Model Transformation
Table of Contents

- Features and categorization of model transformation tools
  - ATL
  - VIATRA
  - Moflon
  - Online demo
Model Transformation Tool Features I.

- Categorization based on [1]
- LHS and RHS of transformation rules
  - Variables
    - Untyped: no type information is checked
    - Syntactically typed: object assigned to a variable should be of a specific type
    - Semantically typed: “expression evaluating to an integer value”
  - Patterns
    - Form: string, term, graph
    - Syntax: abstract, concrete (textual, graphical)
    - Typing: untyped, syntactically typed, semantically typed
  - Logic
    - Non-executable: only specifies the relationship between models
    - Executable: supports model handling
      - Declarative: OCL queries / constraints to retrieve / implicitly create elements
      - Imperative: program that calls repository API to manipulate elements
Model Transformation Tool Features II.

- Syntactic separation of LHS and RHS
  - LHS and RHS are clearly separated: e.g., GT rule
  - Transformation rule implemented in Java (as a black box)
- Bidirectionality
  - Rule executable in 1 or 2 directions
- Rule parameterization
  - Transformation rules with control parameters
    - Configuration
    - Tuning
- Intermediate structures
Model Transformation Tool Features III.

- Rule application scoping
  - Restrictions the part of model participating in the transformation
  - Source vs. target

- Relationship of source and target models
  - New target model: always a new target model is created
  - Existing target model
    - In-place: source and target models are the same
    - Update: whether the target model is destructively manipulated
      - Destructive
      - Extension only

- Rule application strategy (i.e., match selection)
  - Deterministic (e.g., DFS along containment hierarchy)
  - Non-deterministic (single rule application, ‘as long as possible’, parallel)
  - Interactive
Model Transformation Tool Features IV.

- Rule scheduling
  - Form
    - Implicit: no way to influence scheduling (only via appropriate rule design)
    - Explicit: dedicated control flow constructs
      - External: separate control flow machinery
      - Internal: transformation rule invoke each other
  - Rule selection: explicit condition, non-deterministic choice, priorities, interactive

- Rule organization
  - Modularity mechanisms: packaging rules into modules
  - Reuse mechanisms: rule based on other rules
  - Organizational structure
    - Source model: actions attached to the elements of the source language
    - Target model: actions attached to the elements of the target language
    - Independent: no language preference
Model Transformation Tool Features V.

- **Traceability**
  - Implicit support: traceability links as regular model elements
  - Explicit support: traceability links and model elements are different
    - Manual vs. automated
  - Storage location: source, target, independent

- **Directionality**
  - Unidirectional
  - Bidirectional (model synchronization)
    - Bidirectional rules
    - Pairs of unidirectional rules
Model-to-Model Transformation Approaches I.

- Direct manipulation approaches
  - Model repository API available
  - All others (rules, control flow / scheduling) implemented from scratch

- Relational approaches: QVT
  - Relation of source and target element types by constraints
  - Execution engine based on logic programming (Prolog, etc.)
  - Declarative
  - Bidirectional
  - Side effect free
Model-to-Model Transformation Approaches II.

- Graph transformation based approaches: Moflon, VIATRA
  - Explicit LHS and RHS
  - Some additional logic (string, integer handling)
  - Declarative
- Structure-driven approaches: OptimalJ
  - Create the hierarchical structure of the target model
  - Set attributes and references in the target model
- Hybrid approaches: ATL
  - Combines different techniques
- Other approaches
  - CWM
  - XSLT
Table of Contents

- Features and categorization of model transformation tools
  - ATL
  - VIATRA
  - Moflon
  - Online demo
Atlas Transformation Language (ATL)

- http://www.eclipse.org/atl/
- Hybrid approach: declarative + imperative
- Textual syntax
- OCL-based queries (functionality similar to pattern matching)
- Workflow
  - ATL specification is compiled into a byte code
  - Byte code is executable (interpreted) by the ATL virtual machine
- Relationship of source and target models
  - Source models read only
  - Target model write only
- Automatic traceability support
  - Target elements are automatically connected to the corresponding source element
  - Automated explicit support with independent storage location
ATL Language Constructs

- Helper
  - Methods that can be invoked from different ATL constructs
- Query
  - Model to primitive type value transformation

```plaintext
module Src2Trg;
create OUT : SrcMM from IN : TrgMM;

helper def: gVar : Integer = 6;
	helper context Integer def: bisect() : Integer = self / 2;

query NumberOfSrcElems = SrcMM!SrcElem.allInstances() ->size().toString();
```
Matched Rule

- Declarative
- Source pattern
  - Objects of 1 type from the source metamodel
  - Condition: OCL constraint
- Target pattern
  - Target model elements to be created
  - Initialization statements
- Optional (imperative) action block
- Each object on the source side is matched *exactly once*
Matched Rule (Example)

```
rule SrcCont2TrgCont {
  from
    sCont : SrcMM!SrcCont(elements.size() > 20)
  to
    tCont : TrgMM!TrgCont {
      children <- sCont.elements
    }
}

rule SrcElem2TrgElem {
  from
    s : SrcMM!SrcElem
  to
    t : TrgMM!TrgElem {
      name <- s.name
    }
  do {
    t.id <- thisModule.gVar.bisect();
  }
}
```
Called Rule

- Imperative
- ~ Procedure with name and parameters
- Target pattern
- Optional action block
- Can be invoked explicitly many times
Called Rule (Example)

```java
rule SrcCont2TrgCont {
  from
      sCont : SrcMM!SrcCont
  to
      tCont : TrgMM!TrgCont()
  do {
      for (sC in SrcMM!SrcCont.allInstances()) {
          if (sCont.elements.size() > 20) {
              tCont.children <- thisModule.NewTrgElem(sCont.name);
          }
      }
  }
}

rule NewTrgElem(inName:String) {
  to
      t : TrgMM!TrgElem {
          name <- inName
      }
  do {
      t.id <- thisModule.gVar.bisect();
      t;
  }
}
```
Execution Semantics

- Transformation execution in 3 phases
  - Module initialization
  - Matching
    - Match source model elements
    - Allocate target model elements
  - Target model elements initialization
    - Set attributes and references of the target model elements
    - Use thisModule.resolveTemp() operation
- resolveTemp(var, targetElementName)
  - Applies source-to-target mapping for source element var
  - Name of the target element must be targetElementName (as a string)
  - Returns the target element
ATL Categorization

- LHS and RHS of transformation rules
  - Syntactically typed variables
  - Syntactically typed, graph patterns in textual concrete syntax
  - Declarative and executable approach (on the rule level)

- Unidirectional

- Rule parameterization: partially (called rule)

- Rule application scoping: on the source model

- Rule application strategy: deterministic

- Rule scheduling: implicit (matched rule) or internal explicit (called rule)

- Rule selection: explicit condition

- Rule organization
  - Rules can be packed into modules
  - Rule inheritance
  - Independent organizational structure
Table of Contents

- Features and categorization of model transformation tools
  - ATL
  - VIATRA
  - Moflon
  - Online demo
Visual Automated Model Transformation (VIATRA)

- Graph transformation based approach
  - Declarative GT rules
  - Imperative control flow (based on abstract state machines (ASM))
- Textual syntax
- 4 major parts
  - Term evaluation (i.e., non-structural (e.g., attribute) checks)
  - Graph patterns
  - GT rules
    - LHS + RHS (+ optional imperative ASM rules)
  - ASM rules
- Interpreter
- Own model representation layer (i.e., not EMF-based internal model representation)
Graph Patterns in VIATRA

- Graph pattern: declarative specification by a set of structural constraints
  - Parameters: in/out style
  - If specified: restricts the search
  - If not specified: calculated by pattern matching
- Pattern alternatives
- Pattern invocation: recursive structures can be expressed

```java
pattern moduleWithInputPattern(M, P) = {
    module(M);
    module.inputs(I, M, P);
    port(P);
}
pattern simpleEdge(Src, Trg) = {
    node(Src);
    node.edge(Edge, Src, Trg);
    node(Trg);
} or {
    node(Src);
    node.edge(OtherEdge, Src, Middle);
    find simpleEdge(Middle, Trg);
}
```
Graph Transformation Rules in VIATRA

- Different representations
  - LHS + RHS (+ optional action block)
  - Graph pattern + modifications (+ optional action block)
- Patterns and graph transformation rules can be used at several locations
  - Similar to method declaration and invocation in Java

```java
grule sampleGTRule(in M, inout P) = {
  precondition pattern lhs(I, M, P) = {
    module(M);
    module.inputs(I, M, P);
    port(P);
  }

  postcondition pattern rhs(O, M, P) = {
    module(M);
    module.outputs(O, M, P);
    port(P);
  }

  action {
    print("Message");
  }
}
```
ASM-Based Control Flow in VIATRA

- Abstract state machines (proposed by Yuri Gurevich [2])
  - Formal mathematical framework
  - For semantic description purposes
  - Uses programming language constructs

- Constructs
  - ASM functions (~ globally accessible functions, i.e., key-value pairs)
  - ASM rules
    - User-defined (~ methods with in, out, inout parameters)
  - Simple: `skip`, `fail`, `print(X)`, `update X = Y`, `call asmRule(X)`
  - Composite
Composite ASM Rules (Example)

```
rule main(in M) = let C1 = false, PTemp = undef in seq {
    // Pattern matching as a condition of an if statement
    if (find moduleWithInputPattern(M, PTemp))
        skip
    else
        fail
    // Calls another ASM rule
    call anotherASMRule(M);
    // As long as possible rule application
    iterate
        choose P apply sampleGTRule(M, P) do
            print("Successfully applied GT rule");
    // Apply rule in parallel
    forall P2 apply sampleGTRule(M, P);
    // Pattern matching that finds all values for P2 (with a fix M)
    forall P2 with find moduleWithInputPattern(M, P2) do
        print(name(Port));
}
```
Composite ASM Rules (Summary)

- If-then-else rule
- Try-else rule (≈ try-catch block in Java)
- Sequential rule (seq { })
- Random rule (random { })
- Local variable declaration and initialization (let)
- Iteration (iterate) until the following ASM rule is evaluated to true
- Non-deterministic match calculation or GT rule application
  \[
  \text{choose} \ Var \ \text{with} \ \text{patternOrGTRule}(\Var) \ \text{do} \ \text{ASMRule}
  \]
- Calculation of all matches or parallel GT rule application
  \[
  \text{forall} \ Var \ \text{with} \ \text{patternOrGTRule}(\Var) \ \text{do} \ \text{ASMRule}
  \]
VIATRA Categorization

- LHS and RHS of transformation rules
  - Untyped or syntactically typed variables
  - Untyped or syntactically typed graph patterns in a textual concrete syntax
  - Declarative and executable approach (on the rule level)
- LHS and RHS are clearly separated
- Unidirectional
- Rule parameterization
- Relationship of source and target models
  - In-place, destructive transformation
- Rule application strategy (i.e., match selection)
  - Non-deterministic with single, ‘as long as possible’, and parallel rule application
- Explicit external rule scheduling
- Independent rule organization structure
  - Rules can be packed into modules
  - Reuse mechanisms: pattern and GT rule invocation
- Traceability: implicit (regular model elements in the common model space)
Table of Contents

- Features and categorization of model transformation tools
- ATL
- VIATRA
- **Moflon**
- Online demo
EMoflon Features

- LHS and RHS of transformation rules
  - Syntactically typed variables
  - Syntactically typed graph patterns in graphical concrete syntax
  - Declarative and executable approach (on the rule level)
- LHS and RHS are clearly separated
- Unidirectional + Bidirectional rules
- Rule parameterization
- Relationship of source and target models
  - Unidirectional: target model is destructively manipulated
  - Bidirectional: new target model, extension only strategy
- Rule application strategy (i.e., match selection)
  - Non-deterministic with support for single and ‘as long as possible’ rule application
- Explicit external rule scheduling
- Independent rule organization structure
- Traceability
  - Unidirectional: implicit support
  - Bidirectional: automated (explicit?) support with independent storage location
Specifying Control Flow in eMoflon I.
Specifying Control Flow in eMoflon II.

- Activity diagram (nodes + edges)
  - Specifies the content of a method
- Initial (start) node
  - Object (this) and parameters
- Final (stop) node
  - Return value
- Story node
  - Single graph transformation step
- For each story node
  - GT with ‘as long as possible’ execution
- Statement node
  - Originally: arbitrary Java code
    - Is this a model-driven approach?
  - Now: method invocation

M:Module
P:Port
O:outputs
I:inputs
P.method()
Specifying Control Flow in eMoflon III.

- Edge (transition)
- Guard
  - NONE
  - SUCCESS (true)
  - FAILURE (false)
  - EACH_TIME
  - END
Structural Restrictions I.

- Story node without guard
  - 1 outgoing edge with NONE guard
  - Perform a regular graph transformation step
    - Pattern matching
    - Modifications
  - Follow the outgoing edge
- Statement node without guard
  - 1 outgoing edge with NONE guard
  - Perform the statement
    - Invoke method() on the object mapped to P
  - Follow the outgoing edge
Structural Restrictions II.

- Story node as a decision point
  - 1 SUCCESS edge and 1 FAILURE edge
  - Is pattern matching successful?
  - Yes:
    - All nodes are bound by PM
    - Perform modifications
    - Follow the SUCCESS edge
  - No:
    - Nodes might be unbound (null)
    - No modifications
    - Follow the FAILURE edge
Structural Restrictions III.

- For each story node
  - 1 END edge and 0..1 EACH_TIME edge
  - Is pattern matching successful?
  - Yes:
    - All nodes are bound by PM
    - Perform modifications
    - If EACH_TIME edge exists
      - Follow the EACH_TIME edge
        - Activities must end at the for each story node
    - Go to the pattern matching step
  - No:
    - No modifications
    - Follow the END edge
Complex Structural Restrictions

- Result of the design process: Java code is generated
- Can Java code be generated from an arbitrary activity diagram?
  - No
    - Previous syntactic restrictions
    - Additional structural analysis required

```
[Failure]  

[Success]  

[Success]  

[Success]  

[Failure]  

[Failure]  
```

3. Mai 2012 | Real-Time Systems Lab | Prof. Dr. Andy Schürr | Dr. Gergely Varró | 35
For Each Structure
Head-Controlled Loop (While Loop)
Tail-Controlled Loop (Do Loop)
If-Then-Else Structure

[Diagram of an If-Then-Else structure with states labeled Success and Failure, and transitions indicated by arrows.]
Complex Structural Analysis: An Example
Variable Binding Analysis

Story diagram for the Port.handle() method

Module M was bound in the previous step

Port.handle() returns P

M:Module

this:Port

this Object is bound by default

M:Module

P:Port = M.method()

The return value of method() defines port P

P:

O:outputs

O2:outputs
Thesis Ideas

- Visual appearance for the logic gate network simulator
- SDM validation
References

(1) K. Czarnecki, S. Helsen: Classification of Model Transformation Approaches

Table of Contents

- Features and categorization of model transformation tools
- ATL
- VIATRA
- Moflon
- Online demo
Online Demo I.

entity GATE1 is
  port(A:in std_logic;
       B:in std_logic;
       F1:out std_logic);
end GATE1;

architecture behv of GATE1 is
begin
  F1 <= A and B;
end behv;

entity GATE2 is
  port(X:in std_logic;
       Y:in std_logic;
       F2:out std_logic;
       F3:out std_logic);
end GATE2;

architecture behv of GATE2 is
begin
  F2 <= X or Y;
  F3 <= not X;
end behv;

entity COMB is
  port(i1:in std_logic;
       i2:in std_logic;
       i3:in std_logic;
       o:out std_logic);
end COMB;

architecture behv of COMB is
signal wire: std_logic;
begin
  GATE1 port map (A => i1,
                  B => i2,
                  F1 => wire);
  GATE2 port map (X => wire,
                  Y => i3,
                  F2 => o);
end behv;
Online Demo III.