Automatic generation of behavioral code - too ambitious or even unwanted?

Gregor Engels

University of Twente, The Netherlands
23 June 2009

Professional Activities

Gregor Engels

- University of Paderborn
  - Head of Board s-lab (Software Quality Lab)
  - Member Board International Graduate School (IGS)

- Scientific Director Capgemini sd&m Research, Munich

- International
  - Member Founding Board Informatics Europe (http://www.informatics-europe.org/)
  - Steering Committee MODELS Conference, Visual Languages and Human-Centric Computing (VLHCC), International Conference on Graph Transformations (ICGT)
Software is everywhere!

Objectives of modern software development
• efficient development process
• software of high quality
  • correct
  • adaptable
  • reusable
  • …

Solution / Dream (?)
• Model-driven Development (MDD)
• Model-driven Architecture (MDA)
• Service-oriented Architecture (SOA)
Automatic code generation needs:
- complete models
- semantics-preserving transformations

Can we achieve this in general?

Model-driven Software Development (MDD)

Traditional Manual Software Development
The Dream

MDD

designer

problem domain

model (specification) model transformation

program code

compiler

executable program

The Reality

MDD

restricted or no model

program code

executable program

Objectives

MDD

designer

problem domain

model (specification) model transformation

program code

compiler

executable program

Objectives

problem domain

restricted or no model

program code

compiler

executable program

objectives: • acceptable models • semantics-preserving transformations
Only a Dream?  No! Database Design!

MDD

- designer creates model (specification)
- model transformation
- program code
- compiler
- executable program

Model-driven Database Design

- designer creates Entity-Relationship Diagram
- model transformation
- Relational DB Schema
- compiler
- executable program

- semantics-preserving
- • lossless join
- • dependency-preserving
- language/tool support
- • 4th GL

Only a Dream?  No!? Fujaba

MDD

- designer creates model (specification)
- model transformation
- program code
- compiler
- executable program

MDD

- designer creates Story-driven Modelling
- model transformation
- Java Code
- compiler
- executable program
The Fujaba Approach

Fujaba - From UML to Java and Back Again
- Open Source UML CASE Tool Project
- started in 1997 at the University of Paderborn
- Wilhelm Schäfer (University of Paderborn)
  - http://wwwcs.uni-paderborn.de/cs/fujaba/
- Albert Zündorf (University of Kassel)
  - http://www.se.eecs.uni-kassel.de/se/index.php?fujabaproject
- Holger Giese (HPI Potsdam)
  - http://www.hpi.uni-potsdam.de/giese/projekte/fujaba.html

Fujaba – main features
- based on UML
- extended by Story Driven Modeling (SDM)
- deploys graph transformation platform
- combines UML class diagrams and UML behavior diagrams (Story Diagrams) to a powerful, easy to use, yet formal system design and specification language
- generation of Java source code out of the whole design which results in an executable prototype
- re-engineering so that Java source code can be parsed and represented within UML
Behavior modeling with Fujaba

- **object diagrams**
  - are used as typed graphs
  - are the base to define graph transformation rules
- **control flow** described by UML activity diagrams
- **object behavior** described by graph transformation rules

- Running example:
  - autonomous shuttles

Object diagrams as typed graphs

Class Diagram:

Object graph:

(\textit{edges are typed, too})
Story Diagrams

- combine:
  - UML Activity diagrams specify **control flow**
  - Story patterns specify **graph transformation rules**
- specify (more complex) operations on object structures
- have formally defined semantics

**Control flow:**
- sequences
- loops
- conditions
- alternatives

**Graph transformation:**
- object structure modification
- attribute value changes
- attribute conditions

-----

**Story Diagrams - Control Flow Example**

**Modeling alternatives with Story Diagrams**

Guards **success** and **failure**:
Success: Transition fires iff previous activity has been executed successfully, i.e. pattern completely matched and all conditions satisfied.
Motivation for code generation

- model-based software engineering
  - (mainly) working on design level (with models)
    - reduces complexity
  - automatically generate code
    - reduces implementation effort
    - reduces implementation errors
- maintainability and readability of generated code
  - reason: sometimes code has to be reviewed or adapted manually
  - integration with frameworks, platforms, etc.
  - manual code optimization (e.g. for efficiency)

Code Generation with Fujaba

- Structural diagrams (Class diagrams)
  - Inheritance
  - Associations
- Behavioral diagrams which refine class diagrams
  - Activity diagrams (usually for control flow of methods)
  - Story diagrams (refine activity diagrams)
Code generation of structural information

- Code generation defined for:
  - inheritance
  - bidirectional associations
  - composition and aggregation
  - ...

Code generation of activity diagrams

- Activity Diagram specifies single method
- (well-formed) control flow maps to (Java) control structures
- actions / guards translate directly to code (makes diagrams language dependent)
Generated Java code for story diagrams

```java
public class Shuttle {
    ...
    public void gotoTrack(Track destination) {
        boolean fujaba_Success = false;
        Track currentTrack = null;
        try {
            fujaba_Success = false;
            // check object destination is really bound
            javaSDM.ensure(destination != null);
            // bind currentTrack
            currentTrack = this.getTrack();
            javaSDM.ensure(currentTrack != null);
            // check isomorphic binding
            javaSDM.ensure(! (destination.equals(currentTrack)));
            // delete link
            this.setTarget(null);
            // write link
            this.setTarget(destination);
            fujaba_Success = true;
        } catch (JavaSDMException fujaba_InternalException) {
            ...
        }
    } ...
}
```

Summary of Fujaba Approach

- complete structure needs to be modelled to provide object diagrams
- control flow definition is programming language dependent
- code generation must embed methods and its behavior into structural skeleton
- syntax and semantics definition of story diagrams is mapped to programming language instructions
Evaluation of Fujaba

Acceptable model?
- detailed, fine-grained specification
- programming-language dependent

Semantics-preserving transformation?
- compiler-based semantics of Fujaba
- transformation as semantics definition

Not a semantics-preserving transformation!

Objectives:
- acceptable models
- semantics-preserving transformations

Story-driven Modelling

Programmer creates

Java code

Compiler

Executable program

Semantics-preserving model transformation

MDD

designer creates

model

transformation

code

Compiler

Executable program

ML meta model

source language

transformation specification

target language

PL meta model

Model

Transformation engine

<instance of>

<instance of>

<instance of>

<instance of>

Input

Output

Code
What does semantics-preserving transformation mean?

Semantics preserving means that the resulting model/code still conforms to the initial requirements.

Correctness of transformation in the sense of semantics preserving.

What is needed for a proof?

- formal semantics of behavioral models
- formalized properties
- formal transformation
Semantics-preserving model transformation

- semantics-preserving means that all properties of the source model hold for the target model, too

The way to show „semantics-preservation“

Transition systems are equivalent ⇔ All properties that hold on a source model also hold on a target model
Why is it difficult to prove this?

- languages are different – not every element has a counterpart in the other language.

Example:
- UML Activity Diagram
  - several tokens may traverse through a diagram
- Textual Language (e.g., Java)
  - only one program counter

How to map the semantics in this case?

Proof of bisimulation

- Goal: to show that transition system (TS) of each source model is bisimilar to the TS of target model received as result of transformation
Double Check Approach

Joint Project:
- University of Paderborn: G. Engels, M. Semenyak, Chr. Soltenborn, H. Wehrheim
- TU Twente: A. Rensink (Groove approach)

- Bisimilar relation between Transition Systems
  - Systems behave in the same way in the sense that one system simulates the other and vice-versa


Evaluation

- acceptable model?
  - detailed, fine-grained specification
  - programming-language dependent

- semantics-preserving transformation?
  - hard to prove
  - needs semantics definition of languages
Title of the Talk: Automatic generation of behavioral code – too ambitious or even unwanted?

- acceptable model?
  - detailed, fine-grained specification
  - programming-language dependent

- semantics-preserving transformation?
  - hard to prove
  - needs semantics definition of languages

Are there alternatives??

Alternative Approaches

- Use of partial models
  - visual contracts

- Model-driven Monitoring (MdM)
- Model-based Testing (MbT)
Alternative Approaches

- Use of partial models
  - visual contracts

- Model-driven Monitoring (MdM)
- Model-based Testing (MbT)

Graph-Transformation-Based Modeling

**Modelling Alternatives**

- DPO: double pushout
  - complete control of change
  - no context change in addition to rule

- SPO: single pushout
  - complete control of change
  - deletion of dangling edges
  - no context change in addition to rule

- ZPO: “zero” pushout
  - complete control of change
  - context change with embedding descriptions

- DPB: double pullback
  - partial control of change
  - uncontrolled context deletion
  - uncontrolled context extension
Reuse of an old idea: Design by Contract (Eiffel)

- **Contract**: A formal agreement between an operation and its clients
- textual logical expressions
- allows partial description of behavior

Concepts of Design by Contract:
- Describe behavior of operations partially with contract.
- Monitor implementation (correctness) at runtime

1. test pre-condition
2. execute operation
3. test post-condition
Visual Contract Example

Behavioral Aspects:
Visual Contract

Visual Contracts:
- define “minimal” requirements
- semantic concept of loose graph transitions
- formalized by double-pullback

Static Aspects:
Class Diagram

Alternative Approaches

- Use of partial models
  - visual contracts

- Model-driven Monitoring (MdM)
- Model-based Testing (MbT)
Model-Driven Monitoring

- Problem domain
- Model
- Program code
- Executable program

- Designer creates model
- Programmer creates program code
- Compiler generates executable program

- Use visual contracts on model level
- Monitor manual implementation against model

Visual Contracts

- Problem domain
- Model
  - Class Diagram
  - Visual Contracts
- Program code
- Executable program

- Designer creates model
- Programmer creates program code
- Compiler generates executable program

- Loose semantics
- Double-pullback

- How to monitor manual implementation against model?
Model-Driven Monitoring

**Model**
- Model
- Class Diagrams
- Visual Contracts
  - designer

**Designer**
- creates
- model transformation

**Programmer**
- completes
- model transformation

**Implementation**
- executable program
- run-time assertions
  - compiler

**Class Skeletons**
- executable program-code

**Assertsions**
- model transformation

Transformation into Java and JML

JML (Java Modeling Language)
- Design by Contract Extension for Java
- JML Assertions are based on Java expressions

```java
/*@ public normal_behavior
@ requires (exists Product pr;
@   getProducts().values().contains(pr);
@   pr.getASIN().equals(asin))
@   & & (exists Cart c;
@   getCartItems().values().contains(c);
@   c.getCartID().equals(cid));
@ ensures (exists Product pr;
@   getProducts().values().contains(pr);
@   pr.getASIN().equals(asin))
@   & & (exists Cart c;
@   getCart().values().contains(c);
@   c.getCartID().equals(cid));
@   & & (exists CartItem citem;
@   item.getASIN().equals(asin))
@   & & citem.getQuantity() == x));
@*/
public String cartAdd (String asin, String cid, int x);
```

Gregor Engels, BM-MDA 2009

43
Transformation into Java and JML

/*@ requires (exists Product pr; @> getProducts().values().contains(pr); @> pr.getASIN().equals(asin)) @> && (exists Cart c; @> getCartID().equals(cid)))@*/
public String cartAdd (String asin, String cid, int x);

JML (Java Modeling Language)
- Design by Contract Extension for Java
- JML Assertions are based on Java expressions

JML (Java Modeling Language)
- Design by Contract Extension for Java
- JML Assertions are based on Java expressions

Transformation into Java and JML
Model-Driven Monitoring: Tool Support

Visual Contract Workbench

- Visual Contract Workbench is implemented as Eclipse plug-in
- allows for modeling of class diagrams and visual contracts
- complemented by code generation facilities

Visual Contract Workbench
Alternative Approaches

- Use of partial models
  - visual contracts
- **Model-driven Monitoring (MdM)**
  - Model-based Testing (MbT)

Quality check by accident!!

Planned quality check!!
Traditional Manual Software Testing

- Problem domain
- Restricted or no model
- Program code
- Test code
- Tester
- Programmer

Test code creation may be restricted or nonexistent, leading to an inadequate test strategy.

Model-based Testing (MbT): Scenario 1

- Problem domain
- Model (specification)
- Test model
- Test designer
- Program code
- Test code

BUT!
- Complete test code generation
  - Needs complete/low-level test models (expensive)
  - Needs flexible transformation (difficult)
- Thus, not realistic in general
Model-based Testing (MbT): Scenario 2

**BUT!**
- extraction must be specified
- few/no redundancy between program code and test code
- “test it against it”
- thus, not effective in general

Model-based Testing (MbT): Scenario 3

- extraction must be specified
- automated test design
- automated test execution
Model-based Testing (MbT): Our combined Approach

![Diagram of MbT process]

Unit Testing: Test Case Generation

Test case inputs: call parameters + system state

1. Generation of call parameters
   \[ P = \{ \text{cid=abc, prNo=def, num=1} \} \]

   Well-known techniques:
   - Boundary analysis
   - Equivalence classes
   - Random

2. Generation of system state
   \[ S_{\text{input}} = \]

3. Setting system state
   - Simulate
   - Naturally generate
### Unit Testing: Test case generation

Test case inputs: call parameters + system state

1. Generation of call parameters
   \[ P = \{ \text{cid} = \text{abc}^*, \text{prNo} = \text{def}^*, \text{num} = 1 \} \]

2. Generation of system state

3. Setting system state

Well-known techniques:
- Boundary analysis,
- Equivalence classes,
- Random

### Unit Testing: Simulating Test Cases
Unit Testing: JUnit Test Script

cartAdd(Product product, int quantity, String cid)

Test input
- quantity= 1
- cid = "xyz"

Product product = new Product();
product.setTitle("abc");
product.setASIN("def");
int quantity = 1;
String cid = "xyz";

OnlineShop self = new OnlineShop();
Cart c = new Cart();
c.setCartId("xyz");
self.addCart(c);
self.addProduct(product);
self.cartAdd(product, quantity, cid);

Model-driven Monitoring used to check correctness!

public void testCartAdd_0()
{
    Call parameters
    further Objects
    Object variables
    Object relations
    Invoke method

    Generate
    Test input
    1. Generation of call parameters
    \[ P = \{\text{cid} = \text{abc}, \text{prNo} = \text{def}, \text{num} = 1\} \]
    Well-known techniques:
    - Boundary analysis,
    - Equivalence classes,
    - Random

    2. Generation of system state

    3. Setting system state
    \[ s \supseteq s_{input} \]
    simulate system state
    call other class operations
    until desired system state is reached naturally
    e.g. using mock objects or stubs
    e.g. using model checking

    Test case generation

    Test case inputs: call parameters + system state

    1. Generation of call parameters
    \[ P = \{\text{cid} = \text{abc}, \text{prNo} = \text{def}, \text{num} = 1\} \]
    Well-known techniques:
    - Boundary analysis,
    - Equivalence classes,
    - Random

    2. Generation of system state

    3. Setting system state
    \[ s \supseteq s_{input} \]
    simulate system state
    call other class operations
    until desired system state is reached naturally
    e.g. using mock objects or stubs
    e.g. using model checking
Unit Testing: Setting System State Naturally

3. Generation of system setting sequence

Model checking techniques for computation of the system setting sequence: cartCreate, ..., operation
Title of the Talk:
Automatic generation of behavioral code – too ambitious or even unwanted?

The Dream

Our Proposal: MdM + MbT

Acknowledgments

Thanks to the PhD students who helped me in preparing this talk:

Baris Güldali

Maria Semenyak

Michael Spijkermann
References Model-based Testing

- G. Engels, B. Güldali, C. Soltenborn, H. Wehrheim: 

- G. Engels, B. Güldali, M. Lohmann 
  *Towards Model-Driven Unit Testing*

- B. Güldali, M. Mlynarski, A. Wübbeke, G. Engels: 

References Model-driven Monitoring

- G. Engels, M. Lohmann, S. Sauer, R. Heckel: 
  *Model-Driven Monitoring: An Application of Graph Transformation for Design by Contract*

- M. Lohmann, S. Sauer, G. Engels: 
  *Executable Visual Contracts*
References Semantics

  *From UML Activities to TAAL - Towards Behaviour-Preserving Model Transformations*

- G. Engels, C. Soltenborn, H. Wehrheim:
  *Analysis of UML Activities Using Dynamic Meta Modeling*

- G. Engels, J. H. Hausmann, R. Heckel, S. Sauer
  *Dynamic Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML*

The End

engels@upb.de