Anatomy of a domain-specific language project in an industrial context

Development and examination of a DSL demonstrator for elevator controllers

Christoph Wienands & Michael Golm
Software Engineering, Architecture and Platform Technologies
Siemens Corporate Research, Princeton, NJ
Overview

- Motivation and goals
- Elevator controllers as research subject
- Domain analysis
- Inception of new, abstract domain concepts
- DSL Tooling Architecture
- Meta-model
- Encountered challenges
- Support for another target platform (C) and code generator optimization
- Collected metrics
Background of DSL project for elevator controllers

**Motivation**
- Introduction of DSLs into organizations is difficult.
  - Learning curve
  - Language and model evolution
  - Initial investment
- Development of visual DSLs is a labor-intensive task.
  - Experience of DSL authors
  - Difficulties finding appropriate visual representations
- Complexity: Multi-diagram DSLs, constraints, customization, visual-text hybrids
- Generated code said to perform poorly.

**Goals**
- Gain more experience building DSL, especially using Eclipse GMF.
- Collect metrics throughout development and evaluate them.
- Examine challenge of generation of performant code.
- Develop a fully documented, easy-to-understand, publishable DSL and tool chain for research purposes and as a showcase.

Pretty much everything Steve Mellor said in Keynote.
Elevator controllers – Research subject for DSLs

- Elevators are grouped in elevator banks
- High-level control consists of: Opening/closing doors, accelerating/decelerating individual cars, and reacting to floor calls/car calls (buttons pressed inside cars and outside on floors).
- Floor calls require the concept of committed direction: The direction an elevator car will continue in after opening and closing the doors at the next stop.
- Support for multiple special modes: Maintenance, fire-fighter mode, etc.

→ Elevator controller domain is non-trivial and provides enough variability to justify the development of a DSL, yet it is strictly scoped.
Domain analysis of elevator controllers

Elevator controller functionality is separated into:

- Common behavior shared across all controllers
  - Accelerating / decelerating to reach target floor
  - Closing of doors: Closing delay, reopen if obstruction detected
  - Releasing of pressed buttons upon reaching target floor
- Variable, mode-specific behavior:
  - Determining next target floor
  - Setting of committed direction
  - Decision to open doors (only if stopped)

- Common behavior is mostly safety-critical and will not be part of the modeling language.
- Actions strongly depend on state of elevator components and events from subsystem.
- Modeling language based on state machines seems most suited.
Advanced domain concepts in elevator controller DSL

- **2 named call lists**
  - Dynamic lists providing trigger events 'Added' and 'Removed' to state machine transitions
  - *Elevator calls*: Buttons pressed inside car
  - *Assigned floor calls*: Assigned by runtime

- **Call list projections**
  - Dynamic filter queries.
  - Filter configurations: Calls ahead, calls behind, all calls, same direction, opposite direction.
  - Filter result depends on current position, speed, acceleration and committed direction.
  - Can combine calls from *ElevatorCalls* and *AssignedFloorCalls*.
  - Provide events just like underlying call lists.

### Call List Projection Example

*CallsAhead – SameDirection*

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>▲</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>▲</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>▲</td>
<td>◼</td>
</tr>
<tr>
<td>4</td>
<td>▲</td>
<td>◼</td>
</tr>
<tr>
<td>3</td>
<td>▲</td>
<td>◼</td>
</tr>
<tr>
<td>2</td>
<td>▲</td>
<td>◼</td>
</tr>
<tr>
<td>1</td>
<td>▲</td>
<td>◼</td>
</tr>
</tbody>
</table>

- Four persons waiting on floors
- One passenger in car, target floor 6
- Elevator speed 15ft/s, next possible stop 4

**Result of call list projection query: 4, 6**
# DSL Tooling Architecture

## Technology Description

<table>
<thead>
<tr>
<th>Technology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphical Modeling Framework (GMF)</td>
<td>Eclipse-based workbench for visual DSLs</td>
</tr>
<tr>
<td>Eclipse Modeling Framework (EMF)</td>
<td>Underlying (meta) model library</td>
</tr>
<tr>
<td>Graphical Editing Framework (GEF)</td>
<td>Underlying library for editors</td>
</tr>
<tr>
<td>Java Emitter Templates (JET)</td>
<td>Code generation, model-to-text</td>
</tr>
<tr>
<td>openArchitectureWare Xtext</td>
<td>Parser for mini scripting language</td>
</tr>
<tr>
<td>openArchitectureWare Xpand</td>
<td>Code generation for scripting language</td>
</tr>
</tbody>
</table>

---

![Diagram](image)
Example of ‘Up-Only’ elevator controller

Call lists

Call list projections

State machine

Conditions

Actions
Encountered challenges

- Steep learning curve for GMF.
- Implementing an efficient concrete syntax:
  - Weak support for decomposition (subdiagram of different type) in GMF
  - Complexity of mapping ecore meta model to concrete syntax due to EMF flexibility (trade-off)
- Customization of DSL editor:
  - Implemented customizations
    - Read-only, calculated attribute
    - Propagate attribute and reference change notifications
    - Initialize model graphs and prepopulate diagrams upon creation
    - Context menu
    - Integration visual DSL and textual scripting language
  - Most customizations needed to be done manually by modifying or supplementing generated code. Little tool support.
Support for alternative C platform: Optimizations performed by code generator

- Statically allocated arrays
  - Constant memory footprint
- Statically linked event chains
- Not shown: Optimized filter queries for call list projections
- Performant execution of state machine
- Conditional exclusion of status updates
  - Faster execution for simple controllers

```cpp
double* m_ElevatorSpeed;
void tick() {
    m_ElevatorSpeed = new double[m_ElevatorCount];
}

void elevatorCallsAdded(int elevator, int floor) {
    ... elevatorCallsAheadAdded(elevator, floor); ...
}

void elevatorCallsAheadAdded(int elevator, int floor) {
    ... if (m_State[elevator] == STATE_MOVING &&
           movingCallsAheadAdded4Condition(elevator)) {
        movingCallsAheadAdded4Action(elevator);
    } ...
}

for (int i = 0; i < m_ElevatorCount; i++) {
    #ifdef STATUS_SPEED
        m_ElevatorSpeed[i] = m_Sim->getElevatorSpeed(i);
    #endif
}
```

Copyright © Siemens AG 2009. All rights reserved
Christoph Wienands SCR / SE / APT
# Some Collected Metrics

## Effort for creation of a elevator controller by domain expert vs. domain novice

<table>
<thead>
<tr>
<th>Development method</th>
<th>Effort expert (hours)</th>
<th>Effort novice (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using Elevator DSL (therefore works on both target platforms)</td>
<td>1.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Manually developed against C domain framework</td>
<td>1.5</td>
<td>-</td>
</tr>
<tr>
<td>Manually developed for C platform (no domain framework)</td>
<td>4.0</td>
<td>8.0+ (out of time, believed to be at least 12h until complete)</td>
</tr>
</tbody>
</table>

## Efficiency of generated code

<table>
<thead>
<tr>
<th>Metric</th>
<th>DSL-based controller</th>
<th>Manually dev. controller</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Static analysis</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC</td>
<td>791 domain framework + 462 generated controller = 1253 total</td>
<td>508 controller</td>
</tr>
<tr>
<td>Binary size</td>
<td>56kB</td>
<td>40kB</td>
</tr>
<tr>
<td>Implementation effort</td>
<td>4 call list projections, 5 states, 13 transitions and 29 lines of TSL script (conditions, actions)</td>
<td>508 lines of C code</td>
</tr>
<tr>
<td><strong>Runtime analysis</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg. instructions/cycle</td>
<td>184</td>
<td>223</td>
</tr>
</tbody>
</table>
Conclusion and future work

Findings
- Domain analysis and prototyping yielded two domain concepts not found in existing elevator controllers → raising level of abstraction for users.
- Used many Java/Eclipse modeling technologies to create user-friendly DSL → additional experience
- Quantitative analysis of collected metrics support previous findings that DSLs increase productivity.

Future work
- Port DSL to other visual DSL workbenches for comparison
- Extend DSL to support modeling call distribution algorithms across elevator bank.
Thank you for your attention

Christoph Wienands
Christoph.wienands@siemens.com