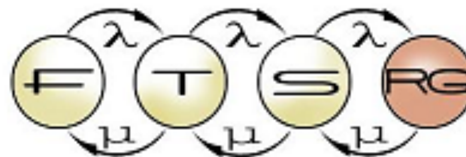
Model Transformations in the VIATRA2 Framework

István Ráth
PhD candidate

Dániel Varró
Associate professor

Budapest University of Technology and Economics
Dept. of Measurement and Information Systems



Overview - Outline

- Introduction
  - Motivation for model transformations
  - Main concepts of MTs
- The VIATRA2 transformation language
  - Models and Metamodels
  - Graph Patterns and Graph transformation
  - Controlled transformations (Abstract state machines)
- Advanced Transformation concepts
  - Synchronization with Event-driven Transformations
  - Constraint satisfaction problems over models
  - Model simulation
- Applications in practice: Model-driven tool integration
  - SENSORIA
  - Embedded domain

Challenges for Software Development

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, …

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

Challenges for Software Development

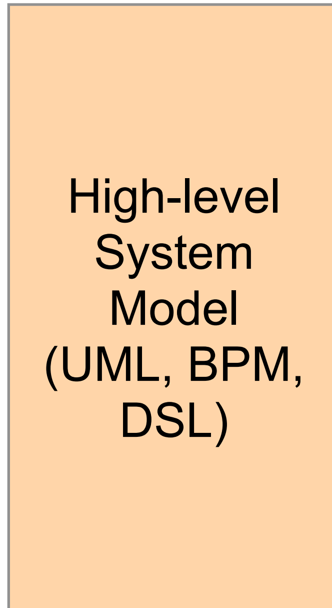
- A typical design process of a large system involves
  - Many stakeholders
  - Many development teams
  - Many manmonths
  - **MANY TOOLS**
    - Requirements, Analysis, Design, Testing, Maintenance, …
- Tool integration is a major challenge
  - Design of Embedded / Critical systems:
    Cost of tool 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

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

Model Transformation for Tool Integration

System design



Model Transformation for Tool Integration

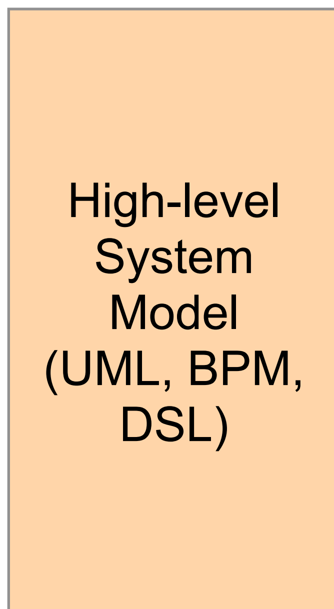
System design

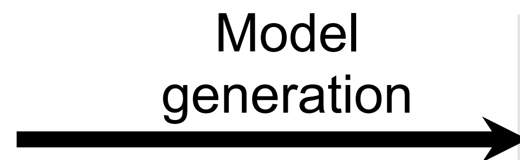
High-level System Model (UML, BPM, DSL)

Model generation

Model Transformation for Tool Integration

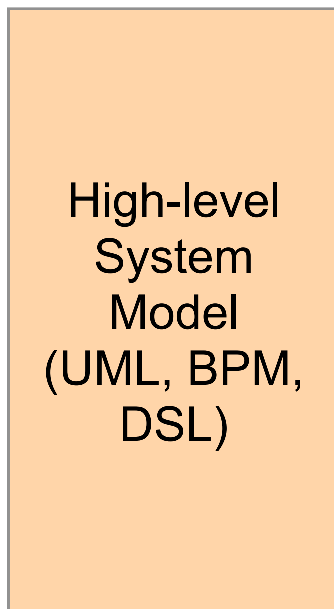
System design

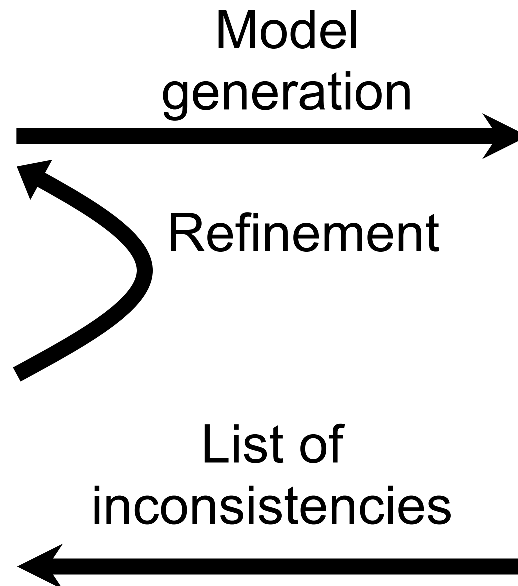




Model Transformation for Tool Integration

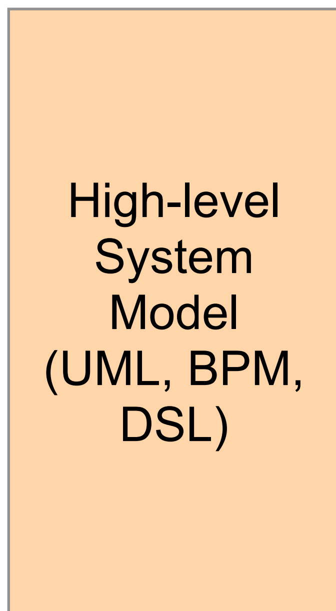
System design

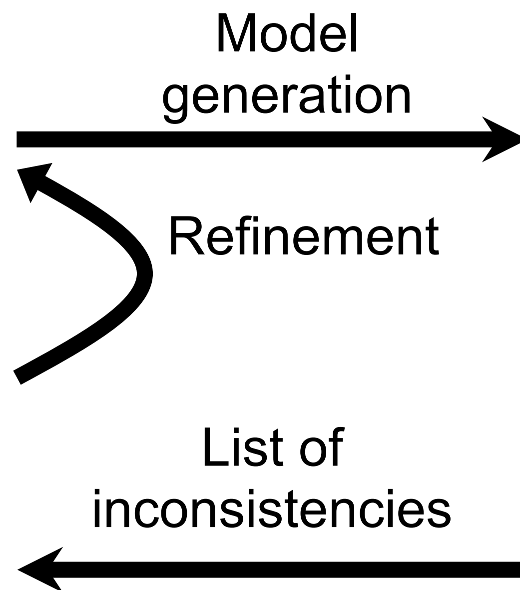




Model Transformation for Tool Integration

System design

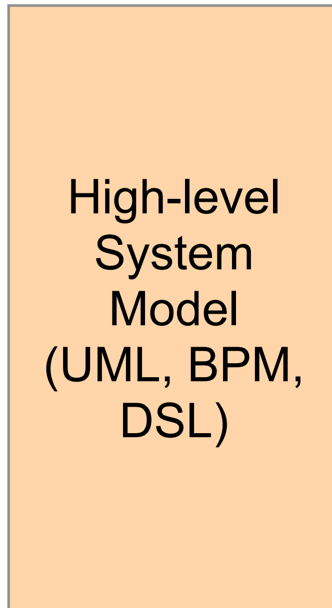


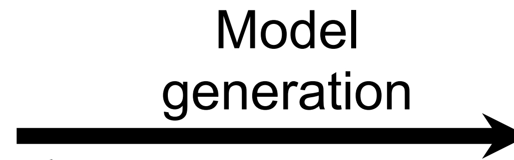


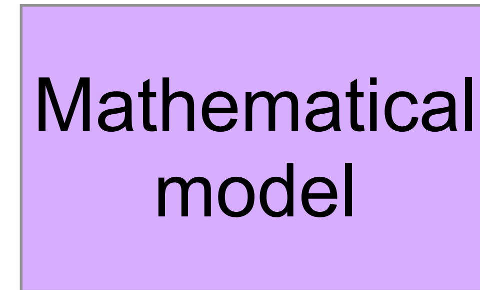
Model Transformation for Tool Integration

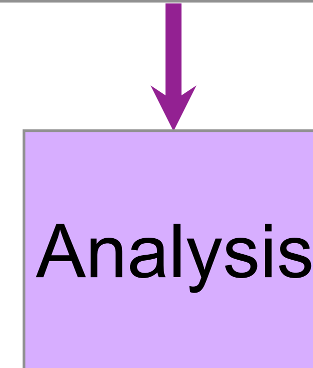
System design

Mathematical analysis







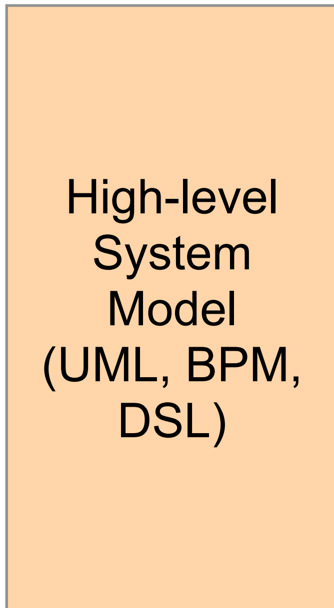


Model Transformation for Tool Integration

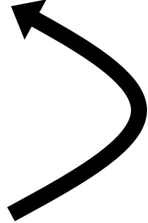
System design

VIATRA2

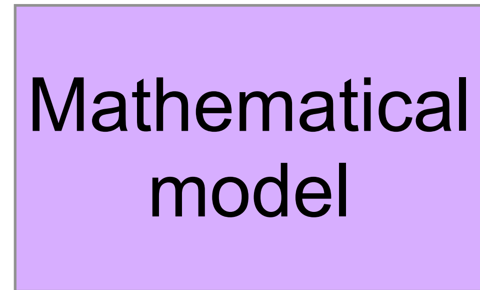
Mathematical analysis



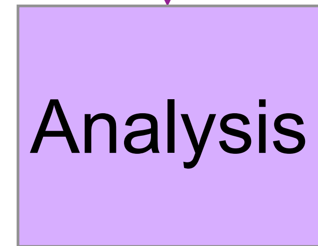
Model generation



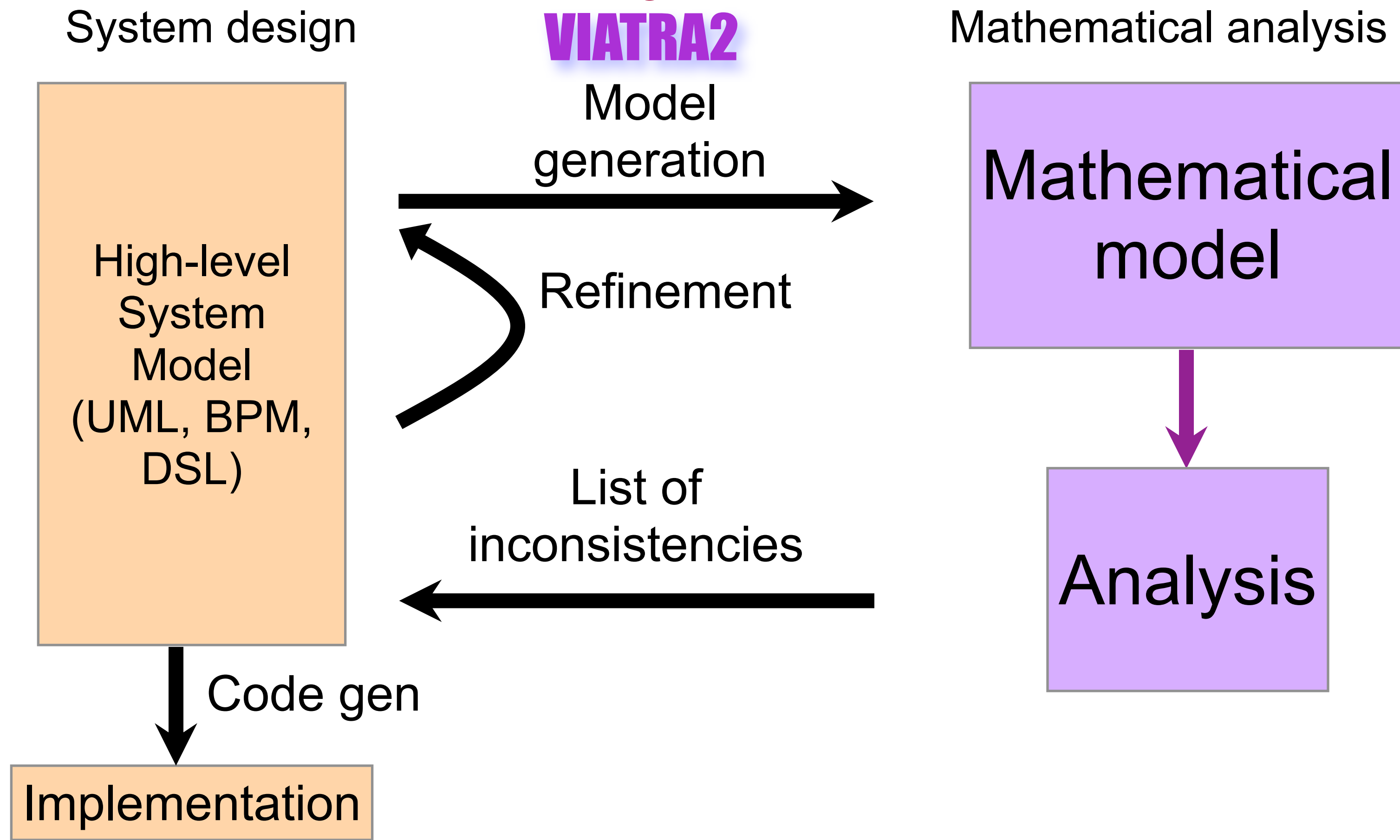
Refinement



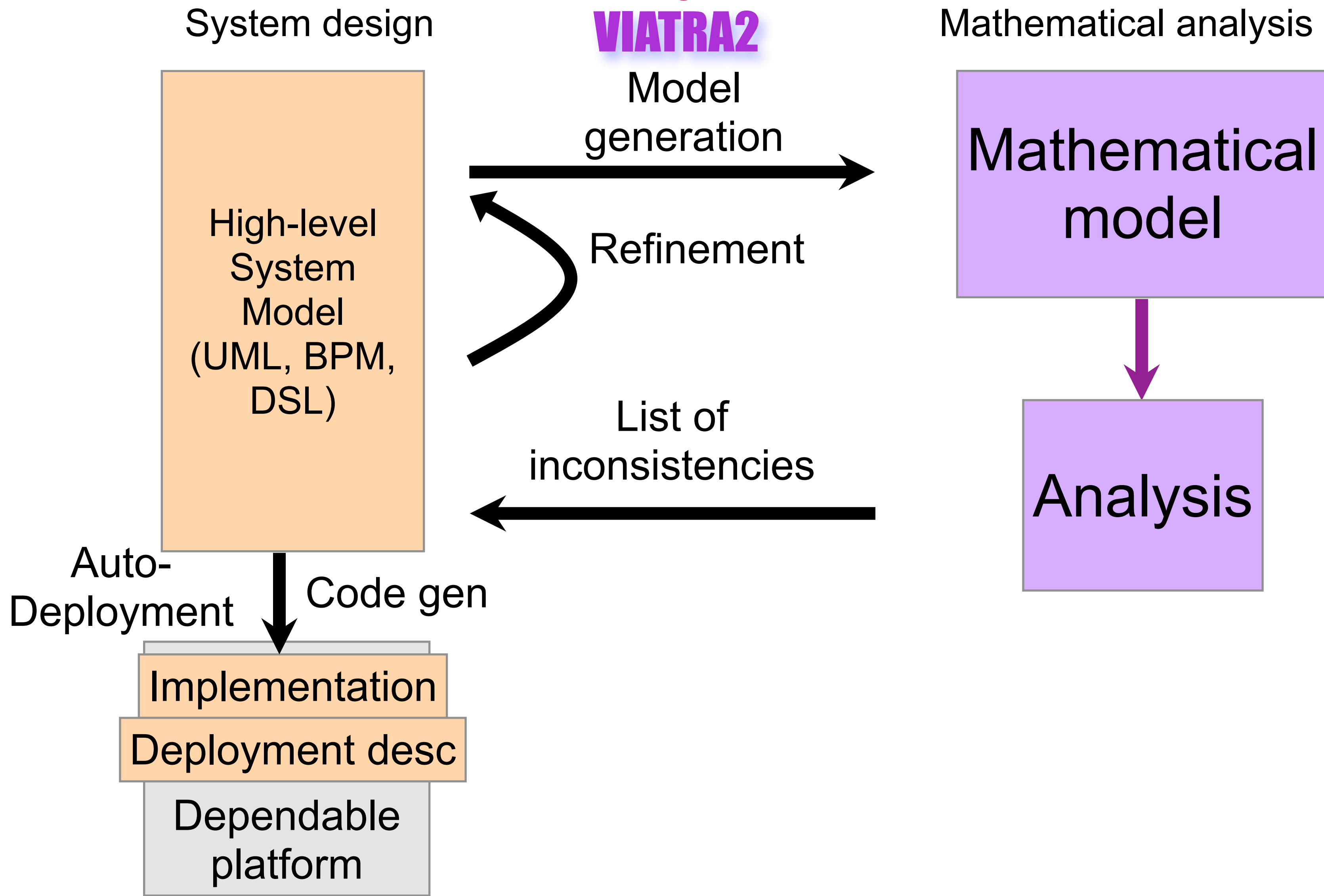
List of inconsistencies



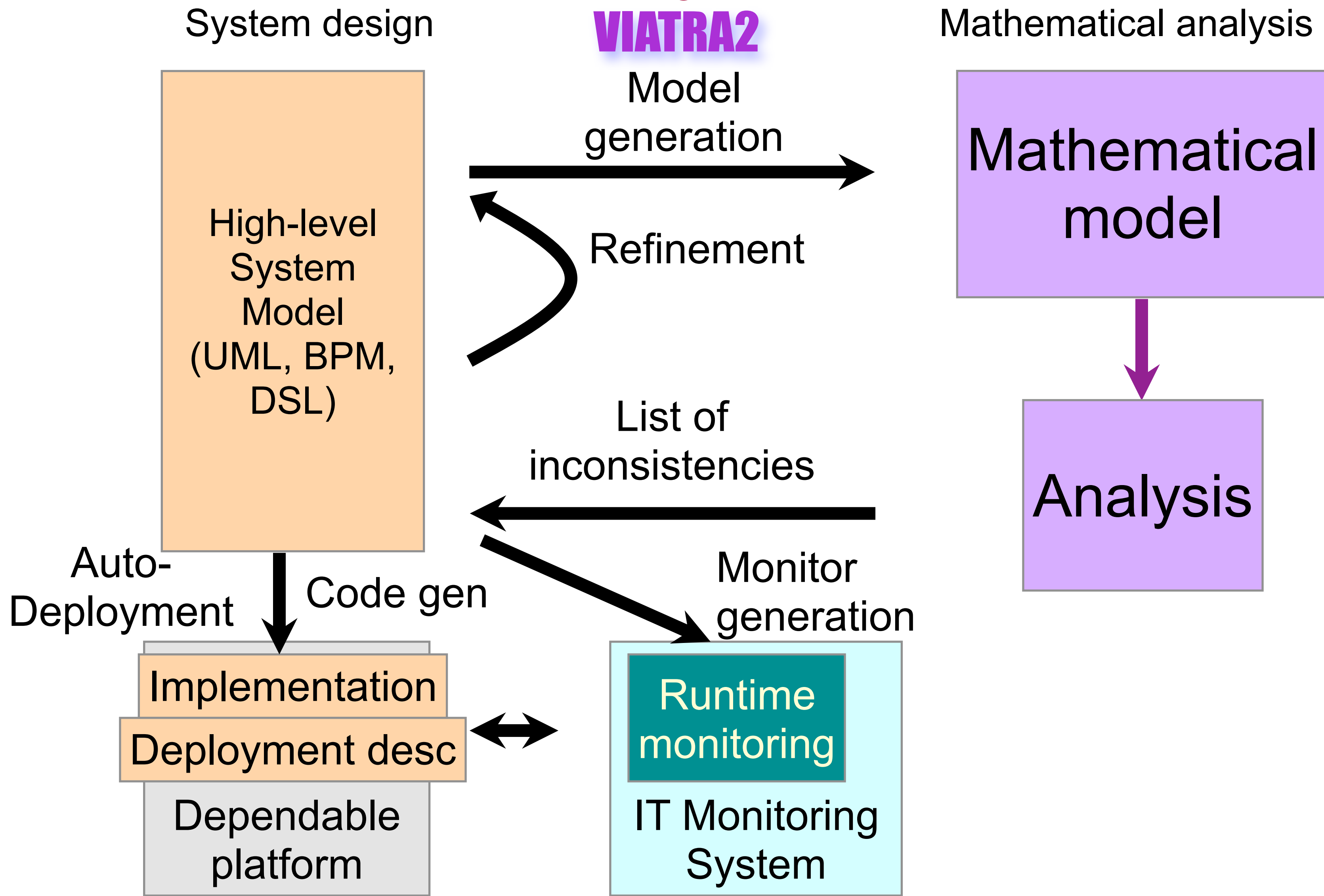
Model Transformation for Tool Integration



Model Transformation for Tool Integration

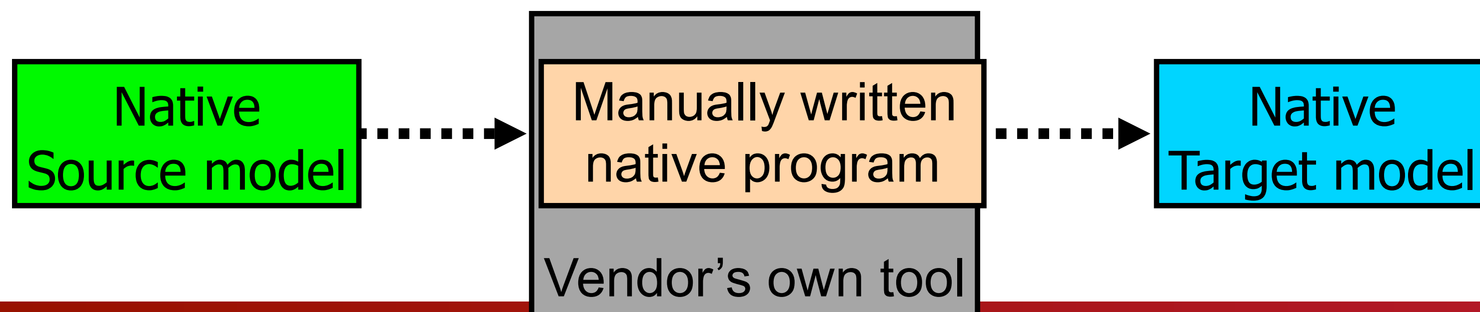


Model Transformation for Tool Integration



THE MODEL TRANSFORMATION PROBLEM

Native (Ad Hoc) Model Transformations



Native (Ad Hoc) Model Transformations

Native source models:

- EMF
- XML documents
- Databases
- Domain-specific models
- UML

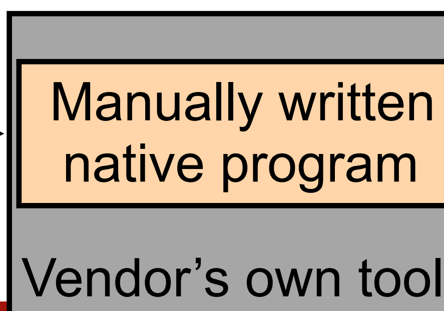
Native Source model → Manually written native program (Vendor's own tool) → Native Target model

Native (Ad Hoc) Model Transformations

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

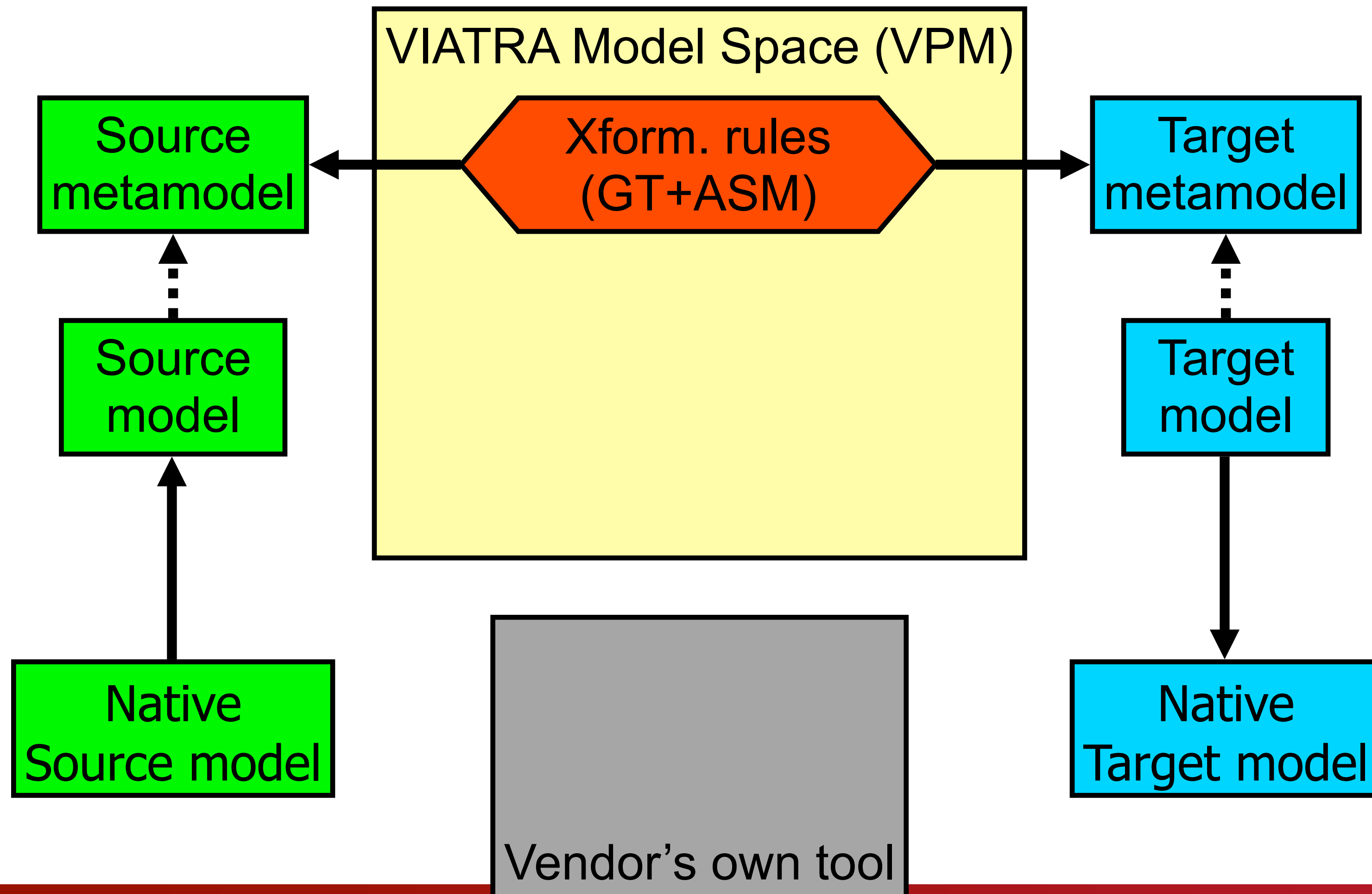
Native target models:
- EMF
- App source code
- XML deployment descript.
- Databases
- Analysis tools

Native Source model

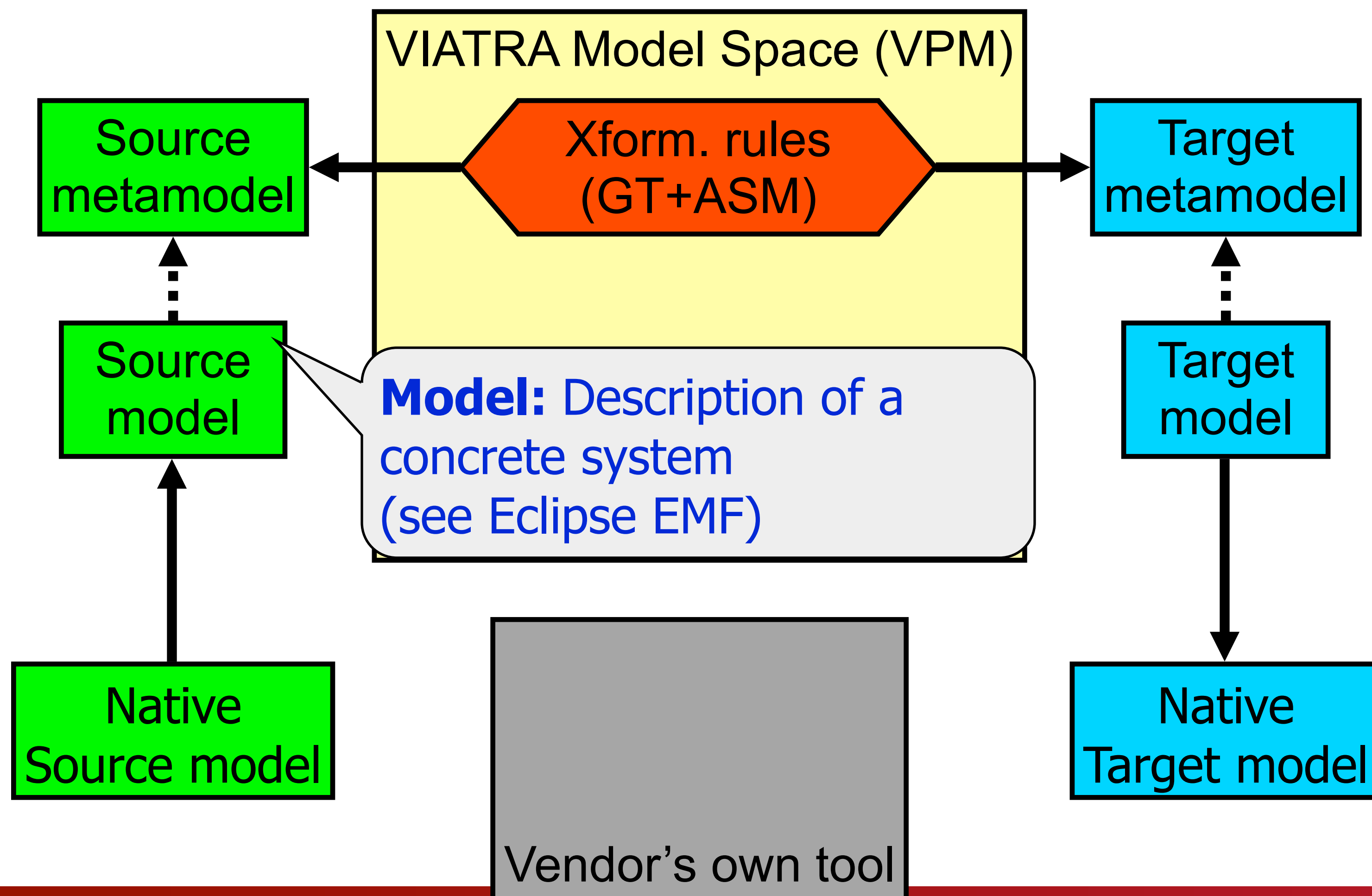


Native Target model

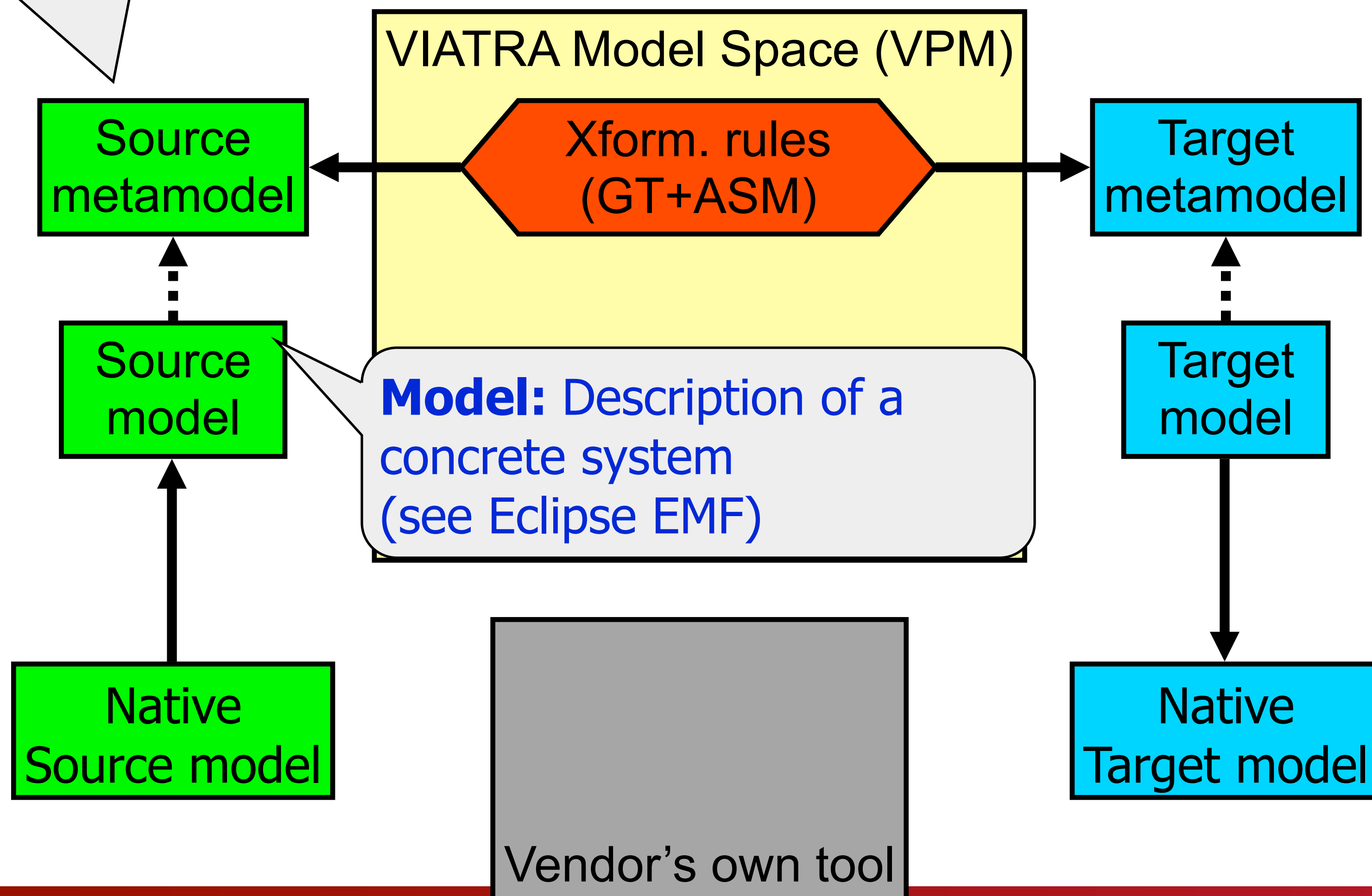
Models, Metamodels, Transformations



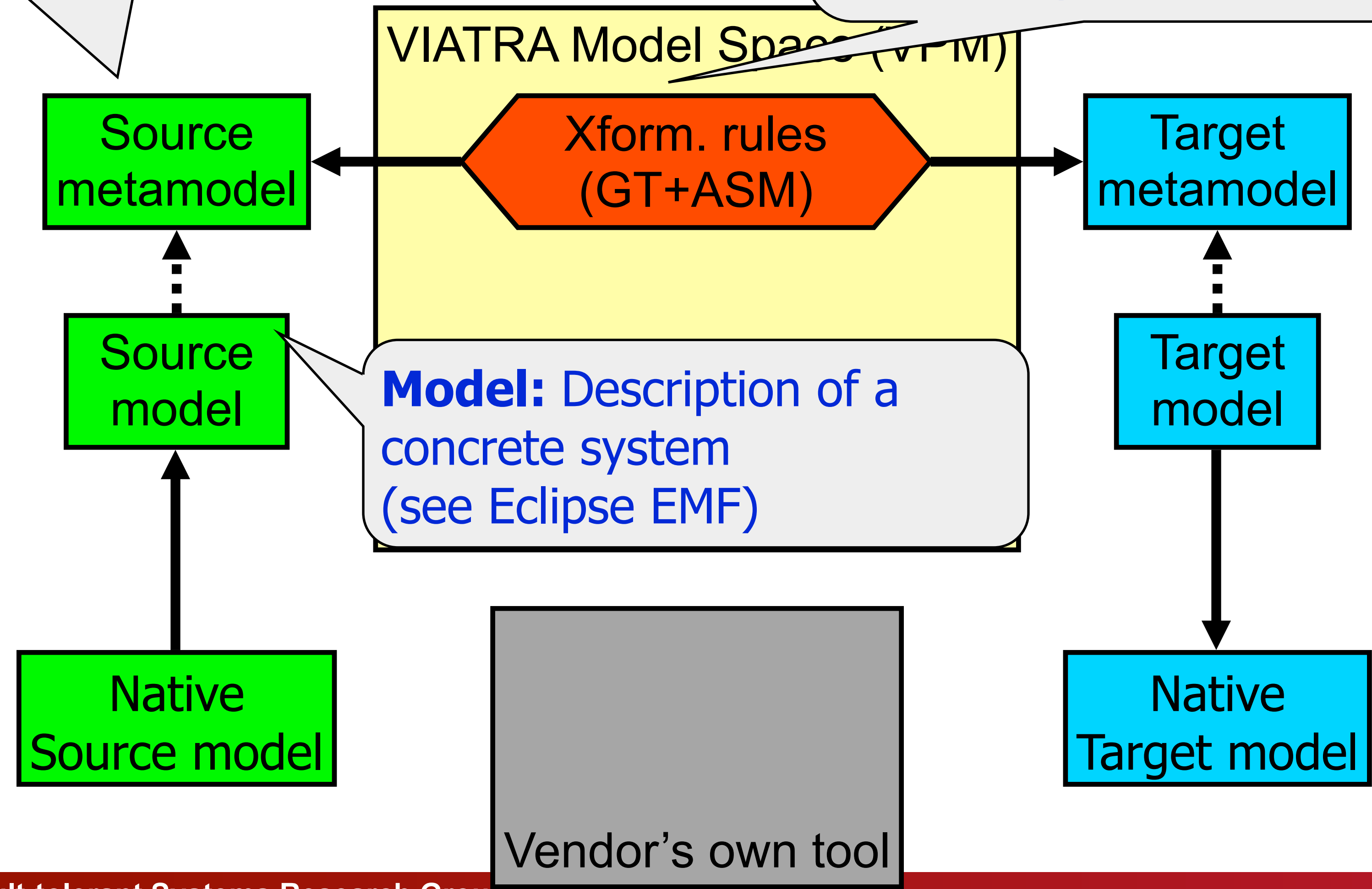
Models, Metamodels, Transformations



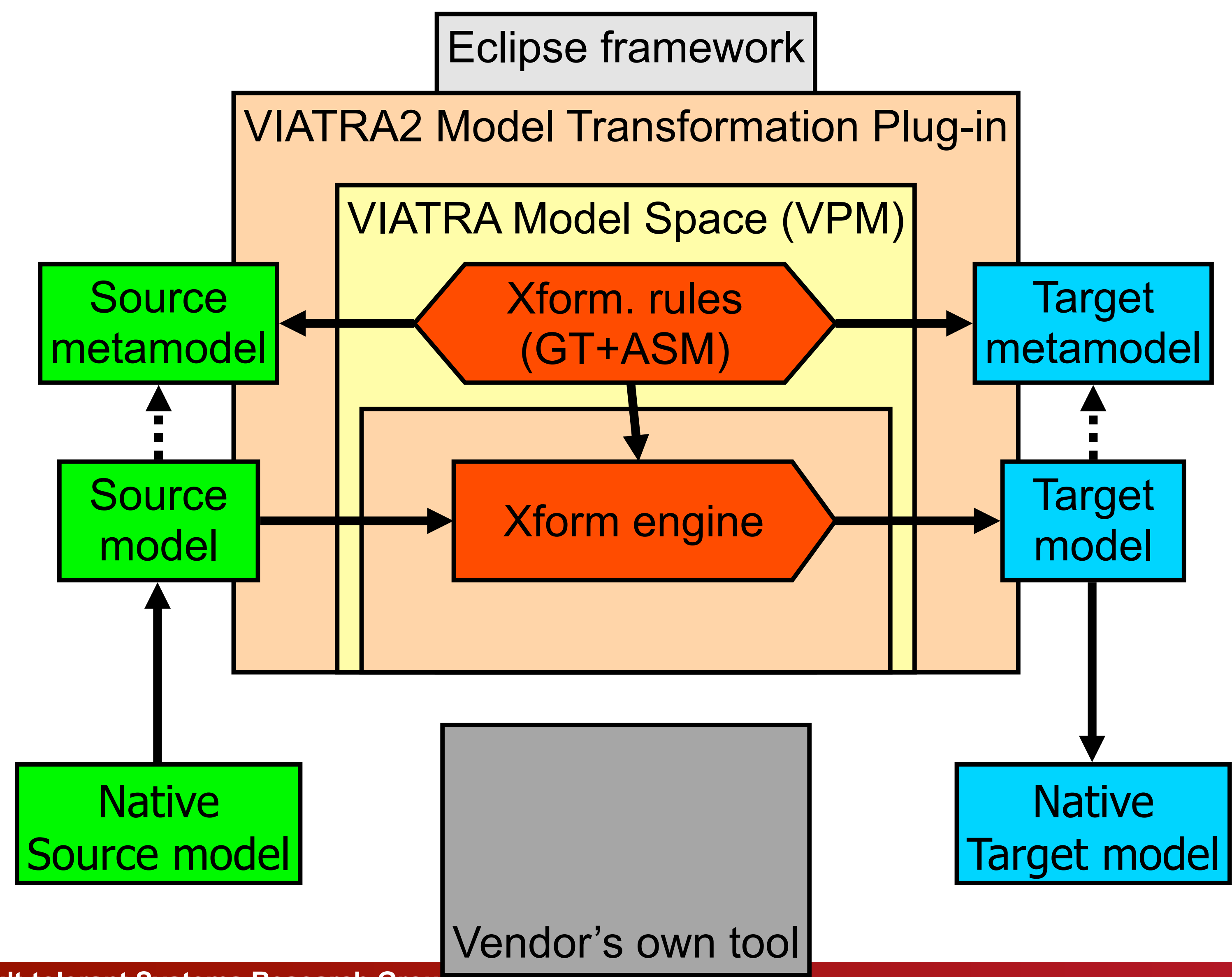
Models, Metamodels, Transformations



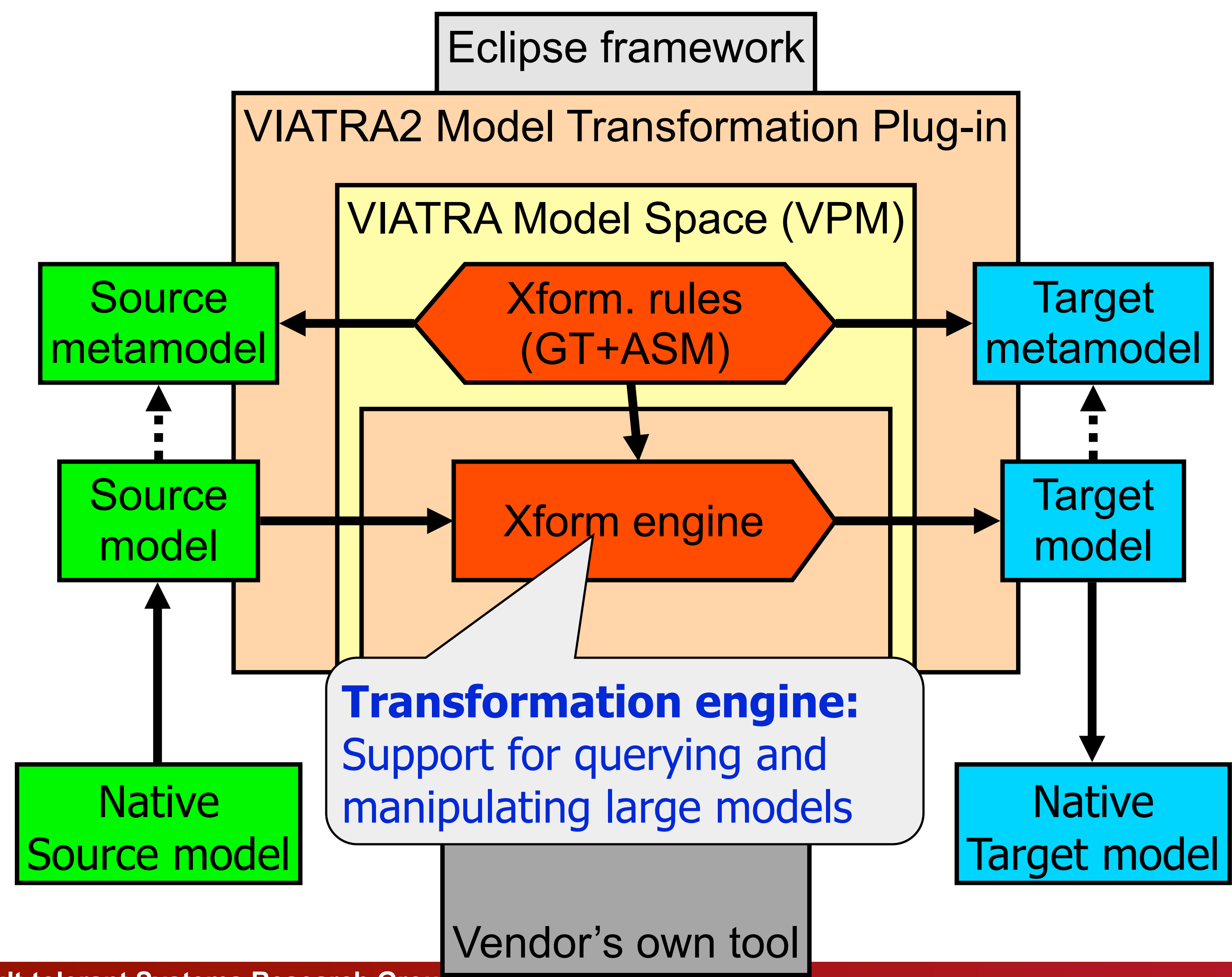
Models, Metamodels, Transformations



Transformation Engine

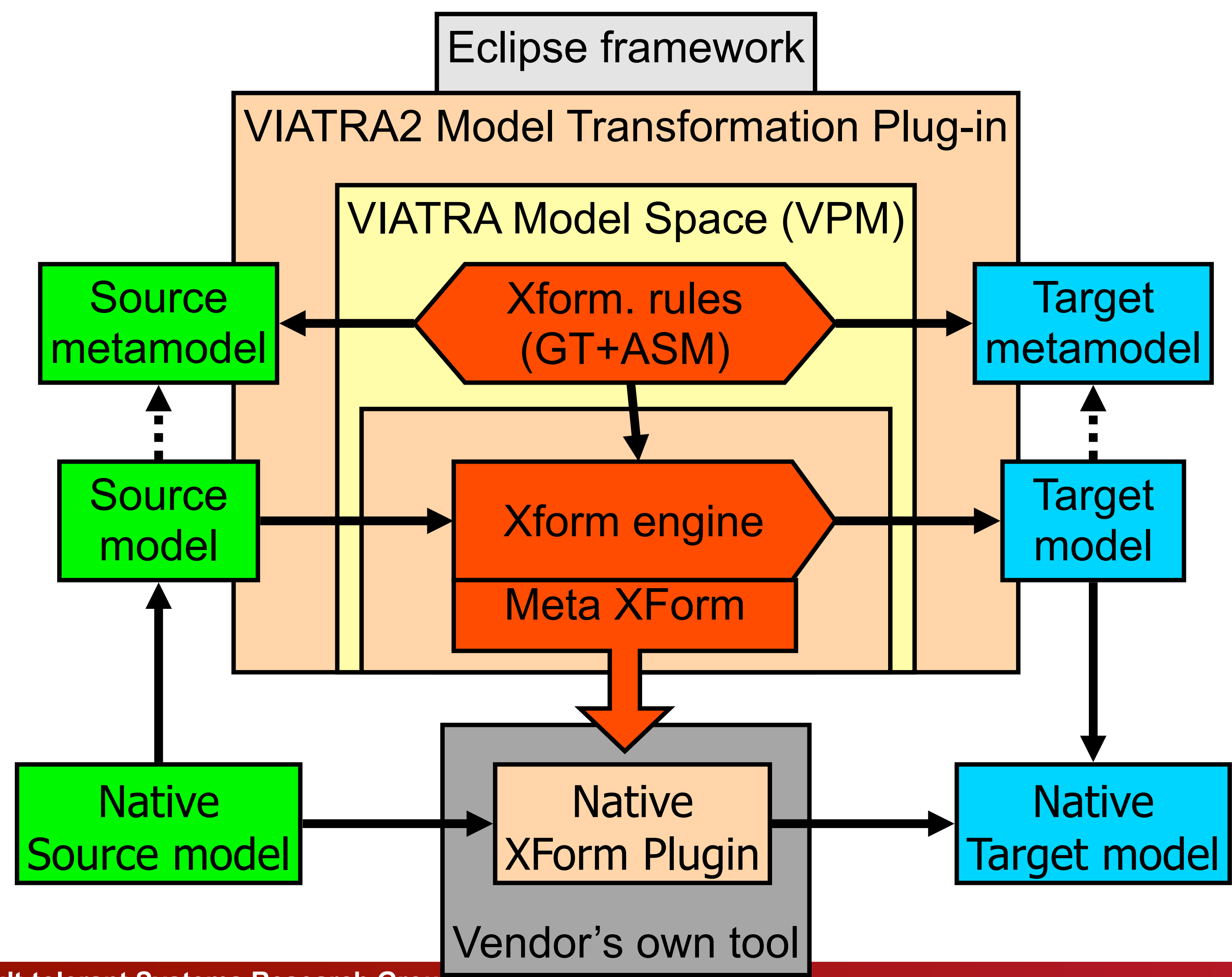


Transformation Engine

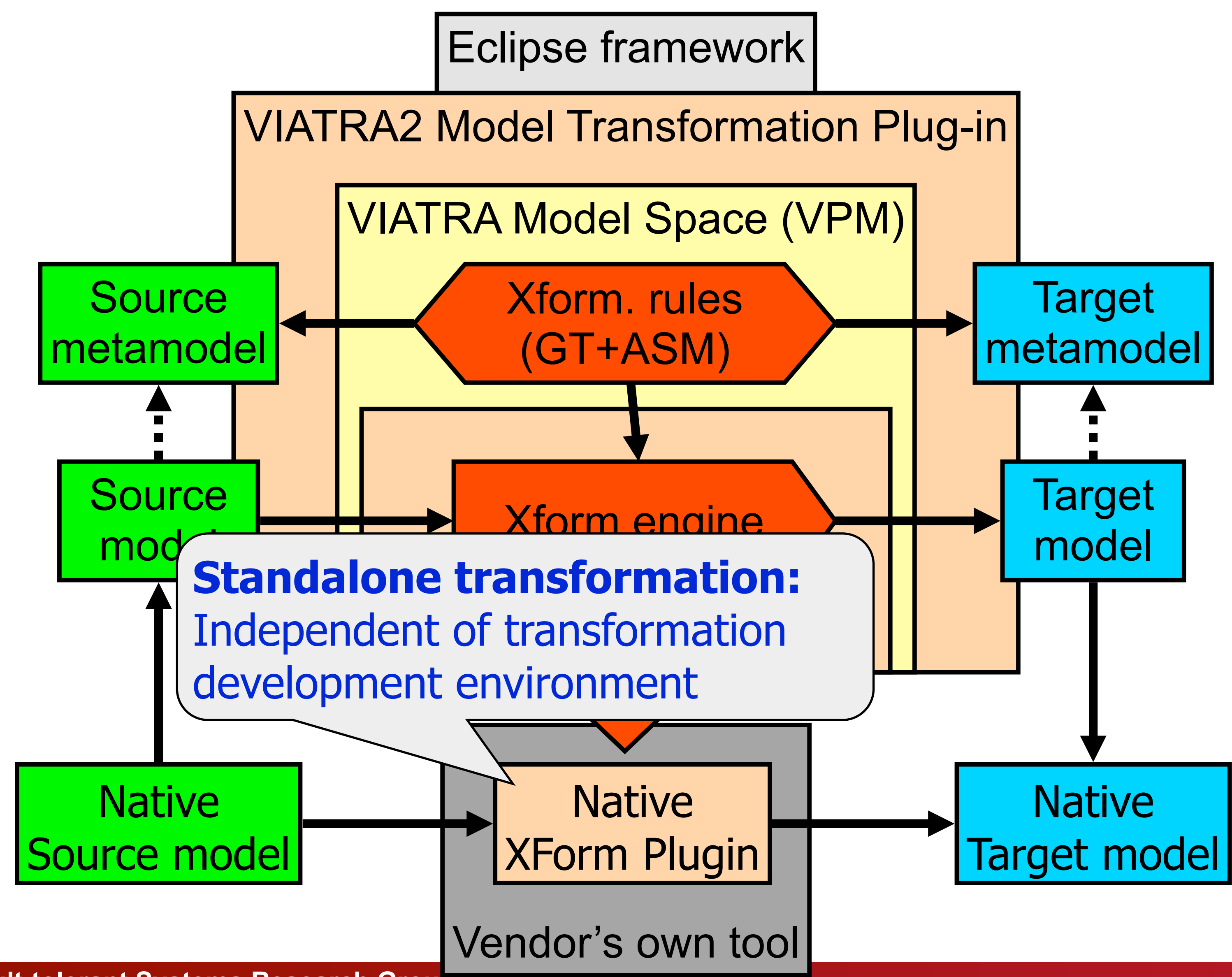


SENSORIA

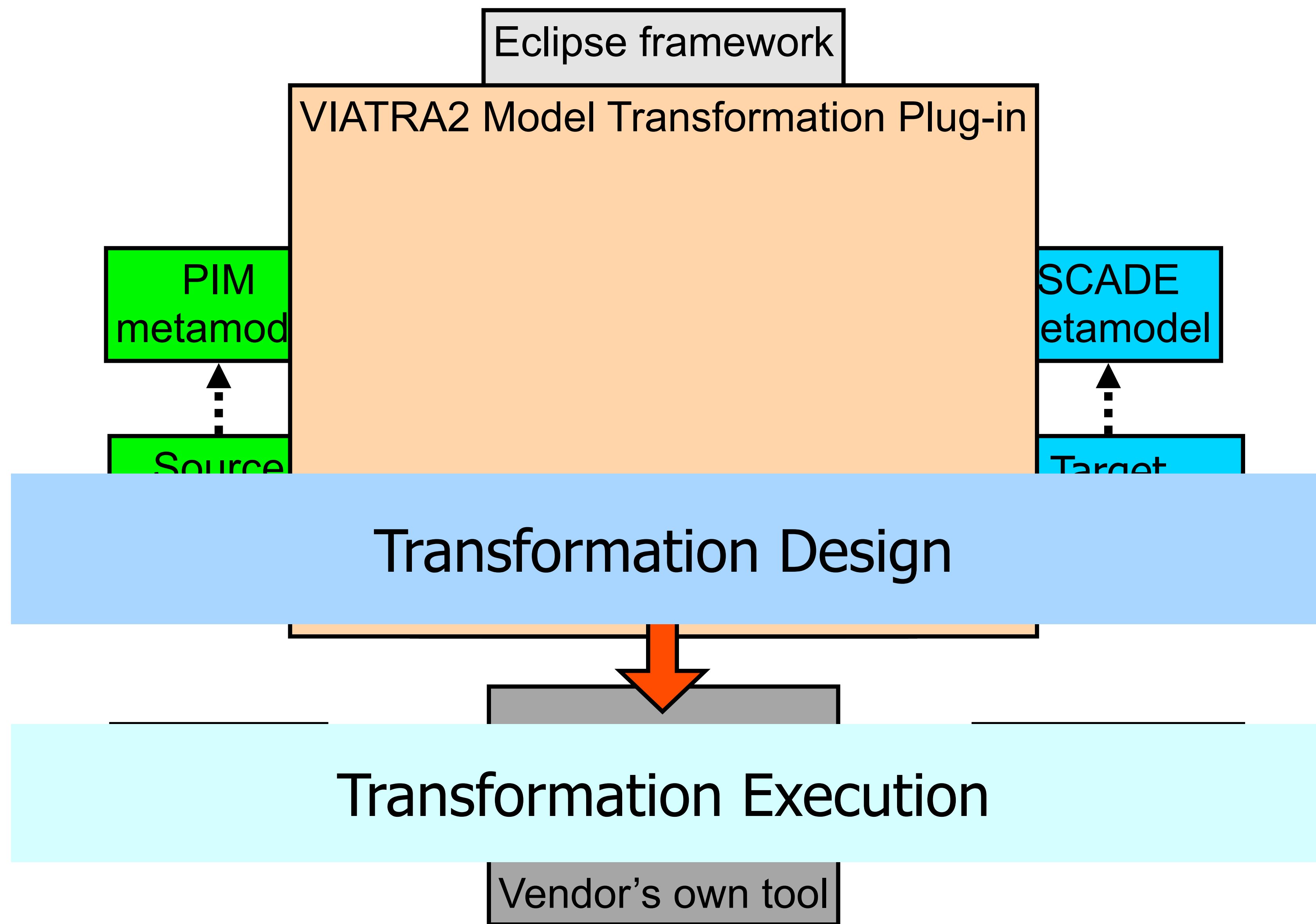
Standalone Transformation Plugins



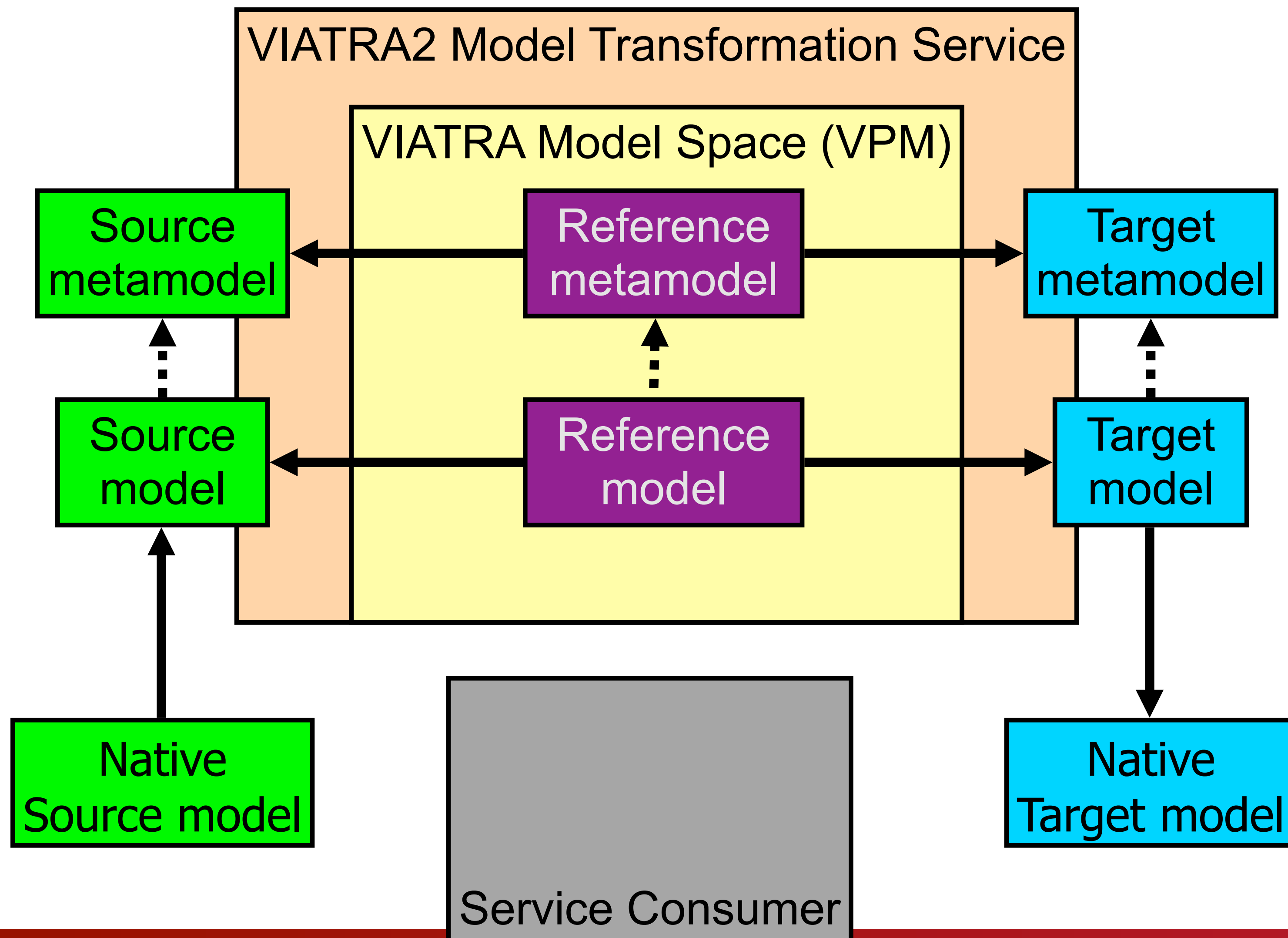
Standalone Transformation Plugins



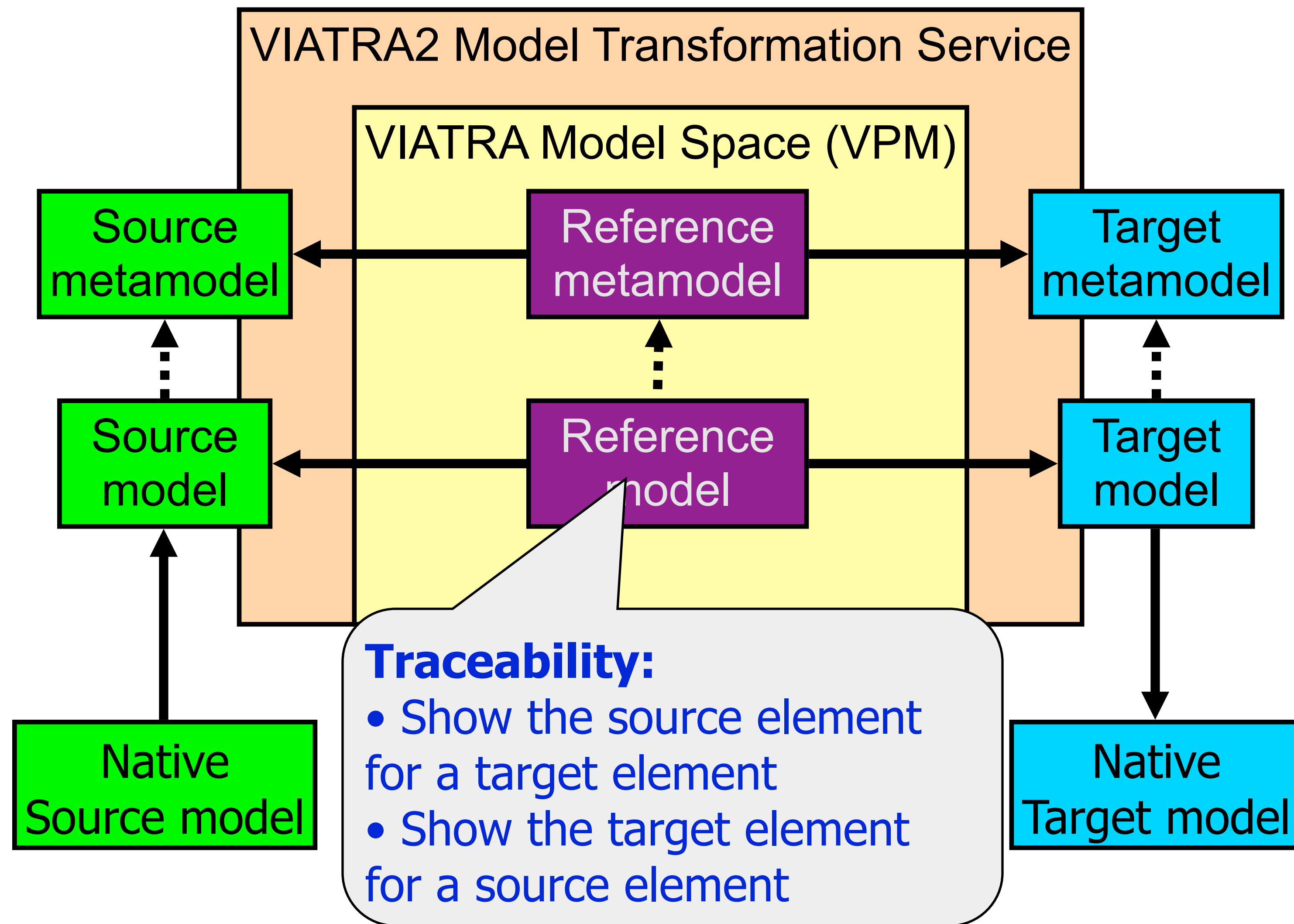
The VIATRA2 framework



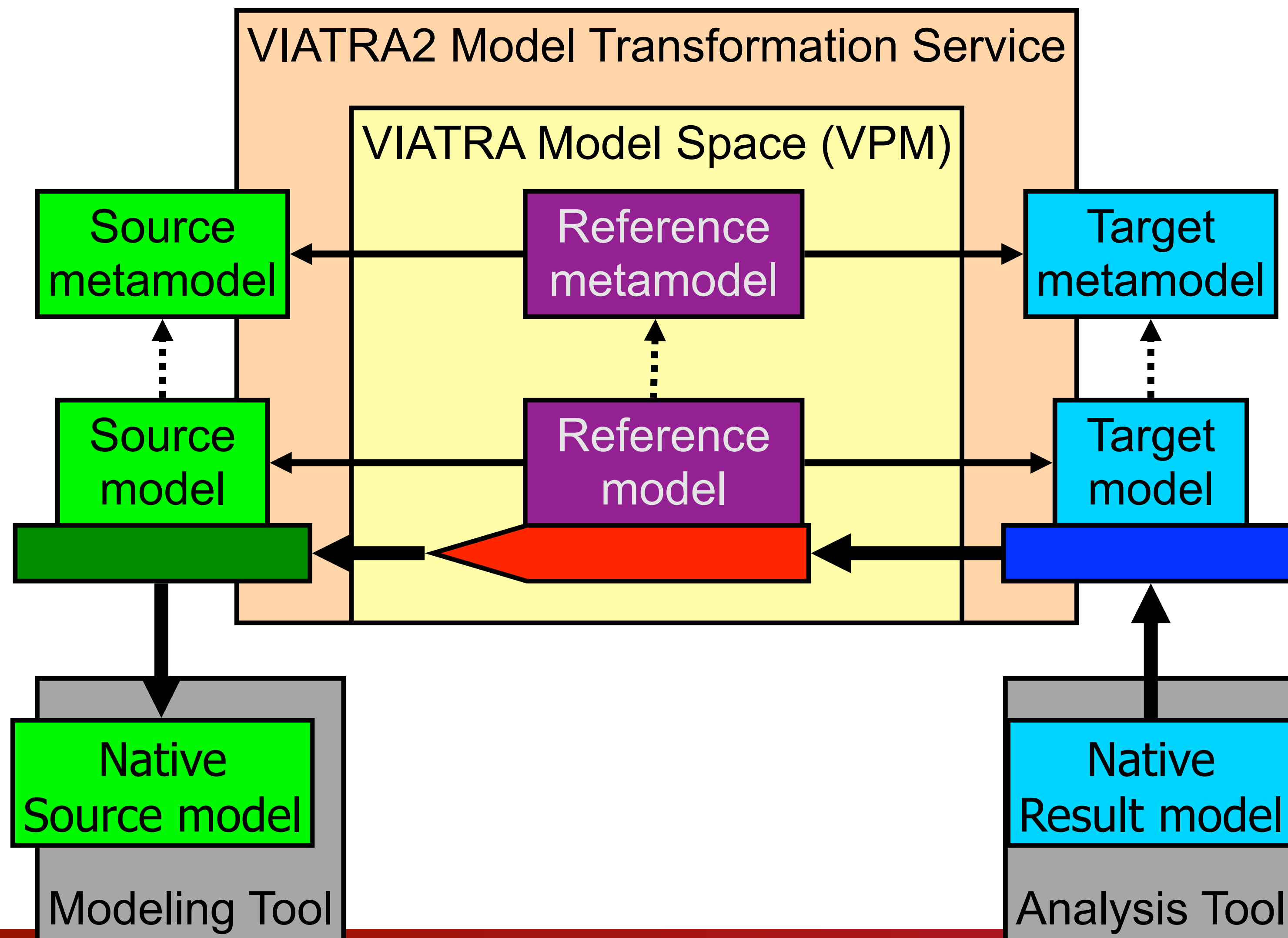
Traceability of Transformations



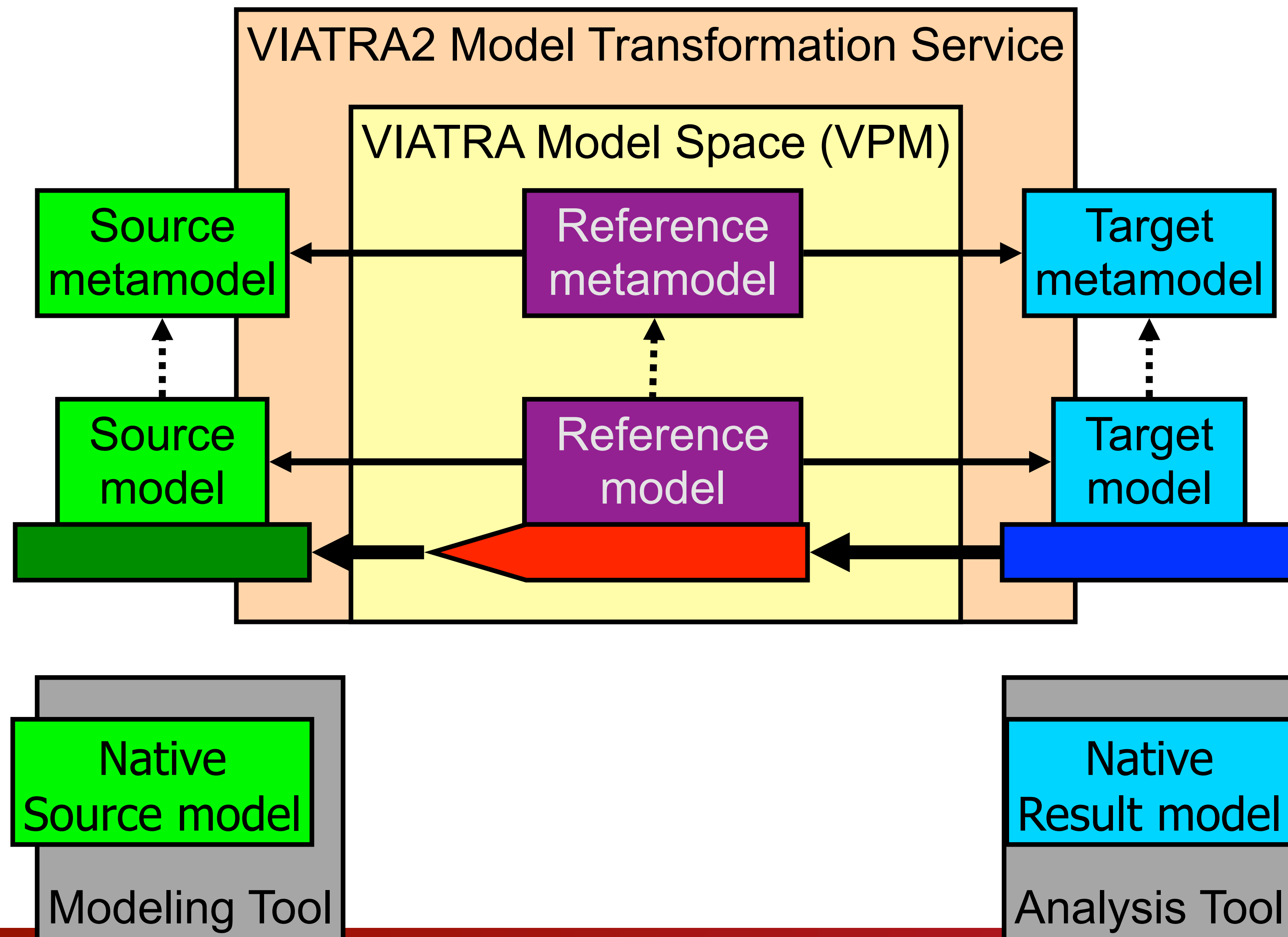
Traceability of Transformations



Back-Annotation of Analysis Results



Back-Annotation of Analysis Results



The VIATRA2 Framework @ Eclipse

eclipse

CONTACT | LEGAL

HOME | USERS | MEMBERS | COMMITTERS | DOWNLOADS | RESOURCES | PROJECTS | ABOUT US

Search

eclipseCON 2009 — 5 Days Left for Advance Registration Prices! — March 23rd - 26th Santa Clara, CA

- GMT Home
- Download
- Documentation
- VIATRA2
  - Documentation
  - Download
  - VIATRA2 Wiki
  - Contact

Project Overview

The main objective of the VIATRA2 (VIsual Automated model TRAnsformations) framework is to provide a general-purpose support for the entire life-cycle of engineering model transformations including the specification, design, execution, validation and maintenance of transformations within and between various modeling languages and domains.

VIATRA2 intends to complement existing model transformation frameworks in providing

- a **model space** for uniform representation of models and metamodels
- a **transformation language** with
  - both declarative and imperative features
  - based upon popular formal mathematical techniques of *graph transformation (GT)* and *abstract state machines (ASM)*
- a high performance transformation engine
  - supporting **incremental model transformations**
  - trigger-driven **live transformations** where complex model changes may trigger execution of transformations
  - handling well over 100,000 model elements (see our **benchmarks**)
- with main **target application domains** in
  - model-based tool integration framework
  - model analysis transformations

Further issues frequently asked by users:

- **Conformance to standards**
- **Known uses of VIATRA2**

Quick Navigator

- Documentation
- Download
- VIATRA2 Wiki

VIATRA2 News

- **October 15th 2008**
  VIATRA project migration complete, we are now official part of the Modeling project (as the rest of GMT).
  The homepage was updated with new sections, further additions to the wiki. The official Release3 source tree is available from Eclipse.org SVN as well as our new update site (check the downloads section).
- **June 2th 2008**

http://www.eclipse.org/gmt/VIATRA2

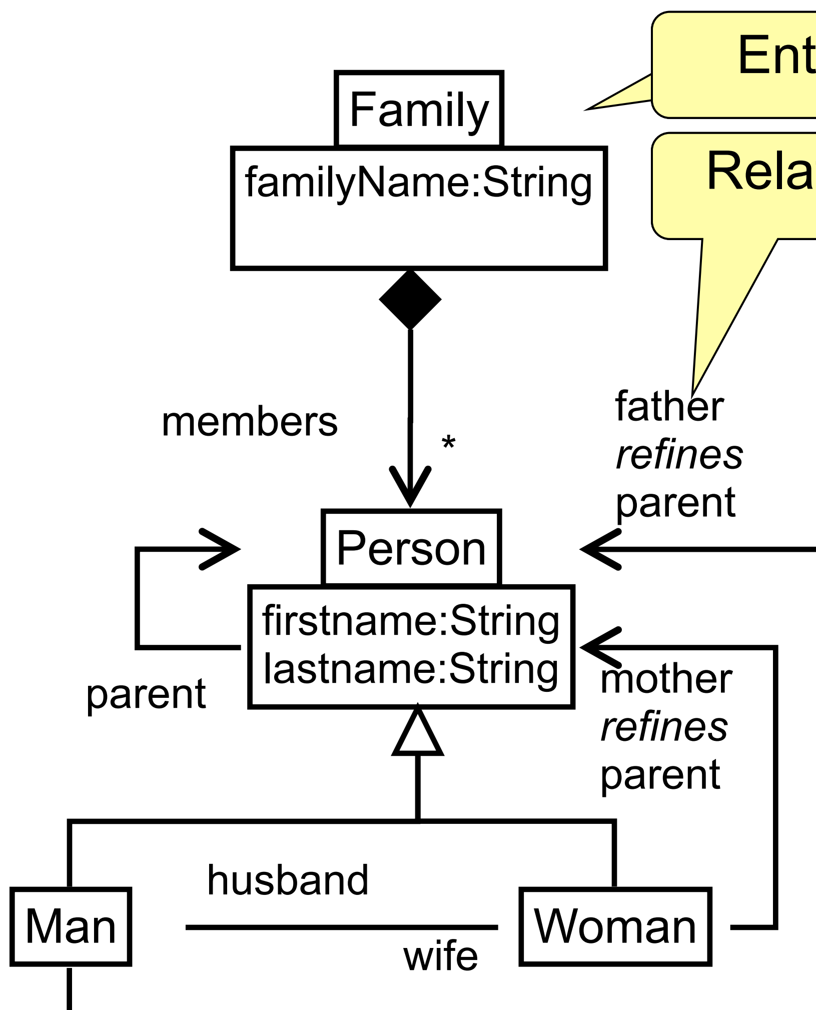
The VIATRA2 Approach

- 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

The VIATRA2 Approach

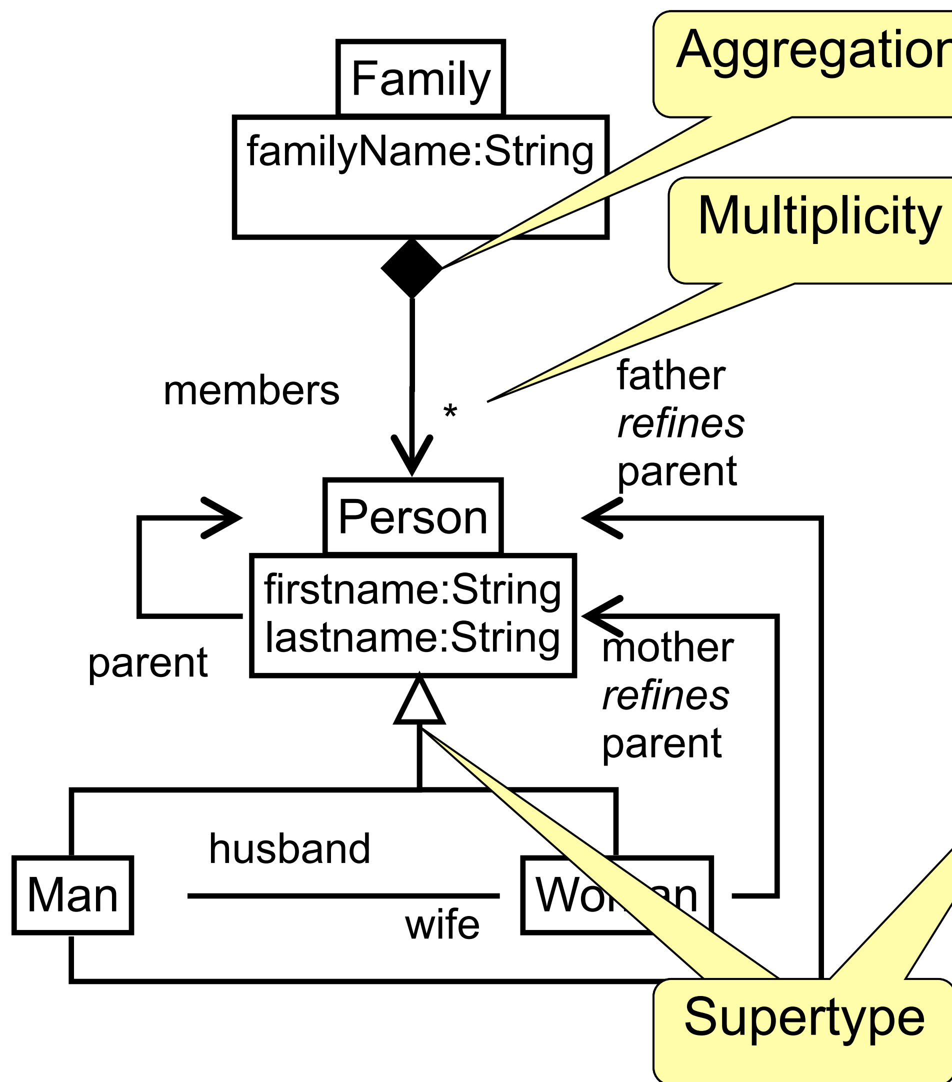
- Model management:
  - **Model space**: Unified, global view of models, metamodels and transformations
    - Hierarchical graph model
    - Complex type hierarchy
    - Multilevel metamodeling
- Model manipulation and transformations: integration of two mathematically precise, **rule** and **pattern-based** 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)



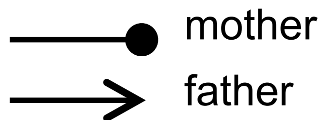

```
entity(family) {
    relation(members, family, person);
    isAggregation(members, true);
}
entity(person) {
    relation(parent,person,person);
    relation(father,person,man);
    multiplicity(father, many_to_one);
    supertypeOf(parent,father);
    relation(firstname, person,
    datatypes.String);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

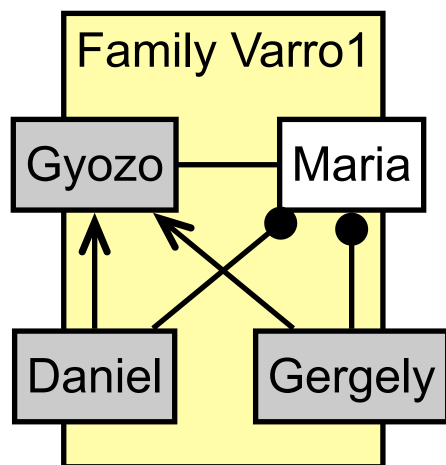
Metamodeling in VIATRA2 (VTML)




```
entity(family) {
    relation(members, family, person);
    isAggregation(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);
```

Models in VIATRA2 (VTML)





Namespace, Imports

Instance definition

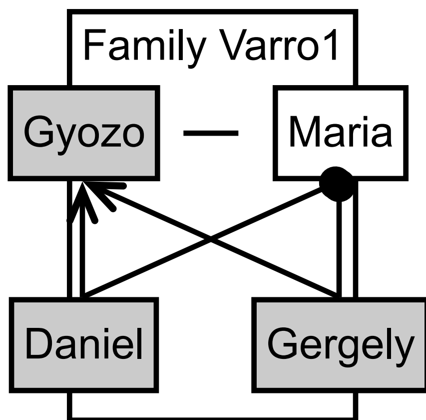
Explicit instance-of

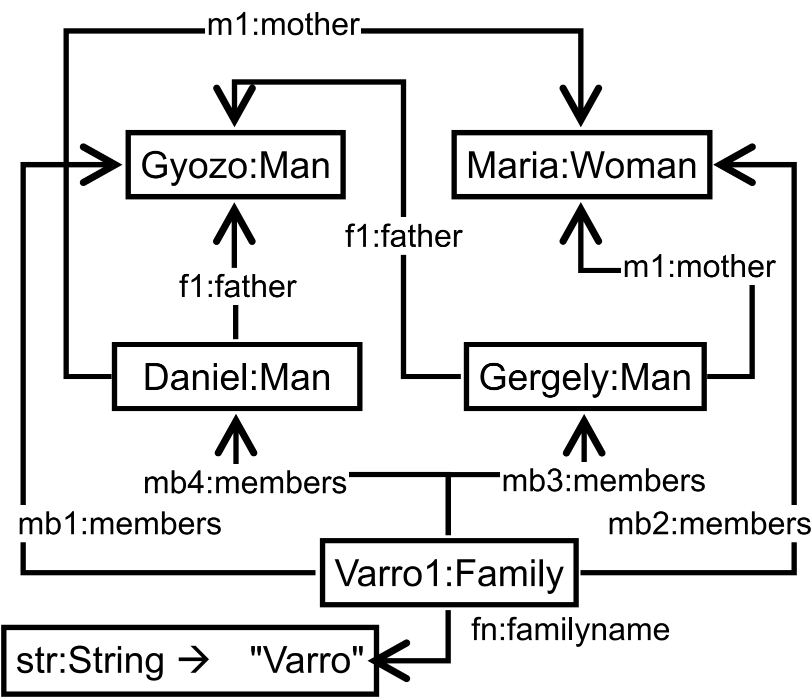
Value OO datatype

```
namespace people.models;
import people.metamodel;

family('Varro1') {
    man('Gyozo') {
            man.wife(wf1, Gyozo, Maria);
    }
    woman('Maria') {
            woman.husband(hb1, Maria, Gyozo);
    }
    man('Daniel');
    entity('Gergely');
    instanceOf(Gergely, man);
    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');
```

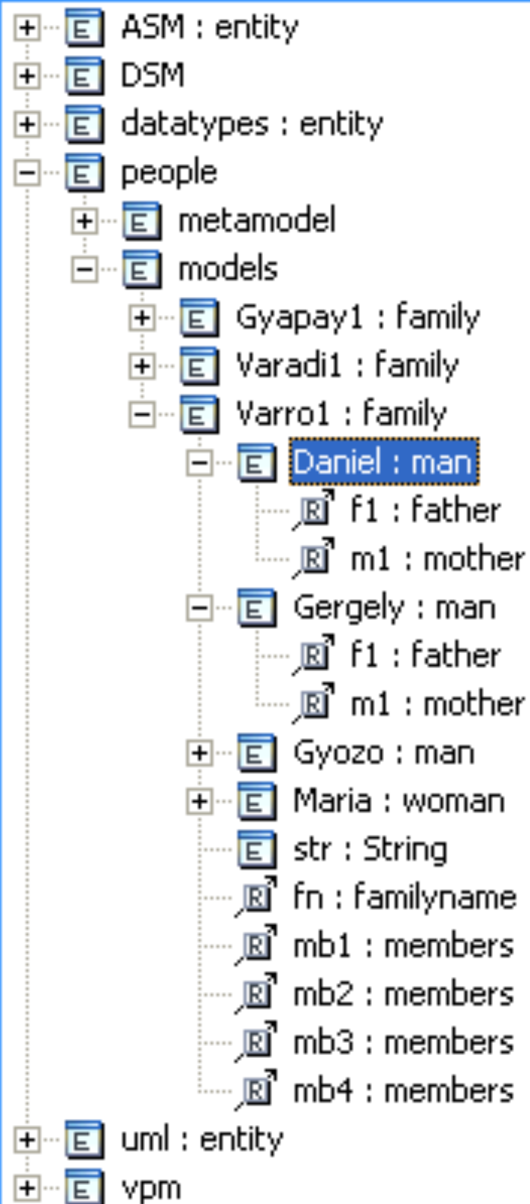
Graphical notation





Graph view

Containment View



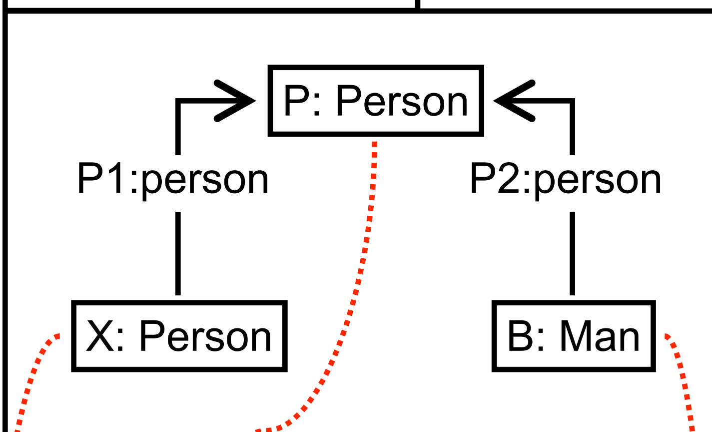
Comparison to EMF

	EMF	VPM / VIATRA
Namespace	Only for metamodels	Both models and metamodels
Navigation	As explicitly defined	Always bidirectional
Generalization	Only for classes	Boths for classes and references
Containment	Along references	Independent from references
Generics	Uni-directional instance-of	Bi-directional instance-of

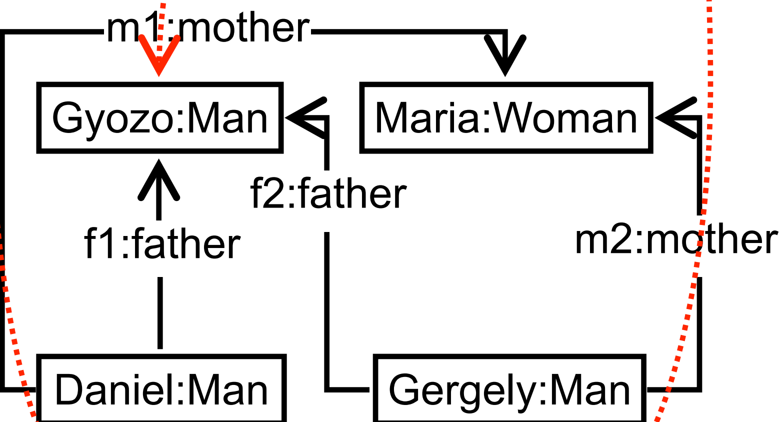
The VIATRA2 Approach

- 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

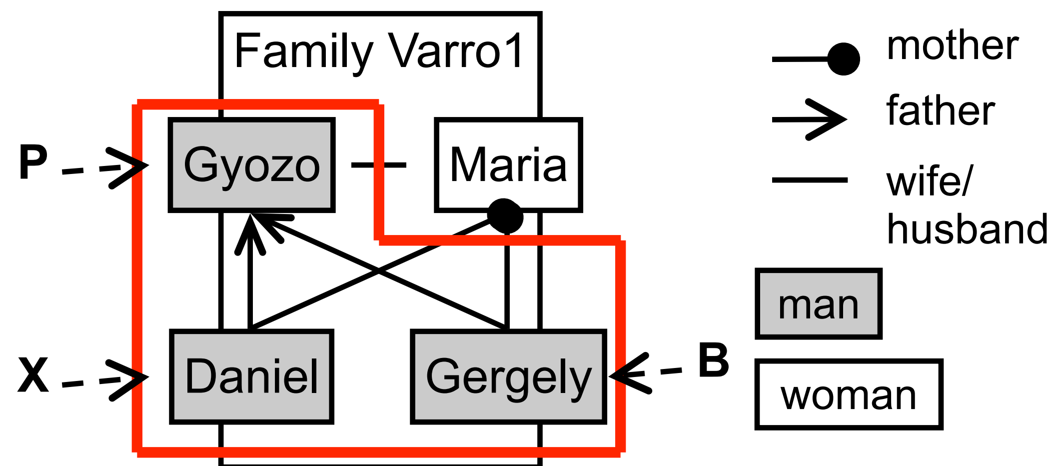


matching



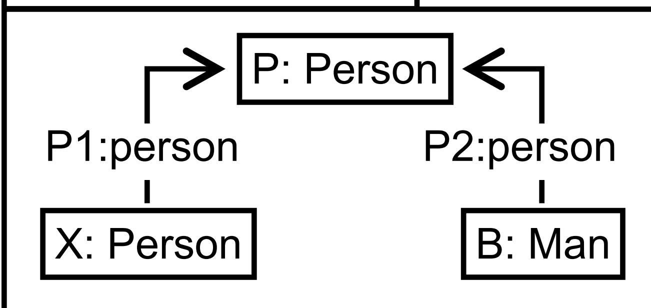
Instance Model

Graphical notation



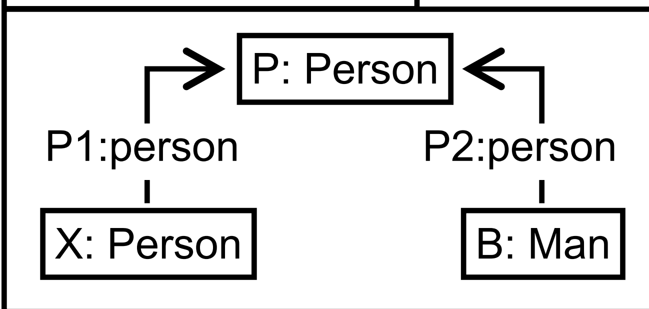
- Graph Pattern:
  - Structural condition that have to be fulfilled by a part of the model space
- Graph pattern matching:
  - A model (i.e. part of the model space) can satisfy a graph pattern,
  - if the pattern can be matched to a subgraph of the model

Graph patterns (VTCL)

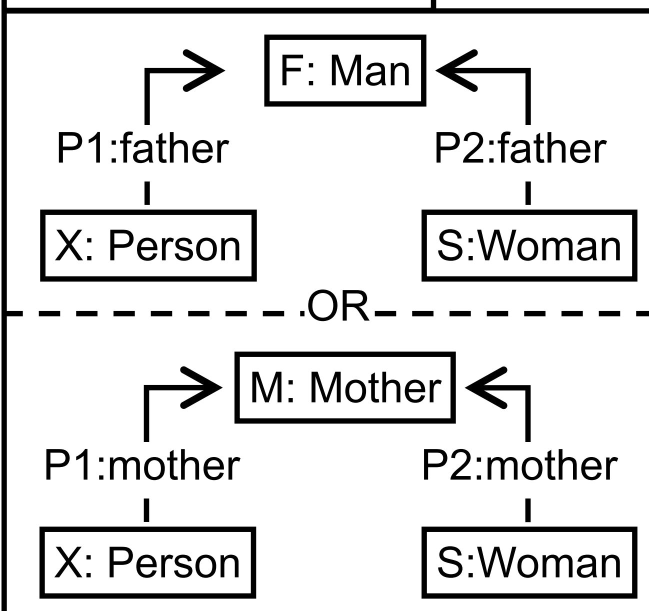



```
// B is a brother of X
pattern brother(X, B) =
{
    person(X);
    person.parent(P1, X, P);
    person(P);
    person.parent(P2, B, P);
    man(B);
```

Graph patterns (VTCL)



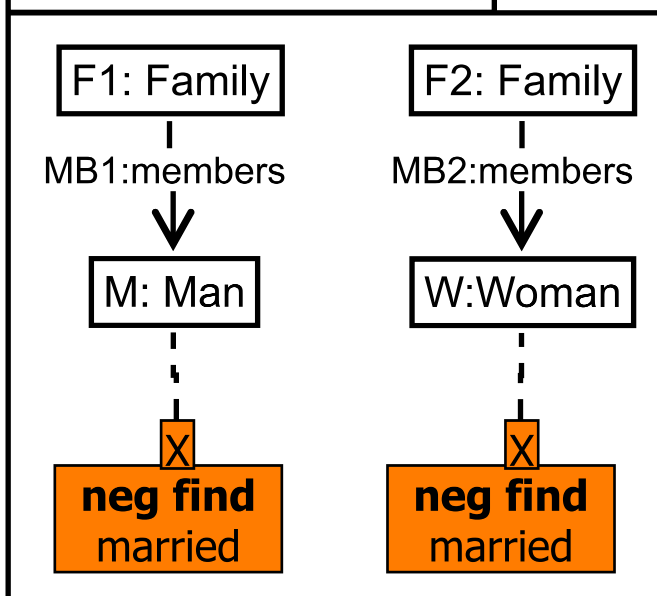

```
// B is a brother of X
pattern brother(X, B) =
{
    person(X);
    person.parent(P1, X, P);
    person(P);
    person.parent(P2, B, P);
    man(B);
```

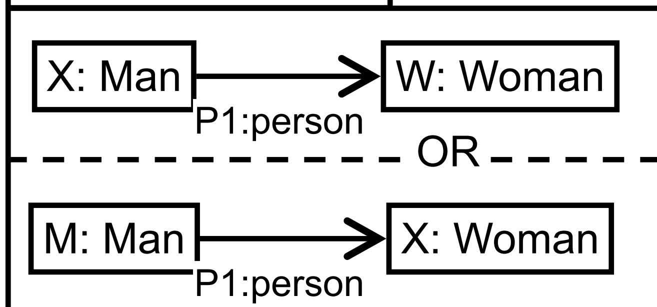


OR pattern

```
// 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);
```

Negative patterns (VTCL)



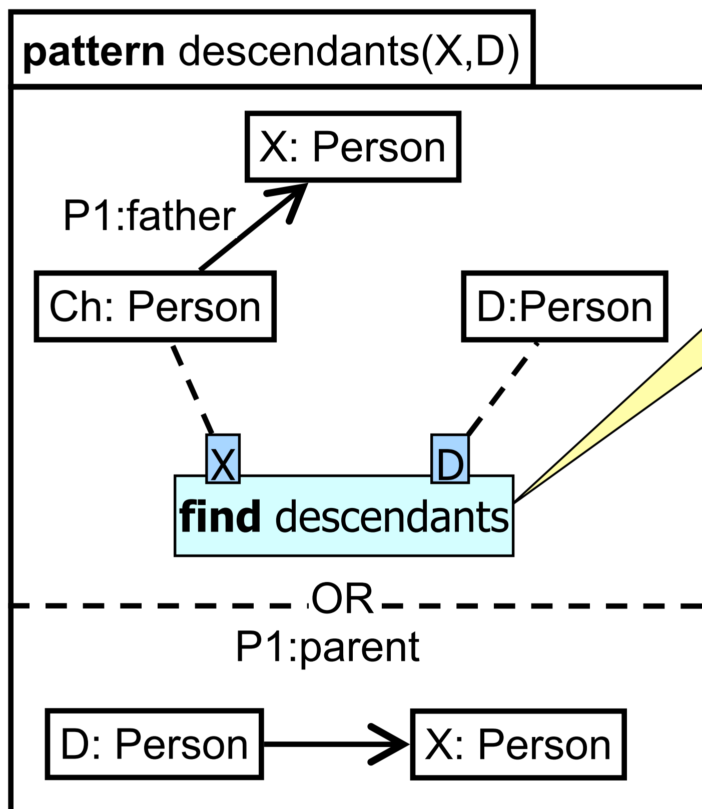


Negative pattern

```
pattern mayMarry(X, D) = {
    man(M);
    family(F1);
    family.members(MB1, F1, M);
    woman(W);
    family(F2);
    family.members(MB2, F2, W);
    neg find married(M);
    neg find married(W);
    check (F1 != F2);
}

pattern married(X) =
{
    man(X);
    man.wife(WF, X, W);
    woman(W);
} or {
    woman(X);
```

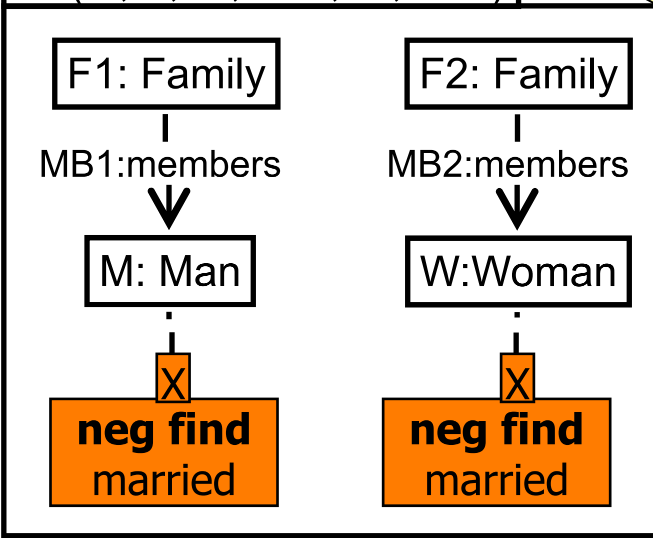
Recursive patterns (VTCL)



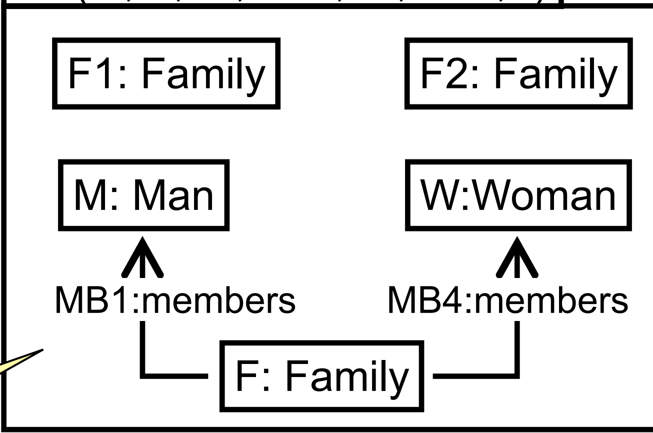
Recursive pattern

```
pattern descendants(X, D) = {
    person(X);
    person.parent(P2, Ch, X);
    person(Ch);
    find descendants(Ch, D)
    person(D);
} or {
    person(X);
    person.parent(P1, D, X);
    person(D);
```

Graph transformation rules (VTCL)



Precondition pattern

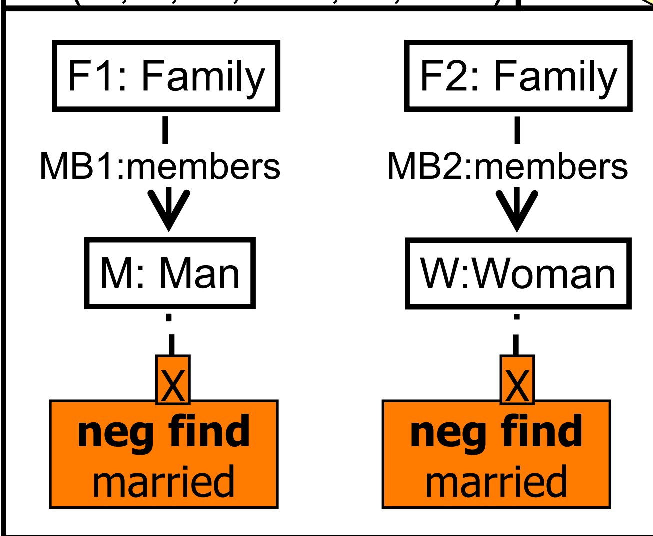


Postcondition pattern

```
gtrule marry(in M, in W, out F) =
precondition pattern
lhs(M,W,F1,MB1,F2,MB2) = {
    family(F1);
    family.members(MB1, F1, M);
    man(M);
    family(F2);
    family.members(MB2, F2, W);
    woman(W);
    neg find married(M);
```

```
postcondition pattern
rhs(M,W,F1,MB1,F2,MB2) = {
    family(F1);
    man(M);
    family(F2);
    woman(W);
    family(F);
    family.members(MB3, F, M);
    family.members(MB4, F, W);
```

Graph transformation rules (VTCL)



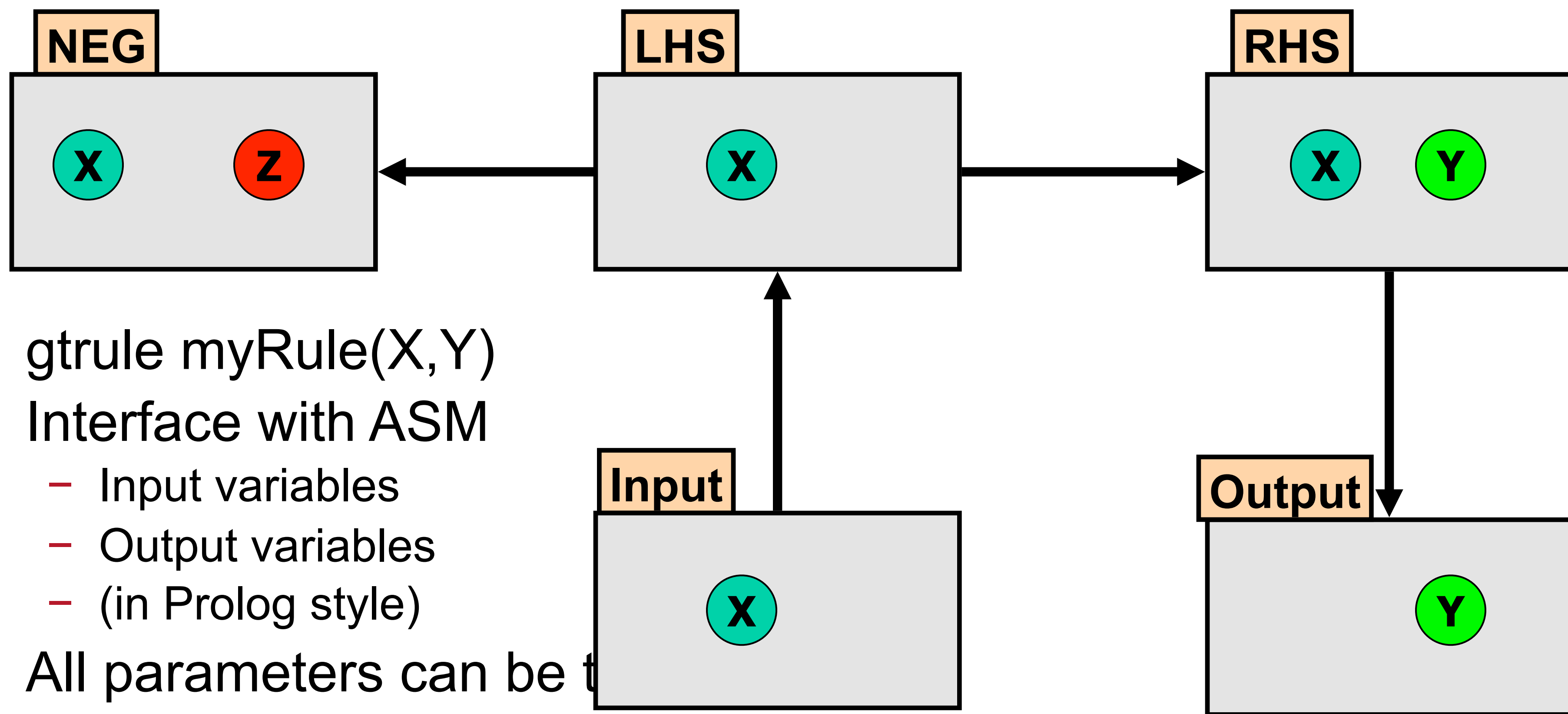
Precondition pattern

Action Part

```
gtrule marry(in M, in W, out F) =
precondition pattern
lhs(M,W,F1,MB1,F2,MB2) = {
    family(F1);
    family.members(MB1, F1, M);
    man(M);
    family(F2);
    family.members(MB2, F2, W);
    woman(W);
    neg find married(M);
```

```
action
  {
    delete(MB1);
    delete(MB2);
    new(family(F));
    new(family.members(MB3, F, M));
    new(family.members(MB4, F, W));
```

Structure of GT rules



- gtrule myRule(X,Y)
- Interface with ASM
  - Input variables
  - Output variables
  - (in Prolog style)
- All parameters can be t

Generic GT rules (VTCL)

gtrule parentIsAncR(Par,Child)

precondition lhs(Par,Child)

P1:parent

Par:Class ← Child:Class

postcondition rhs(Par,Child)

E:anc

Par:Class ← Child:Class

P1:parent

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

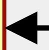
precondition lhs(Par, Child, P1
ClsE,ParR,AncR)

ParR: relation

ClsE:entity

AncR: relation

P1:relation



Par:entity ← Child:entity

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

ParR: relation

ClsE:entity

AncR: relation

P1:relation

Par:entity ← Child:entity

E: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,
```

GT extensions to ASMs

• Permanent states (models) + elementary model manipulation

Iterate choose vs. Forall

- Executes a GT rule as long as possible
- Potential nontermination

- Executes a GT rule for each match one by one
- Potentially conflicting effects

```
// This may cause nontermination if the same
// match for X and B is enabled after the
// body of seq is executed
iterate choose X, B with find brother(X, B) do
    seq {

    }
```

```
// This may cause a conflict when X and B
// are both man, and they are brothers

forall X, B with find brother(X, B) do seq {
        delete(X);
    }
```

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 driven by graph patterns
  - Code templates and/or print rules

SENSORIA

Code templates (Ongoing)

- Code generation
  - Code templates
  - Code formatters
- Code templates (alpha)
  - 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

```
template printClass(in C) =
{
public class $C {
#(forall At,Typ with attrib(C,At,Typ) do seq{)
private $Typ $At;
#(})
}
}

// Result
rule printClass(in C) = seq {
    print("public class " + C + "{");
    forall At,Typ with attrib(C,At,Typ) do
      seq {
        print("private " + Typ + " " + At + ";");
      }
    print("}");
```

Other advanced MT issues

- Reusability
  - Pattern composition (**find**)
  - Rule calls (**call**)
- Traceability
  - Reference metamodels
  - Explicit storage of reference models
- Extensibility
  - Native transformations:
    written in Java, called from MT programs
  - Integration of domain-specific models

ADVANCED TRANSFORMATION CONCEPTS

Incremental graph pattern matching

- … previous talk

What *really* is the incremental pattern matcher?

- Graph patterns ~ complex constraint sets
  - Matching set: overlay of “valid” elements
- RETE: a complex overlay cache of the model graph
  - Stores matching sets explicitly
- A historical cache
  - Can store “previous” overlay states

What *really* is the incremental pattern matcher?

- Graph patterns ~ complex constraint sets
  - Matching set: overlay of “valid” elements
- RETE: a complex overlay cache of the model graph
  - Stores matching sets explicitly
- A historical cache
  - Can store “previous” overlay states

Idea

- Use the historical overlay to
  - Easily compute non-trivial change information (beyond elementary changes)
    - Detect high-level “events”
    - Detect (the net effect of) complex change sequences
  - Assign actions to these changes
    - Event-Condition-Action formalism in rule-based systems
    - → event-driven transformations
  - Or, easily track the validity set of constraints as the model is evolving
    - → constraint satisfaction solving
  - Or, easily track enabledness conditions for simulation rules
    - → stochastic model simulation

Event-driven and live transformations

- Core idea: represent *events* as changes in the matching set of a pattern.
  - More general than previous approaches
  - Use events and reactions as transformation specification
- Live transformations
  - maintain the context (variable values, global variables, …);
  - run as a "daemon", react whenever necessary;
  - as the models change, the system can react instantly, since everything needed is there in the RETE network: *no re-computation is necessary.*
- Paper at ICMT2008: Live model transformations driven by incremental pattern matching.

Change-driven transformations

- Generalization of the live transformation approach
- New formalism supports more precise *change queries*
  - To distinguish between various execution trajectories that result in the same net change
  - A more complete adaptation of the ECA approach to graph transformations
- Application scenarios
  - Incremental model synchronization for non-materialized models (paper at MODELS2009: Change-driven transformations)
  - Back-annotation of simulation traces (paper at SEFM2010)

Constraint satisfaction over models

- Motivation
  - Very pragmatic problem area: constraint evaluation, "quick fix" computation, design or configuration-space exploration etc.
  - CLP(FD) limited: only finite problems can be formulated
    - Problematic: object birth-and-death
  - CLP(FD) difficult to use
- GT-based approaches
  - Easier-to-use formalism
  - Infinite (dynamic) problems
  - "Traditional" problem: scalability ☹
- Our aim: combine ease-of-use and flexibility with performance

CSP(M)

- Described by *(M0,C,G,L)*
  - *M0 initial model* (typed graph)
  - *C* set of *global constraints* (graph patterns)
  - *G* set of *goals* (graph patterns)
  - *L* set of *labeling rules* (GT rules)
- Goal: Find a model Ms which satisfies all global constraints and goals (One, All, Optimal solutions)
- Extensions
  - Flexible CSP
    - Strong-weak constraints
    - Quality metrics
  - Dynamic CSP: Add-remove constraints, goals, labeling rules on-the-fly
    - Re-use already traversed state space

Performance

- "Pure" and "Flexible" CSP(M)
  - Execution times significantly faster than KORAT or GROOVE
- Dynamic CSP(M)
  - We can even beat Prolog CLP(FD) in some cases ☺
- In general
  - VERY problem specific results
  - Lots of potential for future research

Stochastic simulation by graph transformation

- Joint work with prof. Reiko Heckel's group (University of Leicester)
- Conceptually similar to the CSP engine
  - Simulation rule: precondition + transformation
  - Assign probability distributions to simulation rules
  - Execute as-long-as-possible
  - Observe system state changes
- Papers
  - FASE2010: P. Torrini, R. Heckel, I. Ráth: Stochastic simulation of graph transformation systems
  - GT-VMT2010: P. Torrini, R. Heckel, I. Ráth, G. Bergmann: Stochastic graph transformation with regions
  - ASMTA2010: A. Khan, P. Torrini, R. Heckel and I. Ráth: Model-based stochastic simulation of P2P VoIP using graph transformation

Applications

- VoIP P2P network behavioral simulation (Skype)
- Other simulation scenarios
  - Social networks
  - Traffic networks
  - Crystal growth
- Evaluation
  - +: Scales well (comparable to dedicated simulators)
  - +: GT is a high-level behavioral specification formalism that is **easy-to-use**
  - -: does not (yet) support certain topological constraints (e.g. transitive closure)

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 and extensible execution strategies
  - Usable for tool integration problems in practice

VIATRA2 IN TOOL INTEGRATION

Main Application Fields

- Analysis of Business Process Models
  - Verification by MC, Fault simulation, Security analysis (Bell-LaPadula)
  - Optimization (ILog Solver)
  - ➔ IBM Faculty Awards
- MDD in the Enterprise
  - Incremental correspondence maintenance between Requirements models and
    - Architecture models
    - Design models
  - Change impact analysis
  - ➔ SecureChange IP
- Domain-specific modeling languages
  - Design and transformation of domain specific languages
  - Model simulation and stochastic simulation (→ GraSS)
  - Model-based generation of graphical user interfaces
- SOA
  - Performance & Availability analysis
  - Configuration generation
  - Service Analysis and Deployment descriptor generation
  - Verification by MC (BPEL2SAL and back)
  - ➔ 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
  - Test generation using SAL-ATG
  - Valid PSM generation based on constraints (→ CSP(M))
  - ➔ DECOS, DIANA, MOGENTES IP

Summary

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

Many thanks to

D. Varró, G. Bergmann, Á. Horváth, Á. Hegedüs, Z. Ujhelyi, G. Varró

A. Pataricza, A. Balogh, L. Gönczy, B. Polgár, D. Tóth
Z. Balogh, A. Ökrös, Zs. Déri

And many more students

THANK YOU FOR YOUR ATTENTION