE and Requirements Tracing
First C Code working

July 17, 2011 by repurpose

As you may know, our project relies on the idea of extending the C programming language with domain specific extensions. For that to work, we first have to make C available in MPS. While we had done this to some extent in our proof of concept, we are now implementing C much more thoroughly. As you can see in the screenshot below, some essential things are already working.
Tooling!
Editor, Debugger, Testing, Groupware, Scalable, „All in Eclipse“
Tools Tooling

Language Definition Tools
abstract syntax, concrete syntax, constraints

Editor Frameworks
Transformation Languages
Code Generation Tools

xtex

Meta Programming System

INTENTIONAL SOFTWARE
xtext
Open Source (EPL)
Eclipse-based, Eclipse Project
Very flexible, very popular!
Version 2.0 is current:
  - improved performance
  - Xbase: expressions for reuse
  - Xtend2: „Better Java“, with support for Xpand-like templates

MPS
Open Source (Apache 2.0)
Projectional Editor
Very good at lang. Compososition
Version 2.0 is current:
  - Improved performance
  - Unified generate/compile/build
  - Debug MPS in MPS
  - Tables in the editor
  - (Diagrams planned for 2.1)
Commercial Tool.  
Projectional Editor  
Very flexible notations  
Version 1.8 is current

Way More:  
Spoofax  
Rascal  
oomega  
The Whole Platform

see also  
http://languageworkbenches.net
THE END.

coordinates
web www.voelter.de
email voelter@acm.org
skype schogglad
twitter markusvoelter
xing http://www.xing.com/profile/Markus_Voelter
linkedin http://www.linkedin.com/pub/0/377/a31