Model Transformation

Krzysztof Czarnecki
Generative Software Development Lab
University of Waterloo, Canada
gsd.uwaterloo.ca

Modeling Wizards Summer School, Oct. 1, 2010, Oslo, Norway
What is model transformation?

- **Model manipulation**
  - Automated reading, creation, modification
  - Form of metaprogramming
- **Program vs. model transformation**
  - Blurred boundary
  - Program vs. model
  - String, trees, graphs
  - Grammars vs. class models
  - Model trafo is more diverse

1) Model trafo is the automated manipulation of models, which entails—at minimum—reading and creating model elements, and possibly also element modification ("in-place transformation").

At first, model trafos, just as any other programs, are data manipulation programs: take input data and produce output data.

Model trafo is a form of metaprogramming, however, i.e., programs that manipulate metadata, such as programs, specifications, schemata, etc. Manipulating *semantically rich* data, such as programs or specification, requires specialized facilities. That’s why there is also the separate field of program trafo and compiler construction.

2) The distinction between program and model trafo is not clear-cut.

2.1) The first distinction is bias towards tree vs. graph rewriting.

Traditionally, program trafo has used term rewriting—that is, tree transformations—as its underlying theory. The reason is that any practical programming language contains expressions, and these are often deeply nested tree structures. However, the abstract syntax trees (ASTs) manipulated by program trafos also include cross-tree references, such as the reference from a variable use to its declaration. Typically, program transformation systems use “tricks” to deal with these, by adding so-called context (i.e., symbol table), where the declaration of an identifier can be looked up. Thus, program transformation systems, such as Stratego, can be described as “term-rewriting with context”.

Good part of the model trafo community that works on the theoretical aspects of model transformations view graph transformation as the more appropriate paradigm for model transformations. This view applies well to models such as Petri nets, which are truly graph-like. Some models may have natural tree-like nesting (often represented as composition in their respective metamodels), however. These models can still be subjected to graph transformations—the edges participating in the tree structures are treated as any other edges, but some model transformation approaches exploit these tree-structures to their benefit. Thus, viewing model trafo as transforming graphs without special attention to the tree structures is unnecessarily narrow.

The bottom line is that both program and graph trafo systems transform graph structures and both can be used to transform programs or models. In practice, the choice between the two boils down more to how well the source and target languages are supported. For example, a system that can transform Java should come with a rich library of built-in program analyses, starting with the simplest query for all the interfaces a given class implements. Model transformation systems that were not developed or used in this context will likely lack such facilities.

2.2) Another distinction is that programming languages traditionally used grammars as their syntax definition formalisms; most of the modeling world uses richer formalisms such as class models (e.g., MOF). The latter are more properly “concept definition” formalisms with generalization and property relations (with properties possibly divided into references and composition).

2.3) The final distinction is that models are more diverse. Although the term “models” normally refers to system abstractions above the implementation code, that is, artifacts such as requirements and design specifications or analysis models, model driven environments sometimes represent programs in the same form as models and treat them alike. Thus, we can conclude that model trafos operate on more diverse artifacts than program trafos, as these artifacts may include programs, but also other artifacts such as specifications and schema definitions. In other words, one could view program trafo as a narrower field than model trafo.
Applications

- Deriving lower-level models from higher-level ones
  - Compilation – automatic
  - Generating designs – partially automated
- Mapping and synchronizing among models at same or across abstraction levels
- Creating query-based views of a system
- Model evolution, e.g., refactoring
- Reverse engineering higher-level models from lower-level ones
- Transforming between different formats or languages
  - E.g., tool integration
Note that this picture represents a very basic scenario with a single source and target. In general, there can multiple sources and targets, and the same model can play both roles. An alternative representation would place the execution of a transformation between the source and target.
Choice of metamodeling representation has significant impact on model transformation.

Representations such as EMF enforce single-typing of modeling elements, leading to some rigidity. In particular, extensions or changes to the metamodel might be necessary to allow for extra intermediate data or intermediate forms of the model that do not conform to the source or target metamodel. Representations that allow assigning model elements to multiple types, such as RDF, do not suffer from these problems.

Mappings are specifications, which are likely non-executable; transformations are executable functions that implement mappings.

The field of verification and validation of model trafoes is a very active research field today.
The library of queries raises the abstraction level of the source metamodel towards a particular task.

Best Practice

- Augment the source metamodel with a library of queries
- Reuse such a library if available
Transformation Systems and Languages

• Huge number of systems and growing...
  – Over 30 systems and languages in the 2006 survey
  – Vast majority academic
  – A few industrial-strength (improved somewhat compared to 2006)

• Very diverse in terms of paradigms, features, and capabilities
  – Choice depends on application
Classification of language features

Czarnecki, K. & Helsen, S.
Feature-Based Survey of Model Transformation Approaches.
IBM Systems Journal, special issue on Model-Driven Software Development,
45, no. 3, pp. 621-645, July 2006

Download at http://gsd.uwaterloo.ca/node/68
Language Features I

- Query
  - Explicit navigation
  - Source patterns
    - Abstract vs. concrete syntax
    - Graphs, trees, strings
- Element creation
  - Explicit creation
  - Target patterns
- Rules combine source and target patterns
Language Features II

• Logic
  – Language paradigm
    • OO, functional, logic, procedural
• Arity and mode
  – Number of participating models (1, 2, 3, ...)
  – In, out, inout
• Directionality
  – One-way, bi-directional
• Source-target relation
  – New target (extract) vs. in-place (destructive or extension only)
Language Features III

- **Traceability**
  - Built-in or custom; creation, storage
- **Incrementality**
  - Target, source
- **User-edit preservation**
  - Protected regions, separation, update policy
- **Organization and reuse**
  - Source, target, independent
  - Modules, calls, inheritance, polymorphism, aspects

Note that needs for traceability (how much to trace and where and how to store) are very diverse depending on applications; thus, users are typically given the facilities to develop their own support rather than providing a build-in mechanism.

Target incrementality tries to minimize the amount of the target needs to be rebuild when the source changes. Source incrementality tries to minimize the amount of source that needs to be re-examined upon the source changes.

A practical way to deal with user edits is to separate the generated parts from the manually written ones; e.g., the super classes are generated and the subclasses are manually written.

Synchronization systems use update policies to re-merge manual changes with automatically propagated changes in the target.

Practical trafo languages support reuse mechanisms such as modules, inheritance and polymorphism.
Major Categories

- Model-to-text
  - Traverse and print
    - Example: roll your own visitor pattern in Java
  - Template-based ("text with holes")
    - E.g., Xpand, JET, (other: MOF2Text, MOFScript)

- Model-to-model
  - ...

- Text-to-model
  - Parsing
    - E.g., XText
Xpand Template

«DEFINE Root FOR data::DataModel»
«EXPAND Entity FOREACH entity»
«ENDDEFINE»

«DEFINE Entity FOR data::Entity»
«FILE name + ".java"»
  public class «name» {
    «FOREACH attribute AS a»
      private «a.type» «a.name»;
    «ENDFOREACH»
  }
«ENDFILE»
«ENDDEFINE»
Model-to-model (I)

- Direct manipulation
  - Just use a programming language, e.g., Java
  - Often an API over the metamodels
- Operational
  - Programming language with dedicated facilities, e.g., query, pattern matching
  - Often an extension of the metamodeling formalism
  - Examples: Kermeta
- Template-based
  - Models with holes
  - Optional parts and fragment replacement
  - Examples: Model Templates [CA06], CVL


CVL = Common Variability Language = an OMG standard under preparation
Kermeta (operational)

```java
operation transform(source:PackageHierarchy): Database is do
result := Database.new
trace.initStep("uml2db")
source.hierarchy.each { pkg |
  var scm: Schema init Schema.new
  scm.name := String.clone(pkg.name)
  result.schema.add(scm)
  trace.addlink("uml2db", "package2schema", pkg, scm)
}
end
```
Sample Model Template

Elements annotated by addrFeature are included in the result iff addrFeature is selected during configuration.
Model-to-model (II)

- Graph-transformation-based
  - Rule-based, in-place
  - graph patterns with elements to be created, deleted, left; possibly separated in RHS and LHS
  - Rule scheduling: predetermined (e.g., fixpoint) or programmable
  - Rich theory exists
  - Examples: VIATRA, Fujaba, AGG, ATOM3, GReAT, MOFLON (TGG)
Graph Trafo Rules

Fujaba rule

MoTif rule
gtrule liftAttrR(inout CP, inout CS, inout A) =
{
    precondition pattern lhs(CP,CS,A,Par,Attr) =
    {
        Class(CP);
        Class.parent(Par,CS,CP);
        Class(CS);
        Class.attr(Attr,CS,A);
        Attribute(A);
    }
    postcondition pattern rhs(CP,CS,A,Par,Attr,Attr2) =
    {
        Class(CP);
        Class.parent(Par,CS,CP);
        Class(CS);
        Class.attr(Attr2,CP,A);
        Attribute(A);
    }
    action {
        print("Rule liftAttrR is applied.");
        // execution of a GT rule for one attribute of a class
    }
}

// calling the rule for all attributes of a class
choose A apply liftAttrR(Class1,Class2,A);

// calling the rule for all possible matches in parent
forall C1, C2, A do apply liftAttrR(C1,C2,A);

// Apply a GT rule as long as possible for the entire class hierarchy
choose C1, C2, A apply liftAttrR(C1,C2,A)
Model-to-model (III)

- Relational
  - Declarative relation between models
  - Operational meaning derived (interpreted or generated)
  - Limited if general purpose; powerful if specialized
  - Examples: QVT Relational, MOFLON (TGG), Code Views
QVT Relational comes close to spec, but it is challenged when the structural gap between input and output is wide.
Model-to-model (IV)

- Hybrid
  - Combination of different styles
  - Example: ATL combines rules with operational style
  - Many systems become increasingly hybrid as adapted to new problems
    - e.g., VIATRA
OCL suffers from the problem that navigation code needs to check for `oclIsUndefined()`; Alloy avoids this problem by treating navigation as relational composition. ATL inherits OCL along with its problems; however, transitive closures can be sometimes used to avoid the problem.
Model-to-model (V)

- Other
  - Logic-programming based (e.g., Tefkat)
  - Can use other types of trafo systems as well
    - Program trafo systems: Spoofax (Stratego), TXL
    - MPS (AST-based)
    - XSLT
Spoofax (Stratego)

Spoofax comes form the program trafo tradition (term rewriting with context), but can be applied to model trafo.
Meta-programming system (MPS) represents programs as fully resolved ASTs (i.e., ASTs with explicit references to declarations). It offers trafo facilities. The system is attempting to unify programming and modeling.
DSLs in MPS

```java
package stuff.

public class TestClass2
{
    public void calcLift()
    {
        double lift = 0;
        lift = lift + A_CL * C_L;
        lift = lift + lift;
        lift = lift + lift;
    }

    public static void main(String[] args)
    {
        System.out.println(lift);
    }
}
```

```java
import blocks.

public class TestClass
{
    public void calcLift()
    {
        double lift = 0;
        lift = lift + A_CL * C_L;
        lift = lift + lift;
        lift = lift + lift;
    }

    public static void main(String[] args)
    {
        System.out.println(lift);
    }
}
```
Comparing model trafos in action

• Graph Transformation Tool Contest
Summary

• Distinction between program and model transformation fuzzy
• Great diversity of applications
• Great number and diversity of systems and languages
• Mostly academic, but situation improves
  – Spin-offs from academia
  – Still no prominent model transformation tool from a major vendor
References

• For an extensive bibliography in model trafos see
  – and the IBM Systems Journal 2006 survey at
    http://gsd.uwaterloo.ca/node/68
Acknowledgments

• Thanks to
  – Eugene Syriani and Hans Vangheluwe for allowing me to reuse some of the sample system screenshots from their DSM-TP2010 model trafo tutorial
  – Istwan Rath from the VIATRA team for discussing with me the ins and outs of graph transformation systems
  – Markus Voelter for allowing me to reuse the MPS screenshots

• The lions share of the material is based on the model trafo survey with Simon Helsen that appeared in the IBM Systems Journal in 2006
Questions?