Concepts in Code Generation

COMP6209

Julian Rathke
jr2@ecs.soton.ac.uk
Lecture Goals

• introduce and define basic concepts
• *sketch* process of building a generator
Code Generation

Definition: Code generation is the automatic derivation of source code in a conventional language from an input model.

• … (roughly) looks like compilation, but:
  – semantic gap is bigger
    “|model – source code| >> |AST – object code|”
  – steps interleaved and/or repeated (search & control)
• … is more knowledge-intensive
  – “[...] optimizations which are beyond the then current state of the compiler art.” Balzer, 1985
  – “[...] euphemism for programming in a higher-level language than was then available to the programmer.” Parnas, 1985
Code generation changes the software development process.

- **Requirements**
- **Domain Knowledge**
- **Design Knowledge**

- **Reduces conceptual gap**
- **Formalizes and internalizes domain and design knowledge**

- **Can focus on requirements**

---

```java
/// <summary>
/// This is the interface used by the LocalService.
/// </summary>
[ExternalDataExchange]
public interface LocalService
{
    /// <summary>
    /// This is the method that the workflow calls to send data back to the host
    /// </summary>
    /// <param name="complexObject">complexObject</param>
    public void SendDataToHost(ComplexObject complexObject);

    /// <summary>
    /// This is the event that is used to pass information into the workflow
    /// </summary>
    /// <param name="ObjectName">ObjectName</param>
    public void EventHandler<LocalServiceEventArgs> ExecuteEvent();
}
```
Code generators can also generate non-code artefacts from the same model.

- documentation
  - description of input/output, `man` pages, manual
  - description of design knowledge, code derivation, and justification
- installation and control scripts
- test data, test data generators, and simulations
- proofs of safety, effectiveness, correctness
- trace information between model and code
  - round-trip engineering
  - incremental consistency
Benefits of Code Generation

• increased *productivity*
  – from high-level specification to system implementation
  – fast turn-around
• increased *reliability*
  – no (manual) coding errors
• increased *portability*
  – re-generate for new platform
• increased level of *intentionality*
  – algorithms represented in domain-specific concepts
    (instead of efficient [convoluted] low-level code)
Anatomy of a Code Generator

- separation between engine and transformations
  - basic transformations built into code generator
  - additional transformations extracted from model
  - reflection allows reasoning over transformations
- ASTs as internal representation

\[
\text{grammar(s)} \quad \begin{array}{c}
X \rightarrow a Y \\
\ldots
\end{array}
\]

parse \quad \text{engine} \quad \text{unparse}

\text{source}
\text{code}

\text{model} \quad \text{extract} \quad \text{AST} \quad \text{reflect} \quad \text{transformation} \quad \text{database} \quad \text{meta-information}

\bullet \text{ strategies}
\bullet \text{ rules}
\bullet \text{ grammar}
\bullet \ldots

\text{X} \rightarrow a \text{ Y}
\ldots
Most code generators follow one of the basic code generation paradigms.

Paradigms given by main operation or data structure:

• **generative** or **code-based**
  – assemble code from fragments
    ▶ macros, C++ templates, aspects
    ▶ template engines (JET, Velocity, …)

• **transformative** or **model-based**
  – refine model into code
    ▶ MDA / MBSE / …

• **deductive** or **proof-based**
  – logically deduce code from specification
    ▶ Amphion, KIDS / PlanWare / SpecWare, …
Software transformations are a fundamental concept in code generation.

- **Horizontal transformations**
  - evolution: evolve specification
  - refactoring: evolve architecture

- **Vertical transformations**
  - refinement: implement/refine to code

(Czarnecki / Eisenecker, 2000)
Compositional generators apply only vertical transformations.

- introduced by Batory
- typical for CASE tools
  - models built via GUI
  - code generation via simple final traversal
- structure-preserving
  - changes do not cross model element boundary
  - generated code can be traced back to initial model
- specifications often executable (at each level)


(Czarnecki / Eisenecker, 2000)
**Holistic** generators apply vertical and horizontal transformations.

- “whole system” transformations
  - optimization
  - refactoring
  - weaving

- can bridge bigger gaps
- cannot be composed from vertical / horizontal transformations
  - more difficult to implement

(Czarnecki / Eisenecker, 2000)
Domain Modeling

• establish and define
  – vocabulary
  – concepts and roles
  – features (“options”)

• define common and variable properties of systems
  – semantics
  – dependencies

• define common architecture(s)

Rule of thumb:
  good domains (for code generation) have good domain models!
Domain modeling is a hard problem.

- requires domain experience (a lot!)
- domain experts often think too abstract for code generation

\[
\begin{align*}
\text{Process model: } & \quad x_{k+1} = F_k x_k + G_k u_k \\
\text{Measurement model: } & \quad z_k = H_k x_k + v_k
\end{align*}
\]

\[
\begin{align*}
& \text{State variables: } (n \times 1) \\
& \text{Process noise: } (n \times 1) \\
& \text{Coupling matrix: } (n \times n) \\
& \text{Process transition matrix: } (n \times n) \\
& \text{Process noise variables: } (l \times 1) \\
& \text{Measurement noise: } (m \times n) \\
& \text{Measurement model: } (m \times 1) \\
& \text{Measurement sensitivity matrix: } (m \times n) \\
& \text{Measurement variables: } (m \times 1)
\end{align*}
\]

\[
\begin{align*}
u_k & \sim N(0, Q_k) \\
E[u_k u_i^T] & = \begin{cases} 
Q_k, & i = k \\
0, & i \neq k 
\end{cases} \\
v_k & \sim N(0, R_k) \\
E[v_k v_i^T] & = \begin{cases} 
R_k, & i = k \\
0, & i \neq k 
\end{cases} \\
E[v_k u_i^T] & = 0
\end{align*}
\]
Domain modeling is a hard problem.

- requires domain experience (a lot!)
- domain experts often think too abstract for code generation

\[
\begin{align*}
\text{Process model} & : \quad x_{k+1} = F_k x_k + G_k u_k + \nu_k \\
\text{Measurement model} & : \quad z_k = H_k x_k + v_k
\end{align*}
\]

\[
\begin{align*}
u_k & \sim N(0, Q_k) \\
E[u_k u_i^T] & = \begin{cases} Q_k, & i = k \\
0, & i \neq k \end{cases} \\
\nu_k & \sim N(0, R_k) \\
E[v_k v_i^T] & = \begin{cases} R_k, & i = k \\
0, & i \neq k \end{cases} \\
E[v_k u_i^T] & = 0
\end{align*}
\]
Domain modeling requires a formal modeling language.

- **UML most common**
  - classes as concepts
  - relations as roles
  - OCL to restrict combinations

- **tool support**
  - graphical editors
  - transformation tools

- **other approaches exist**
  - description logics, ...
Generative domain modeling must cover problem space and solution space.

**Problem space**
- application domain-specific:
  - high-level concepts
  - features
  - functional model
  - non-functional QoS
  maps directly or indirectly to components

**Generic configuration knowledge:**
- dependencies: defaults, mandatory, illegal

**Implementation knowledge:**
- constructions, optimizations
- express transformations using meta-information

**Transformation**

**Specific requests:**
- optimize space / time
- use specific algorithms
- generate additional artifacts
- target languages

**Solution space**
- implementation components
  - procedures
  - functions
  - fragments
  - templates
  minimize redundancy
  maximize reuse

But not always a clear separation …

(Czarnecki / Eisenecker, 2000)
Generative domain model adds solution concepts.

- solution concepts
  - code fragments
  - structure
  - relevant variables
  - types
Generative domain model adds solution concepts.

- solution concepts
  - code fragments
  - structure
  - relevant variables
  - types
- difficult to express
  - order
  - dependencies
- use specialized notations
  - feature models
Generator frameworks provide dedicated tools to build generators.

- Typically provide tools for:
  - Meta-modeling (MetaEdit+, GME, …)
  - Meta-programming (C++ templates, …)
  - Transformation and search (Stratego, …)

- Often used to build itself (bootstrapping)
  - Requires *metamodel* of system
    - UML defined within MOF
  - Supports customization
    - Modify specification language
    - Retarget implementation language
    - Extend transformation base
Summary

• code generator structure
• code generation approaches
  – generative, transformative, deductive
• transformations
• domain modeling