Code generation

- Automatic transformation of domain specific models into software models
- Automatic translation from software models into executable code
- Ingredients:
  - syntax and semantics of modeling formalisms should be described
  - correctness preserving transformation steps should be defined
  - code generators should be developed
Code generation

- **Advantages**
  - *Increase in productivity:*
    - generating tedious and boring parts of the code
    - code generators produce thousands of lines of code in seconds
    - changes are quickly propagated
    - agile development
  - *Increase of quality:*
    - bulky handwritten code tend to have inconsistent quality because increase of knowledge during development
    - bug fixes and code improvements can be consistently rolled out using a generator
Code generation

• Advantages
  • *Increase of Consistency*:  
    – in API design and naming convention  
    – single point of definition  
    – explicit documented design decisions  
  • *Architectural consistency*:  
    – Programmers work within the architecture  
    – Well-documented and -maintained code generator provides a consistent structure and approach  
  • *Abstraction: language-independent definition*  
    – Lifting problem description to a higher level  
    – Easier porting to different languages and platforms  
    – Design can be validate on an abstract level
Model transformations

- **EGL: Overview**
  - Model-to-text transformation language
  - Two types of sections
    - Static: content appears verbatim in generated text
    - Dynamic: executable code (EOL)
- **Templates**
  - Generate files
- **Protected regions**
- **Beautification**
Model transformations

• EGL – A Template Language
  • EGL is a template language (e.g. PHP)
    [% for (i in Sequence{1..5}) { %]
    i is [%= i %]
    [% } %]

• Dynamic sections: contents executed
• Static sections: contents appear verbatim in output
Model transformations

- EGL- Preprocessor for EOL
  - EGL is minimally derived from EOL
    
    ```
    [% for (i in Sequence{1..5}) { %]
    i is [%= i %]
    [% } %]
    ```
  - becomes:
    ```
    for (i in Sequence{1..5}) {
    out.print('i is '); out.println(i);
    }
    ```
  - produces:
    ```
    i is 1
    i is 2
    i is 3
    i is 4
    i is 5
    ```
Model transformations

• **EGL – feature summary**
  • Common M2T language features:
    – Support for defining and utilising protected regions
    – Beautification
    – Traceability
  
• **Novel / uncommon features**
  – Co-ordination engine: encourages decoupling
  – Strong integration with other model management languages
Model transformations

• EGL – readability
  • Templates should be readable
    – But so should generated text

• Philosophy: make templates readable
  – And run a post-processor on the generated text

• Beautifiers provided for Java and XML
  – Extensible; invoked via Epsilon workflow
  – Similar concept available in Xpand
Model transformations

- **EGL example: Nixon’s enemy list**

  ```plaintext
  [% var rmn = EnemyMap!Enemy.all().
      select( e | e.name = 'Richard Nixon' ); %]
  
  Richard M. Nixon\’s Enemy List
  ---------------------------------------
  
  [% for ( e in rmn.isEnemyOf ) { %]
    I hate [%= e.name %]
  [% } %]
  
  will produce:
  
  Richard M. Nixon’s Enemy List
  ---------------------------------------
  
  I hate Dick Dastardly
  I hate the Novels of Jacqueline Susann
Model transformations

• Conclusions
  • No guarantee on correctness of generated code, see book “Code Generation with Templates”
  • For more information see Chapter 7 of EpsilonBook.
Overview of workshop

- Model driven software engineering in general
- Grammars, signatures and meta-models
- Model transformations and code generation
- DSL Design
DSL Design

Model-driven engineering

Goal:
• Raising the level of abstraction
  … from the computing domain to the problem domain

MDE combines:
• Domain-specific modeling languages
• Model transformations

“a language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain” [2]

Examples of DSLs

- HTML
- SQL
- MediaWiki
- (La)TeX
- PROMELA (SPIN)
- YACC
- ATL
- ASF+SDF
- EBNF
- Flash
- …
## DSL Design

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression of the solution in terms of domain concepts</td>
<td>Cost of DSL implementation and education</td>
</tr>
<tr>
<td>Enhanced productivity, reliability, maintainability, and portability</td>
<td>Difficulty of finding the right scope and balancing between domain-specificity and general-purpose constructs</td>
</tr>
<tr>
<td>Domain knowledge contained in language</td>
<td>Solutions for a limited set of problems</td>
</tr>
<tr>
<td>Mostly concise and largely self-documenting</td>
<td>Potentially less efficient code</td>
</tr>
</tbody>
</table>
Domain-specific language design:

1. Domain analysis
2. Language design
3. Language implementation
DSL Design

Domain Analysis

Identifying the objects, operations and relationships between them

“A problem domain is defined by consensus, and its essence is the shared understanding of some community” [3]

DSL Design

Language Implementation Strategies

- Stand-alone
- Embedding
- Translation
• Static semantics is needed for:
  • catching errors before code generation

• Static semantics defines restrictions on syntactically valid programs:
  • cannot be expressed in context-free grammar
  • calculated before executing a program
  • rules for:
    – scoping
    – identification of variables, functions, etc.
    – type systems
    – data/control flow
• Xtext provides primitives for basic tasks
  • identification
  • basic scoping

• Alternatives to define static semantic rules
  • use XTypeS
    – experimental and only simple rules
  • use standard model transformation language like ATL
    – verbose and again simple rules
  • use Java
    – flexible but complex and involved
DSL Design

• Type systems
  • A type system of a (programming) language groups values into types
  • Prevents illegal operations, like multiplication of strings by booleans: type error

• Statically typed language: each variable and expression has a fixed type
  – all operands can be type-checked at compile-time
• Dynamically typed language: values have fixed type, but variables and expressions have no fixed type.
  – operands can be only type-checked at run-time
What data and their types will be used by the DSL?

Values are entities manipulated by programs
Different types of (programming) languages support different types of values

Values are grouped into types
A type is a set values with operations that can be applied uniformly to all these values
DSL Design

• **Primitive types**
  • *Primitive value* can not be decomposed into simpler values

• *Primitive type* is a type of which the values are primitive
  - Booleans = {‘true’, ‘false’}
  - Integer = {…, -2, -1, 0, 1, 2, …}
Lifetime of a variable is the time between creation (allocation) and destruction (deallocation)

- Global variable’s lifetime is the program’s run-time
- Local variable’s lifetime is an activation of a block
- Heap variable’s lifetime is arbitrary, but maximum is program’s run-time
- Persistent variable’s lifetime is arbitrary and not restricted to program’s run-time
• A global variable is a variable that can be used anywhere in a program
• A local variable is only available within the block where it is declared
  • A block is a program construct that includes local declarations
  • An activation of a block is the time interval that the block is executed
• **Bindings and environments**
  
  • If identifiers occur in an expression, such an expression cannot be understood in isolation:
    - its meaning depends on the declarations of these identifiers elsewhere in the program

  • A *binding* is an association between an identifier and an entity such as value, variable or procedure

  • An *environment* (or *name space*) is a set of bindings
The scope of a declaration is the part of the program where the declaration is effective.

A block is a language construct that delimits the scope of declarations within it.
- monolithic block structure, e.g. Cobol
- flat block structure, e.g. Fortran
- nested block structure, e.g. Algol-like language, C, Java
Blocks:

- A block command is a form of command that contains a local declaration \( D \) and a subcommand \( C \)
  - In C/C++ and Java: \( \{ D \ C \} \)
    
    ```
    if (x > y) \{ int z = x; x = y; y = z; \}
    ```

- A block expression is a form of expression that contains a local declaration \( D \) and a subexpression \( E \)
  - Haskell provides block expressions: \( \text{let } D \ \text{in } E \)
**Scope and visibility**

- A *binding occurrence* of identifier $I$ is an occurrence where $I$ is bound to some entity $X$.
- An *applied occurrence* of $I$ is an occurrence where use is made of the entity $X$ to which $I$ is bound.
- Each applied occurrence of $I$ should correspond to exactly one binding occurrence of $I$.
  - An identifier $I$ may be defined in multiple blocks.
- Nested blocks, some outer block contains a declaration of $I$:
  - If inner block does not contain a declaration of $I$, then declaration is visible throughout outer and inner blocks.
  - If inner block contains a declaration of $I$, then the inner block declaration hides the outer block declaration.
A *declaration* is a language construct to produce a binding.

Types of simple declarations:
- type
- constant
- variable
- procedure
DSL Design

- A *procedure definition* binds an identifier to a procedure
  - function
  - procedure

  - A function definition:
    ```c
    bool even(int n) { return (n % 2 == 0); }
    ```
  - A procedure definition:
    ```c
    void double(int& n) { n *= 2; }
    ```
• Parameters and arguments
  
  • An *argument* is a value or other entity that is passed to a procedure
  
  • An *actual parameter* is an expression that yields an argument
  
  • A *formal parameter* is an identifier through which a procedure can access an argument
• An identifier is *overloaded* of it denotes two or more distinct procedures in the same scope
• In older programming languages operators for certain built-in functions are overloaded
• “-” operator:
  - integer negation \((\text{Integer} \rightarrow \text{Integer})\)
  - floating-point negation \((\text{Float} \rightarrow \text{Float})\)
  - integer subtraction \((\text{Integer} \times \text{Integer} \rightarrow \text{Integer})\)
  - floating-point subtraction \((\text{Float} \times \text{Float} \rightarrow \text{Float})\)
Dynamic Semantics

- Represents the intended meaning of a language and language constructs
- Is a means to represent our understanding of a model/program (what it does)
- To communicate our understanding to other entities and
- To understand what happens in a computer/machine when a program/model is executed
• Not every language is suitable for describing dynamic semantics

• Dynamic semantics ideally is described in a formal language (formal semantics) because:
  - ambiguities and inconsistencies can be detected in a model which appears to be “ok”
  - this is the basis for analysis, validation and verification, but also implementation
• Dynamic semantics in practice
  • Hardly any modeling language used in industry has formal semantics
  • The semantics of languages is often defined by translation or interpretation
  • Languages evolve and more ambiguities are introduced
DSL Design

(domain) model

(DS) Language

(implementation) engine
DSL Design

- Ambiguous representation and miscommunication
- Different understandings of models
DSL Design

- Model to implementation inconsistency
  - SW constructs do not use semantics, only syntax
  - Code is manually added in the parts generated

- Cognitive feedback gap between modeling and model/implementation debugging
  - No possibility to transfer results of execution/debugging back to the original model

- Lack of early design correctness analysis
• Transformation from one environment to another
  - Semantic transformation gap
  - Model transformation correctness reasoning
• Multi-disciplinary project
  - Semantic gap between models
• (Reference) Implementation
  - Model/program is executed
  - Conclusions about the behaviour based on observations
  - if the code is complex (as usual) no behavioural analysis is possible

• Translational
  - Select target language(s)
  - Model based in most of the cases

• Interpretation

• Formal semantics
  - structural operational semantics (SOS, MSOS)
  - denotational semantics
  - axiomatic
  - algebraic
  - …
**DSL Design**

Translational semantics:
- Model based
- Ad-hoc syntactic mapping
- No semantic mapping -> no ambiguities detection
- No support for language evolution
- Inconsistency and Properties preservation?
- Traceability/reverse mapping
Translational semantics
- Ad-hoc semantic mapping
- No semantic mapping -> no ambiguities detection
- No support for language evolution
- Inconsistency and properties preservation?
- Traceability/reverse mapping
DSL Design

- (domain) model
- DSL
- (implementation) engine
- formal semantics
- Target model
- Target language
- engine

Translation for free! Correct by construction!

Design decision?
DSL Design

• Formal dynamic semantics
  • Modern Systems models
    – complex
    – (possibly) high level of abstraction
  • Reasoning about the models using rigorous methods
    – need to find existing ambiguities and inconsistencies
    – need to keep a (modeling) language “clean and simple”
  • Formal semantics allows for
    – analysis, validation and verification, but also (correct) implementation
    – model/program comparison, thus optimization, modification
Dynamic semantics

• Overview of transformations

SLCO Code → T2T → LTS

T2M

SLCO Model

M2M

Poools Model

M2T

Poools Code

NQC Code

M2T

NQC Model

M2M

Promela Model

M2T

Promela Code

SLCO Model

M2M

T2M

M2M

Promela Model

M2T
• **Operational semantics**
  - specifies HOW (step-by-step) program/model executes
  - specifies HOW states/configurations are modified during the execution
  - all possible executions are generated
  - underlying model is the model of Transition systems (a program/model execution is turned into transition system)
# DSL Design

## Target languages

<table>
<thead>
<tr>
<th></th>
<th>(A)Synchronous communication</th>
<th>Lossy/lossless communication</th>
<th>Concurrent objects</th>
<th>Connectivity for Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLCO</td>
<td>Both</td>
<td>Both</td>
<td>Unlimited</td>
<td>Point-to-point</td>
</tr>
<tr>
<td>POOSL</td>
<td>Synchronous</td>
<td>Lossless</td>
<td>Unlimited</td>
<td>Point-to-point, Broadcast</td>
</tr>
<tr>
<td>PROMELA</td>
<td>Both</td>
<td>Lossless</td>
<td>Unlimited</td>
<td>Point-to-point</td>
</tr>
<tr>
<td>NQC</td>
<td>Asynchronous</td>
<td>Lossy</td>
<td>Limited</td>
<td>Broadcast</td>
</tr>
</tbody>
</table>

- **SLCO Model**: Bridging all gaps at once with 1..N transformations for each gap.
- **NQC Code**: Target languages include SLCO, POOSL, PROMELA, and NQC.
DSL Design

- **Endogenous transformations**
  - Transformation within one meta model
  - Bridge gaps between platforms

- **Exogenous transformations**
  - Transformation between different meta models
  - Transform SLCO to Promela/POOSL/NQC/…
DSL Design

- **Motivation**
  - Quite a number transformations have been implemented
  - To verify endogenous transformations
    - semantics of SLCO is needed
DSL Design

Running Example

```
Initial

receive Start([true]) from Q1

receive V() from Q2
[n < 1]
/n := n + 1

receive Stop() from Q3

/n >= 1

State

Initial

receive Start([true]) from Q1

receive V() from Q2
```

```
p: P
P1
P3
P2

q: Q
Q1
Q3
Q2

p1_q1(Boolean)
p3_q3(Integer)
q2_p2()
```
The state machines specify the following communication between p and q:

- the two objects first communicate synchronously over channel p1_q1
- after which q repeatedly sends signals to p over the lossy channel q2_p2
- if p has received at least one signal sent by q, it sends a signal over channel p3_q3 and terminates
- after receiving this signal, q terminates as well
• Configuration Step (CS) language
  • specifies explicitly low-level activities, implicit in SLCO
  • serves as underlying language to express semantics of SLCO
  • the CS representation of an SLCO model refines its behavior by splitting the execution in more basic activities

• CS together with the SLCO2CS transformation captures the semantics of SLCO
DSL Design

Configuration

Step

\(<\langle p, P, \text{Initial}\rangle \langle q, Q, \text{Initial}\rangle, [\langle\langle p, n\rangle, 0\rangle], [\langle\langle p3_q3, p, P3, q, Q3\rangle, \langle\langle q2_p2_, q, Q2, p, P2\rangle, \rangle], \text{initial}\rangle>

\(<\langle p, P, \text{State}\rangle \langle q, Q, \text{State}\rangle, [\langle\langle p, n\rangle, 0\rangle], [\langle\langle p3_q3, p, P3, q, Q3\rangle, \langle\langle q2_p2_, q, Q2, p, P2\rangle, \rangle], \text{receiving } V()\rangle>

\(<\langle p, P, \text{State, 0, 1}\rangle \langle q, Q, \text{State}\rangle, [\langle\langle p, n\rangle, 0\rangle], [\langle\langle p3_q3, p, P3, q, Q3\rangle, \langle\langle q2_p2_, q, Q2, p, P2\rangle, \rangle]>

>
DSL Design

states
0
1
final 2
final 3
4
5
6
7
8
9
initial 10

transitions
initial 10 "Start(true)" 9
9 "lost V()" 9
9 "sending V()" 8
8 "receiving V()" 7
8 "lost V()" 8
7 "n := 1" 6
7 "lost V()" 7
7 "sending V()" 0
6 "sending Stop()" 5
6 "lost V()" 6
6 "sending V()" 1
5 "lost V()" 5
5 "sending V()" 4
5 "receiving Stop()" final 2
4 "lost V()" 4
4 "sending Stop()" final 3
1 "sending Stop()" 4
1 "lost V()" 1
0 "n := 1" 1
0 "lost V()" 0
DSL Design
Validating Model Transformations

- For small LTSs, properties can be manually verified.
- For larger LTSs, reduction techniques can be applied to reduce the LTS:
  - \texttt{ltsconvert} (part of the mCRL2 toolset) can reduce LTSs, by means of an equivalence relation, for instance branching bisimilarity relation.
- When LTSs get too large for reduction and manual inspection, other tools can be used for verification:
  - CADP offers tools that take an LTS and a temporal logic property and perform verification of the property on the LTS.
DSL Design

P: Producer

C: Consumer

Ps
Ps to Cs()

Ps to Cs()

Pas
Pas to Cas()

Lossy2Lossless

C: Consumer

Ps

P: Producer

Cs

Cas

C_PABP_Receiver: ABP_Receiver

ARAS

AASReceiver

AASAR

C_P_Receiver_to_Ac()
**DSL Design**

**P: Producer**
- Cs
- Cas

**C: Consumer**
- Ps
- Pas

**Lossy2Lossless**

- **Initial**
  - receive Start() from Ps
  - State1
    - receive Signal("Bericht") from Pas
    - State2
      - receive Stop() from Ps

- **C: Consumer**
  - Ps
  - Pas

- **C_PABP_Recover: ABP_Recover**
  - AROriginal
  - ARSender
  - ARAS

- **C_PABP_AS: ABP_AS**
  - AASReceiver
  - AASAR
  - P_C_AS_to_AR(Integer)

- **P: Producer**
  - Cs
  - Cas

- **P_CABP_Sender: ABP_Sender**
  - ASReceiver
  - ASOriginal

- **P_CABP_AR: ABP_AR**
  - AARAS
  - AARSender

- **P_COriginal_to_Sender(String)**
- **P_C_Send_to_Receiver(String, Integer)**
- **P_CS to_Cs()**
DSL Design

---

Start()

sending Signal('Bericht')

receiving Signal('Bericht')

Stop()

---

LTSview
(Alternative Visualisation)

Lossy2Lossless

LTSconvert
(State Space Reduction)
DSL Design

• Points made so far:
  • Concepts of DSL
  • An attempt to define semantics formally

• Questions/remarks
Code generation

• That is all!
• Questions?