DSL Design
A conceptual framework for building good DSLs

Markus Voelter
independent/itemis
voelter@acm.org

www.voelter.de
voelterblog.blogspot.de
@markusvoelter
+Markus Voelter

Based on material from a paper written with Eelco Visser

Introduction
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.

**Concepts** (abstract syntax)  
(concrete) **Syntax**  
semantics (generators)  
**Tools and IDE**
<table>
<thead>
<tr>
<th></th>
<th>more in GPLs</th>
<th>more in DSL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain Size</strong></td>
<td>large and complex</td>
<td>smaller and well-defined</td>
</tr>
<tr>
<td><strong>Designed by</strong></td>
<td>guru or committee</td>
<td>a few engineers and domain experts</td>
</tr>
<tr>
<td><strong>Language Size</strong></td>
<td>large</td>
<td>small</td>
</tr>
<tr>
<td><strong>Turing-completeness</strong></td>
<td>almost always</td>
<td>often not</td>
</tr>
<tr>
<td><strong>User Community</strong></td>
<td>large, anonymous and widespread</td>
<td>small, accessible and local</td>
</tr>
<tr>
<td><strong>In-language abstraction</strong></td>
<td>sophisticated</td>
<td>limited</td>
</tr>
<tr>
<td><strong>Lifespan</strong></td>
<td>years to decades</td>
<td>months to years (driven by context)</td>
</tr>
<tr>
<td><strong>Evolution</strong></td>
<td>slow, often standardized</td>
<td>fast-paced</td>
</tr>
<tr>
<td><strong>Incompatible Changes</strong></td>
<td>almost impossible</td>
<td>feasible</td>
</tr>
</tbody>
</table>

**General Purpose**

- C
- Components
- State Machines
- Sensor Access

**Domain Specific**

- LEGO Robot Control
Big Language
with many first class concepts!

Small Language
with a few, orthogonal and powerful concepts
Modular Language

with many optional, composable modules

Case Studies
namespace com.mycompany {
    namespace datacenter {
        component DelayCalculator {
            provides aircraft: IAircraftStatus
            provides console: IManagementConsole
            requires screens[0..n]: IIInfoScreen
        }
        component Manager {
            requires backend[1]: IManagementConsole
        }
        public interface IIInfoScreen {
            message expectedAircraftArrivalUpdate
                (id: ID, time: Time)
            message flightCancelled(flightID: ID)
        }
        public interface IAircraftStatus {
        }
        public interface IManagementConsole {
        }
    }
}

namespace com.mycompany.test {
    system testSystem {
        instance dc: DelayCalculator
        instance screen1: InfoScreen
        instance screen2: InfoScreen
        connect dc.screens to
            (screen1.default, screen2.default)
    }
}
appliance KIR {
  compressor compartment cc {
    static compressor c1
    fan ccfan
  }
  ambient tempsensor at
  cooling compartment RC {
    light rclight
    superCoolingMode
    door rcdoor
    fan rcfan
    evaporator tempsensor rceva
  }
}

parameter t_absztaatur: int
parameter t_absztauder: int
parameter t_absztaend: int
var tuertenschlurfschelle: int = 0

start: entry { state noCooling }
state noCooling:
  check {! (RC->needsCooling) && (cc.c1->status > 333) } {
    state rccooling
    on isDown { RC.rcdoor:open } {
      set RC.rcfan-active = true
      set RC.rclight->active = false
      perform rcfanabschalttask after 10 {
        set RC.rcfan-active = false
      }
    }
  }
state rccooling:
  entry { set RC.rcfan-active = true }
  check { ! (RC->needsCooling) } {
    state noCooling
    on isDown { RC.rcdoor:open } {
      set RC.rcfan-active = true
      set RC.rclight->active = false
      set tuertenschlurfschelle = curStep = 30
    }
    exit { perform rcfanabschalttask after max(5, tuertenschlurfschelle - curStep ) {
      set RC.rcfan-active = false
    }
  }
31.03.2012

Refrigerators

module main imports OsekKernel, ECASI, BitLevelUtilities {
constant int WHITE = 500;
constant int BLACK = 700;
constant int SLOW = 20;
constant int FAST = 40;

stateMachine linesfollower {
event initialized;
initial state initializing {
initialized (true) -> running
}
state running {
}
}

initialize {
ecrobot_set_light_sensor_active (SENSOR_PORT_T::NEXT_PORT_S1);
event linesfollower:initialized
}

terminate {
ecrobot_set_light_sensor_inactive (SENSOR_PORT_T::NEXT_PORT_S1);
}

task run cyclical prio = 1 every = 2 {
stateSwitch linesfollower
state running
int32 light = 0;
light = ecrobot_get_light_sensor (SENSOR_PORT_T::NEXT_PORT_S1);
if ( light < ( WHITE + BLACK ) / 2 ) {
updateMotorSettings(SLOW, FAST);
} else {
updateMotorSettings(FAST, SLOW);
}
default
<noop>;
}
void updateMotorSettings( int left, int right ) {
nxt_motor_set_speed(MOTOR_PORT_T::NEXT_PORT_C, left, 1);
nxt_motor_set_speed(MOTOR_PORT_T::NEXT_PORT_B, right, 1);
}

Example

Extended C
**Example**

```
entity Post {
    key :: String (id)
    blog -> Blog
    urlTitle :: String
    title :: String (searchable)
    content :: WikiText (searchable)
    public :: Bool (default=false)
    authors -> Set<User>
}  

function isAuthor(): Bool {
    return principal() in authors;
}

function mayEdit(): Bool {
    return isAuthor();
}

function mayView(): Bool {
    return public || mayEdit();
}

access control rules

rule page post(p: Post, title: String) {
    p.mayView()
}

rule template newPost(b: Blog) {
    b.isAuthor()
}

section posts

define page post(p: Post, title: String) {
    title: output(p.title)
    bloglayout(p.blog)
    placeholder view [ postView(p) ]
    postComments(p)
}

define permalink(p: Post) {
    navigate post (p, p.urlTitle) { elements }
}
```
Model

A schematic description of a system, theory, or phenomenon that accounts for its known or inferred properties and may be used for further study of its characteristics

www.answers.com/topic/model

Model

A representation of a set of components of a process, system, or subject area, generally developed for understanding, analysis, improvement, and/or replacement of the process

www.ichnet.org/glossary.htm
an abstraction or simplification of reality

ecosurvey.gmu.edu/glossary.htm

Model

which ones?

an abstraction or simplification of reality

what should we leave out?

ecosurvey.gmu.edu/glossary.htm
Model Purpose

... code generation
... analysis and checking
... platform independence
... stakeholder integration

... drives design of language!
Model Purpose

... code generation
... analysis and checking
... platform independence
... stakeholder integration

Example
Refrigerators

Model Purpose

... code generation
... analysis and checking
... platform independence
... stakeholder integration

Example
Extended C
Model Purpose

... code generation
... analysis and checking
... platform independence
... stakeholder integration

Example
Pension Plans

Programs Languages Domains
Domain body of knowledge in the real world

deductive top down

existing software (family)

inductive bottom up

Example
Penion Plans

Example
Refrigerators
A DSL \( L_D \) for \( D \) is a language that is specialized to encoding \( P_D \) programs.

more efficient

smaller
Programs are trees

Fragments are subtrees with root
Parent-Child Relation

Programs and Fragments
Programs are graphs, really.

\begin{center}
\begin{tikzpicture}
  \node (M) at (0,0) {M}
  \node (A) at (-1.5,-2) {A}
  \node (B) at (0,-2) {B}
  \node (C) at (-1.5,-3.5) {C}
  \node (D) at (0,-3.5) {D}
  \node (E) at (-1.5,-5) {E}
  \node (F) at (0,-5) {F}
  \node (G) at (0.5,-5) {G}
  \draw (A) -- (B);
  \draw (C) -- (D);
  \draw (B) -- (E);
  \draw (F) -- (G);
  \draw (E) -- (D);
  \node at (0,-1.5) {reference};
\end{tikzpicture}
\end{center}

Programs are graphs, really.

\begin{center}
\begin{tikzpicture}
  \node (M) at (0,0) {M}
  \node (A) at (-1.5,-2) {A}
  \node (B) at (0,-2) {B}
  \node (C) at (-1.5,-3.5) {C}
  \node (D) at (0,-3.5) {D}
  \node (E) at (-1.5,-5) {E}
  \node (F) at (0,-5) {F}
  \node (G) at (0.5,-5) {G}
  \draw (A) -- (B);
  \draw (C) -- (D);
  \draw (B) -- (E);
  \draw (F) -- (G);
  \draw (E) -- (D);
  \node at (0,-1.5) {to};
  \node at (0.5,-3.5) {from};
  \node at (0.5,-2) {Refs_f};
\end{tikzpicture}
\end{center}
Languages are sets of concepts

$L = \{C_1, C_2, C_3, \ldots, C_n\}$

Languages are sets of concepts

$L = \{C_1, C_2, C_3, \ldots, C_n\}$

lo ⇒ concept → language
Programs and languages

Language: concept inheritance
Language

does not depend on any other language

\[ \forall r \in \text{Refs}_l \mid \text{lo}(r) = \text{lo}(r) = 1 \]
\[ \forall s \in \text{Inh}_l \mid \text{lo}(s) = \text{lo}(s) = 1 \]
\[ \forall c \in \text{Cdh}_l \mid \text{lo}(c) = \text{lo}(c) = 1 \]

Independence

Fragment
does not depend on any other fragment

\[ \forall r \in \text{Refs}_f \mid \text{fo}(r) = \text{fo}(r) = f \]
\[ \forall e \in \text{Ef} \mid \text{lo}(e) = \text{lo}(e) = 1 \]

Example Refrigerators

Hardware:

```
compressor compartment cc {
    static compressor cl
    fan ccfan
}
```

Cooling Algorithm

```
macro kompressorAus {
    set cc.cl->active = false
    perform ccfanabschalttask after 10 {
        set cc.ccfan->active = false
    }
}
```
Homogeneous

Fragment

everything expressed
with one language

\[ \forall e \in E_f \mid lo(e) = 1 \]
\[ \forall c \in C_{dnf} \mid lo(c.parent) = lo(c.child) = 1 \]

Heterogeneous

Example

Extended C

module CounterExample from counterc imports nothing {

var int theI;
var boolean theB;
var boolean hasBeenReset;

stateMachine Counter {
    in start() < no binding>
    step(int[0..10]) size < no binding>
    out someEvent(int[0..100], x, boolean b) < no binding>
}

var int[0..10] currentVal = 0
var int[0..100] LIMIT = 10
states (initial = initialState)
    on start [ ] -> countState { send someEvent(100, true & false || true); }
    state countState {
        on step [currentVal + size > LIMIT] -> initialState { send reseted(); }
        on step [currentVal + size <= LIMIT] -> countState { currentVal = currentVal + size; }
        on start [ ] -> initialState { }
    }
}

end stateMachine

var Counter c1;

exported test case test1 {
    init(c1);
    assert(c) isInState(c1, initialState);
    trigger(c1, start);
    assert(c) isInState(c1, countState);
    test(test case);
}

ICSE 2012 Tutorial F4: DSL Design © 2012 Markus Völter
Heterogeneous Statemachines Testing

Domain Hierarchy

```
module CounterExample from counterd imports nothing {
  var int the1;
  var boolean the5;
  var boolean hasBeenReset;

  statemachine Counter {
    in start() <no binding>
    step(int[0.10] size) <no binding>
    out someEvent(int[0..100] x, boolean b) <no bindings>
    reseted() <no binding>
  }

  vars int[0..10] currentVal = 0
  int[0..100] LIMIT = 10
  states (initial = initialState)

  state initialState {
    on start [ ] -> countState { send someEvent(100, true & false || true); }
  }

  state countState {
    on step [currentVal + size > LIMIT] -> initialState { send reseted(); }
    on step [currentVal + size <= LIMIT] -> countState { currentVal = currentVal + size; }
    on start [ ] -> initialState {
      ... } // ...more
  }
}
```

Example Extended C

ICSE 2012 Tutorial F4: DSL Design © 2012 Markus Völter
Domain Hierarchy

Design Dimensions

expressivity
coverage
semantics
separation of concerns

completeness
paradigms
modularity
concrete syntax

process
Expressivity

expressivity
coverage
semantics
separation of concerns

completeness
paradigms
modularity
concrete syntax

process

Shorter Programs

More Accessible Semantics
For a limited Domain!

Domain Knowledge encapsulated in language

Def: Expressivity

A language $L_1$ is more expressive in domain $D$ than a language $L_2$ if for each $p \in P_D \cap P_{L_1} \cap P_{L_2}$, $|p_{L_1}| < |p_{L_2}|$
Smaller Domain
\[\downarrow\]
More Specialized Language
\[\downarrow\]
Shorter Programs

The do-what-I-want language
\[\psi\]
Single Program vs. Class/Domain

No Variability!

Domain Hierarchy

more specialized domains
more specialized languages
Reification

\[ D_n \]

Reification

\[ D_{n+1} \]

\[ \Rightarrow \]

\[ D_n \]
Reification

Transformation/Generation

Language Definition

Overspecification!
Requires Semantic Analysis!

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

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

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.
Def: DSL cont’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.

**Declarative!**

Another Example

```c
if (isConnected(port) || true) {
    port.doSomething();
}
```

**Turing Complete!**

**Requires Semantic Analysis!**
Example

```
with port (port) {
    port.doSomething();
}
```

Example

```
exported component AnotherDriver extends Driver {
  ports:
    requires optional ILogger logger
    provides IDriver cmd
  contents:
    field int count = 0
    int setDriverValue(int addr, int value) <- op cmd.setDriverValue {
      with port (logger) {
        logger.log("some error message");
      } with port
      return 0;
    }
}
```
Linguistic Abstraction

In-Language Abstraction

Libraries
Classes
Frameworks

Linguistic Abstraction

Analyzeable
Better IDE Support

In-Language Abstraction

User-Definable
Simpler Language
Linguistic Abstraction
Analyzable
Better IDE Support

In-Language Abstraction
User-Definable
Simpler Language

Special Treatment!
Example
Refrigerators

lib stdlib {
    command compartment::coolOn
    command compartment::cool0ff
    property compartment::totalRuntime: int readonly
    property compartment::needsCooling: bool readonly
    property compartment::couldUseCooling: bool readonly
    property compartment::targetTemp: int readonly
    property compartment::currentTemp: double readonly
    property compartment::isCooling: bool readonly
}

Language Evolution Support
Customization vs. Configuration

Precision vs. Algorithmics
Coverage

expressivity  completeness
coverage  paradigms
semantics  modularity
separation of  concrete
concerns  syntax

process

Domain $D_L$ defined inductively by $L$

(the domain that can be expressed by $L$)

$C_L(L) == 1$ (by definition)

not very interesting!
**Def: Coverage**

to what extend can a language $L$ cover a domain $D$

\[ C_D(L) = \frac{\text{number of } P_D \text{ programs expressible by } L}{\text{number of programs in domain } D} \]

**Def: Coverage**

why would $C_D(L)$ be $\neq 1$?

1. $L$ is deficient
2. $L$ is intended to cover only a subset of $D$, corner cases may make $L$ too complex

Rest must be expressed in $D_{-L}$
Def: Coverage

Coverage is full.
You call always write C.

Def: Coverage

Only a particular style of web apps are supported.

Many more are conceivable.
Def: Coverage

DSLs are continuously evolved so the relevant parts of the deductive domain are supported.

<table>
<thead>
<tr>
<th>Example</th>
<th>Example</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
<td>Refrigerators</td>
<td>Pension Plans</td>
</tr>
</tbody>
</table>

Semantics & Execution

- expressivity
- coverage
- semantics
- separation of concerns
- completeness
- paradigms
- modularity
- concrete
- syntax
- process
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!

Thinking of all constraints is a coverage problem!

Assign fixed types
Derive Types
Calculate Common Types
Check Type Consistency

What does a type system do?

var int x = 2 * someFunction(sqrt(2));
<table>
<thead>
<tr>
<th>Intent + Check</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>var int x = 2 * someFunction(sqrt(2));</td>
<td>var x = 2 * someFunction(sqrt(2));</td>
</tr>
<tr>
<td>More code</td>
<td>More convenient</td>
</tr>
<tr>
<td>Better error messages</td>
<td>More complex checkers</td>
</tr>
<tr>
<td>Better Performance</td>
<td>Harder to understand for users</td>
</tr>
</tbody>
</table>

**Example Refrigerators**

```java
macro kompressorAus {
    set cc.c1->active = "string"
    perform ccfanausschalter
    set cc.ccfan->active
}
```

**ICSE 2012 Tutorial F4: DSL Design © 2012 Markus Volter**
What does it all mean?

Execution Semantics

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

\[ \downarrow \]

Transformation

\[ \downarrow \]

Interpretation

\[ L_D \]

\[ L_{D-1} \]
Transformation

Example

Exten ded C

Transformation

\[ L_D \]

Transformation

\[ L_{D-1} \]

Known Semantics!
Transformation

Correct? Transformation

\[ L_D \downarrow \]
\[ L_{D-1} \]

Tests (D)

\[ \downarrow \]

Tests (D-1)

Transformation

\[ \downarrow \]

\[ L_{D-1} \]

Run tests on both levels; all pass. Coverage Problem!
Example
Refrigerators

Transformation

<table>
<thead>
<tr>
<th>Name</th>
<th>Documentation</th>
<th>Tags</th>
<th>Valid time</th>
<th>Transaction time</th>
<th>Fixture</th>
<th>Product</th>
<th>Element</th>
<th>Expected value</th>
<th>Actual value</th>
</tr>
</thead>
</table>
| Accrued right of
  extreme                |               |        | 2006-12-31 | 2007-9-24        | Jan De Jong  | Old Age Pension    | Accrued right          | 761.0402       | 761.0402     |
| Accrued right last pay   |               |        | 2004-1-1   | 2007-9-24        | Jan De Jong  | Old Age Pension    | Accrued right          | 705.0589       | 705.0589     |
| premium per year         |               |        | 2006-1-1   | 2007-9-24        | Jan De Jong  | Old Age Pension    | Premium old age pension| 329.0625       | 329.0625     |
| Accrued right at extreme |               |        | 2006-12-31 | 2007-9-24        | Pet Van Dijk | Old Age Pension    | Accrued right          | 740.94         | 724.7658     |
|                           |               |        | 1985-12-31 | 2007-9-24        | Jan De Jong  | Old Age Pension    | Accrued right in service period | 73.661         | 73.661       |
|                           |               |        | 1985-12-31 | 2007-9-24        | Jan De Jong  | Old Age Pension    | Years of service in service period | 3.7534         | 3.7534       |
|                           |               |        | 1987-12-31 | 2007-9-24        | Jan De Jong  | Old Age Pension    | Pension base average IP | 7750           | 7750         |
|                           |               |        | 1998-12-31 | 2007-9-24        | Jan De Jong  | Old Age Pension    | Accrued right in service period | 387.7445       | 387.7445     |
|                           |               |        | 1998-12-31 | 2007-9-24        | Jan De Jong  | Old Age Pension    | Years of service in service period | 10.8162        | 10.8162      |
|                           |               |        | 1998-12-31 | 2007-9-24        | Jan De Jong  | Old Age Pension    | Pension base average IP | 8250           | 8250         |
Transformation

Tests
Simulators
Documentation

Transformation

Example Refrigerators
Multi-Stage

L₃

L₂

L₁

L₀

Modularization

Multi-Stage: Reuse

L₃

L₂

L₁

L₀

Reusing Later Stages

Optimizations!
Multi-Stage: Reuse

Robot Control
State Machine

Components

C (MPS tree)

C Text

Example
Extended C

Consistency
Model Checking

Efficient Mappings

C Type System

Syntactic
Correctness,
Headers

ICSE 2012 Tutorial F4: DSL Design © 2012 Markus Völter
Multi-Stage: Reuse

Reusing Early Stages

Portability

Example

Extended C
Multi-Stage: Preprocess

Adding an optional, modular emergency stop feature

Platform

Generated Application
- Domain Frameworks
- Libraries
- Middleware
- Drivers
- Operating System
A program at \( D_0 \) that acts on the structure of an input program at \( D_0 \).

- **Imperative**: step through
- **Functional**: eval recursively
- **Declarative**: ? solver?
<table>
<thead>
<tr>
<th>Name</th>
<th>Documentation/Tags</th>
<th>Valid time</th>
<th>Transaction time</th>
<th>Fixture</th>
<th>Product</th>
<th>Element</th>
<th>Expected value</th>
<th>Actual value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquired right at retirement</td>
<td></td>
<td>2006-12-31</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>Acquired right</td>
<td>761.0402</td>
<td>761.0402</td>
</tr>
<tr>
<td>Acquired right at death</td>
<td></td>
<td>2004-1-1</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>Acquired right</td>
<td>705.0589</td>
<td>705.0589</td>
</tr>
<tr>
<td>premium last year</td>
<td></td>
<td>2006-1-1</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>Premium old age pension</td>
<td>329.5625</td>
<td>329.5625</td>
</tr>
<tr>
<td>Acquired right at retirement</td>
<td></td>
<td>2006-12-31</td>
<td>2007-9-24</td>
<td>Piet Van Dijk</td>
<td>Old Age Pension</td>
<td>Acquired right</td>
<td>740.94</td>
<td>724.7658</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1985-12-31</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>Acquired right in service period</td>
<td>73.661</td>
<td>73.661</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1985-12-31</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>Years of service period</td>
<td>7.534</td>
<td>7.534</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1987-12-31</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>Pension base average PP</td>
<td>7750</td>
<td>7750</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1998-12-31</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>Acquired right in service period</td>
<td>387.7449</td>
<td>387.7449</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1998-12-31</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>Years of service period</td>
<td>10.8082</td>
<td>10.8082</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1998-12-31</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>Pension base average PP</td>
<td>8250</td>
<td>8250</td>
</tr>
</tbody>
</table>

### Example

**Pension Plans**

```prolog
param: int

start() {
  state noCooling
}

state noCooling:
  check (RC->needCooling) && (cc.cl->state == state rcCooling)
  on isDown (RC->door open) {
    set RC.rfan active=true
    set RC.rfan active=false
    perform rcFanSchaltTask after 10
  }

state rcCooling:
  state noCooling
  on isDown (RC->door open) {
    set RC.rfan active=true
    set RC.rfan active=false
    set rcFanSchaltSchelle = curStep + 30
  }
  exit {
    perform rcFanSchaltTask after max(5, rcFanSchaltSchelle-curStep) {
      set RC.rfan active=false
    }
  }
```

### Example

**Refrigerators**
A program at $D_0$ that acts on the structure of an input program at $D_{>0}$
**Interpretation**

An interpreter :-)
**Def: Semantics**

... via mapping to lower level

\[ L_D \]

Transformation

\[ L_{D-1} \]

---

**Multiple Mappings**

... at the same time

\[ L_D \]

\[ L_x \]

\[ L_y \]

\[ L_z \]

Similar Semantics?
Multiple Mappings
... at the same time

**L_D**

**L_x** **L_y** **L_z**

Similar Semantics?

all green!
Multiple Mappings

... alternatively, selectably

\[ L_D \]

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

\[ L_X \quad L_Y \quad L_Z \]

Restricted Port leads to reduced overhead C

Example

Extended C
Multiple Mappings

... alternatively, selectably

External Data:
- Switches
- Annotation Model

Switch Control
Java vs. C

Example
Pension Plans
**Multiple Mappings**

... alternatively, selectably

![Diagram](heuristics.png)

Heuristics: Analyze model to try to decide

\[ L_D \quad \Downarrow \quad L_X, L_Y, L_Z \]

---

**Multiple Mappings**

... alternatively, selectably

![Diagram](testing.png)

Heuristics: Analyze model to try to decide

\[ L_D \quad \Downarrow \quad L_X, L_Y, L_Z \]

TESTING!
Reduced Expressiveness

bad? maybe.
good? maybe!

Model Checking
SAT Solving

Exhaustive Search, Proof!

Unique State Names
Unreachable States
Dead End States
Guard Decidability
Reachability

Exhaustive Search, Proof!
Separation of Concerns

expressivity  completeness
coverage       paradigms
semantics      modularity
separation of  concrete
cconcerns      syntax

process

Several Concerns

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

integrated into one fragment

separated into several fragments

Viewpoints

∀r ∈ Refs_f | fo(r.to) = fo(r.from) = f
∀e ∈ E_f | lo(co(e)) = l

independent
dependent
Viewpoints

Hardware

Behaviour

Tests

Example Refrigerators

Viewpoints: Why?

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

Hardware

Product Management

Behaviour
Thermodynamics-Experts

Tests

Example
Refrigerators

Viewpoints

independent

sufficient?
contains all the data for running a meaningful transformation
Viewpoints

- sufficient implementation code
- sufficient Hardware structure documentation

Example
Refrigerators

Viewpoints: Why?

1:n Relationships
Viewpoints

Well-defined Dependencies
No Cycles!
Avoid Synchronization!
(unless you use a projectional editor)
**Viewpoints: Why?**

\[ L_D \xrightarrow{\text{Annotation}} L_{D-1} \]

**Progressive Refinement**

![Diagram of progressive refinement]
Views on Programs

Example Pension Plans

Completeness

expressivity coverage semantics separation of concerns

completeness paradigms modularity concrete syntax process
Can you generate 100% of the code from the DSL program?

More generally: all of $D_{-1}$

Semantics:

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

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

Introduce \( G \) ("generator"):

\[ OB(p) == OB(G(p)) == OB(q) \quad \text{complete} \]

\[ OB(G(p)) \subset OB(p) \quad \text{incomplete} \]
Incomplete: What to do?

\[ F_D \]
\[ \Downarrow \]
\[ \text{OB}(F_D) \neq F_{D-1} \]

Incomplete: What to do?

Manually written code!

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

Call "black box" code
(foreign functions)

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

Call "black box" code (foreign functions)

Embed $\mathbb{LD}_{-1}$ code in $\mathbb{LD}$ program

Embed C statements in components, state machines or decision tables

Example

Refrigerators

Use composition mechanisms of $\mathbb{LD}_{-1}$ (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, …)

Generate base classes with abstract methods; implement in subclass

Use protected regions (if you really have to…)

ICSE 2012 Tutorial F4 DSL Design © 2012 Markus Völter
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!

Incomplete: When?

Good for technical DSLs: Devs write L_{D-1} code

Bad for business DSLs. Maybe use a L_{D-1} std lib that L_D code can call into?
Incomplete: When?

**Good** for technical DSLs: Devs write \( \mathcal{L}_{D-1} \) code

**Bad** for business DSLs.
Maybe use a \( \mathcal{L}_{D-1} \) std lib that \( \mathcal{L}_D \) code can call into?

---

Prevent Breaking Promises!

```java
class B extends BBase {
    public void doSomething() {
        Registry.get("A").someMethod();
    }
}
```
Prevent Breaking Promises!

Better:

Dependency Injection
Static analysis tools

Roundtripping

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

Semantic Recovery!

Fundamental Paradigms

expressivity | completeness
coverage | paradigms
semantics | modularity
separation of concerns | concrete
process | syntax
Structure

Modularization, Visibility

Namespaces,
public/private
importing

divide & conquer
reuse
stakeholder integration
Structure

Partitioning (Files)

VCS Unit
Unit of sharing
Unit of IP

!= logical modules
may influence language design

Structure

Modularization, Visibility

Example

Extended C

```c
module Module1 from HPL.main imports Module2 {
    exported var int aReallyGlobalVar;

    struct aLocallyVisibleStruct {
        int x;
        int y;
    };

    exported int anExportedFunction() {
        return anImportedFunction/Module2();
    } anExportedFunction (function)
}
```
Structure

Modularization, Visibility

```java
namespace com.mycompany.test {
    system testSystem {
        instance dc: DelayCalculator
        instance screen1: InfoScreen
        instance screen2: InfoScreen
        connect dc.screens to
            (screen1.default, screen2.default)
    }
}
```

Example Components

Structure

Partitioning (Files)

- change impact
- link storage
- model organization
- tool integration
Structure

Spec vs. Implementation

plug in different Impl.

different stakeholders

---

Example

Extended C
**Structure**

**Specialization**

Liskov substitution P leaving holes ("abstract")

variants (in space)
evolution (over time)

---

**Structure**

**Specialization**

Pension Plans can inherit from other plans.

Rules can be abstract;

Plans with abstract rules are abstract
Structure

Superposition, Aspects
merging
overlay
AOP

modularize cross-cuts

Example Components

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
}
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**

**Imperative**

sequence of statements changes program state

Example

Refrigerators

---

**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>
Behavior

Functional

functions call other functions. no state. No aliasing.
Behavior

Functional

pure expressions are a subset of functional (operators hard-wired)
guards
preconditions
derived attributes

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

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

Declarative

Exten C

synchronous blockType org:recipeLabs::doma::library::base::Discrete::DiscreteDerivative
input u
output y
parameter initialCondition = 0
parameter gain = 1.0 // normalized
behavior {
    stateful func main(initialCondition, gain, fs>u) -> y {
        check@, I(x), x/I/>(real) -> real
        static assert u is real() ;
        error "Input value must be numeric"
        static assert initialCondition is real() ;
        error "Initial condition must be numeric"
        static assert initialCondition is real() & u is real() -> unit(initialCondition) == unit(u) ;
        error "Initial condition and input value must have same unit"
        static assert gain is real() ;
        error "Gain value must be numeric"
        eq y(-1) = initialCondition
        eq y(n) = fs * gain * (u(n) - u(n-1))
    } }
Behavior

Reactive

Example
Refrigerators

Behavior

Data Flow

chained blocks consume continuous data that flows from block to block

write  understand  debug  analyze  performance

simple -  simple/hard  hard  simple  can be good
Behavior

Data Flow

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

Embedded Programming
Enterprise ETL & CEP
Behavior

State Based

states, transitions, guards, reactions

event driven, timed

write understand debug analyze performance

simple - simple/hard s/h simple + can be good

Example Refrigerators
Behavior

Combinations

data flow uses functional, imperative or declarative language inside block

Behavior

Combinations

state machines use expressions in guards and often an imperative language in actions
Behavior

Combinations

Example
Refrigerators

start:
  entry { state noCooling }

state noCooling:
  check ( !(RC:needsCooling) && (cc.c1 > stdbrkt > 333) ) {
    state rcooling
    on isDown ( RC:rcdoor>xopen ) {
      set RC:rfan-active = true
      set RC:rlight-active = false
      perform rfanAbschalttask after 30 {
        set RC:rfan-active = false
      }
    }
  }

state rcooling:
  entry { set RC:rfan-active = true }
  check ( !(RC:needsCooling) ) {
    state noCooling
    on isDown ( RC:rcdoor>xopen ) {
      set RC:rfan-active = true
      set RC:rlight-active = false
      set turnMaschungScheule = currStep + 30
    }
    exit {
      perform rfanAbschalttask after max( 5, turnMaschungScheule-currStep ) {
        set RC:rfan-active = false
      }
    }
  }

Behavior

CallCycle

CallHandling interface user:
in event hangup
in event accept
interface phone:
in event callIncoming: string in event callFinished
out event acceptCall
out event hangupCall
internal:
event finished = callFinished || hangup
var timer: integer
Behavior

Combinations

purely structural languages often use expressions to specify constraints

```
c/s interface IDriver {
    int setDriverValue(int addr, int value)
    pre value > 0
}
```

Language Modularity

expressivity coverage semantics separation of concerns
completeness paradigms modularity concrete syntax

process
Language Modularity, Composition and Reuse (LMR&C) increase efficiency of DSL development

4 ways of composition:

Referencing
Reuse
Extension
Reuse

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:

Referencing

Reuse

Extension

Embedding

Referencing

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

Homogeneous and heterogeneous fragment structure
Referencing

Dependent

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

<table>
<thead>
<tr>
<th>homogeneous</th>
<th>heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>fragment structure</td>
<td></td>
</tr>
</tbody>
</table>

No containment

Used in Viewpoints
Referencing

Fragment

references

Fragment

references

Fragment

Example Refrigerators

Referencing

Fragment

references

Fragment

references

Fragment

Referencing

prolog {
set RC->accumulatedRuntime = 0
}
step 10
assert-currentstate-is noCooling
mock: set RC->accumulatedRuntime = 100
step
assert-currentstate-is abtauen
assert-value cc.cl->active is false
mock: set RC->accumulatedRuntime = 0
step 5
assert-currentstate-is abtauen
assert-value cc.cl->active is false
step 15
assert-currentstate-is noCooling

Example Refrigerators
Extension

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

Dependent

Extension

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

Containment
more specialized domains
more specialized languages
**Extension**

$D_{n-1} \Rightarrow D_n = \bigtriangleup$

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

Example
Extension

Example

ICSE 2012 Tutorial F4: DSL Design © 2012 Markus Völter

Reuse

I1

A1

A2

A3

I2

B3

B4

B5

Dependency

Reuse

Embedding

Referencing

Extension

Independent

Homogeneous

Fragment structure

Dependent

Heterogeneous

ICSE 2012 Tutorial F4: DSL Design © 2012 Markus Völter
Often the referenced language is built expecting it will be reused.

*Hooks may be added.*
Embedding

<table>
<thead>
<tr>
<th>independent</th>
<th>languages dependencies dependent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuse</td>
<td>Embedding</td>
</tr>
<tr>
<td>Referencing</td>
<td>Extension</td>
</tr>
</tbody>
</table>

homogeneous | heterogeneous
fragment structure

I_2

B3 → B4

I_A

B5

I_1

A1 → A2

A3

I_2

B3 → B4

I_A

B5

I_1

A1 → A2

A3
**Embedding**

**Independent**

- **Containment**

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

**Example Pension Plans**
Embedding often uses Extension to extend the embedded language to adapt it to its new context.

Challenges - Syntax

Extension and Embedding requires modular concrete syntax

Many tools/formalisms cannot do that
Challenges - Type Systems

**Extension:** the type system of the base language must be designed to be extensible/overridable.

Challenges - Type Systems

**Reuse and Embedding:** Rules that affect the interplay can reside in the adapter language.
Challenges - Trafo & Gen
Referencing (I)

Written specifically for the combination

Can be Reused

Two separate, dependent single-source transformations

Challenges - Trafo & Gen
Referencing (II)

A single multi-sourced transformation
Challenges - Trafo & Gen
Referencing (III)

A preprocessing trafo that changes the referenced frag in a way specified by the referencing frag

Challenges - Trafo & Gen
Extension

Transformation by assimilation, i.e. generating code in the host lang from code expr in the extension lang.
Challenges - Trafo & Gen

Extension

```c
module impl imports <<imports>> {
    int speed( int val ) {
        return 2 * val;
    }

    robot script stopAndGo
    block main on bump
        accelerate to 12 + speed(12) within 3000
        drive on for 2000
        turn left for 200
        decelerate to 0 within 3000
        stop
    }
}
```

Example

Extende d C

Challenges - Trafo & Gen

Reuse (I)

Reuse of existing transformations for both fragments plus generation of adapter code
Challenges - Trafo & Gen

Reuse (II)

composing transformations

Challenges - Trafo & Gen

Reuse (III)

generating separate artifacts plus a weaving specification
Challenges - Trafo & Gen
Embedding (I)

a purely embeddable language
may not come with a generator.

Assimilation (as with Extension)

Challenges - Trafo & Gen
Embedding (II)

Adapter language can coordinate the
transformations for the host and for
the embedded languages.
Concrete Syntax

- expressivity
- coverage
- semantics
- separation of concerns

- completeness
- paradigms
- modularity
- concrete syntax

process

UI for the language!
Important for acceptance by users!

- Textual
- Symbolic
- Tabular
- Graphical
Reuse existing syntax of domain, if any!

Tools let you freely combine all kinds.

Default: Text

Editors simple to build
Productive
Easy to integrate w/ tools
Easy to evolve programs

... then add other forms, if really necessary
Default: Text

Editors simple to build
Productive
Easy to integrate w/ tools
Easy to evolve programs

Graphical in case...

Relationships
Graphical in case...

Flow and Dependency

Graphical in case...

Causality and Timing
Symbolic

Either Mathematical, or often highly inspired by domain

Tables

<table>
<thead>
<tr>
<th>Name</th>
<th>Documentation</th>
<th>Tags</th>
<th>Valid time</th>
<th>Transaction time</th>
<th>Fixture</th>
<th>Product</th>
<th>Expected value</th>
<th>Actual value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actuared right at extreme</td>
<td></td>
<td></td>
<td>2006-12-31</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>761.0402</td>
<td>761.0402</td>
</tr>
<tr>
<td>Actuared right at last pay</td>
<td></td>
<td></td>
<td>2004-1-1</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>705.0589</td>
<td>705.0589</td>
</tr>
<tr>
<td>premium of last year</td>
<td></td>
<td></td>
<td>2006-1-1</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>329.5625</td>
<td>329.5625</td>
</tr>
<tr>
<td>Actuared right at extreme 2</td>
<td></td>
<td></td>
<td>2006-12-31</td>
<td>2007-9-24</td>
<td>Piet Van Dijk</td>
<td>Old Age Pension</td>
<td>740.94</td>
<td>724.7638</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1985-12-31</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>73.661</td>
<td>73.661</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1985-12-31</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>3.7534</td>
<td>3.7534</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1987-12-31</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>7750</td>
<td>7750</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1990-12-31</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>387.7449</td>
<td>387.7449</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1998-12-31</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>10.8082</td>
<td>10.8082</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1998-12-31</td>
<td>2007-9-24</td>
<td>Jan De Jong</td>
<td>Old Age Pension</td>
<td>8250</td>
<td>8250</td>
</tr>
</tbody>
</table>
Combinations

```c
int decide(int x, int y) pre

component AComp extends nothing {
  ports:
    provides Decider decider
  contents:
    int decide(int x, int y) <- op decider.decide {
      return int, 0
        x == 0 x > 0 ;
        y == 0 0 1
        y > 0 1 2
    }
}
```

Combinations

![Flowchart diagram](image)
Combinations
Process

expressivity coverage semantics separation of concerns completeness paradigms modularity concrete syntax

Domain Analysis

Interview Experts
Get Feedback
Structure their Knowledge
Create Examples
Build the Language
Iterate to goal

language size/complexity

ss

time

Documentation

Create example-based tutorials!
Domain Folks Programming?

Precision vs. Algorithmics!

- DU Coding
- DU/Dev Paired
- Dev Coding
- DU Reviewing
DSL as a Product

Release Plan
Bug Tracker
Testing!
Support
Documentation

Reviews

Reviews become easier --- less code, more domain-specific
The End.
This material is part of my upcoming (early 2013) book DSL Engineering.
Stay in touch, it may become a free eBook 😊

voelter.de/dslbook

www.voelter.de
voelterblog.blogspot.de
@markusvoelter
+Markus Voelter