# Towards Customizable and Bi-directionally Traceable Transformation between VDM++ and Java

Fuyuki Ishikawa
National Institute of Informatics, Japan

# Motivation: Positioning in Dev. Process

- Write formal specification on the basis of the one in natural languages
  - Forced to remove ambiguity and to have some types of completeness and consistency (e.g., in type def.)
  - Validate through test, review, and other analyses

⇨ Should be reflected to the implementation

(not only improving and using the specification in natural languages)

  - Avoid dual cost of formalization (spec. and impl.)
  - Inherit what are validated to the implementation (VDM itself does not force to be fully formal for this, e.g., unlike stepwise refinement in B)

# Difficulty: Abstraction in VDM

```
class EventManager

instance variables
private t : real;
private s : set of real;
private user : token;
private state : State;
inv state.isValid();

operations
compute1 : nat ==> nat
compute1(x) == (
  let p in set s be st p in s and p > avg(s)
   in return round p * x;
)
pre forall i in set s & i <= t;
```

Don't care about "how" on computers
- real (don't say float or double)
- set (don't say HashSet or TreeSet)

May abstract away nonessential data structures

May include elements only for verification purpose, and may exclude implementation details (e.g., loggers)

Use declarative notations

# Motivation: Gaps between VDM and Impl.

- There are “VDM2Java” and “Java2VDM” tools
  - The implementation strategy is basically fixed by the code generator
  - Translation basically overwrites the other side
- ⇨ Difficulties in introducing and managing implementation-specific decisions
- The same stands for the VDM-UML Link tools (even in the interface or skeleton level)
- ⇨ How to correlate a class diagram of the VDM model with one of the Java code, with different abstraction levels (essentially in vocabularies)?

# Motivation: Gaps between VDM and Impl.

VDM

```
class TestClass

instance variables

private x : nat;
private a : real;
private b : real;
private c : set of int;
private state: State;

end TestClass
```

Java

```
public class TestClass{



  private int x;
  private double a;
  private float b;
  private HashSet<Integer>c;

  private Logger log;

}
```

# Motivation: Essential Requirement

*Necessary to distinguish and manage*

- What parts in formal specification (VDM) are essential decisions, inherited to implementation
  - As they are (possibly with syntax translation)
  - With additional decisions (e.g., how to implement on memory, using array? hash table?)
- What parts in formal specification (VDM) are tentative and not necessary in implementation
  - e.g., assertions, tentative mock to let it run early
- What additional parts are newly introduced in implementation
  - e.g., logger, encryption, exception handling

# Approach

VDM++ to Java transformation (not translation)

- Specify the gaps, or implementation decisions, explicitly as transformation rules
- ⇨ Customize, and explicitly keep traces

- Technical approaches
  - Specify transformation rules syntactically (not link invariants), and use them also for Java test code generation from VDM++ test specification
  - ⇨ Leverage lightweight usages (“specify, run and test”) supported in the current tools
  - Apply a bidirectional transformation theory and tool
  - ⇨ Explore potentials in “code to spec” change reflection

# Illustrating Usage Scenario

## 1. Describe a specification in VDM

**VDM**

```
class TestClass

instance variables

private x : nat;
private a : real;
private b : real;
private c : real;
private state : State;

end TestClass
```

# Illustrating Usage Scenario

## 2. Define transformation rules

VDM

```
class TestClass

instance variables

private x : nat;
private a : real;
private b : real;
private c : real;
private state : State;

end TestClass
```

Default Rules
- Implement the *nat* type as the *int* type
- Implement the *real* type as the *double* type

*(formal rule syntax omitted)*

Rules defined by Developers
- Implement the *real* type of variable *b* as the *float* type
- The variable *state* only exists in the model

# Illustrating Usage Scenario

## 3. Call the forward transformation

VDM

```
class TestClass

instance variables

private x : nat;
private a : real;
private b : real;
private c : real;
private state : State;

end TestClass
```

Java

```
public class TestClass{

  private int x;
  private double a;
  private float b;
  private double c;

}
```

# Illustrating Usage Scenario

4. Modify the generated codes if necessary
   (choosing whether to reflect insertions or not)

Java

```
public class TestClass{

  private int newx;
  private double a;
  private float b;
  private double c;
  private Logger log;
```

Rule
- The variable *log* exists only in implementation

# Illustrating Usage Scenario

## 5. Call the backward transformation

VDM

```
class TestClass

instance variables

private newx : nat;
private a : real;
private b : real;
private c : real;
private state : State;

end TestClass
```

Java

```
public class TestClass{

  private int newx;
  private double a;
  private float b;
  private double c;
  private Logger log;
```

Log of the previous forward transformation is used

# Test Case Inheritance

**VDM**

```
public op1 :
  seq of int ==> bool
op1(s) == … ;
```

*Test*

```
s = [1, 2, 3];
assertTrue(op1(s))
```

**Java**

```
public boolean
  op1(LinkedList<Integer> s){
    …
}
```

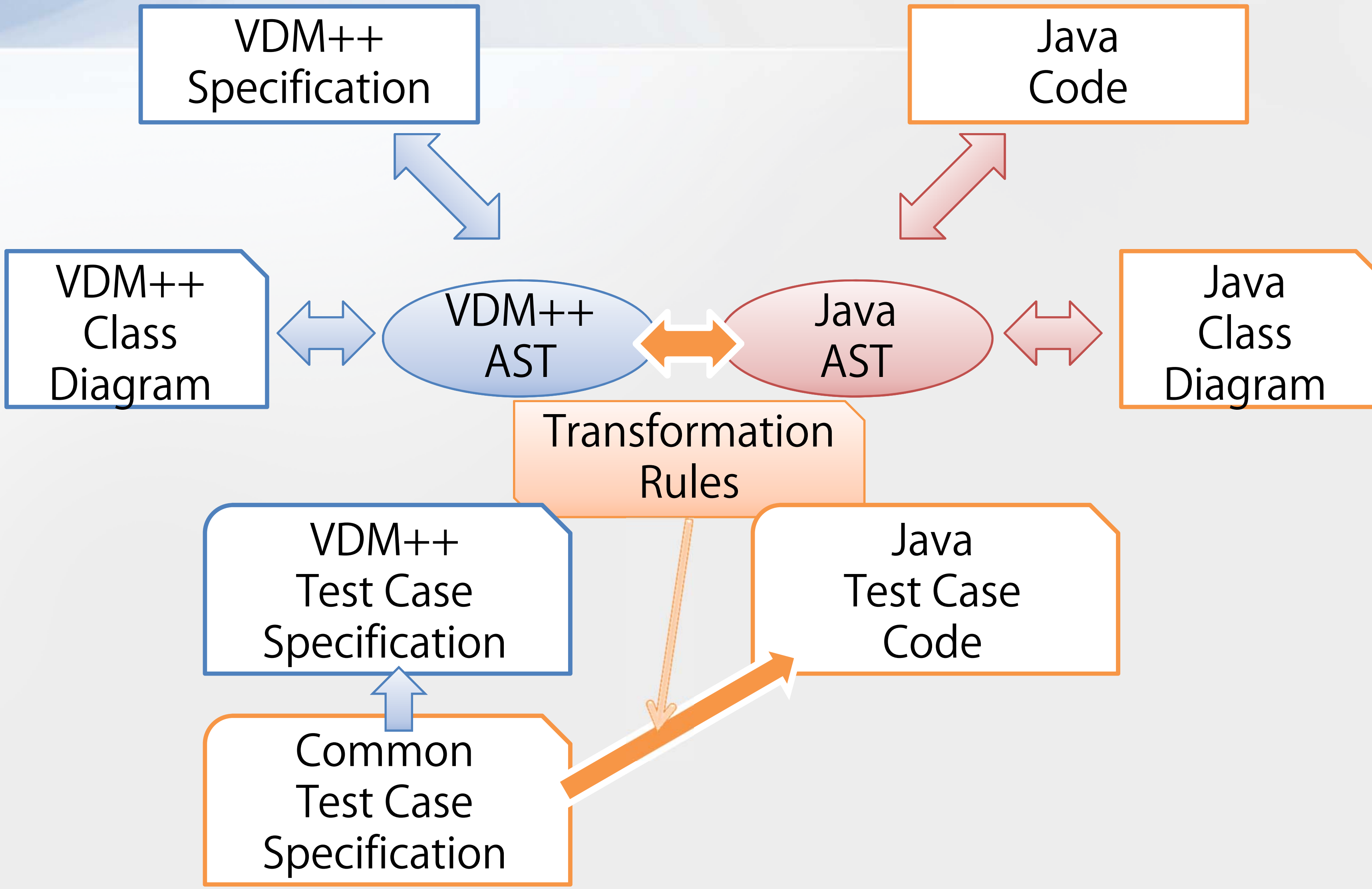
*Test*

```
s = new LinkedList();
s.add(1);
s.add(2);
s.add(3);
assert(op1(s) == true);
```

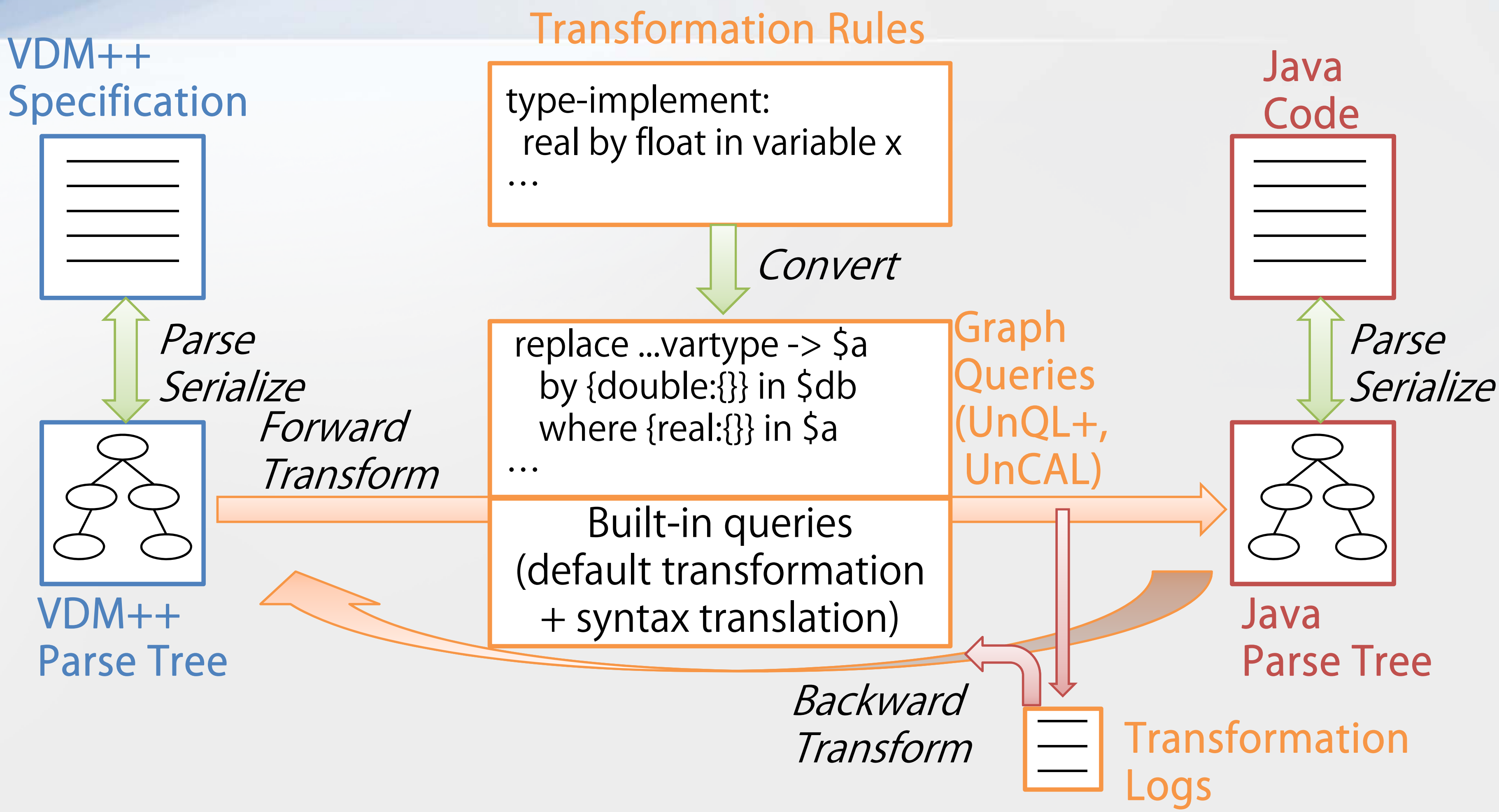
***Transformation rule***
*Implement the seq type as LinkedList in op1*

[reported in the 7th workshop at FM 2009]

# The Whole Picture

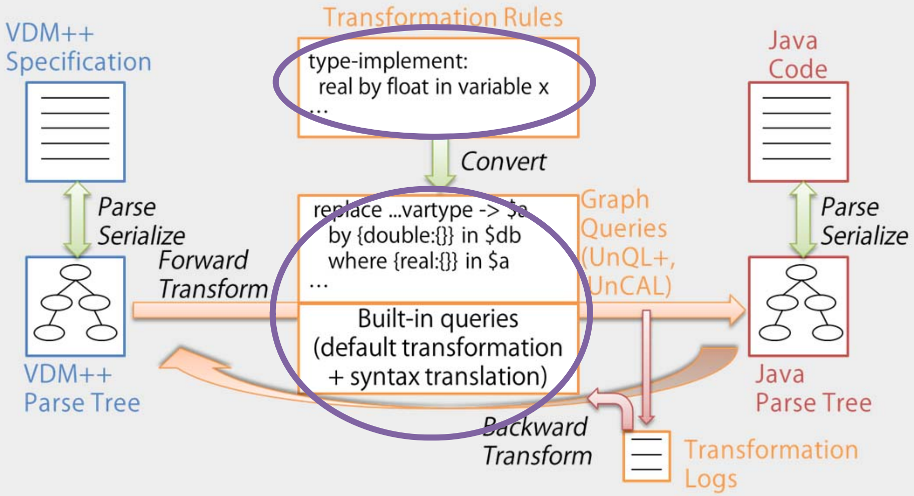


# Internal Mechanism: Overview



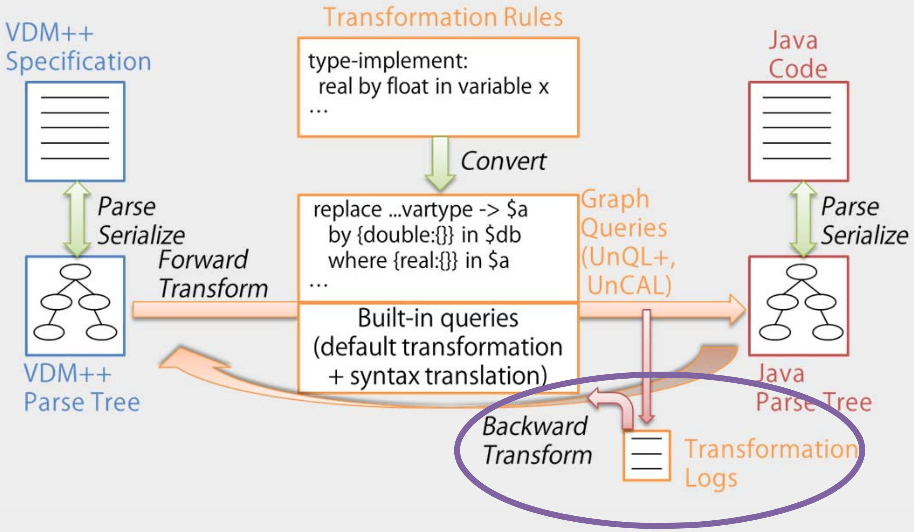
# Internal Mechanism: Transformation

- Rule expressivity: change (types), remove or add (variables, arguments, methods, sentences, …)
  - The underlying theory/tool potentially support "select", "*replace*", "*delete*" and "*insert*" operations on the parse trees
- Override: overriding rules are first applied, and then the defaults for remaining elements



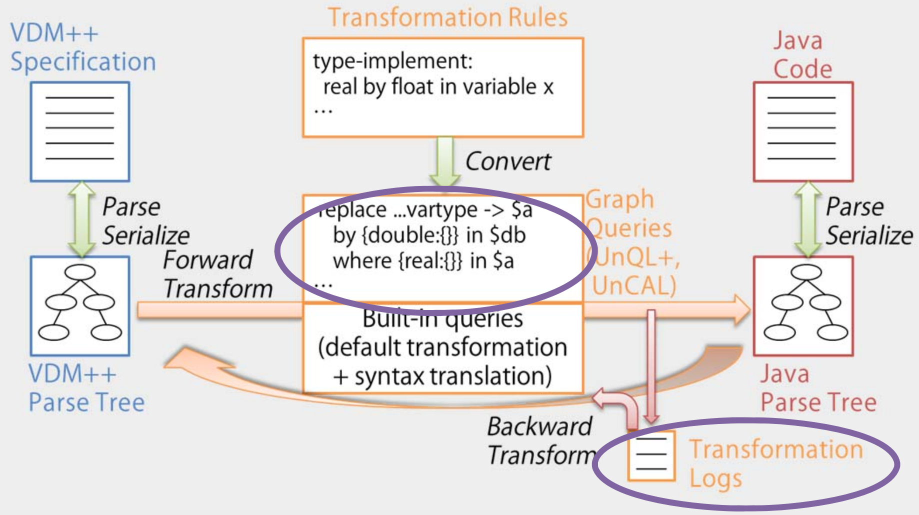
# Internal Mechanism: Bidirectionality

- Transformation supported by GRoundTram
  [ Hidaka, ICFP10 / http://www.biglab.org/ ]
  - Bidirectionality: *Suppose Java J is generated from VDM++ V, then J is modified into J', and VDM++ V' is generated from J'. Generation of Java code from V' results in J'.*
  - Limitation: insertion at the Java side may lead to possible multiple VDM++ (need user decision, or default)

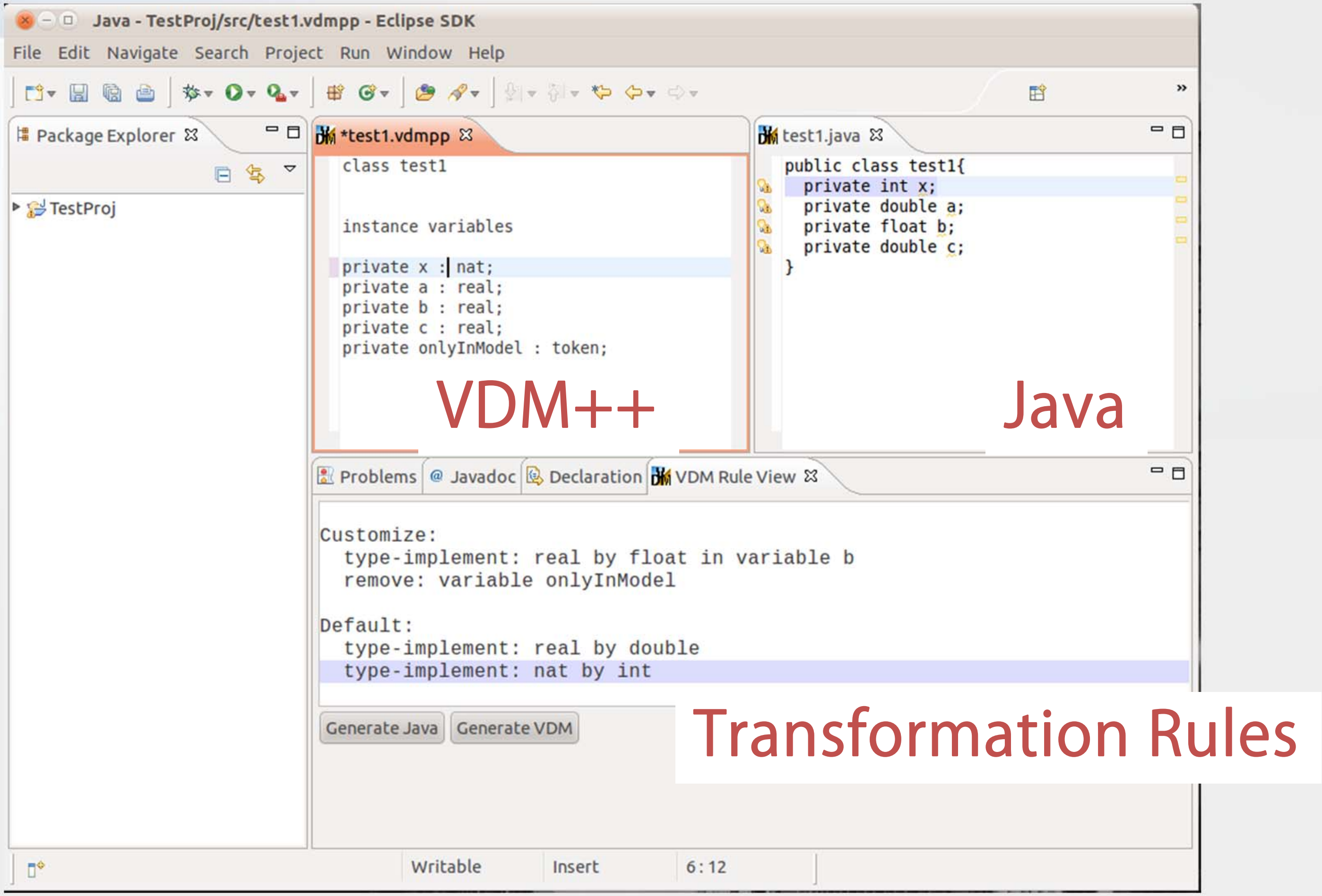


# Internal Mechanism: Traceability

- Extraction of corresponding parts
  - VDM-ReplaceRules/DeleteRules: A select query is generated from each replace/delete query to identify which parts of VDM++ are processed by it
  - VDM-Java: Transformation logs include from which VDM++ node each Java node is derived
    - InsertRules-Java: At the same time, it is possible to extract which Java nodes are newly inserted by each rule



# GUI Prototype




*Highlighting the corresponding parts in the two other views*

# Discussion (1): Expected Advantages

- Syntactical transformation with test case generation
- ⇨ Match with the lightweight “specify, run and test” usages primarily supported in the current tools
- Customization of code generation
- ⇨ Match most with situations where the (default) generated code is almost acceptable
  - e.g., customize the generated skeleton with implementation-specific types
  - e.g., situational applications without so tight NFR
- Traceability and bidirectional transformation
- ⇨ Match with iterated or derivative development, which often appears in many present projects

# Discussion (2): Limitations and Future Work

- A lot!
  - Coverage of the syntax by default rules
    - Libraries of domain-specific custom rules
  - Method for semantics validation
    - Validation of default rules (by the provider)
    - Validation framework for custom rules by users
  - Sophisticated user interface
    - Extraction of rules from Java code
  - Case studies and applications
  - …

Thank you!