Domain-Specific Languages as Rosetta-Stone for Engineering Cognitive Systems

Dr.-Ing. Sebastian Wrede, MLR Colloquium, Stuttgart, 01.07.2013
Goal: Development of interactive cognitive systems which learn by interacting

From islands of functionality to integrated technical systems

Functional architecture & system integration are major challenges [1]

Means: cognitive systems engineering

Hypothesis test cycle in advanced robotics projects is extremely complex.

System-level hypotheses are often formulated only implicitly.

Experiments realized often in an ad-hoc fashion and with limited reproducibility.

Better engineering methodology required to answer architectural questions.

New system analysis approaches required to understand and monitor learning machines.

Agile, but reproducible?
Cognitive Systems Engineering Challenges

- Cognitive systems projects are inherently multidisciplinary (true as well in industry?!)
- Different domains & individual expertise
- Handling of inherent & accidental complexity
- Development process (i.e. iterative vs. sequential)
- Staying flexible while keeping implementation quality
- Agility is a must for collaborative (research) projects due to the nature of ongoing research

https://www.cit-ec.de/events/summerschool2013
Cognitive Systems Engineering
Topics and Activities

I. Modeling
II. Construction
III. Analysis
IV. Application

http://www.cor-lab.de/cognitive-systems-engineering
Cognitive Systems Engineering
Topics and Activities

I. Modeling

II. Construction

III. Analysis

IV. Application

http://www.cor-lab.de/cognitive-systems-engineering
Planning, development & maintenance of a system requires understanding of:

- **Domain** (domain experts, e.g. customer and algorithm engineers)
- **System structure** (system architects)
- Many other aspects...

**Domain-specific Models** capture and link such knowledge in a formal way.

Our focus is on software modeling and we aim at:

- Tight integration of models with executable code
- Utilization of automated software engineering techniques
Model-driven Software Development with Domain-Specific Languages

- A Domain-Specific Language (DSL) is created specifically to **solve problems in a particular domain**
- **Natural abstractions and notation for domain experts**
- In contrast to general-purpose languages, DSLs are **semantically richer than GPL programs**, emphasis on meaning
- **Hide implementation details, platform independent**
- **Allow for semantic analysis vs. mere syntax checking** (correct by construction)

**Code generation**
- DSL models are **incrementally translated** to code (through Model-to-Model & Model-to-Text transformations)
Domain-Specific Languages
Main Types and Robotics Examples

**Internal**
- Reuses syntax & semantics of host language
- Examples: Lisp, Ruby, ...

**External**
- Custom Syntax
- Graphical, Textual, Matrix, ...

DSLs have a long tradition, but were typically hard to develop!

Frigerio et al. 2012

Lego NXT-G

(C) M. Fowler
Language Workbenches for DSL Developers and Users

- Language and “DSL-IDE” development env.
- (Projectional) model editing
- Language modularization & composition
- Code generation
- Java, C(++), XML, ... base languages

(C) Martin Fowler
Example: A Domain-Specific Language for Movement-Control Architectures

- **Domain:** Modular Motor Skill Architectures for movement control with dynamical systems
- **Movement primitives:** periodic or goal-directed movements modeled as **Dynamical Systems**
- **Complex movements through composition**
- **Flexibility by adding machine learning techniques**
- **Primitives applicable to different types of robots**
- **Challenges** for hypotheses testing
  - Domain utilizes **differing terminology** & informal architecture models
  - Architectures hard to compare
  - Large **gap between descriptions and code level**
Example: A Domain-Specific Language for Movement-Control Architectures

Core concepts identified in domain analysis:

- **Dynamical Systems** provide the dynamics of movements
- **Adaptive Modules** combine Dynamical Systems and Machine Learning alg. to resemble a *Movement Primitive*
- **Adaptive Components** are recurring patterns of control logic around an Adaptive Module, e.g. *Pattern Generator, Tracking Controller*, ...
- **Control Spaces** are sets of explicit variables jointly manipulated or sensed
- **Mapping and Transformations** as operations between spaces
- ...

---

"Skill Sequencer"<br>
"TimedProcessing"

"Skill Control"<br>
"Control"<br>
"3-dim.

"Right Hand"<br>
"c.r. Translation"<br>
"3-dim.

"Weight Up Control"<br>
"Control"<br>
"1-dim.

"Weight Down Control"<br>
"Control"<br>
"3-dim.

"Left Hand"<br>
"c.r. Translation"<br>
"3-dim.

"Adaptive Module"<br>
"SVML Weight Up"<br>
"Dynamical Systems: VelocityField"

"Adaptive Module"<br>
"SVML Weight Down"<br>
"Dynamical Systems: VelocityField"

"Adaptive Module"<br>
"SVLM Padle Left"<br>
"Dynamical Systems: VelocityField"

"Adaptive Module"<br>
"SVLM Padle Right"<br>
"Dynamical Systems: VelocityField"

--

"ZeroVelocity"

"ZeroVelocity"

"InvKin"

"InvKin"

"InvKin"

"c.r. JointAngles"<br>
"14-dim.

"c.r. JointAngles"<br>
"14-dim.

"c.r. JointAngles"<br>
"14-dim.

---

www.uni-bielefeld.de
A Domain-Specific Language for Movement-Control Architectures

```plaintext
AMARSi Circuit Demo {

Beat:
   Beat heartbeat (every 10 ms)

Components:
   lfcctrl<Onicilla Leg> ctrl left fore |
   rfctrl<Onicilla Leg> ctrl right fore |
   lhctrl<Onicilla Leg> ctrl left hind |
   rhctrl<Onicilla Leg> ctrl right hind |
   | left fore angles  | left hind angles  |
   | right fore angles | right hind angles |

State Spaces:
   State Space Command Left Fore (spacetype=Onicilla Leg rootscope=/onicilla/ctrl/leftfore/) Properties: << ... >> Connections:
      <- outgoing to component lfcctrl
   State Space Command Right Fore (spacetype=Onicilla Leg rootscope=/onicilla/ctrl/rightfore/) Properties: << ... >> Connections:
      <- outgoing to component rfctrl
   State Space Command Left Hind (spacetype=Onicilla Leg rootscope=/onicilla/ctrl/lethhind/) Properties: << ... >> Connections:
      <- outgoing to component lhctrl
   State Space Command Right Hind (spacetype=Onicilla Leg rootscope=/onicilla/ctrl/lethhind/) Properties: << ... >> Connections:
      <- outgoing to component rhctrl
   State Space Status Left Fore (spacetype=Onicilla Leg rootscope=/onicilla/status/lefthind/) Properties: << ... >> Connections:
      -> ingoing from component lstatus
   State Space Status Right Fore (spacetype=Onicilla Leg rootscope=/onicilla/status/lefthind/) Properties: << ... >> Connections:
      -> ingoing from component rstatus
   State Space Status Left Hind (spacetype=Onicilla Leg rootscope=/onicilla/status/lefthind/) Properties: << ... >> Connections:
```

Towards Simplified Hypothesis Testing with Domain-Specific Modeling

System description
(technology/platform-independent)

Experiment generation
including computational aspects
(for target platform)

Execution and validation
(in simulation or hardware)

Cognitive Systems Engineering
Topics and Activities

I. Modeling

II. Construction

III. Analysis

IV. Application

http://www.cor-lab.de/cognitive-systems-engineering
Construction
How to Get from Models to Systems?

Not everything required to execute experiments is inside the DSL (Option 0), so what are the targets of code generation?

Option 1: Runtime parsing of component models
  - Component types (according to DSL concepts)
  - Input / output ports (for control spaces)
  - Deployment details (for code generation)
  - Workbench loads component descriptions at design time & makes them available as DSL models

- Code generation targets platform integration
  - Communication code (if necessary)
  - Coordination logic (processing strategies, ...)
  - Configuration (and parametrization)
Construction
How to Get from Models to Systems?

Not everything required to execute experiments is inside the DSL (Option 0), so what are the targets of code generation?

Option 2: Abstractions available in a DSL and as functional libraries (e.g. OpenCV, Eigen, ...)

- Generators target (object-oriented) library APIs
- Transforms domain-specific code into **building blocks conforming to DSL-based architecture**
- **Pure forward generation** of code, no reverse engineering

- **Code generation targets architecture integration**
  - Communication code (if necessary)
  - Coordination logic (processing strategies, ...)
  - Configuration (and parametrization)
  - **Additional instrumentation** (e.g. OpenCV previews)
Construction Guidelines for (DSL-friendly) Library Design

Usability inside DSL imposes requirements on libraries

- Usage in an environment usually not envisioned by developers
- External programmatic control and configuration
  - e.g., inputs and outputs
- Decoupling from a specific control architecture
- Abstraction from communication frameworks and hardware I/O

Clear Interface Abstractions
High Cohesion
Separation of Concerns
Dependency Injection

Images (C) http://www.planetgeek.ch
Construction
Robotics Service Bus (RSB)

- Lightweight **event-driven middleware**
- Realizes **ESB concept for robotics** systems
- Full **observability / introspection** for analysis
- **Cross-platform** & language (c++, py, java, cl)

- Multiple **extension points**
  Serialization, Transport, Transformation, Filtering, Threading & Configuration

---

https://code.cor-lab.org/projects/rsb

[Wienke et al. 2011]
Construction
Rosetta: DSL-based Interoperability

**Problem**
- Imported **components use different data types**
- Manual bridging is expensive

**Solution**
- DSL-based **type mapping and code generation** [Wienke et al. 2012]
- Operates on type descriptions

- **data-holder:**
  - `rst.kinematics.JointAngles`
- **wire-schema:** `yarp.icub.torso.command`

- **unpack-rules:**
  - `len(.angles) = 3`
  - `.angles[0] = .angles.a0`
  - `.angles[1] = .angles.a1`
  - `.angles[2] = .angles.a2`
Enabling Agile Development
Continuous Integration and Testing

- Modularized **DSL-based architecture facilitates agility**, but **needs testing** (build, deployment, development issues)
- Manual testing and monitoring not efficient and error prone

(C) http://www.adfkickstart.com
Benefit: continuous monitoring of source code changes

- Continuous error checking
  - Builds on different platforms
  - Unit tests
  - Checks of downstream projects
  - Immediate feedback to developers

- Continuous analysis
  - Static code analysis
  - Evaluation of runtime analysis results (e.g. code coverage)
  - Also: immediate feedback

Challenges
- Support non-Java projects
- Simulation-based testing
- Hardware in the Loop
- Scalability and maintenance
Enabling Agile Development
Scalable Ecosystem Required

Our Current Setup

- > 350 jobs
- Many dependencies
- Job information present in other systems, too:
  - Dependency structure
  - Build instructions
  - Repetitive aspects across jobs
  - Maintenance issue

Idea

Build job descriptions from DSL!
Enabling Agile Development
Continuous Integration as a Central Component
Enabling Agile Development
Continuous Integration as a Central Component
Enabling Agile Development
Continuous Integration as a Central Component
Enabling Agile Development
Continuous Integration as a Central Component

CoR-Lab – Research Institute for Cognition and Robotics
Enabling Agile Development
Continuous Integration as a Central Component
Cognitive Systems Engineering
Topics and Activities

I. Modeling

II. Construction

III. Analysis

IV. Application

http://www.cor-lab.de/cognitive-systems-engineering
Application and Experimentation
Test for Methodology and System Ideas

- Focus on application & evaluation of engineering methodology of novel system hypotheses incorporating CITk methods both in fundamental and applied research projects

- Experimentation typically requires real users

- Larger # of interactions / long-term operation (week)

- Consideration of many stakeholders

Example: FlexIRob Field Study
Idea: Simplify kinesthetic teaching in task and configuration space such that implicit constraint modeling is possible.

Application and Experimentation
Example: FlexIRob Field Study

Programming Experiment
- Inspired by “wire-loop” game
- Teach-in of task trajectory in confined space with redundant robot manipulator
- Assisted / non-assisted condition

Example 1
FlexIRob - A Compliant Robot System with Dedicated Support for Decomposed PbD

Three essential interaction modes
• Based on LWRIV joint impedance

Configuration phase [1]
• Gravity Compensation to configure pose in a specific workspace area
• Compliant Recording for recording training samples
• Neural network (RNN) is trained w/ samples

Three essential interaction modes

‣ Based on joint impedance mode of KUKA LWR IV

Configuration phase [1]

‣ User trains a neural network in kinesthetic teaching

Gravity Compensation to configure in joint space

‣ Compliant Recording to acquire training data

FlexIRob - A Compliant Robot System with Dedicated Support for Decomposed PbD

FlexIRob - A Compliant Robot System with Dedicated Support for Decomposed PbD

Three essential interaction modes

‣ Based on LWRIV joint impedance

Configuration phase [1]

‣ Gravity Compensation to configure pose in a specific workspace area
‣ Compliant Recording for recording training samples
‣ Neural network (RNN) is trained w/ samples

Assisted programming phase [2]

‣ Assisted Gravity Compensation to teach a trajectory in task space
‣ Trained neural network provides redundancy resolution
‣ Hierarchical controller fuses task & joint space

FlexIRob User Study
Phases and Conditions

(1) Warm-up Phase
- Familiarize w/ interaction modes
- Video instructions
- 2 min free interaction

(2) Configuration Experiment
- To test environment modeling
- 2 workspace areas
- Feedback with reference trajectory
- 3 trials w/ escalating instructions

(3) Programming Experiment
- Inspired by “wire-loop” game
- Styrofoam parcour as target trajectory
- Assisted / non-assisted condition
- Assisted group utilized pre-trained controller

(4) Questionnaire

Field Experiment
- Conducted in March 2012
- On-site at HARTING [1] Technology Group
- Automatic production with manual assembly
- 49/50 participants actually took part in the experiment
- Final sample: 44 participants, 15 males and 29 females

Programming Experiment
Example 2 - Condition: Assisted
Our expectations in terms of
- improved task-space **accuracy**
- **time** required for programming
- **collision avoidance** in joint space
in the assisted condition (using AGC) were **entirely** confirmed.
# Questionnaire
## Subjective Assessment of Interaction

### General Experience

<table>
<thead>
<tr>
<th>Question</th>
<th>Mean</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do you think the robot is threatening?</td>
<td>4.27</td>
<td>1.06</td>
</tr>
<tr>
<td>How intelligent do you rate the robot?</td>
<td>1.78</td>
<td>0.80</td>
</tr>
<tr>
<td>How reliable do you think the robot is?</td>
<td>2.07</td>
<td>0.90</td>
</tr>
<tr>
<td>Do you think the feedback was helpful?</td>
<td>1.57</td>
<td>0.95</td>
</tr>
</tbody>
</table>

1: yes/very much  
...  
5: no/not at all

### Main effects (ANOVA)

<table>
<thead>
<tr>
<th>Effect</th>
<th>F</th>
<th>df</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>on simplicity of handling (w/o covariates)</td>
<td>8.59</td>
<td>1.42</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>on reasonableness (w/o covariates)</td>
<td>3.34</td>
<td>1.42</td>
<td>&lt; 0.10</td>
</tr>
<tr>
<td>on simplicity of handling (with covariates)</td>
<td>3.17</td>
<td>1.37</td>
<td>&lt; 0.10</td>
</tr>
<tr>
<td>on instinctiveness of handling</td>
<td>2.83</td>
<td>1.42</td>
<td>&lt; 0.10</td>
</tr>
<tr>
<td>of knowing industrial robots on perceived threat</td>
<td>7.58</td>
<td>1.42</td>
<td>&lt; 0.01</td>
</tr>
</tbody>
</table>

Covariates included:  
- employment time  
at HARTING  
- experience with  
- computers  
- spatial vision  
- spatial imagination

---

Methods & Tools used in FlexIRob

- Continuous Integration
- Simulation-based testing
- Continuous Deployment
  - Reduction of system setup time
  - Ensures quality of rolled-out components
- Utilizes DSL-based job description
- Component architecture modeled in DSL
  - Utilizes RST type language*
- Generated aspects
  - FSM-based coordination logic
  - Components and executables
- Analysis tools for introspection, recording and transformation of experiment data [1]

---

Conclusion

**DSLs, a Rosetta Stone for Systems Engineering?**

Well, challenges remain...
- Finding the right **domain abstractions**
- **Learning curve** for users / developers
- **DSL modularization & composition**
- Fixing your **development process**

**Further aspects**
- Many technical aspects...
- **Platform representation** reusing existing DSLs
- Research on **modular DSLs for cognitive robotics**
- Extension of **behavioral aspects** towards interactive learning
- Exploitation of model-based information in analysis & testing

---

Paul Hudak (1998)
Conclusion

DSLs, a Rosetta Stone for Systems Engineering?

Sure, it helps!

☑ Forces stakeholders to agree on concepts & domain terminology
☑ Promotes understandability of models in a multidisciplinary project team
☑ Separation of concerns and roles
☑ DSL models make the implicit explicit
☑ Agile development with a DSL-based architecture helps to stay flexible

Take home message

➤ Towards a shared collection of composable DSLs for robotics
➤ Focus on language concepts, not on specific APIs

Thank you for your attention!