Logic-based
Software Analysis and Transformation

Dr. Günter Kniesel
gk@cs.uni-bonn.de

ROOTS Group
Computer Science Department III
University of Bonn
Software Transformation

- Software Transformation
  - Modification (or sequence of modifications) of software artefacts.

Examples
- Program Synthesis
  - derive implementation from specification
- Reverse engineering
  - Extract higher abstraction from lower level artefacts
- Migration
  - translate to other language at same level of abstraction
- Rephrasing
  - transform to same language
Transformation Categories

- **Translation:**
  Transform program in language A to program in language B
  - Migration
  - Synthesis
  - Reverse engineering
  
- **Rephrasing:**
  Transform program in language A to program in the same language
  - Normalization
  - Optimization
  - Refactoring
  - Renovation
  - Enhancement

Translation:
Transform program in language A to program in language B

Rephrasing:
Transform program in language A to program in the same language
Software Analysis

- Software Analysis
  - Derivation of implicit information from software artefacts.

Examples
- Control-flow analysis → A calls B
- Data-flow analysis → A is assigned from B
- "Bad smell" detection → "Long method"
- Design Pattern detection → "Pull variant of Observer"
- Concern identification → "Logging"
- ...

© 2006 Günter Kniesel, gk@cs.uni-bonn.de
Software Analysis and Transformation

- Transformation needs prior analysis!

- Analysis determines
  - whether a transformation is necessary
    - bad small detection for refactorings
  - whether a transformation is legal
    - check behaviour preservation of refactorings
  - the program elements to be transformed
    - method with long parameter list
  - context information necessary for the refactoring
    - methods of parent class for which to create forwarding methods in a decorator

→ Combine transformation and analysis!
Conditional Transformations (CTs)

- CT = Condition + Transformation
  - Condition is true $\Rightarrow$ Transformation may be executed

Example: "Add Class" Transformation
- Condition: Class does not exist

$\neg \text{exists(Class)} \Rightarrow \text{add(Class)}$

© 2006 Günter Knesel, gk@cs.uni-bonn.de

Logic-Based Software Analysis and Transformation 6
Overview

- Logic based Software Artefact Representation
- Logic-based Software Analysis

- Introduction of JTransformer
- Analysis Example: Design Pattern Detection

- Logic based Conditional Transformations (CTs)

- Transformation example: Build your own Refactoring

- CTC: A Language-Parametric Transformation System
- How to define own languages

- Exercises (implement yourself)
Logic-Based Software Representation
package demo;

class C {
    int m(int i) {
        m(i);
    }
}

packageT(1, 'demo')
classDefT(2, 1, 'C', [3])
methodDefT(3,2,'m',[4],int,[],5)
paramDefT(4, 3, 's', int)
blockT(5, 3, 3, [6])
callT(6, 5, 3, null, 3)
identT(7, 6, 3,'i',4)
Logic-Based Program Representation

```java
package demo;

class C {
    int m(int i) {
        m(i);
    }
}
```

**Diagram Details:**
- **Package:** `demo`
- **Class:** `C`
- **Method:** `m(int i)`
  - **Parameters:** `int i`
  - **Body:** `m(i);`

**Nodes and Edges:**
- **Package Node:** `packageT(1, 'demo')`
- **Class Node:** `classDefT(2, 1, 'C', [3])`
- **Method Node:** `methodDefT(3, 2, 'm', [4], int, [], 5)`
- **Parameter Node:** `paramDefT(4, 3, 's', int)`
- **Block Node:** `blockT(5, 3, 3, [6])`
- **Call Node:** `callT(6, 5, 3, null, 3)`
- **Variable Node:** `identT(7, 6, 3, 'i', 4)`
package demo;

class C {

    int m(int i) {
        m(i);
    }
}

classDefT(2, 1, 'C', [3])

methodDefT(3, 2, 'm', [4], int, [], 5)

paramDefT(4, 3, 's', int)

blockT(5, 3, 3, [6])

callT(6, 5, 3, null, 3)

identT(7, 6, 3, 'i', 4)
class ClassName {

    int m(int i) {
        m(i);
    }
}

© 2006 Günter Kniesel, gk@cs.uni-bonn.de
Logic Fact Representation of AST

```java
public class ClassName {
  modifierT(1, 'public').
  classDefT(1, Package1, 'ClassName').
  toplevelT(0, null, '/JTTest/src/ClassName.java', [1]).

  void m(int i) {
    methodDefT(2, 1, 'm', [9], type(basic, void, 0, [], 10).
    paramDefT(9, 2, type(basic, int, 0), 'i').
    blockT(10, 2, [12]).
    m(i);
    applyT(12, 10, null, 'm', [13], 2).
    identT(13, 12, 'i', 9).
  }
}
```
Logic-based Program Analysis

classMethodReturnsOwnInstance(T, M, F) :-

    methodDefT(M, T, _, [], type(_, T, 0), _, _),
    modifierT(M, static),

    fieldDefT(F, T, type(_, T, 0), _, _),
    modifierT(F, static),

    getFieldT(_, _, M, _, _, F).

A static method in T returns an instance of T by accessing a static field that has type T

→ Singleton Pattern
Program Element Facts (PEFs) for Java

- Complete representation of Java 1.4 Abstract Syntax Tree
  - Files and packages
  - Interface elements (types and their members)
  - Code elements (Statements and expressions)
- See http://roots.iai.uni-bonn.de/research/jtransformer

- JTransformer
  - Eclipse Plug-In
  - Automatic creation and incremental update of PEFs for Java project
  - Development environment for program analyses and transformations
  - Reverse engineering of Java code from PEF representation
JTransformer Tutorial
– Program Representation and Analysis –
Double click on PEF shows reverse engineered source code

Context menu shows Java source in editor or internal representation in PEF navigator

Query factbase or run CTs
Conditional Transformations
### Conditional Transformations with JT

**Fixed set of Program Element Facts for Java 1.4:**

\[
\text{PEF} \in \{ \text{classDefT}(\text{Id}, \text{Pkg}, \text{Name}, \text{Members}), \\
\text{fieldDefT}(\text{Id}, \text{Class}, \text{Name}, \text{Init}), ..., \\
\text{applyT}(...), ... \}
\]

**Condition Language**

\[
C \rightarrow EC \land C \mid C \lor C \mid \text{not}(C) \\
EC \rightarrow \text{true} \mid \text{false} \mid \text{newId}(\text{Var}) \mid \text{pef} \in \text{PEF}
\]

**Transformation Language**

\[
T \rightarrow ET \mid ET, T \\
ET \rightarrow \text{add}(\text{pef}) \mid \text{delete}(\text{pef}) \mid \text{replace}(\text{pef}_1, \text{pef}_2) : \text{pef} \in \text{PEF}
\]
Create Accessor Method

- **AddGetter CT**
  - for all fields that have no getter method ...
  - ... add method that returns the field's value

```java
public class C {
    B b = new B();
    ...
}

public class C {
    B b = new B();
    B getB() {
        return b;
    }
    ...
}
```
AddGetter CT

tt(addGetter(Class,Field,Type,Getter), ( classDefT(Class,_,_,_),
not(externT(Class)),
fieldDefT(Field,Class,Type,Name,_,),
% No method with signature "<Type> get<Name>()" :
concat(get, Name, Getter),
not( methodDefT(Method,Class,Getter,[],Type,_,_) ),
% Identities of elements to be created:
new_id(Method),...,new_id(Get)
),
%
Create Method "<Type> get<Name>() { return <Field>}":
add( methodDefT(Method,Class,Getter,[],Type,[],Block) ),
add( blockT(Block,Method,Method,[Return]) ),
add( returnT(Return,Block,Method,Get) ),
add( getFieldT(Get,Return,Method,null,Name,Field) ),
add_to_class(Class,Method)
)).
CT are Parametric

- \( \text{CT}(V_1,\ldots,V_n) = \text{condition}(V_1,\ldots,V_n) \Rightarrow \text{transformation}(V_1,\ldots,V_n) \)

- **Variables** describe program elements
  - to be changed
  - needed to determine that the change is legal
  - needed to express context-dependent transformations

- **Application of a CT to a program**
  - \( \forall \) substitutions that make the precondition true:
    - \( \Rightarrow \) perform the transformations subject to these substitutions
CT are Parametric

- Application of a CT to a program
  - \( \forall \) substitutions that make the precondition true:
    \( \Rightarrow \) perform the transformations subject to these substitutions

\[
\begin{array}{c|c}
V_1 & \ldots & V_n \\
\hline
(Val_{11}, \ldots, Val_{1n}), \\
\vdots \\
(Val_{k1}, \ldots, Val_{kn})
\end{array}
\]

- \( CT(V_1, \ldots, V_n) = \text{condition}(V_1, \ldots, V_n) \Rightarrow \text{transformation}(V_1, \ldots, V_n) \)
Replace Read Accesses

- ReplaceReadAccesses
  - for all fields that have a getter method and
    for all read accesses to the field outside of its getter method...
  - ... replace the read access by the getter invocation

```java
public class C {
    B b = new B();
    B getB() {...}
    foo() {
        ...
        b.m();
    }
}
```

```java
public class C {
    B b = new B();
    B getB() {...}
    foo() {
        ...
        getB().m();
    }
}
```
Full „Encapsulate Field“ Refactoring

- Five CT definitions
  - AddGetter
  - AddSetter
  - ReplaceReadAccesses
  - ReplaceWriteAccesses
  - MakeFieldPrivate

- A CT sequence definition
  - invoke all of the above in the specified order

- Syntax of CT invocation in JTransformer
  - `apply_ct( Head )`: invoke a CT
  - `apply_ctlist( [Head1, ..., HeadN])`: invoke a sequence of CTs
Example: Refactorings as CTs

→ Demo

See „encapsulateField.pl“ file provided for the exercise at
Language-Parametric System

Is it possible to do everything for arbitrary languages...

... with the same system?
CTC: Language-Independent CTs

Alphabet of Program Element Terms (choose your own)

\[ \Sigma = \{ \text{classDefT(Id,Pkg,Name,Members)}, \text{fieldDefT(Id,Class,Type,Name,Init)}, \text{methodDefT(Id,Class,Name,...)}, \text{getFieldT(Id,...,Meth,Field)} \} \]

Condition Language

\[ C \rightarrow EC \land C \mid C \lor C \mid \text{not}(C) \]
\[ EC \rightarrow \text{true} \mid \text{false} \mid \text{exists}(\text{elem}) : \text{elem} \in \Sigma \]

Transformation Language

\[ T \rightarrow ET \mid ET, T \]
\[ ET \rightarrow \text{add}(\text{elem}) \mid \text{delete}(\text{elem}) \mid \text{replace}(\text{elem}_1,\text{elem}_2) : \text{elem} \in \Sigma \]
\[ \mid \text{new\_node\_id}(\text{Var}) \]
The Big Picture: Conditional Transformations

CTC Applications

- Aspect Mining
- Aspect Refactoring
- Aspect Analysis
- Aspect Compiler (LogicAJ)

CTC: Conditional Transformation Core

- Dependency analysis (Condor)
- Logic-based representation of models, analyses and transformations
- Composition (ConTraCT)

CT Interpreter (Analysis and Transformation)

- CTC Plug-In for Java (JTransformer)
- CTC Plug-In for ... (cooperations)

© 2006 Günter Kniesel, gk@cs.uni-bonn.de Logic-Based Software Analysis and Transformation
Defining Your Own Language

● 1. Define Program Element Terms
   ◆ Define the AST of your language

● 2. Specify Program Element Term structure in a standard form
   ◆ Use ast_node_def(Language, AST_Node_Type, Argument_Descriptions)

● 3. Implement a "Reader"
   ◆ A parser for your language that generates Prolog facts corresponding to the PET structure specified in 2.

● 4. Implement a "Writer"
   ◆ A generator that converts PETs into source code.

● 5. (Optional): Possibly provide a set of predefined conditions and CTs for working on Programs in your new language

● 2 - 5 are the elements of a plugin for the CTC that enables it to work with your language.
1. Defining Program Element Terms

- Syntax → Program Element Terms
  - Non-terminals are good candidates for PET types

- Determine attributes of PET types
  - Names, ...

- Determine relations between elements
  - e.g. extends(Sub,Super)

- Avoid mutual references

- Consider good database schema design rules
  - functional dependencies
  - normal forms (4NF)
2. Specification of PET structure

- PET structure: \texttt{fieldDefT(id\#, parent\#, type, name, expr\#)}

- Specified by:

  
  \begin{align*}
  &\text{language} & \text{PET type (= AST node type)} \\
  &\text{ast_node_def('Java', fieldDefT, [} \\
  &\quad \text{ast_arg(id, mult(1,1,no), id, [fieldDefT])}, \\
  &\quad \text{ast_arg(parent, mult(1,1,no), id, [classDefT])}, \\
  &\quad \text{ast_arg(type, mult(1,1,no), attr, [typeTerm])}, \\
  &\quad \text{ast_arg(name, mult(1,1,no), attr, [atom])}, \\
  &\quad \text{ast_arg(expr, mult(0,1,no), id, [...])}, \\
  &\quad \text{ast_arg(modifier, mult(0,*no), attr, [atom])}) \\
  \end{align*}

  \begin{itemize}
  \item \textbf{argument name} \\
  \item \textbf{multiplicity} (no = not ordered, ord = ordered) \\
  \item \textbf{order} (id = identity, attr = primitive) \\
  \item \textbf{kind of value} (multiple types are allowed, e.g. an expr, can have many)
  \end{itemize}

© 2006 Günter Kniesel, gk@cs.uni-bonn.de
3. Reader & 4. Writer

- **Reader**
  - Implement a parser
  - Possibly use an existing one and just implement a visitor on its internal AST that creates appropriate Prolog facts

- **Writer**
  - Can be implemented in Prolog, immediately creating source in your language
    - Look at file 'java_writer.pl' for an example how this was done for Java
  - Alternatively you can write out the factbase to a Prolog text file
    - Using the call 'writeTreeFacts(FileToWhichToWrite)'
  - ... and then use any tool you like to convert the generated text to your language syntax
State of Affairs

Done
- jTransformer: CT-based transformation engine for Java
- LogicAJ: Translation of a generic aspect language to CTs
- Condor: CT-based dependency analysis
- CT-Core (CTC): Core for language-parametric system

Ongoing
- *Transformer: language-parametric transformation tool
  - CTC with language-independent GUI tool à la JTransformer
- Integrate the subsystems
- Demonstrate their expressiveness on applications

http://roots.iai.uni-bonn.de/research