Precise Model Transformations in Tool Integration

Dániel Varró
Assistant Professor
Overview - Outline

- Introduction
  - Challenges for tool integration
  - Model-driven approach to tool integration

- The VIATRA2 framework
  - Using a combination of formal methods
  - Abstract state machines and graph transformation

- Success Scenarios
  - SENSORIA
  - DECOS
Challenges for Software Development

A typical design process of a large system involves
- Many stakeholders
- Many development teams
- Many manmonths
- MANY TOOLS
  - Requirements, Analysis, Design, Testing, Maintenance, ...

Tool integration is a major challenge
- Design of Embedded / Critical systems:
  - Cost of tool integration ≈ Cost of the tools themselves

Why?
- Continuous evolution / changes of tools
- Each having its own (modeling / programming) language
- Difficult to build correct and robust bridges between them

Budapest University of Technology and Economics
Fault-tolerant Systems Research Group
Motivating scenario

A car is broken down...

1. discover services
   - Car description
   - Policies
   - GPS data
   - Credit Card data

2a. list services
   - garage
   - car rental
   - tow truck

2b. deposit
   - amount
   - validity
   - certification

3. order services
   - garage
   - tow truck
   - car rental

4a. arrange repair
   - service type
   - deadline
   - price

4b. send car
   - type
   - location
   - price

4c. send truck
   - type
   - location
   - price

5. confirm / cancel
Motivating scenario

We aim to provide justified answers for questions like:

- Is it true that the credit card of the driver will not be charged if there are no available garages?
- Will a tow truck be sent within 15 minutes?
- Will the driver have a proper car ordered automatically?

Diagram:

1. discover services
   - Car description
   - Policies
   - GPS data
   - Credit Card data

2a. list services
   - garage
   - car rental
   - tow truck

2b. deposit
   - amount
   - validity
   - certification

4c. send truck
   - type
   - location
   - price

5. confirm / cancel

Diagram flow:
- Car rental
- Garage
- Tow truck
- Deposit
- Confirm / cancel
How to provide answers?

No single tool which solves all these problems

How can they cooperate?

Tool 1:
Captures service models

Tool 2:
No deposit will be charged if service is cancelled.
The driver will get everything according to his policy if payment if confirmed.

Tool 3:
The car will arrive within 15 minutes with 90% probability.
The GPS location service is a quality bottleneck.

Tool 4:
Standard deployment code generated in WSDL.

Tool 4: Allows to integrate Tool 1-4
SENSORIA approach

- Service-oriented extension of standard visual modelling techniques
- Custom model-driven SOA development process
- Tool integration by model transformations
- Use of integrated, hidden mathematical methods to improve the quality of the overall service
- Deployment process which targets popular service-oriented platforms
Model Transformations for Tool Integration

- System design
  - (Semi-)formal specification, system model
  - Automated model generation
  - Back-annotation
  - Code generation
  - Implementation / Configuration

- Mathematical analysis
  - Mathematical model
  - Analysis

VIATRA2 = VIsual Automated model TRAnformations: a general-purpose model transformation framework
The VIATRA Model Transformation Framework
The VIATRA2 framework

Native source models:
- EMF
- XML documents
- Databases
- Domain-specific models
- UML

Native target models:
- EMF
- App source code
- XML deployment descriptor
- Databases
- Analysis tools

Vendor’s own tool
Manually written native program
Native Target model
The VIATRA2 framework

Metamodel: Precise spec of a modeling language (see Eclipse EMF - Ecore)

Transformation: How to generate a target equivalent of an arbitrary source model

VIATRA Model Space (VPM)

Source metamodel → Xform. rules (GT+ASM) → Target metamodel

Source model → Model: Description of a concrete system (see Eclipse EMF)

Native Source model → Native Target model

Vendor’s own tool
The VIATRA2 framework

Eclipse framework

VIATRA2 Model Transformation Plug-in

VIATRA Model Space (VPM)

Source metamodel → Xform. rules (GT+ASM) → Target metamodel

Source model → Xform engine → Target model

Transformation engine: Support for querying and manipulating large models

Vendor’s own tool

Native Source model → Native Target model
The VIATRA 2.0 framework

Eclipse framework

VIATRA2 Model Transformation Plug-in

VIATRA Model Space (VPM)

Source metamodel

Source model

Xform. rules (GT+ASM)

Xform engine

Target metamodel

Target model

Standalone transformation: Independent of transformation development environment

Native Source model

Native XForm Plugin

Native Target model

Vendor’s own tool
The VIATRA2 framework

Eclipse framework

VIATRA2 Model Transformation Plug-in

Transformation Design

Transformation Execution

Vendor’s own tool
The VIATRA2 Approach

- **Model management:**
  - **Model space:** Unified, global view of models, metamodels and transformations
    - Hierarchical graph model
    - Complex type hierarchy
    - Multilevel metamodeling

- **Model manipulation and transformations:**
  integration of two mathematically precise, **rule** and **pattern-based** formalisms
  - Graph patterns (GP): structural conditions
  - Graph transformation (GT): elementary xform steps
  - Abstract state machines (ASM): complex xform programs

- **Code generation:**
  - Special model transformations with
  - Code templates and code formatters
Metamodeling in VIATRA2 (VTML)

```
entity(family);
relation(members, family, person);
isAggregation(family.members, true);
entity(person) {
  relation(parent, person, person);
  relation(father, person, man);
  multiplicity(father, many_to_one);
  supertypeOf(parent, father);
  relation(firstname, person, datatypes.String);
}
entity(woman) {
  relation(husband, woman, man);
  multiplicity(husband, one_to_one);
}
supertypeOf(person, woman);
entity(man) {
  relation(wife, man, woman);
  inverse(wife, woman.husband);
}
supertypeOf(person, man);
```
Metamodelling in VIATRA2 (VTML)

```vtml
entity(family);
relation(members, family, person);
isAggregation(family.members, true);
entity(person) {
    relation(parent, person, person);
    relation(father, person, man);
    multiplicity(father, many_to_one);
    supertypeOf(parent, father);
    relation(firstname, person, datatypes.String);
}
entity(woman) {
    relation(husband, woman, man);
    multiplicity(husband, one_to_one);
}
supertypeOf(person, woman);
entity(man) {
    relation(wife, man, woman);
    inverse(wife, woman.husband);
}
supertypeOf(person, man);
```
namespace people.models;
import people.metamodel;
import datatypes;
family('Varro1') {
    man('Gyozo') {
        man.wife(wf1, Gyozo, Maria);
    }
    woman('Maria') {
        woman.husband(hb1, Maria, Gyozo);
    }
    family.members(mb4, 'Varro1', 'Daniel');
    family.familyname(fn, 'Varro1', str);
    String(str) -> "Varro";
}

man(active, 'Gyozo');
father(father, 'Gyozo');
mother(mother, 'Gyozo');
wife(wife, 'Gyozo');
instance('Gyozo', woman, man);
instanceOf('Gyozo', woman, man);
daniel(daniel, 'Daniel');
gergely(gergely, 'Gergely');
person.father(f1, 'Gergely', 'Gyozo');
person.mother(m1, 'Gergely', 'Maria');
family.members(mb1, 'Varro1', 'Gyozo');
family.members(mb2, 'Varro1', 'Maria');
family.members(mb3, 'Varro1', 'Gergely');
family.members(mb4, 'Varro1', 'Daniel');
family.familyname(fn, 'Varro1', str);
String(str) -> "Varro";
The VIATRA2 Approach

- **Model management:**
  - **Model space:** Unified, global view of models, metamodels and transformations
    - Hierarchical graph model
    - Complex type hierarchy
    - Multilevel metamodeling

- **Model manipulation and transformations:**
  Integration of two mathematically precise, rule and pattern-based formalisms
  - Graph patterns (GP): structural conditions
  - Graph transformation (GT): elementary xform steps
  - Abstract state machines (ASM): complex xform programs

- **Code generation:**
  - Special model transformations with
  - Code templates and code formatters
**Pattern definition**

**pattern** brother(X,B)

**check** (X ≠ B)

**matching**

- m1: mother
- f1: father
- m2: mother
- f2: father

**Instance Model**

**Graphical notation**

- mother
- father
- wife/husband
- man
- woman

- **Graph Pattern:**
  - Structural condition that have to be fulfilled by a part of the model space

- **Graph pattern matching:**
  - A model (i.e. part of the model space) can satisfy a graph pattern,
  - if the pattern can be matched to a subgraph of the model
Graph patterns (VTCL)

// B is a brother of X
pattern brother(X, B) = {
    person(X);
    person.parent(P1, X, P);
    person(P);
    person.parent(P2, B, P);
    man(B);
    check (X != B)
}

// S is a sister of X
pattern sister(X, S, F) = {
    person(X) in F;
    person.father(P1, X, M);
    man(M) in F;
    person.father(P2, S, M);
    woman(S) in F;
    or {
        person(X) in F;
        person.mother(P1, X, W);
        woman(W) in F;
        person.mother(P2, S, W);
        woman(S) in W;
    }
}
Negative patterns (VTCL)

**Pattern mayMarry(M,W)**

- **F1**: Family
  - **MB1**: members
    - **M**: Man
      - **X**: Man
      - **neg find married**
  - **F2**: Family
    - **MB2**: members
      - **W**: Woman
      - **X**: Woman
      - **neg find married**

- **Check** \((F1 \neq F2)\)

**Pattern married(X)**

- **X**: Man
  - **P1**: person
  - **W**: Woman
  - **OR**
- **M**: Man
  - **P1**: person
  - **X**: Woman

\[
\text{Pattern mayMarry}(X, D) = \{
\text{man}(M);
\text{family}(F1);
\text{family.members}(MB1, F1, M);
\text{woman}(W);
\text{family}(F2);
\text{family.members}(MB2, F2, W);
\text{neg find married}(M);
\text{neg find married}(W);
\text{check}(F1 \neq F2);
\}\]

\[
\text{Pattern married}(X) = \{
\text{man}(X);
\text{man.wife}(WF, X, W);
\text{woman}(W);
\}\ \text{or} \{
\text{woman}(X);
\text{man.wife}(WF, M, X);
\text{man}(M);
\}\]
Recursive patterns (VTCL)

\[
\text{pattern descendants}(X, D) = \{
\text{person}(X); \\
\text{person}.\text{parent}(P2, Ch, X); \\
\text{person}(Ch); \\
\text{find descendants}(Ch, D) \\
\text{person}(D); \}
\text{or}
\{
\text{person}(X); \\
\text{person}.\text{parent}(P1, D, X); \\
\text{person}(D); \\
\}
\]
Graph transformation rules (VTCL)

**precondition pattern**
\[ \text{lhs}(M, W, F_1, MB_1, F_2, MB_2) = \{
    \text{family}(F_1);
    \text{family.members}(MB_1, F_1, M);
    \text{man}(M);
    \text{family}(F_2);
    \text{family.members}(MB_2, F_2, W);
    \text{woman}(W);
    \text{neg find married}(M);
    \text{neg find married}(W);
\} \]

**postcondition pattern**
\[ \text{rhs}(M, W, F_1, MB_1, F_2, MB_2, F) = \{
    \text{family}(F_1);
    \text{man}(M);
    \text{family}(F_2);
    \text{woman}(W);
    \text{family}(F);
    \text{family.members}(MB_3, F, M);
    \text{family.members}(MB_4, F, W);
\} \]

**gtrule marry**(in \( M \), in \( W \), out \( F \)) =

**precondition pattern**
\[ \text{lhs}(M, W, F_1, MB_1, F_2, MB_2) = \{
    \text{family}(F_1);
    \text{family.members}(MB_1, F_1, M);
    \text{man}(M);
    \text{family}(F_2);
    \text{family.members}(MB_2, F_2, W);
    \text{woman}(W);
    \text{neg find married}(M);
    \text{neg find married}(W);
\} \]

**postcondition pattern**
\[ \text{rhs}(M, W, F_1, MB_1, F_2, MB_2, F) = \{
    \text{family}(F_1);
    \text{man}(M);
    \text{family}(F_2);
    \text{woman}(W);
    \text{family}(F);
    \text{family.members}(MB_3, F, M);
    \text{family.members}(MB_4, F, W);
\} \]
Graph transformation rules (VTCL)

precondition pattern

\[ \text{lhs}(M,W,F_1,MB_1,F_2,MB_2) = \{ \]

- \( F_1: \text{Family} \)
- \( MB_1: \text{members} \)
- \( M: \text{Man} \)
- \( X \)
- \( \text{neg find married}(M) \)

- \( F_2: \text{Family} \)
- \( MB_2: \text{members} \)
- \( W: \text{Woman} \)
- \( X \)
- \( \text{neg find married}(W) \)

\[ \} \]

action

\{ 
- delete(MB_1);
- delete(MB_2);
- new(family(F));
- new(family.members(MB_3, F, M));
- new(family.members(MB_4, F, W));
- new(man.wife(WF_1, M, W));
\}
**Generic GT rules (VTCL)**

**gtrule** `parentIsAncR(Par, Child)`

**precondition** `lhs(Par, Child)`

- `Par:Class`
- `Child:Class`
- `P1:parent`

**postcondition** `rhs(Par, Child)`

- `Par:Class`
- `Child:Class`
- `E:anc`
- `P1:parent`

**gtrule** `parentIsAncR(Par, Child, ClsE, ParR, AncR)`

**precondition** `lhs(Par, Child, P1, ClsE, ParR, AncR)`

- `Par:entity`
- `Child:entity`
- `ClsE:entity`
- `P1:relation`
- `ParR:relation`
- `AncR:relation`

**postcondition** `rhs(Par, Child, P1, ClsE, ParR, AncR)`

- `Par:entity`
- `Child:entity`
- `ClsE:entity`
- `ParR:relation`
- `AncR:relation`
Abstract State Machines

- **ASM:** high-level specification language
  - Control structure for xform
  - Integrated with GT rules

- **Examples**
  - `update location = term;`
  - `parallel {...} / seq {...}
  - `let var = term in rule;`
  - `if (formula) rule1; else rule2;`
  - `iterate rule;`
  - `forall/choose variables with formula do rule;`
  - `forall/choose variables apply gtrule do rule;`

```
forall X below people.models, B below people.models
with find brother(X, B) do seq {
    print(name(X) + "->" + name(B));
}
```

```
let X = people.models.Varro1.Daniel,
Y = people.models.Gyapay1.Szilvia,
F = undef, F2 = undef in
  choose Z below people.models
apply marry(X, Y, F)
do seq {
    rename(F, "Varro2");
    move(F, people.models);
    iterate choose M below people.models, W below people.models
    apply marry(M, W, F2)
do move(F2, people.models);
```

**GT extensions to ASMs**
- Permanent states (models)
- Elementary model manipulation
- Graph pattern matching as advanced logical conditions / ASM functions

Note that:
- extensions are syntactic sugars from spec point of view
- they were specified by ASMs in my PhD thesis
The VIATRA2 Approach

- **Model management:**
  - **Model space:** Unified, global view of models, metamodels and transformations
    - Hierarchical graph model
    - Complex type hierarchy
    - Multilevel metamodeling

- **Model manipulation and transformations:** integration of two mathematically precise, **rule** and **pattern-based** formalisms
  - Graph patterns (GP): structural conditions
  - Graph transformation (GT): elementary xform steps
  - Abstract state machines (ASM): complex xform programs

- **Code generation:**
  - Special model transformations with
  - Code templates and code formatters
Code templates

- Code generation
  - Code templates
  - Code formatters
- Code templates
  - Text block with references to GTASM patterns, rules
  - Compiled into GTASM programs with prints
- Velocity templates

- Code formatters
  - Split output code into multiple files
  - Pretty printing

```
// Generated
rule printClass(in C) = seq {
  print("public class " + C + ":");
  forall At,Typ with attrib(C,At,Typ) do {
    print("private " + Typ + " " + At + ";");
  }
  print("}");
}
```
Other advanced MT issues

- Reusability
  - Pattern calls
  - Rule calls
- Traceability
  - Reference metamodels
  - Explicit storage of reference models
- External transformations
- Live transformations vs. Batch transformations
Transformation engine(s)
Why Not XSLT?

- **Mathematical answer**: Models are not trees but graphs
  - XSLT is for transforming trees
  - GT is for transforming graphs

- **Practical answer**: Problems of XSLT
  - Low level of abstraction ➔ difficult to write complex xforms
  - Low performance ➔ calling other templates is expensive
  - Maintainability ➔ difficult to understand an XSLT code written by other developers
  - Lack of support for
    - Model synchronization / Incremental transformations
    - Multiple Input / Multiple Output transformations
    - …
Model-specific search plans

- Rule (LHS)
  - Typical model 1
    - + statistics
  - Typical model 2
    - + statistics

Algorithm for generating search plans

- Search plan 1
  - cost function
  - Estimate for the size of SST that would be traversed

- Search plan 2
  - cost function

Optimal behaviour for typical model 1

Optimal behaviour for typical model 2

Design/Compile time

Execution time

- Current model
  - + statistics

Pattern matching
Adaptive PM approach

- Rule (LHS)
  - Typical model 1
    - + statistics
  - Typical model 2
    - + statistics

Design/Compile time

Execution time

- Current model
  - + statistics
- Strategy selection
- Pattern matching

Search plan 1
- cost function

Search plan 2
- cost function

Pattern matching
Model Synchronization Problem

- **Batch Approach**
  - Restart transformation from scratch each time

- **Incremental Approach**
  - Store partial matches of patterns
  - Propagate model changes
  - Apply xform for novel model parts

- Implemented by adapting RETE networks
Development of Model Transformations
Problems with traditional MT solution

- Transformation designer need to know
  - Source and target languages
  - Transformation technology
    (how to write rules)

- Source and Target languages significantly
differ from transformation languages
Model Transformation by Example

- Start with prototype mapping models (PMM)
  - sample source and target models enriched with interconnection
- Automatically generate transformation rules

**Advantage:**
- No need to directly write transformation rules
- Sufficient to know the source and target languages

**MTBE = Model Transformation By Example**
Summary of VIATRA

- **VIATRA:** provides a rule and pattern-based language for uni-directional model transformations

- **Main concepts**
  - multi-level metamodeling + model space
  - transformation language
    - graph transformation
    - abstract state machines
  - template-based code generation.

- **Added value:**
  - Rich specification language
  - Advanced execution strategies
  - Usable for tool integration problems in practice
Success Stories of VIATRA in Tool Integration
Main Application Fields

- **Analysis of Business Process Models**
  - Verification by MC
  - Fault simulation
  - Security analysis (Bell-LaPadula)
  - BPEL generation
  => IBM Faculty Award
- **SOA**
  - Performance & Availability analysis
  - Configuration generation
  - Service Analysis and Deployment
  => SA Forum + SENSORIA IP
- **Embedded Systems**
  - PIM & PSM for dependable embedded systems
  - PIM & PSM model store
  - PIM-to-PSM mapping
  - PIM & PSM validation
  - Middleware code generation
  => DECOS IP
- **Other**
  - Design and transformation of domain specific languages
  - Model-based generation of graphical user interfaces
DECOS

Dependable Embedded Components and Systems

(IP-Project #511764 in EU FP6 / Priority [2] IST)

Partner (19)

Industry
- Airbus, AEV, EADS, Infineon, TTEch, Flat, Profactor, Hella, Liebherr, Thales, Estrel

Universities
- TU Vienna, TU Darmstadt, TU Hamburg, Uni Kassel, Uni Kiel
- Budapest Uni of Techn. and Economics

Research Centres
- ARCS, SP Swedish Test. & Res. Inst.
The DECOS Toolchain

- **DECOS Project Goal**

  Uniform platform for *integration* of embedded distributed (real-time) applications of *mixed* (up to highest) criticality with *model-driven development* support
  - hardware reduction
  - flexibility increase
  ⇒ from *federated* to *integrated* systems

- **DECOS tool chain**
  - Supports
    - Specification
    - Design
    - Implementation
    - V&V of embedded systems
Tool Chain: Model-Based Integrated Development Support

"From Requirements To Deployment"

1. **Requirements**
   - functional, performance, dependability

2. **Cluster modelling**
   - nodes, network

3. **Behaviour modelling**
   - of jobs

4. **PIM-PSM mapping**
   - allocation and scheduling

5. **Code generation**
   - Middleware
   - Configuration
   - Job code

6. **Deployment**
   - compile, link, download

7. **Verification & Validation (V&V)**
   - accompanying (Test Bench)
Tools and functionalities based on VIATRA2 modelbus
Summary

- Tool integration by precise model transformations: feasible in
  - Service-oriented applications
  - Dependable embedded systems

- Transformations can be specified by a combination of formal techniques
  - Graph transformation
  - Abstract State Machines

- MDD tools
  - Can be built on open tool platforms
  - Integrate a large set of tools

- Our approach
  - Open, customizable
  - Highly adaptive (new modeling standards, platforms, V&V tools, …)
Thank you for your attention

- And many thanks to
  - And many more students