Model-Driven Development and DSLs in the context of embedded systems

Markus Voelter
Independent/itemis
voelter@acm.org

A little History
Domain Specific Languages
Examples
Tools
Benefits
Architecture DSLs
A more theoretical View
Programming and Modeling
The LWES Project
Bonus: Best Practices
A little History

programming started close to the hardware

abstractions ~ computing
abstractions ¬ computing?

Java

abstractions ¬ computing?

SQL
general purpose

domain specific
tailor made
effective++
specialized, limited
used by experts
together with other specialized tools

2

Domain Specific Languages
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.
DSL Program
(aka Model)

map

automated!

GPL Program

map

Generation
Transformation
Compilation

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

compiled vs. interpreted
customization vs. configuration

graphical vs. textual
Example 1: Embedded Protocol Handler
**Component Specification**

```c
processing DigitalIn "22" moduleType 0x00 hal + DigitalIn2Hal {

dataTypes {
  SinglePointIndicationWithoutTime;
  SinglePointIndicationWithTime;
  IndicatingWithTime;
  IndicatingWithoutTime;
  StringType1(StringWithoutTime);
  StringType2(StringWithoutTime);
}

parameters {
  DataTypes default {
    singlePointIndicationType -- get SinglePointIndicationWithTime;
  };
  ProcessingTime default {
   ời int timeInSeconds = 0;(U6)
    singlePointIndicationType (P == 0x00);
    StringType2(P == 0); (M == 0x00);
  };
  MaximumOscillatingFrequency;
}

function PADATA (i : ProcessData)
function WRITEUNIT (input : ProcessData)

struct ProcessData {
  int8 channel;
  int32 requestId;
}

struct Memory {
  int32 state;
  ProcessData data;
  instance memory Memory;
}
```

**Message Format Definition**

```c
procedure writeRegisteredNumber2 requestCode 0x29 {
  request: struct request1 {
    int8 arcPattern = 2; spU;
    C.parse(requestCode);
  };
  int8 (registerAddress);
  reply: struct donMessage {
    int8 statusType pattern (statusType);
    int8 donCore pattern (defaultReturn);
  };
  struct request2 {
    int8 registerType pattern (4b0000); (registerType);
  };
  int8 registerAddress;
  int8 registerData [2];
};
```
Testing

```c
procedure writeRegisteredNumber2 registerCode 0x37 (  
  request: struct request1 (  
    int8 addr pattern: 2:3:0;  
    struct registerAddress;  
  );  
  int8 registerAddress;  
);  
request: struct request2 (  
  int16 registerType pattern: 4:0:0:0:0:0:0:0;  
  int8 registerAddress;  
  int8 registerData [2];  
);  
);  
```
Example 2: Pension Fund Specification

Textual Documentation
Insurance Mathematics

Calculation Rules and Tests
Example 3:

Radar Systems Engineering
Component Definition

```java
import "/classpath/test.jar"

quantity voltage is double
quantity temperature is double

source component Sensor

  producers: [measurement: voltage, measurement: temperature]

  consumes: [sensorInput]
  produces: [sensorOutput]

  behaviers:
    m.measurements <- measurement
    m.temperature <- temperature

processing component TempCalibration

  consumes: [sensorInput]
  produces: [calibratedMeasurement]

  behaviers:
    calibratedMeasurement <- measurement

sink component Output

  consumes: [TempCalibrated]
  produces: [outputValue]

  behaviers:
    outputValue <- measurement
    outputValue <- originalMeasurement
```

Component Behavior Specification

```
beginwhitespace("beginMainModule")
beginwhitespace("""
beginwhitespace(""
beginwhitespace(""
beginwhitespace(""
beginwhitespace(""
beginwhitespace(""
beginwhitespace(""
beginwhitespace(""
beginwhitespace(""

beginwhitespace("Main")
beginwhitespace(""
beginwhitespace(""
beginwhitespace(""
beginwhitespace(""

endwhitespace("Main")
endwhitespace(""
endwhitespace(""
endwhitespace(""
endwhitespace(""
endwhitespace(""
endwhitespace(""
endwhitespace(""
endwhitespace(""
endwhitespace(""
```
Resulting System Behaviour

```
beginModule["example"]
begin["Procede"]

\text{temp\_correct[\_inc]} := \text{process[\_mixture\_correct[\_inc, \_mix]\_correct[\_inc]]}
\text{original\_mixture[\_inc]} := \text{process[\_mix]}$

\text{End["Procede"]}
\text{EndModule["example"]}
```

Analysis
Example 4:
Alarm System Menus
Menu Structure

```
import "classpath:units.str"
import "classpath:software.str"

namespace s1
  use units

condition locked
  condition blinkingLights

  menu Normal label "Standard Menu"
    item unlockNow apar(TimeoutAlarm) if locked
    button label "Unlock"
      item alarmLevel apar(alarmLevel)
      valerror SoundLevel restrict 10...20
      item soundLevel apar(TurnOffAlarm) if blinkingLights
      end
    menum autoLocking label "Automatic Locking"
      item startTime apar(TurnOffAlarm)
      valerror T30
      item endTime apar(TurnOffAlarm)
      valerror Time
      template areaSettings [size=0, area=, new=apar(TurnOffAlarm)] areaSettings
      template areaSettings [size=0, area=, new=apar(TurnOffAlarm)] areaSettings
    end
  end
  template [size: 0, area: 0, new: apar(TurnOffAlarm)] areaSettings
    iconOff apar(TurnOffAlarm) label "Unlock" on-off
    bool time = label(0) label "On" label "Off"
    item timer startTime label "Next"
    item alarmLevel apar(alarmLevel)
    valerror SoundLevel restrict size=20
  end
  menu Expert extends Normal
    item master apar(UnlockNow) afterItem unlockNow
    end
end
```

Software Components

```
message TurnOffAlarm
message TurnOnAlarm
message AlarmLevel
message UnlockNow

component AlarmManager {
  receives TurnOffAlarm
  receives TurnOnAlarm
  receives AlarmLevel
}

component MasterSwitch {
  receives UnlockNow
}
```
Eclipse Modeling
Eclipse Xtext

Example 5:
Requirements Tracability
Imported Requirements

@DummyRequirementCollection

- **Init**: The system should start operating only after it has been initialized property
- **Efficient**: The program should be as small regarding memory footprint as possible
- **Cyclo**: The actual control of the device should be based on a cyclic task
- **Calibration**: The black/white values should be easily calibrated
- **MaxSpeed**: Speeds per motor can only be up to 80
- **ResetMotorPorts()**: Display output should be optional
- **Initialization**: Initialization should be separate from operation
- **ConsistentSetting**: Motor settings have to be updated consistently

Program Code with Annotations (green)

```c
trace Cyclo
    // This is the cyclic task that is called every 1ms to do the actual control of the
    // task. Run Cyclo every 1 every 2.
    trace TwoPhases
        stateswitch linefollower
            state running
                int32 hung = 0;
                honour = erobot_get_touchsensor(SENSOR_PORT_T::BNT_PORT_02);
                if ( ! honour && ! hung ) { 
                    event linefollower::bumped
                        terminate;
                }
            trace Init
                int32 light = 0;
                light = erobot_get_light(ERobot::PORT_T::KNT_PORT_01);
                if ( light < ( WHITE + BLACK ) ) { 
                    trace ConsistentSetting
                        updateMotorSettings(SLOW, FAST);
                    } else if (trace ConsistentSetting
                        updateMotorSettings(FAST, SLOW);
                    }
            state crash
                updateMotorSettings(0, 0);
                default
                    <noop>;
```
Selecting from the Requirements

```java
fun cycle()
  doc: This is the cyclic task that is called every 1sec to do the actual control of the
  task
  task cycle prio = 1
```

Find Usages of Requirements

![Diagram of JetBrains MPS](image.png)

JetBrains MPS
Example 5: OSGi-based System

Component Specification

```java
subsystem the.world.scenario {
  public:
    // Some code here

  interface media {
    // Some interface methods here
  }

  interface phone {
    // Some interface methods here
  }

  private:
    component ModelA {
      // Some component methods here
    }

    component PrintingFAX {
      // Some component methods here
    }

  }

  subsystem the.house.scenario {
    subsystem the.hall.scenario {
      private:
    }

    component HomeAppliances {
      // Some component methods here
    }

  }
}
```
Component Specification

```java
subsystem the.world.scenario {
    public:
        Immutable type ProblemReport {
            problem: string
            severity: int
            executed: bool
        }

    interface Radio {
        export as ProblemReport; var;
    }

    interface Phone {
        broadcast as string; string
    }

    private:
        component Network {
            provides option: Radio
            requires phone: Phone
        }

        component PrintingPhone {
            provides source: Phone
        }
    }
}

subsystem the.won.scenario {
    uses the.won.scenario

    private:
        component Arctic {
            task: multiple extended nodeNetwork
crane: base radio @...}
    }
}
```

Component Specification

```java
subsystem the.world.scenario {
    public:
        Immutable type ProblemReport {
            problem: string
            severity: int
            executed: bool
        }

    interface Radio {
        export as ProblemReport; var;
    }

    interface Phone {
        broadcast as string; string
    }

    private:
        component Network {
            provides option: Radio
            requires phone: Phone
        }

        component PrintingPhone {
            provides source: Phone
        }
    }
}

subsystem the.won.scenario {
    uses the.won.scenario

    private:
        component Arctic {
            task: multiple extended nodeNetwork
crane: base radio @...}
    }
}
```
Example 6:
Math, Science and Java
env block Aircraft

\[ v : \text{double} \quad \frac{\text{m}}{\text{s}} \] / current aircraft speed

\[ A : \text{double} \quad [\text{m} \cdot \text{m}] \] / cross area of the wing

\[ c_s : \text{double} \quad [\text{]} \] / Auftriebsbeiwert

\[ c_w : \text{double} \quad [\text{]} \] / Widerstandsbeiwert

\[ n_{\text{wings}} : \text{double} \quad [\text{]} \] / Number of Wings

function block fundamental stuff

uses Aircraft, Environment

The dynamic pressure \( p_{\text{dyn}} \) is calculated from the current air density \( \rho \) and the square of the flight speed \( v \):

\[ p_{\text{dyn}} = \frac{1}{2} \cdot \rho \cdot v^2 \]

exported \( p_{\text{dyn}} : \text{double} \quad [\text{Pa}] \)

\[ \rho = 0 \Rightarrow 1225 \Rightarrow 0 \]

\[ v = 0 \Rightarrow 200 \Rightarrow 1225 \Rightarrow 245 \]

function block Stuff on the Wings

uses Environment, Aircraft, Fundamental Stuff

Aus dem Druck \( p_{\text{dyn}} \) lässt sich dann der aktuelle Auftrieb \( F_A \) berechnen; die Form wird auf der \( c_s \) beschrieben und die Fläche durch \( A \):

\[ F_A : \text{double} \quad [\text{N}] = p_{\text{dyn}} \cdot A \cdot c_s \]

\[ c_s \approx 0.3 \quad p_{\text{dyn}} \approx 1225 \quad A \approx 2 \Rightarrow 183.75 \]

\[ p_{\text{dyn}} \approx 1225 \quad A \approx 10 \quad c_s \approx 0.6 \Rightarrow 367.5 \]

Auch der Widerstand \( F_W \) berechnet sich entsprechend mit Hilfe des Beiwertes \( c_w \):

\[ F_W : \text{double} \quad [\text{N}] = p_{\text{dyn}} \cdot A \cdot c_w \]

\[ c_w \approx 0.5 \]

Angenommen haben mehrere Flügel \( n_{\text{wings}} \) an Flugzeug, dann berechnet sich der Auftrieb...
```java
import blocks Environment
    Aircraft
        Stuff on the Wings
public class TestClass extends <name> implements <name> {
    <static fields>
    <static initializers>
    <class>
    <properties>
    <initializers>
public TestClass() {
    <no statements>
}
public void init {
    values aix = 4; Environment.cbo = 1.225 $$;
    values planestatic = 41 Aircraft.A = 10, Aircraft.g.a = 0.6, Aircraft.g.w = 0.3, Aircraft.w = 100 $$;
    double auftrich = stuff on the wings r.A (air, planestatic);
    System.err.print(sauftrich);
}
    <static methods>
    <nested classifiers>
```
Example 7: Fountains
Hardware Structure

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

Behaviour

```
pumping program P1 for AtLeastOneZone + WithAlarm +
    SuperPowerCompartment/f=compl1 { scalar
    parameter defaultWaterLevel : int
    superWaterLevel: int
    event superPowerTimeout

    init {
        set compl1->targetHeight = defaultWaterLevel
    }

    start: on compl1->needsPower -- true & ! (compl1->isPumping) { scalar
        do compl1->pumpOn
    } on compl1->enough { scalar
        do compl1->pumpOff
    } on compl1.superPumping->turnedOn {
        set compl1->targetHeight = superWaterLevel
        raise event superPowerTimeout after 20
    } on compl1.superPumping->turnedOff or superPowerTimeout {
        set compl1->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

Tools

- Eclipse Modeling Project
- Xtext
- Yakindu
Tools

Tooling!
Editor, Debugger, Testing, Groupware, Scalable, „All in Eclipse“
Language Definition Tools
abstract syntax, concrete syntax, constraints

Editor Frameworks
Transformation Languages
Code Generation Tools
Current Version 2.0:
- Improved performance
- Xbase: expressions for reuse
- Xtend2: “Better Java”, with support for Xpand-like templates

Open Source (EPL)
Eclipse-based, Eclipse Project
Very flexible, very popular!
Open Source (Apache 2.0) Projectional Editor

Very good at lang. Composition

Current Version 2.0:

- 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
Automation
faster, deterministic

Increased Quality
well defined structures all through the system
Meaningful Validation
more semantics in the model

Capture
Domain Knowledge
formalized into languages and models
Suitable Notations

textual, graphical, tabular

Technology Independence

generate „technology glue code“
Abstraction w/o Runtime Overhead

generator „optimizes away“

Capture Implementation Strategy

in the generators
Everything is a model
including for example hardware (some) hardware
Airport Management System

Monitors Website Aircraft-Module

Data Center
component DelayCalculator {}
component InfoScreen {}
component AircraftModule {}

component DelayCalculator {
    provides IDelayCalculator
    requires IInfoScreen
}
component InfoScreen {
    provides IInfoScreen
}
component AircraftModule {
    provides IAircraftModule
    requires IDelayCalculator
}

interface IDelayCalculator {}
interface IInfoScreen {}
interface IAircraftModule {/}
component InfoScreen {
    provides IIInfoScreen
}

instance screen1: InfoScreen
instance screen2: InfoScreen
...

component DelayCalculator {
    provides default: IDelayCalculator
    requires screens[0..n]: IIInfoScreen
}

component InfoScreen {
    provides default: IIInfoScreen
}

component AircraftModule {
    provides default: IAircraftModule
    requires calculator[1]: IDelayCalculator
}
component DelayCalculator {
    provides aircraft: IAircraftStatus
    provides managementConsole:
        IManagementConsole
    requires screens[0..n]: IInfoScreen
}

component Manager {
    requires backend[1]: IManagementConsole
}

component InfoScreen {
    provides default: IInfoScreen
}

component AircraftModule {
    requires calculator[1]: IAircraftStatus
}

component DelayCalculator {
    requires screens[0..n]: IInfoScreen
    ...
}

component InfoScreen {
    provides default: IInfoScreen
}

instance dc: DelayCalculator
instance screen1: InfoScreen
instance screen2: InfoScreen

connect dc.screens
to (screen1.default, screen2.default)
namespace com.mycompany.production {
   instance dc: DelayCalculator

   // InfoScreen instances are created and
   // started in other configurations
   dynamic connect dc.screens every 60 query {
      type = IInfoScreen
      status = active
   }
}

interface IAircraftStatus {

   oneway message reportPosition
      (aircraft: ID, pos: Position )

   request-reply message reportProblem {
      request (aircraft: ID, problem: Problem,
                comment: String)
      reply (repairProcedure: ID)
   }
}

struct FlightInfo {
    from: Airport
    to: Airport
    scheduled: Time
    expected: Time
    ...
}

replicated singleton flights {
    flights: FlightInfo[]
}

component DelayCalculator {
    publishes flights
}

component InfoScreen {
    consumes flights
}

interface IAircraftStatus {
    oneway message registerAircraft(aircraft: ID!)
    oneway message unregisterAircraft(aircraft: ID!)
    oneway message reportPosition(aircraft: ID!,
        pos: Position!)
    request-reply message reportProblem {
        request (aircraft: ID!, problem: Problem!,
            comment: String!)
        reply (repairProcedure: !ID)
    }
    protocol initial = new {
        state new {
            registerAircraft => registered
        }
        state registered {
            unregisterAircraft => new
            reportPosition
            reportProblem
        }
    }
}
As you understand and develop your Architecture...

Develop a language to express it!
Language resembles architectural concepts

We express the application(s) with the language.
Architecture

Clear Understanding from building the language
Unambiguous Vocabulary

Concepts independent from Technology
Programming Model can be defined based on Conceptual Architecture

Architecture „executable“ (i.e. more than rules and docs)
Since we already have a formal model....

Generate API
Maps Architectural Concepts to Implementation language (non-trivial!)

Implementation Code

Programming Model API

Glue Code

Runtime Infrastructure (Platform/Middleware)
Implementation

Implementation only depends on the generated programming model API

- Implementation Code
- Programming Model API
- Glue Code
- Runtime Infrastructure (Platform/Middleware)

Programming Model

Generated API + Usage Idioms
Completely Technology-Independent

- Implementation Code
- Programming Model API
- Glue Code
- Runtime Infrastructure (Platform/Middleware)
**Runtime Infrastructure**

Select based on fit wrt. to architectural concepts and non-functional requirements

Implementation Code → Programming Model API → Glue Code → Runtime Infrastructure (Platform/Middleware)

---

**Glue Code**

Aka Technology Mapping Code
Maps API to selected platform

Implementation Code → Programming Model API → Glue Code → Runtime Infrastructure (Platform/Middleware)
Glue Code
Contains Configuration Files for Platform
Might require „mix in models“

Several Platforms
Different Platforms, not Languages
Support for Scaling (non-functional req)

Implementation Code
Programming Model API
Glue Code
Runtime Infrastructure (Platform/Middleware)

Implementation Code
Programming Model API
Glue Code
Glue Code 2
Runtime Infrastructure (Platform/Middleware)
Platform 2
Formal Architecture Description is not new: ADLs, UML

But all of those use existing, generic languages!
This misses the point!

Trying to express your specific architecture with predefined abstractions is not useful!
You want to build a language to capture your own architectural abstractions as you learn things.

Where are standards useful?
People have to learn architectural concepts anyway.

Is UML with a profile still a standard language?
On which meta level do I want to standardize?

M2 (UML), M3 (MOF)?

Isn’t a DSL based on MOF as „standard“ as a profile based on UML?
UML Profiles instead?
You’ll think more about UML-ities than your own concepts

UML Profiles instead?
Tool integration issues (repository, diff/merge, versioning)
UML Profiles instead?
Tools are often complex, heavyweight, bloated. Acceptance limited.

UML Generally Useless?
No. UML can be used for documentation (sequence diagrams, eg)
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?

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

Much better with new linguistic abstraction

```
// A'
for (int i in arr) {
    sum += i;
}
```
```
// B'
seqfor (int i in arr) {
    1.add( arr[i] );
}
```

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

Linguistic Abstraction
What's this? And what's the Problem?

```java
var linefollower_states_enum linefollower_currentstate = linefollower_states_enum::STATE_INITIALIZED;

enum linefollower_events_enum { EVENT_INITIALIZED, EVENT_BUMPED, EVENT_BLOCKED, EVENT_UNBLOCKED }

enum linefollower_states_enum { STATE_INITIALIZING, STATE_RUNNING, STATE_PAUSED, STATE_CRASH }

void linefollower_event_state(linefollower_currentstate_enum linefollower_currentstate, linefollower_events_enum linefollower_event) {
    if (linefollower_currentstate == linefollower_states_enum::STATE_INITIALIZING) {
        if (true) {
            linefollower_currentstate = linefollower_states_enum::STATE_RUNNING;
        }
    }
    if (linefollower_currentstate == linefollower_states_enum::STATE_RUNNING) {
        if (linefollower_event == linefollower_events_enum::EVENT_BUMPED) {
            if (true) {
                linefollower_currentstate = linefollower_states_enum::STATE_PAUSED;
            }
        }
    }
    if (linefollower_currentstate == linefollower_states_enum::STATE_PAUSED) {
        if (linefollower_event == linefollower_events_enum::EVENT_UNBLOCKED) {
            if (true) {
                linefollower_currentstate = linefollower_states_enum::STATE_RUNNING;
            }
        }
    }
}
```

Much better!

```java
statemachine 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>>
    }
}```
Much better!

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

Domains are Hierarchical

```
  ...
  D2-1-1  D2-1-2  ...
  D2-2-1  D2-2-2  ...
  D1-1    D1-2    ...
  ...
  D0
```
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.
Programming and Modeling
### Modeling
- *(Mostly) Graphical Notations*
- Abstract Syntax Storage
- Projecting Editors
- Different editable views for model

### Programming
- *(Mostly) Textual Notations*
- Concrete Syntax Storage
- *(Fancy) ASCII Editors*
- Read-Only Visualizations

---

**Why the difference?**
Programming the way we do Modeling?

Modeling the way we do Programming?

Modeling = Programming
Programming = Modeling
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
We don’t want to model, we want to program!

And always: precise and tool processable

Programming Languages are not MODULAR enough.
Programming Languages are not 
COMPOSABLE enough.

Programming Languages are not 
CONFIGURABLE enough.
Programming Languages are not ADAPTABLE enough.

Programming Language Syntax is not FLEXIBLE enough.
with many first class concepts!

with a few, orthogonal and powerful concepts
Modular Language

with many optional, composable concepts

Modular Language

Like frameworks and libraries, but with syntax and IDE support
Several Concerns in a Domain

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.
Incremental Extension of C with DSLs for Embedded Systems, integrated with Formal Methods and support for PLE and Requirements Tracing
Language Workbenches for Embedded Systems

http://mbeddr.com
EXAMPLE CASE

SOLUTION

```
1. This module represents the code for the line follower Lego robot. It has a couple
   modules main imports ClassKernel, EASt, BitLevelUtilities {
       constant int WHITE = 500;
       constant int BLACK = 700;
       constant int SLOW = 20;
       constant int FAST = 40;
   }

2. State machine to manage the line follower: {
    event init():
        state INIT;
    }

3. Event INIT:
    initial [true] -> running;
    state running[
        ]

4. Event INIT:
    INITIALIZE:
        SCREBOT_SET_LIGHT_SENSOR_ACTIVE;
    state INIT:
        state INIT;
    }

5. This is the cyclic task that is called every loop to do the actual control:

6. void main()
    {
        INIT:
            int32 light = 0;
            light = SCREBOT_GET_LIGHT_SENSOR(SENSOR_PORT_0, NOT_PORT_0);
            if (light <= (WHITE + BLACK) / 2) {
                UPDATE_MOTOR_SETTINGS(FINE, FAST);
            } else {
                UPDATE_MOTOR_SETTINGS(FAST, SLOW);
            }
    }
```

```
This procedure actually configures the motors based on the speed values
void UPDATE_MOTOR_SETTINGS(int left, int right) {
    set_motor_speed(MOTOR_PORT_0, NOT_PORT_0, left, 1);
    set_motor_speed(MOTOR_PORT_0, NOT_PORT_0, right, 1);
    }
```java
exported interface MotorControl {
    void stop( );
    void setLeftSpeed( int8 speed );
    void setRightSpeed( int8 speed );
}

exported component Motors {
    provides motorControl : MotorControl;
}

    exported component implementation MotorsNXT : Motors {
        procedure void motorControl.stop( ) {  
            nxt_motor_set_speed(MOTOR_PORT_V : NXT_PORT_B, 0, 1);
            nxt_motor_set_speed(MOTOR_PORT_W : NXT_PORT_C, 0, 1);
        }
        procedure void motorControl.setLeftSpeed( int8 speed )  {
            nxt_motor_set_speed(MOTOR_PORT_V : NXT_PORT_B, speed, 1);
        }
        procedure void motorControl.setRightSpeed( int8 speed )  {
            nxt_motor_set_speed(MOTOR_PORT_W : NXT_PORT_C, speed, 1);
        }
    }

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

    robot script stopAndGo  
    block main on bump block retreat on bump <no bumpReaction>  
        step  
            accelerate to 0 - 30 within 2000
            drive on for 2000
            decelerate to 0 within 1000
            step  
            accelerate to speed(25) within 3000
            drive on for 2000
            turn left for 2000
            block driveMore on bump <no bumpReaction>  
                accelerate to 80 within 2000
                turn right for 3000
                decelerate to 0 within 3000
                step  
            }
```
JetBrains MPS

First C Code working
July 17, 2011 by rsptevo

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 it much more thoroughly. As you can see in the screenshot below, some essential things are already working.
Bonus: Best Practices

Limit Expressiveness
Notation, Notation, Notation

Graphical vs. Textual
Interpretation vs. Generation

Rich Domain Specific Platform
Checks First
and Separate

Cascading
Annotation Models

Don’t forget Testing
Iterate!

Co-Evolve Language and Concepts
Domain Users Programming?

Compatible Organization
THE END.

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