DSL Design
A conceptual framework for building good DSLs

Code Generation 2012

Markus Voelter
independent/itemis
voelter@acm.org

based on material from a paper written with Eelco Visser

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

E.Visser@tudelft.nl
http://eelcovisser.org/

Code Generation 2012

Markus Voelter
independent/itemis
voelter@acm.org

based on material from a paper written with Eelco Visser

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

E.Visser@tudelft.nl
http://eelcovisser.org/
8+1 Design Dimensions

expressivity  completeness
coverage paradigms
semantics modularity
separation of concrete
concerns syntax

process

Which abstractions go into the language, and why

expressivity  completeness
coverage paradigms
semantics modularity
separation of concrete
concerns syntax

process
Which portions of the domain is covered by the DSL?

expressivity coverage semantics separation of concerns

completeness paradigms modularity concrete syntax

process

What does it all mean?

expressivity coverage semantics separation of concerns

completeness paradigms modularity concrete syntax

process
Should the various concerns in the domain be separated into different viewpoints?

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

Is the DSL able to express all% of the system, or do you have to „manually“ write code?

expressivity  | completeness
coverage     | paradigms
semantics    | modularity
separation of concerns | concrete
process
Which language paradigms exist, and when and how can they be used in DSLs?

- expressivity
- coverage
- semantics
- separation of concerns

- completeness
- paradigms
- modularity
- concrete
- syntax

---

How do you modularize (and then compose and combine) languages (GPLs and DSLs)?

- expressivity
- coverage
- semantics
- separation of concerns

- completeness
- paradigms
- modularity
- concrete
- syntax
### Which syntactic forms are most suitable, and how do you combine them?

<table>
<thead>
<tr>
<th>expressivity</th>
<th>completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td>coverage</td>
<td>paradigms</td>
</tr>
<tr>
<td>semantics</td>
<td>modularity</td>
</tr>
<tr>
<td>separation of concerns</td>
<td>concrete syntax</td>
</tr>
</tbody>
</table>

process

### What do you have to consider in terms of the development process?

<table>
<thead>
<tr>
<th>expressivity</th>
<th>completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td>coverage</td>
<td>paradigms</td>
</tr>
<tr>
<td>semantics</td>
<td>modularity</td>
</tr>
<tr>
<td>separation of concerns</td>
<td>concrete syntax</td>
</tr>
</tbody>
</table>

process
namespace com.mycompany {
    namespace datacenter {
        component DelayCalculator {
            provides aircraft: IAircraftStatus
            provides console: IMangementConsole
            requires screens[0..n]: IInfoScreen
        }
        component Manager {
            requires backend[1]: IMangementConsole
        }
        public interface IInfoScreen {
            message expectedAircraftArrivalUpdate
                (id: ID, time: Time)
            message flightCancelled(flightID: ID)
        }
        public interface IAircraftStatus ...
        public interface IMangementConsole ...
    }
}
```cpp
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 cl
        fan ccfan
    }
    ambient tempsensor at
    cooling compartment RC {
        light rclight
        superCoolingMode
door rcdoor
        fan rcfan
        evaporator tempsensor rceva
    }
}
```
parameter int t_abwaschauf;  
parameter int t_abwaschende; 
var int t_auslaufschwelle; 

start: 
  entry ( state noCleaning )

state noCleaning: 
  check ( (RC->needCleaning) && (cc.cl->stehzeit > 333) ) 
  state rcCooling
  on isDown ( RC->door->open ) 
    set RC.rcfan->active = true 
    set RC.rclight->active = false 
    perform rfanabschalttask after 10 
    set RC.rcfan->active = false 

state rcCooling: 
  entry ( set RC.rcfan->active = true )
  check ( !(RC->needCleaning) )
  state noCleaning
  on isDown ( RC->door->open ) 
    set RC.rcfan->active = true 
    set RC.rclight->active = false 
    set tuerNachlaufSchwelle = curStep + 30
  exit 
  perform rfanabschalttask after max( 5, tuerNachlaufSchwelle-currStep ) 
  set RC.rcfan->active = false 

prolog 
  set RC->accumulatedRuntime = 0

step 10
  assert-currentState-is noCooling
  mock: set RC->accumulatedRuntime = 110
  step
  mock: set RC.rcfan->setTemp = 10
  assert-currentState-is abtauend
  assert-value cc.cl->active is false
  mock: set RC->accumulatedRuntime = 0
  step 5
  assert-currentState-is abtauend
  assert-value cc.cl->active is false
  step 15
  assert-currentState-is noCooling
In this talk: mostly a little bit

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

Reification

\[ D_{n+1} \rightarrow \quad D_n \rightarrow \quad \text{Reified} \]
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<int> l = ...;
for (int i=0; i<arr.size(); i++) {
    l.add(arr[i]);
}
```
**Linguistic Abstraction**

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

**Declarative!**
**Directly represents Semantics.**

```java
seqfor (int i in arr) {
    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.

Linguistic Abstraction

In-Language Abstraction

Libraries
Classes
Frameworks
Linguistic Abstraction
Analyzable
Better IDE Support

Special Treatment!

In-Language Abstraction
User-Definable
Simpler Language
Linguistic Abstraction

Std Lib

In-Language Abstraction

Example Refrigerators

lib stdlib {
    command compartment::coolOn
    command compartment::coolOff
    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
}
# Semantics & Execution

<table>
<thead>
<tr>
<th>expressivity</th>
<th>expressivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>coverage</td>
<td>completeness</td>
</tr>
<tr>
<td>semantics</td>
<td>paradigms</td>
</tr>
<tr>
<td>separation of concerns</td>
<td>modularity</td>
</tr>
<tr>
<td></td>
<td>concrete</td>
</tr>
<tr>
<td></td>
<td>syntax</td>
</tr>
</tbody>
</table>

---

## Static Semantics

---

## Execution Semantics
Static Semantics

Constraints
Type Systems

What does it all mean?

Execution Semantics
Def: Semantics
... via mapping to lower level

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

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

OB: Observable Behaviour (Test Cases)
Mapping

\[ \mathcal{L}_D \]

Transformation

\[ \downarrow \]

Interpretation

\[ \mathcal{L}_{D^{-1}} \]

Known Semantics!

Transformation

\[ \mathcal{L}_D \]

Correct!? \[ \downarrow \]

Transformation

\[ \downarrow \]

Interpretation

\[ \mathcal{L}_{D^{-1}} \]
Transformation

Tests (D) \xrightarrow{L_D} \rightarrow Transformation

Tests (D-1) \xrightarrow{L_{D-1}} \rightarrow Interpretation

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

Example
Refrigerators
Def: Semantics

... via mapping to lower level

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

\[ L_D \]
\[ \downarrow \quad \downarrow \quad \downarrow \]
\[ L_x \quad L_y \quad L_z \]
Similar Semantics?

Multiple Mappings
... at the same time

\[ L_D \]
\[ \downarrow \quad \downarrow \quad \downarrow \]
\[ L_x \quad L_y \quad L_z \]
Similar Semantics?

all green!
Multiple Mappings

... alternatively, selectably

\[ \text{Extend } L_D \text{ to include explicit data that determines transformation} \]

\[ L_D \quad \downarrow \quad \downarrow \quad \downarrow \]

\[ L_X \quad L_Y \quad L_Z \]

Example

Restricted Port leads to reduced overhead C
Multiple Mappings
... alternatively, selectably

External Data:
- Switches
- Annotation Model

Switch Control
Java vs. C

Example
Pension Plans
Multiple Mappings
... alternatively, selectably

Heuristics: Analyze model to try to decide

Multiple Mappings
... alternatively, selectably

TESTING!
Transformation

\[ D_{n+1} \]

\[ D_n \]

---

Transformation

```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 Extended C
**Multi-Stage**

```
L_3
\downarrow
L_2
\downarrow
L_1
\downarrow
L_0
```

**Modularization**

---

**Multi-Stage: Reuse**

```
L_3
\downarrow
L_2
\downarrow
L_1
\downarrow
L_0
```

---

Reusing Later Stages

Optimizations!
Multi-Stage: Reuse

Robot Control
State Machine

Components

C (MPS tree)

C Text

Example
Extended C

Syntactic Correctness, Headers
Efficient Mappings

Consistency
Model Checking

C Type System

Example
Extended C
Multi-Stage: Reuse

Reusing Early Stages Portability

Multi-Stage: Reuse

Java

C#

Example Pension Plans
Reduced Expressivity

bad? maybe.
good? maybe!

Model Checking
SAT Solving
Exhaustive Search, Proof!

Challenges

Model + Property Specifications
Tool
OK COUNTER EXAMPLE
Out Of Memory
Challenges

Model + Property Specifications

Tool

Formalism
Input Language
Property Language
Algorithm
Interpretation

OK
COUNTER EXAMPLE
Out Of Memory

State Based

Model + Property Specifications

Tool

State Machines
NuSMV
LTL / CTL

„Magic“

OK
COUNTER EXAMPLE
Out Of Memory

Out Of Memory
**mbeddr Approach**

- High-Level State Machine
- High-Level Properties

**generate**

**NuSMV Model**

**generate**

**CTL**

**highlight errors**

**NuSMV**

**Text File**

---

**mbeddr Approach**

- easier to use
- hopefully used more
- full power: write
  - CTL/LTL if you want to
Model Checking

http://mbeddr.com

Model Checking

http://mbeddr.com
Model Checking

Finds problems in state machines.

... even ones you didn’t think of!

Much more complete than manual testing.

http://mbeddr.com
Model Checking

Language Modularity

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

Components

State Machines

Sensor Access

LEGO Robot Control

General Purpose

Domain Specific

Big Language

with many first class concepts!
Small Language

with a few, orthogonal and powerful concepts

Modular Language

with many optional, composable modules
Language does not depend on any other language

Independence

Fragment does not depend on any other fragment

Independence

Hardware:
```c
compressor compartment cc {
    static compressor c1
    fan ccfan
}
```

Cooling Algorithm
```c
macro kompressorAus {
    set cc.c1->active = false
    perform ccfanabschalttask after 10 {
        set cc.ccfan->active = false
    }
}
```

Example Refrigerators
Homogeneous

Fragment
everything expressed
with one language

\[ \forall e \in E_f \mid lo(e) = 1 \]
\[ \forall c \in \text{Cdn}_f \mid lo(c.parent) = lo(c.child) = 1 \]

Heterogeneous

Example

Extended C
Language Modularity, Composition and Reuse increase efficiency of DSL development

4 ways of composition:

- Referencing
- Extension
- Reuse
- Embedding
Language Modularity, Composition and Reuse
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?
Dependencies:

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

Fragment Structure:

homogeneous vs. heterogeneous („mixing languages“)
Referencing

Dependent

No containment

Referencing

Used in

Viewpoints
Referencing

Fragment

References

Fragment

References

Fragment

Referencing

Example Refrigerators

prolog

{ set RC->accumulatedRuntime = 00 }

step 10

assert-currentstate-is noCooling

mock: set RC->accumulatedRuntime = 110

step

assert-currentstate-is abort

assert-value cc.cl->active is false

mock: set RC->accumulatedRuntime = 0

step 3

assert-currentstate-is abort

assert-value cc.cl->active is false

step 15

assert-currentstate-is noCooling

parameter t_abbaustart: int

parameter t_abbaustopp: int

parameter t_abbaustopp: int

var tuerenaufwaertshoehe: int = 0

start:

{ state noCooling }

state noCooling:

check ( RC->needsCooling ) && ( cc.cl->state = state noCooling )

on isoDown ( RC->rdoor->open ) {

set RC->rdoor->active = true

set RC->rdoor->active = false

perform <<fanabschaltAufset:after 30 ( set RC->rdoor->active = false

state recooling:

check ( RC->needsCooling ) && ( state recooling )

on isoDown ( RC->rdoor->open ) {

set RC->rdoor->active = true

set RC->rdoor->active = false

set tuerenaufwaertshoehe = currStep = 30

exit

perform <<fanabschaltAufset:after max 5, tuerenaufwaertshoehe=currStep )

set RC->rdoor->active = false

}
more specialized domains
more specialized languages

dependent

containment
**Extension**

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

\[ D_n \]

---

**Extension**

**Drawbacks**

- tightly bound to base
- potentially hard to analyze
- the combined program
Extension

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

statemachine linefollower {
    event initialized {
        initialized [true] -> running
    }
    state running {
        ...
    }
    initiate {
        ecrobot_set_light_sensor_active (SENSOR_PORT_T::NXT_PORT_S1);
        event linefollower:initialized
    }
    terminate {
        ecrobot_set_light_sensor_inactive (SENSOR_PORT_T::NXT_PORT_S1);
    }
}

task run cycloic prio = 1 every = 2 {
    stateswitch linefollower
    state running
    int32 light = 0;
    light = ecrobot_get_light_sensor (SENSOR_PORT_T::NXT_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::NXT_PORT_S1, left);
    nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_S1, right);
}

Extension C

module CounterExample from counter imports nothing {
    var int theI;
    var boolean theB;
    var boolean hasBeenReset;

    statemachine Counter {
        in start() < no binding>
        start(0..10) size < no binding>
        out someEvent(int[0..100] x, boolean b) < no binding>
        reseted() < no binding>
    }
    vars int[0..10] currentVal = 0
    int[0..100] LIMIT = 10
    state (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 [ ]
    }
}

Extension C Statemachines Testing

var Counter cl;
exported test case test1 {
    init(0);
    assert(c.isInState(cl, initialState);
    trigger(cl, start);
    assert(c.isInState(cl, countState);
}

Often the referenced language is built expecting it will be reused.

Hooks may be added.
Embedding

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

**Containment**

**Example Pension Plans**
Embedding

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)

Two separate, dependent single-source transformations

Written specifically for the combination

Can be Reused

Challenges - Trafo & Gen
Referencing (II)

A single multi-sourced transformation
Referencing (III)

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

Extension

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

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
}

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

Combinations

```c/s interface Decider {
    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,
            \begin{tabular}{|c|c|c|}
                \hline
                \multicolumn{2}{|c|}{\textbf{x}} & \textbf{y} \\
                \hline
                \textbf{x = 0} & 0 & 1 \\
                \textbf{x > 0} & 1 & 2 \\
                \hline
            \end{tabular}
        }
```
Combinations

**Diagram:**

- A flowchart showing a PID controller with feedback and feedforward paths.
-Key components: K Ts/z, K(z-1)/z, and the process model 1/(s^2 + 0.5s + 1).

**Table:**

<table>
<thead>
<tr>
<th>Description</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>system SHALL display speed</td>
<td></td>
</tr>
</tbody>
</table>
Combinations

The End.

This material is part of my upcoming (early 2013) book **DSL Engineering**. Stay in touch, it may become a free eBook 😊

http://voelter.de/dslbook

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