MŰEGYETEM 1782

Precise Model Transformations in Tool Integration

Dániel Varró
Assistant Professor

FAULT TOLERANT SYSTEMS RESEARCH GROUP

Overview - Outline

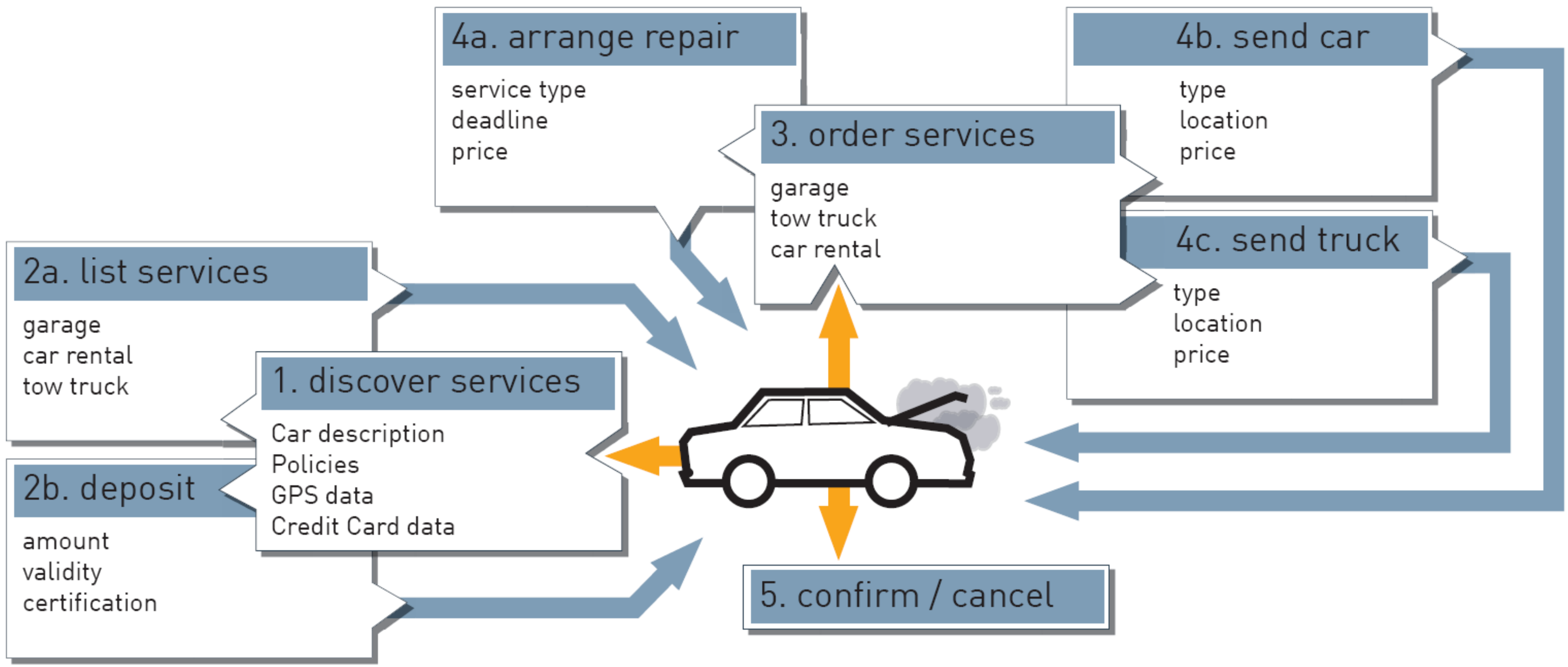
- Introduction
  - Challenges for tool integration
  - Model-driven approach to tool integration
- The VIATRA2 framework
  - Using a combination of formal methods
  - Abstract state machines and graph transformation
- Success Scenarios
  - SENSORIA
  - DECOS

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

Motivating scenario

A car is broken down…



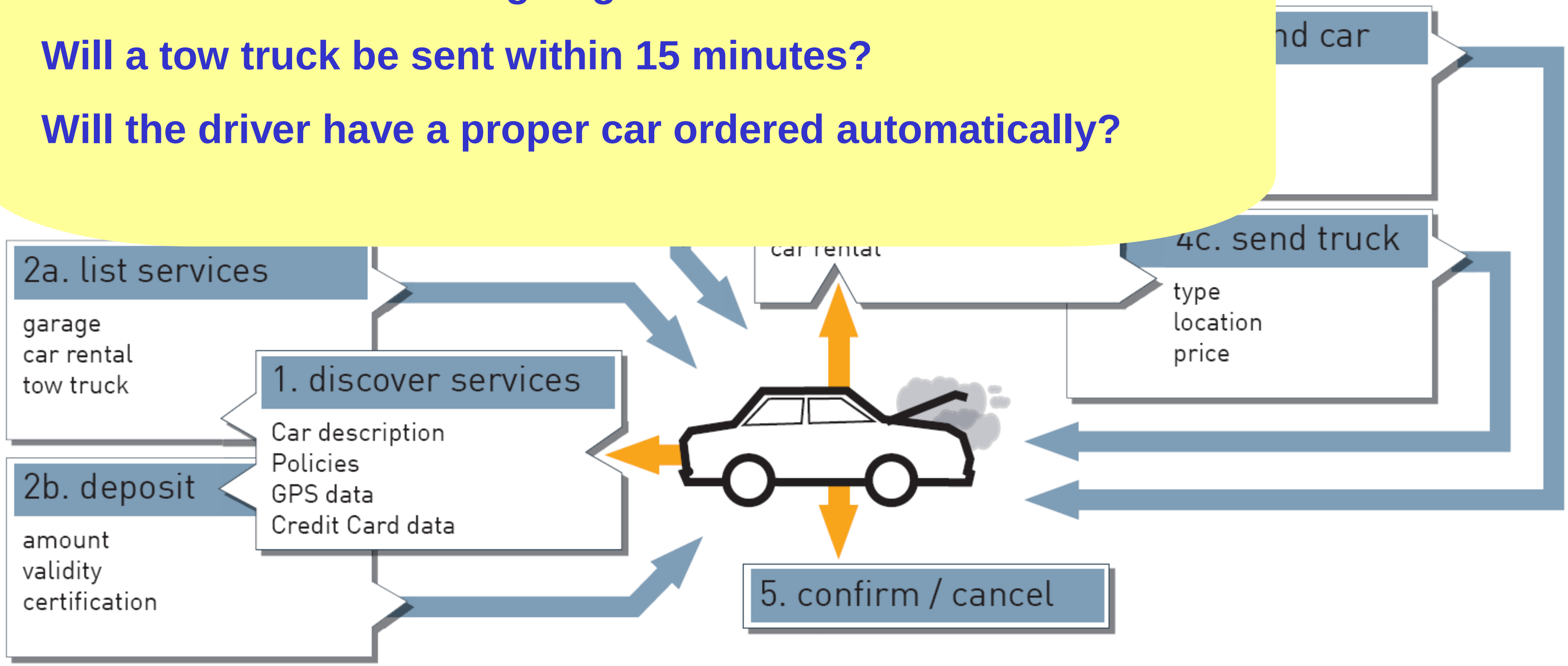
Motivating scenario

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?

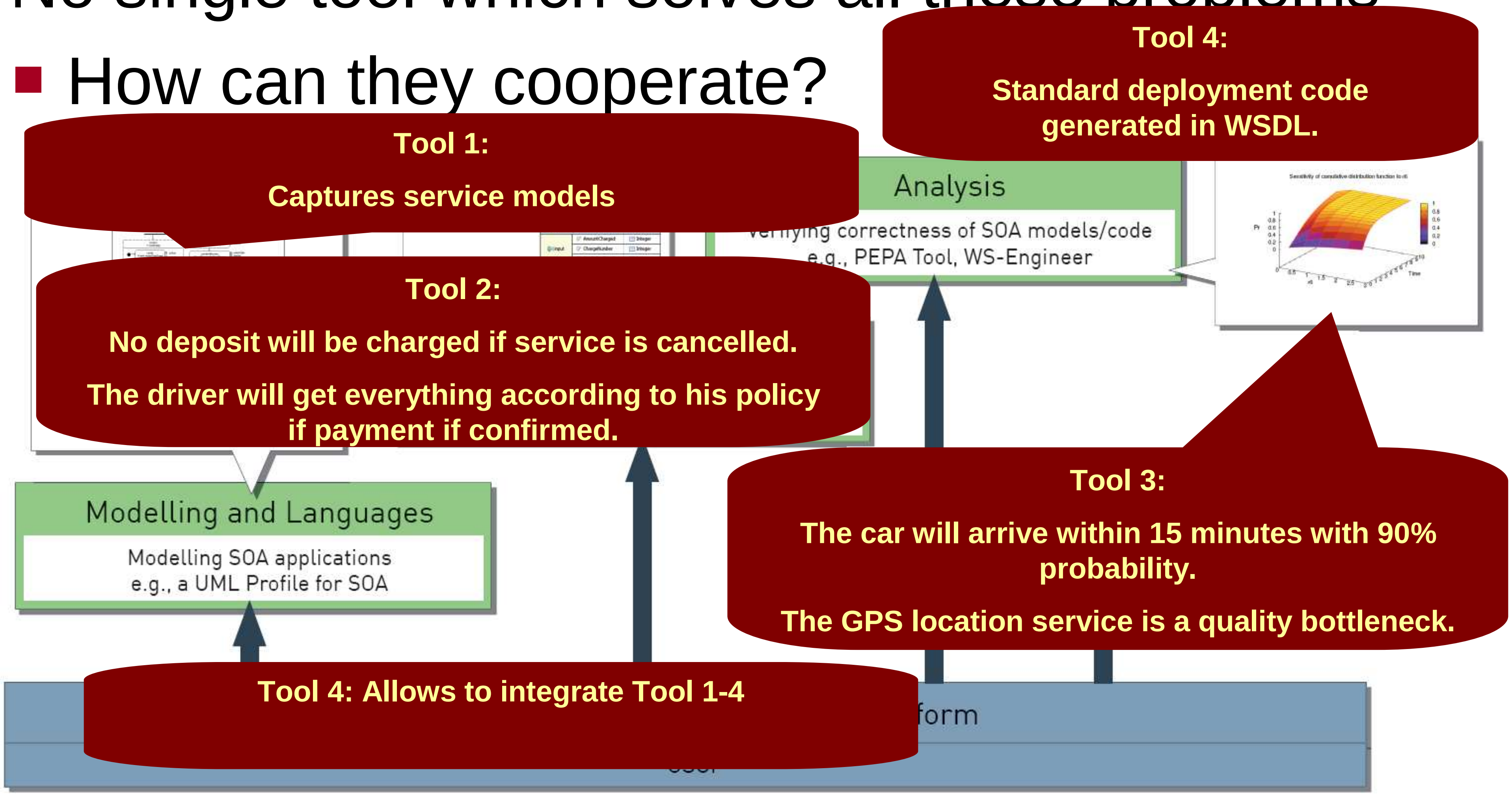
Will the driver have a proper car ordered automatically?



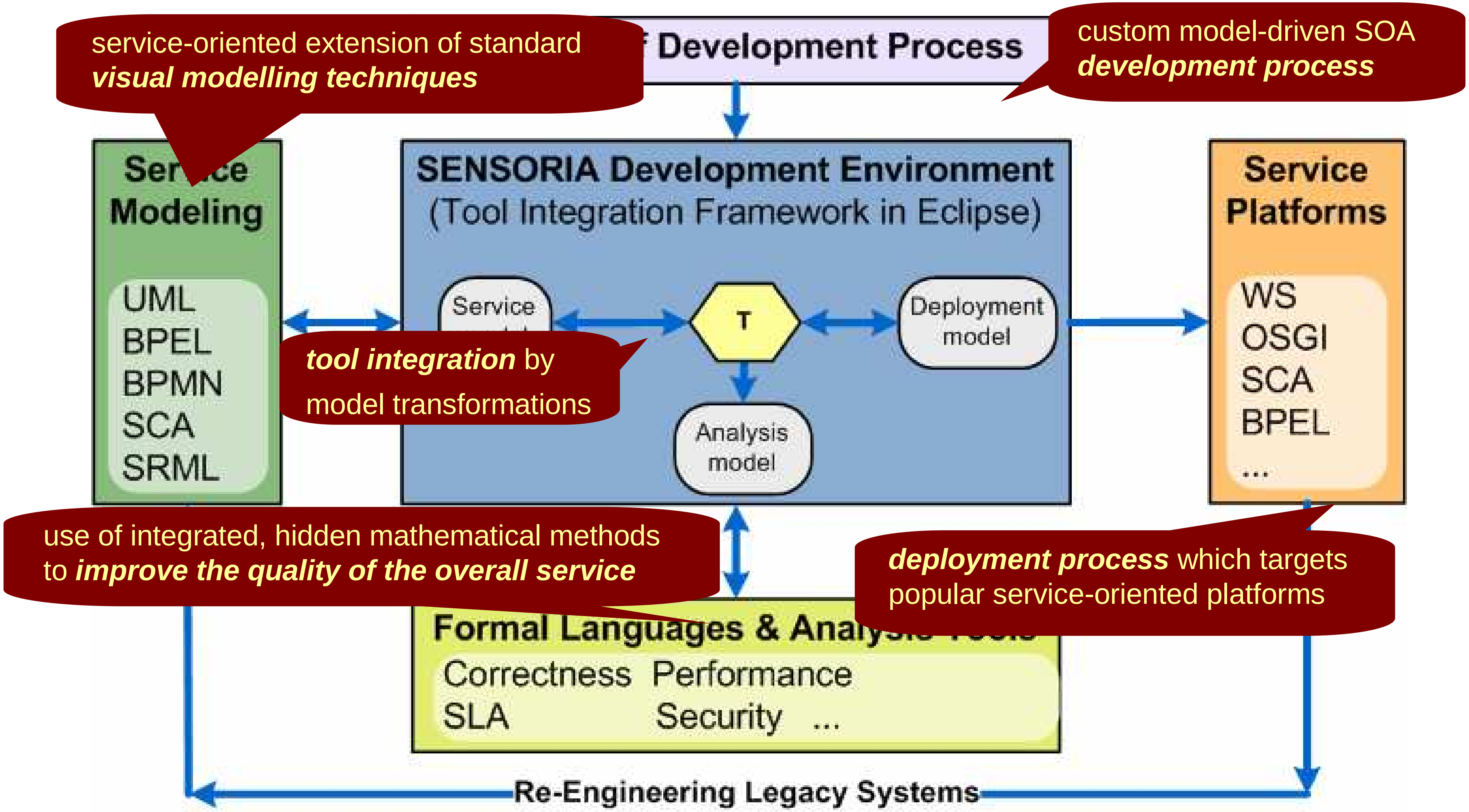
How to provide answers?

No single tool which solves all these problems

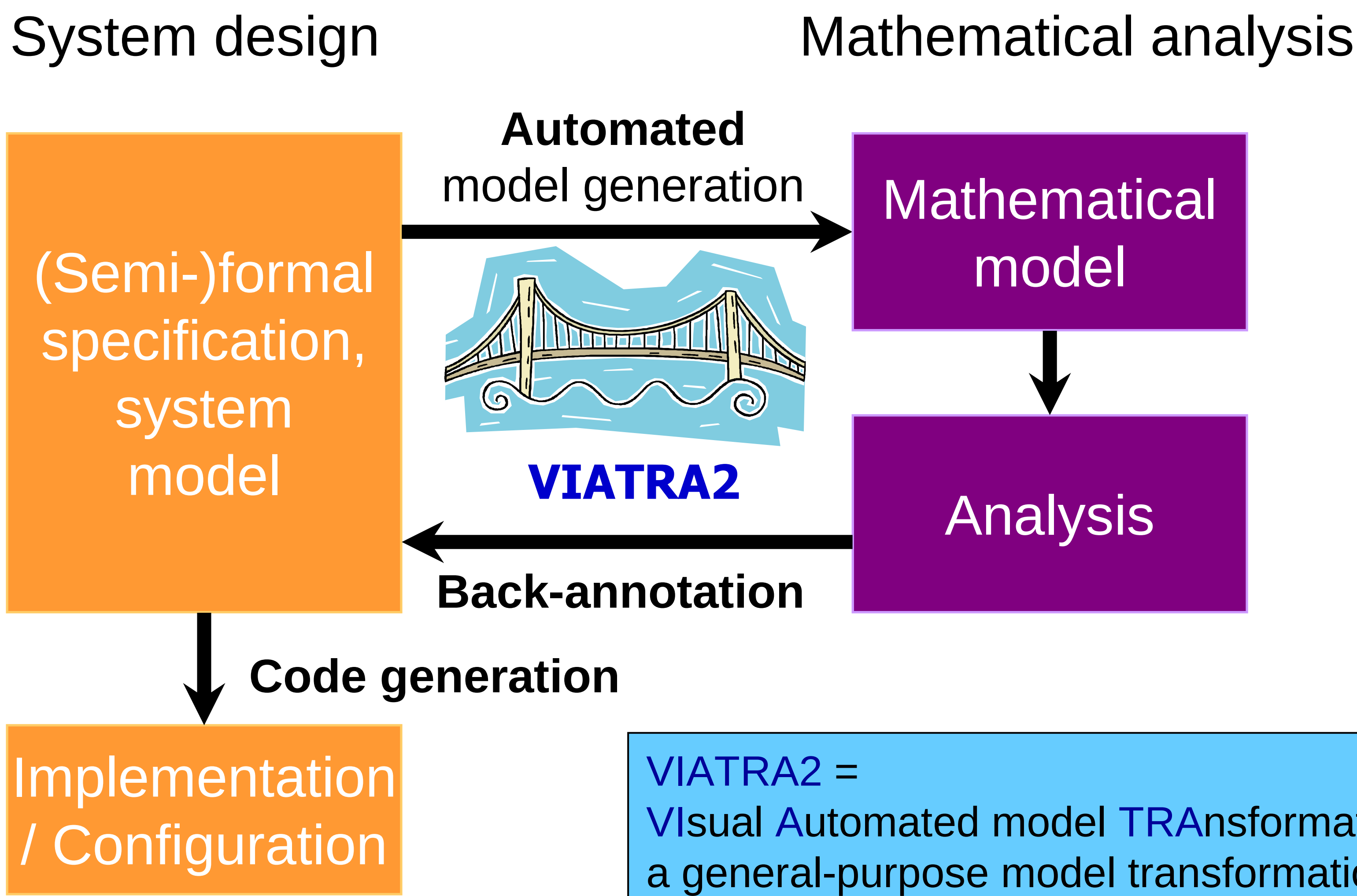
- How can they cooperate?



SENSORIA approach



Model Transformations for Tool Integration



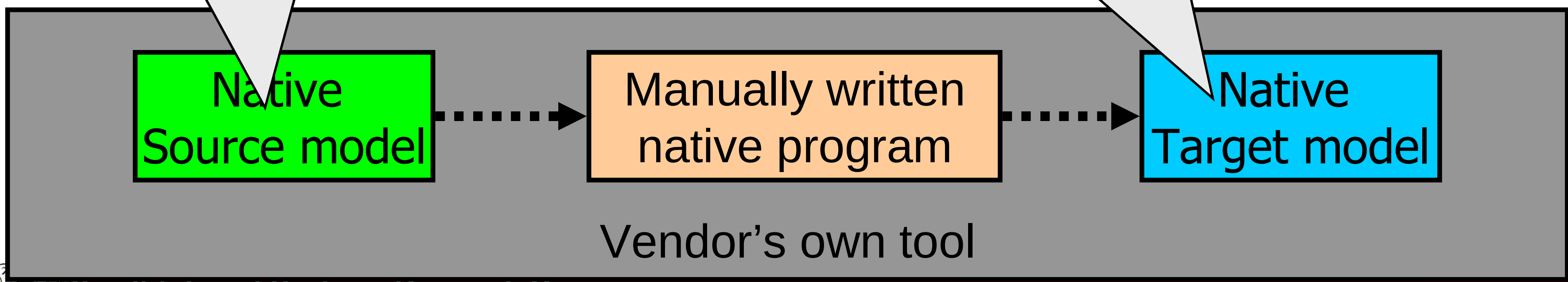
VIATRA2 =
VIsual Automated model TRAnsformations:
a general-purpose model transformation framework

The VIATRA Model Transformation Framework

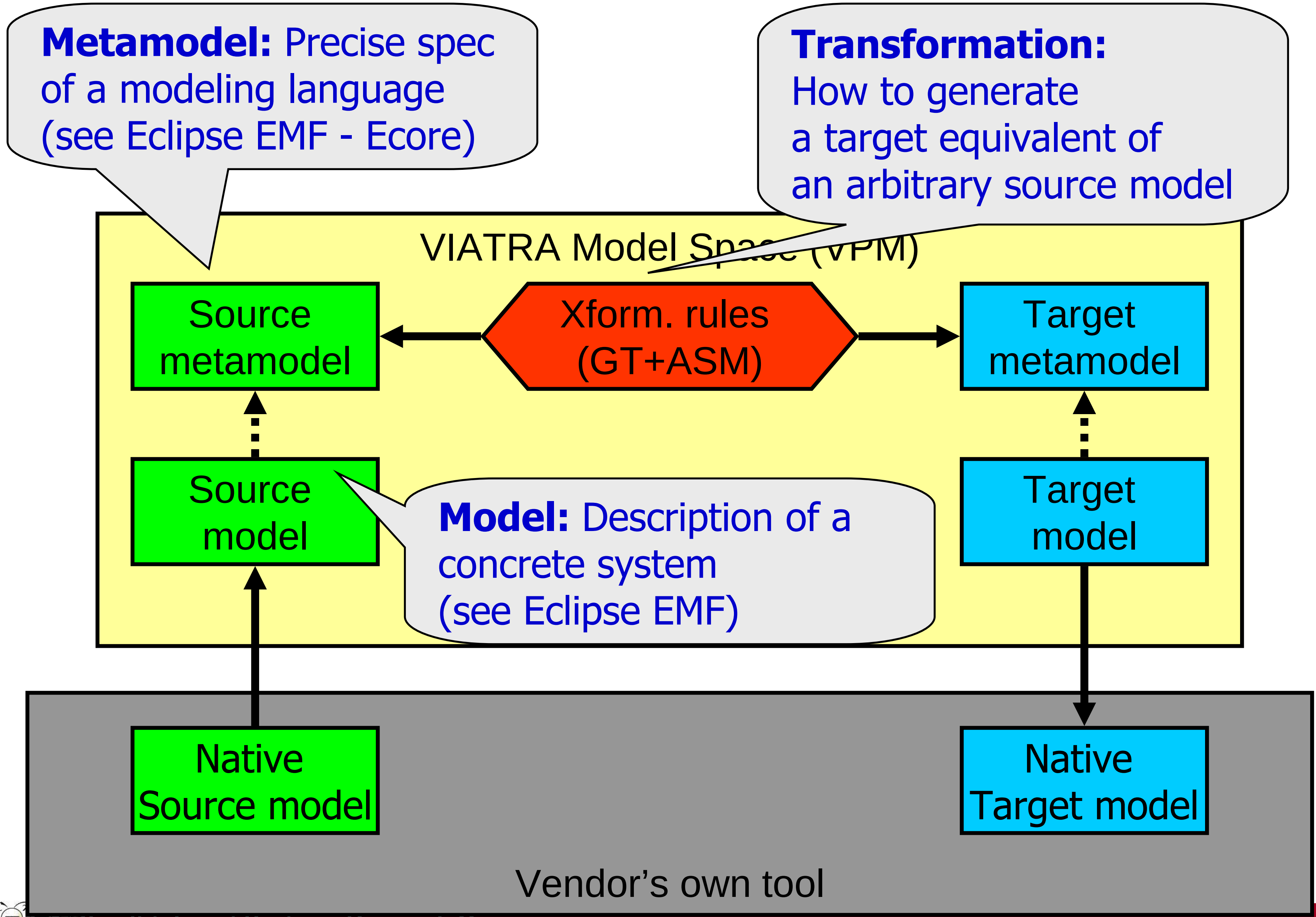
The VIATRA2 framework

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

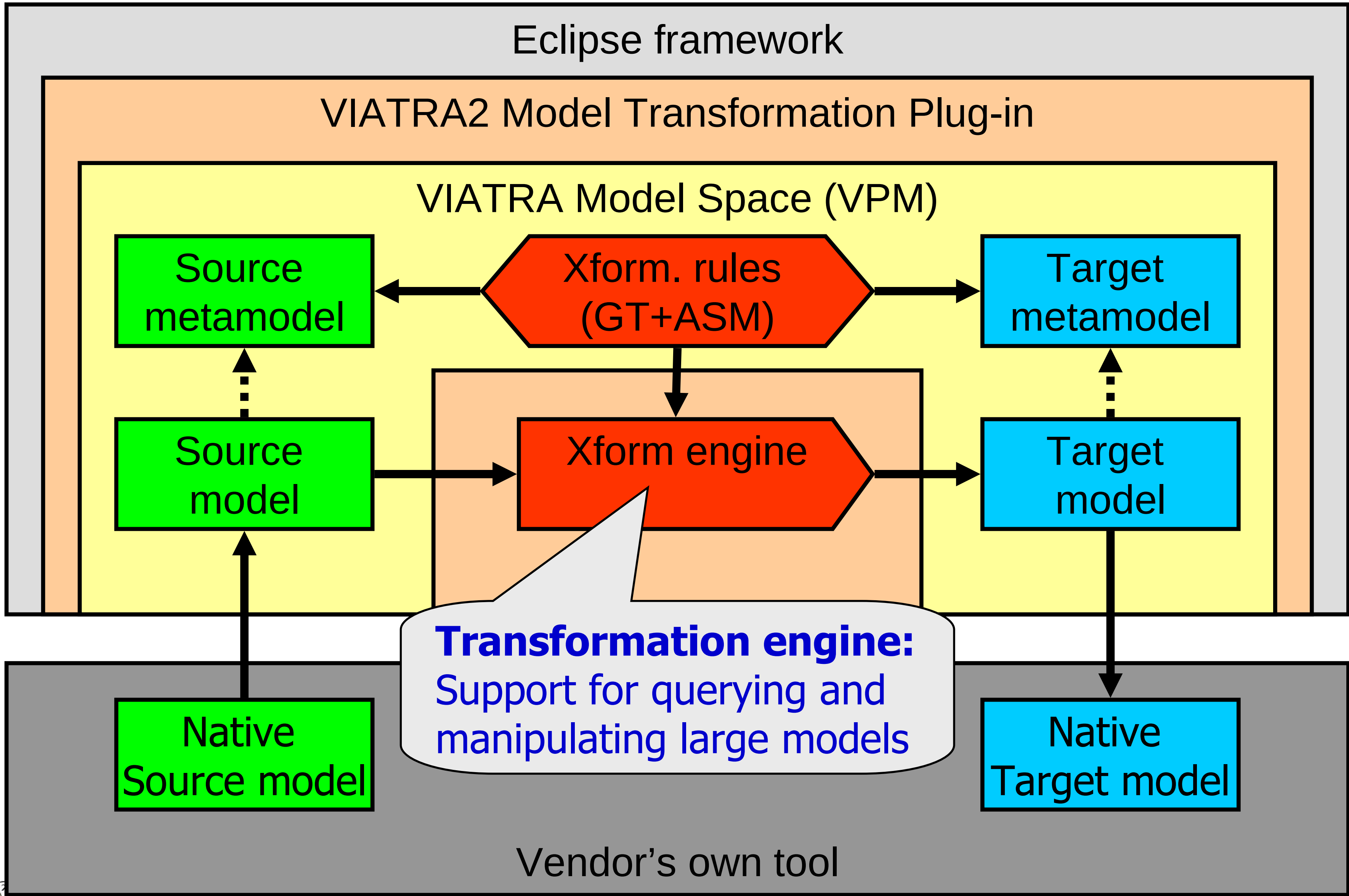
Native target models:
- EMF
- App source code
- XML deployment descript.
- Databases
- Analysis tools



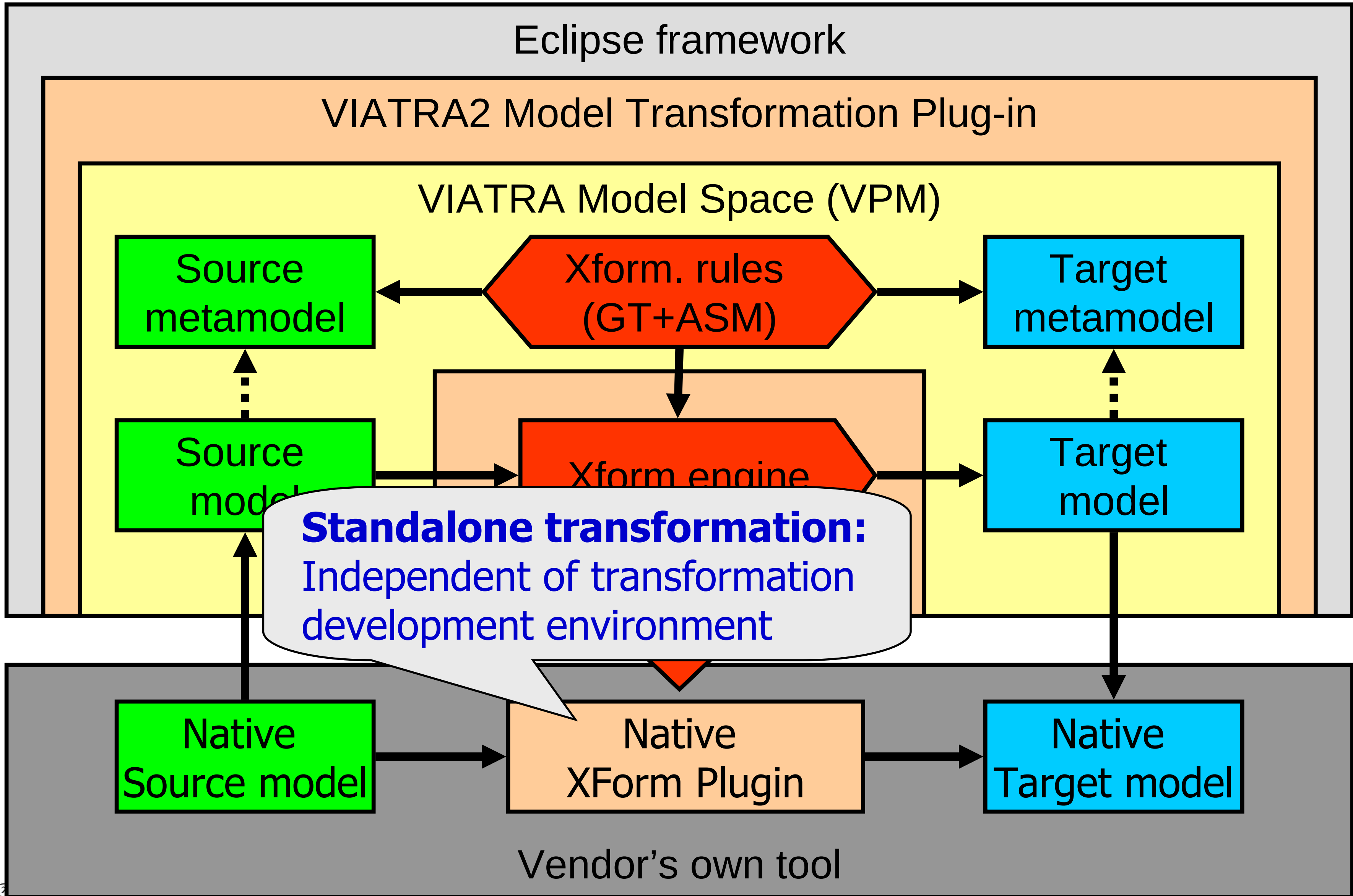
The VIATRA2 framework



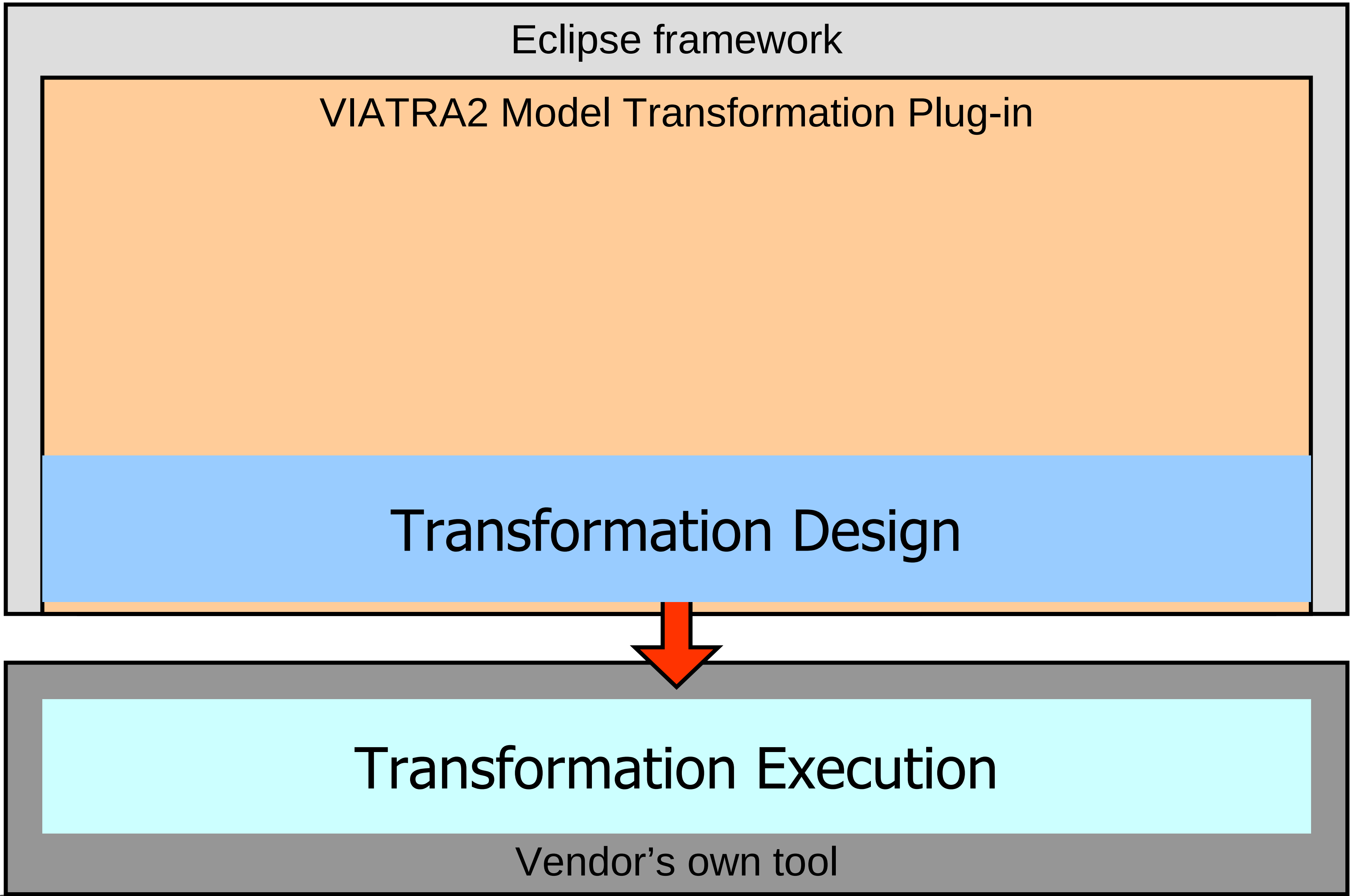
The VIATRA2 framework



The VIATRA 2.0 framework



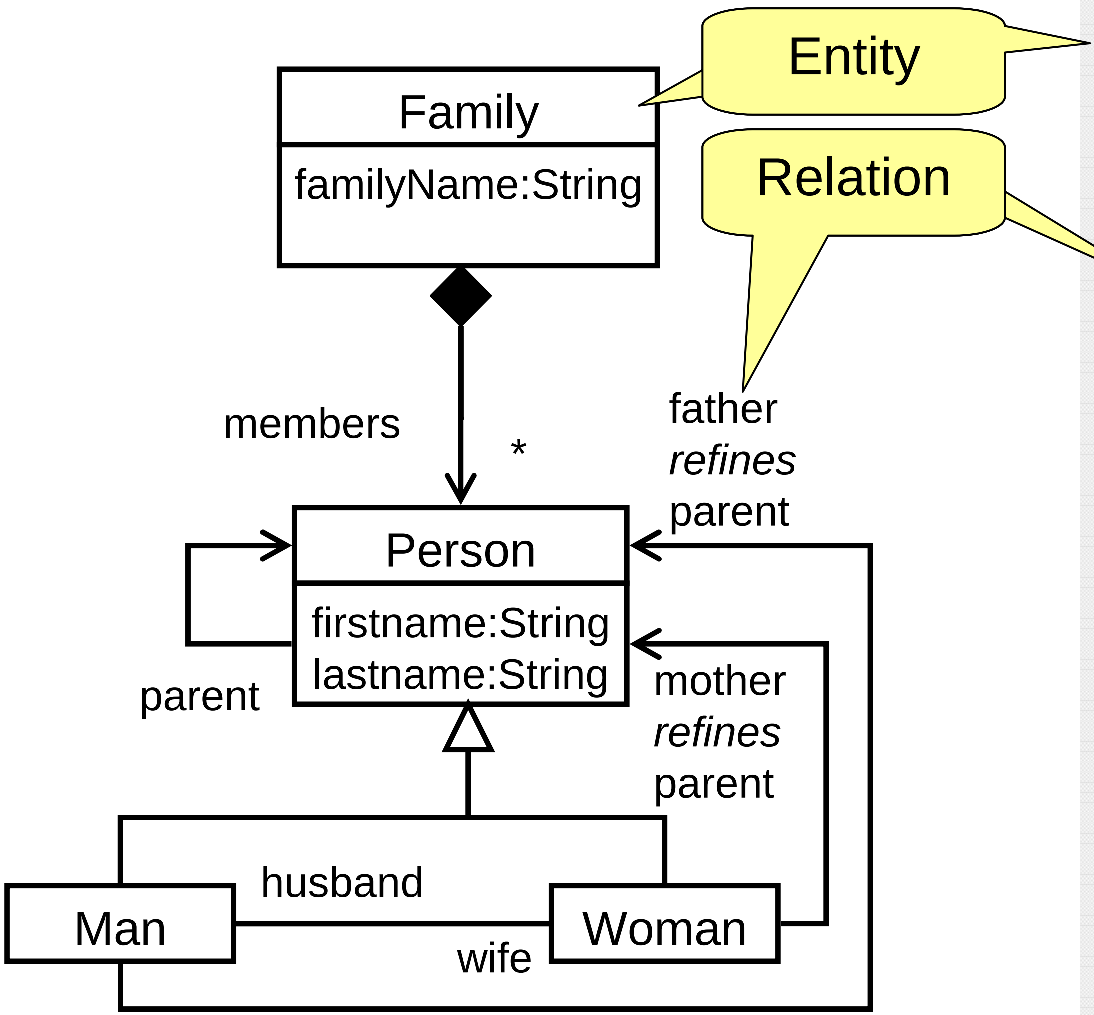
The VIATRA2 framework



The VIATRA2 Approach

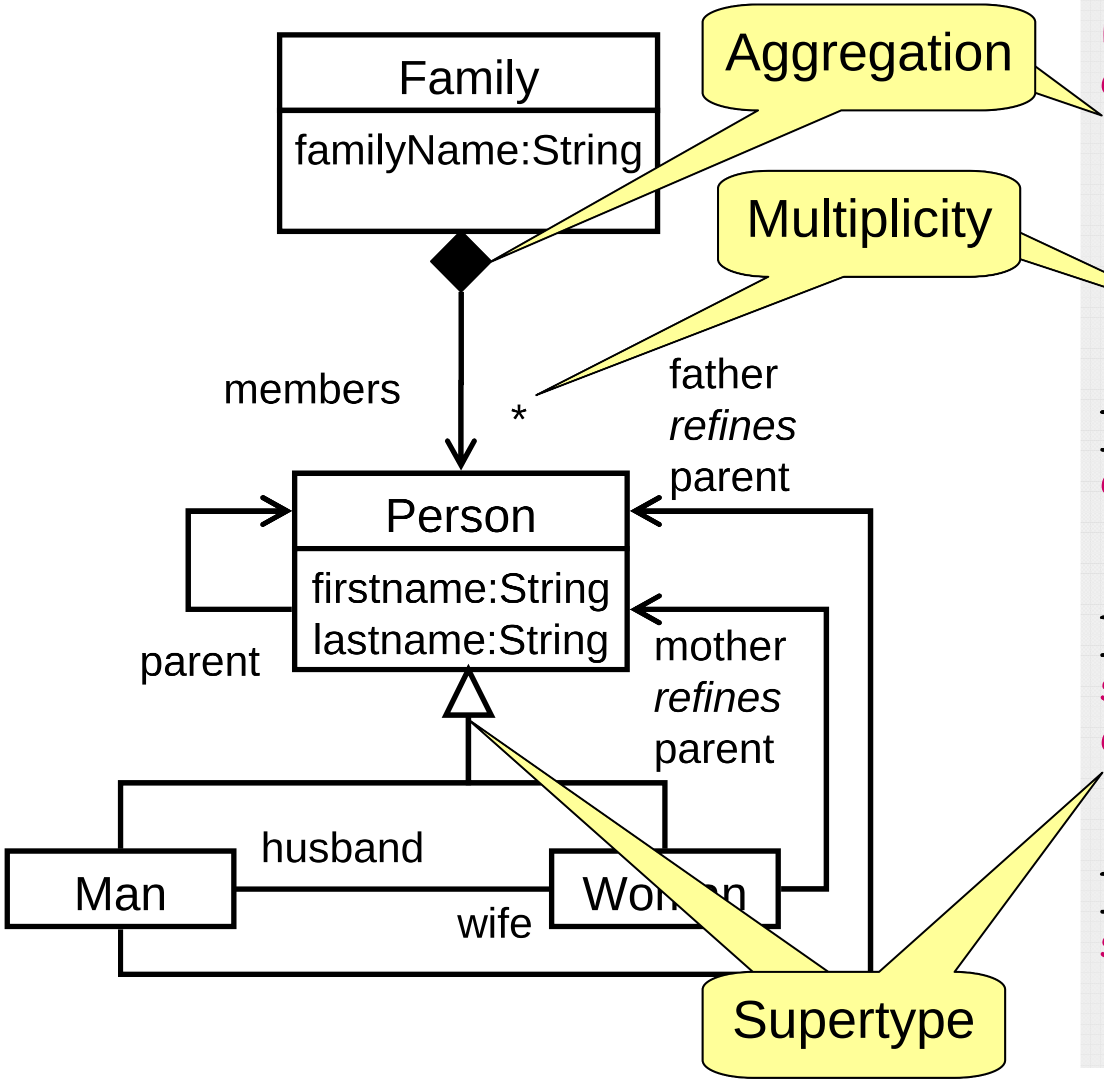
- Model management:
  - **Model space**: Unified, global view of models, metamodels and transformations
    - Hierarchical graph model
    - Complex type hierarchy
    - Multilevel metamodeling
- Model manipulation and transformations: integration of two mathematically precise, **rule** and **pattern-based**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



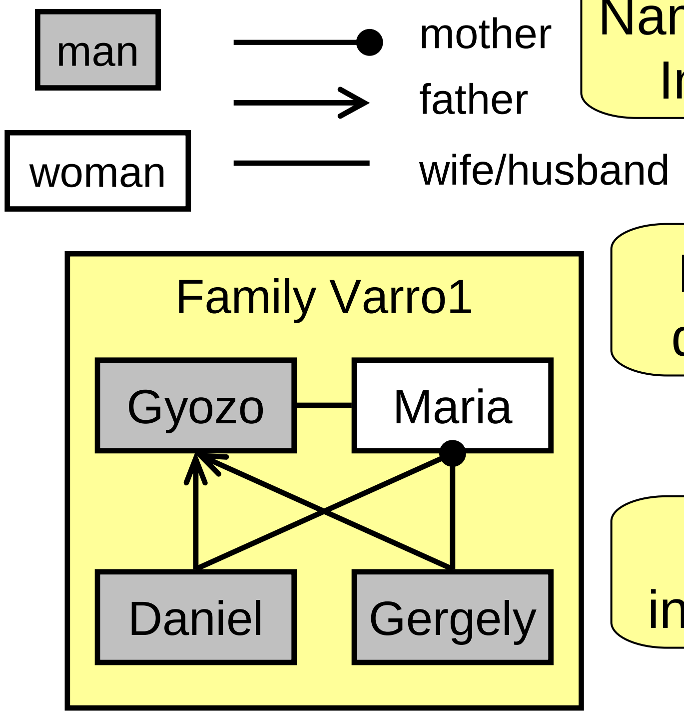

```
entity(family);
relation(members, family, person);
isAggregation(family.members, true);
entity(person) {
    relation(parent,person,person);
    relation(father,person,man);
    multiplicity(father, many_to_one);
    supertypeOf(parent,father);
    relation(firstname, person,
    datatypes.String);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

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);
```

Models in VIATRA2 (VTML)



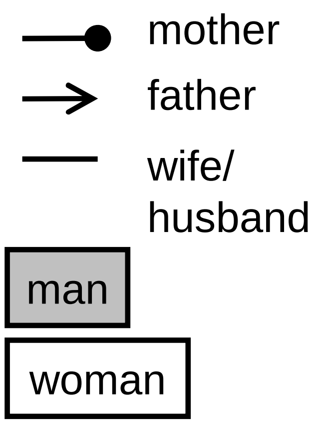
Namespace, Imports

Instance definition

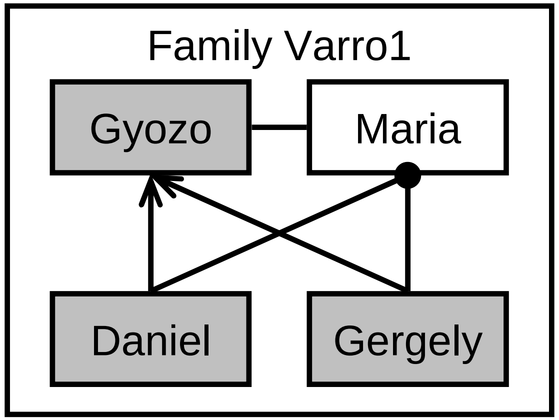
Explicit instance-of

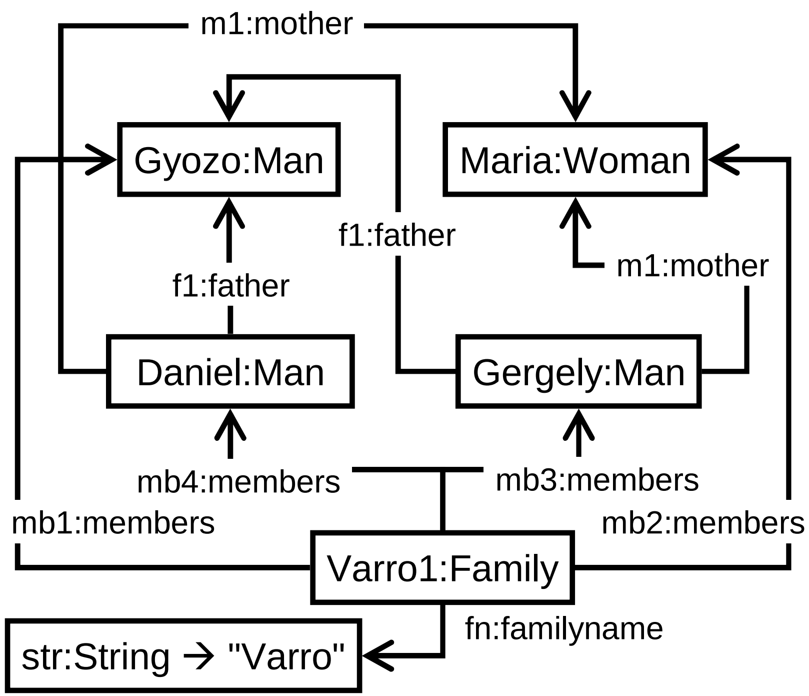
Value OO datatype

```
namespace people.models;
import people.metamodel;
import datatypes;
 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');
     family.familyname(fn, 'Varro1', str);
     String(str) -> "Varro";
}
```



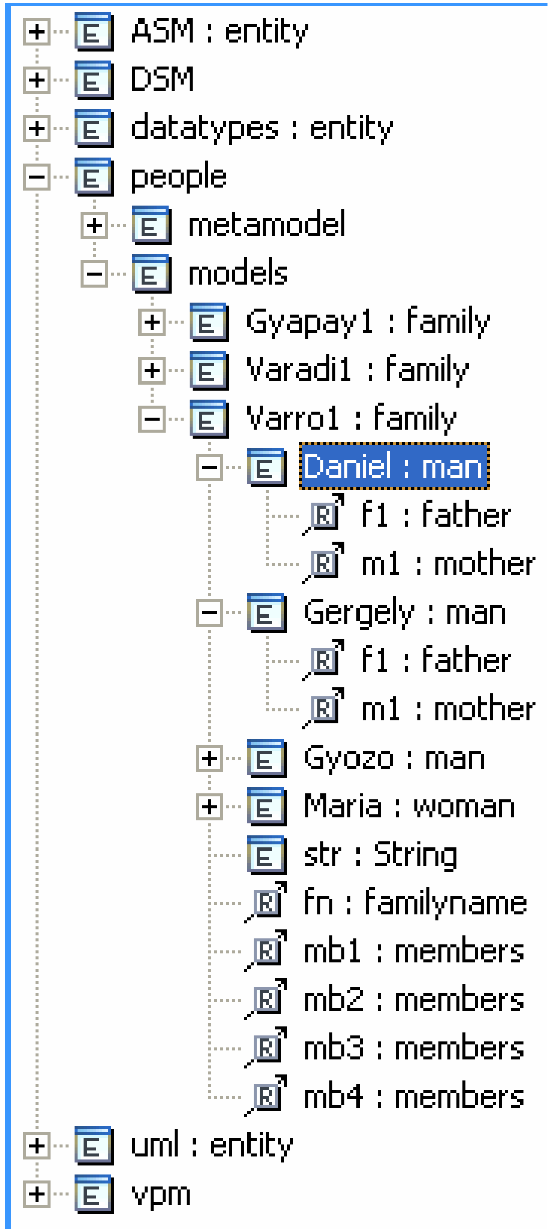
Graphical notation





Graph view

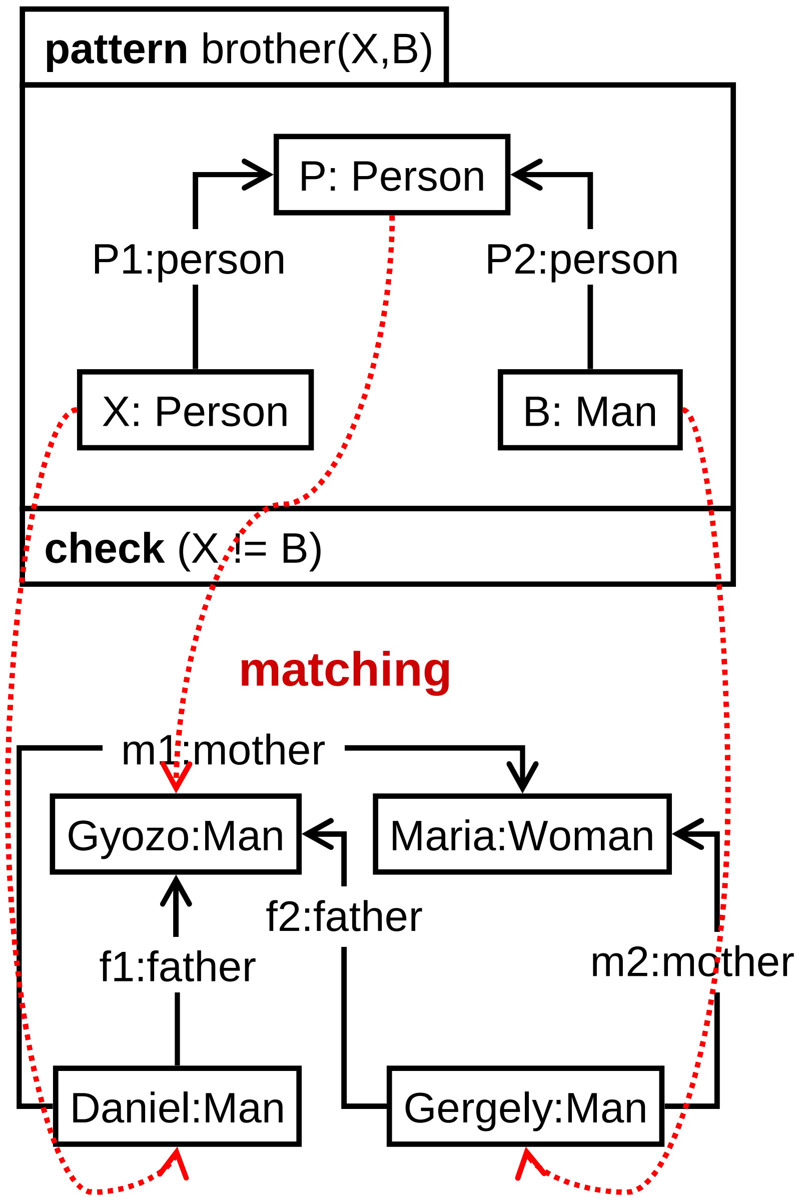
Containment View



The VIATRA2 Approach

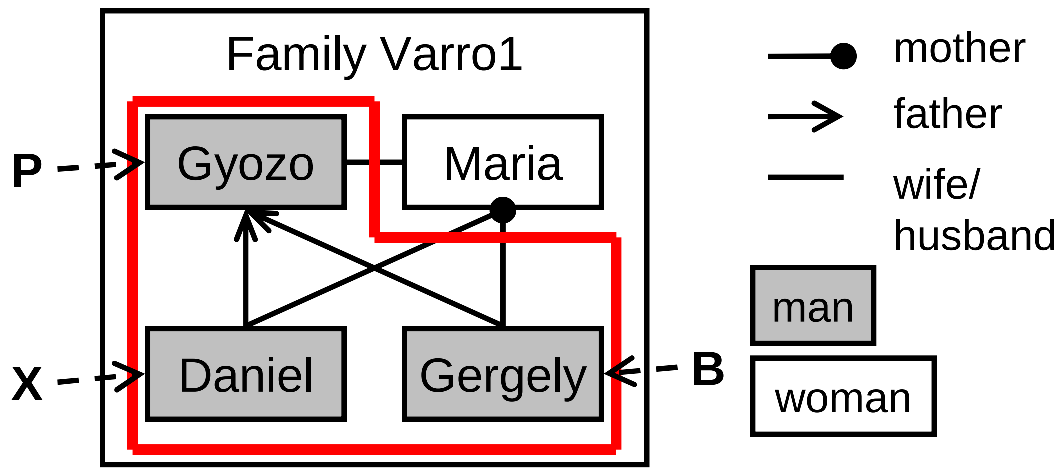
- Model management:
  - **Model space**: Unified, global view of models, metamodels and transformations
    - Hierarchical graph model
    - Complex type hierarchy
    - Multilevel metamodeling
- Model manipulation and transformations: integration of two mathematically precise, **rule** and **pattern-based**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



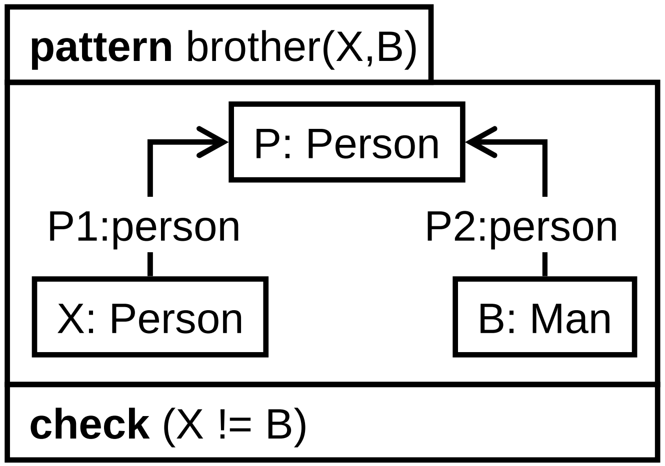
Instance Model

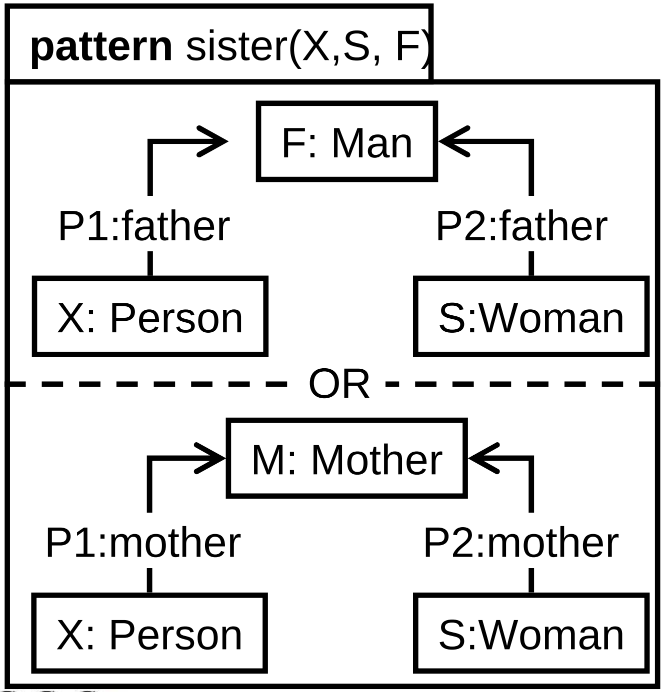
Graphical notation



- 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)



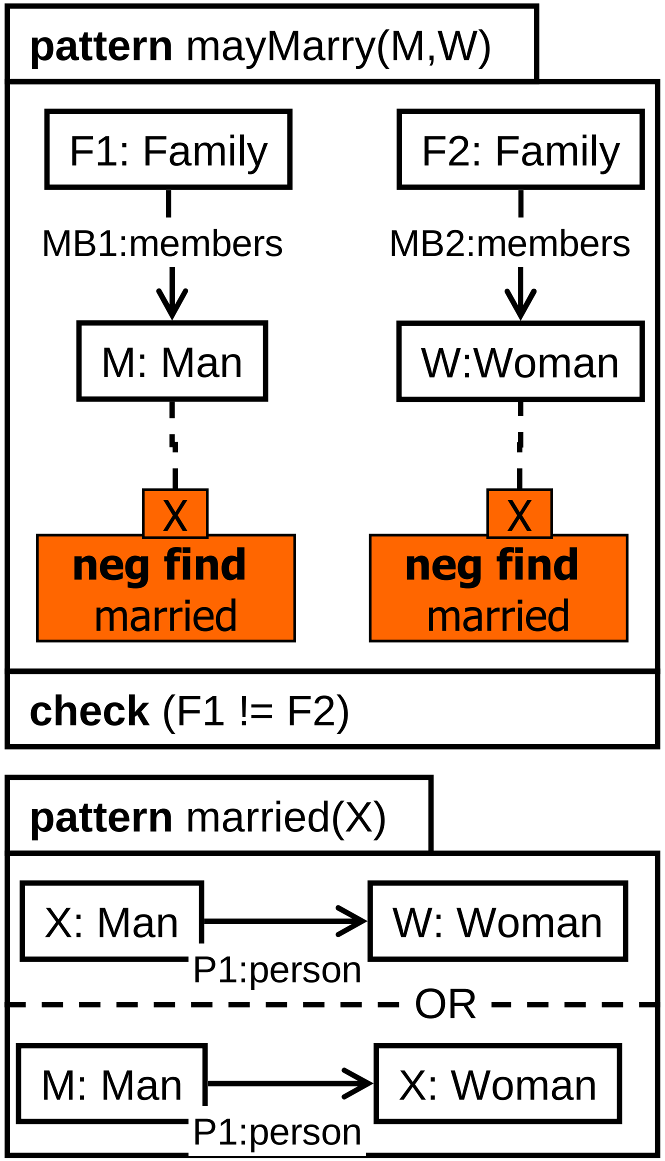



```
// B is a brother of X
pattern brother(X, B) =
{
    person(X);
    person.parent(P1, X, P);
    person(P);
    person.parent(P2, B, P);
    man(B);
    check (X != B)
}
```

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);
    woman(S) in W;
}
```

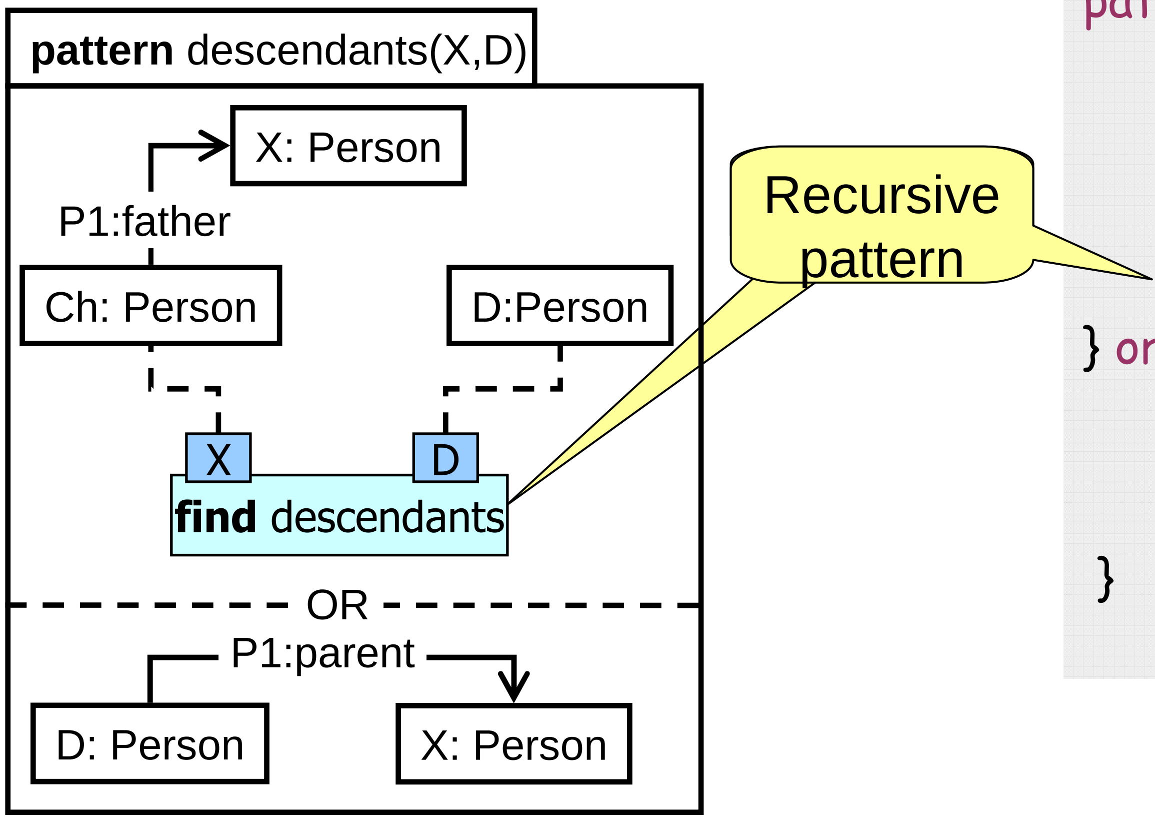
Negative patterns (VTCL)



Negative pattern

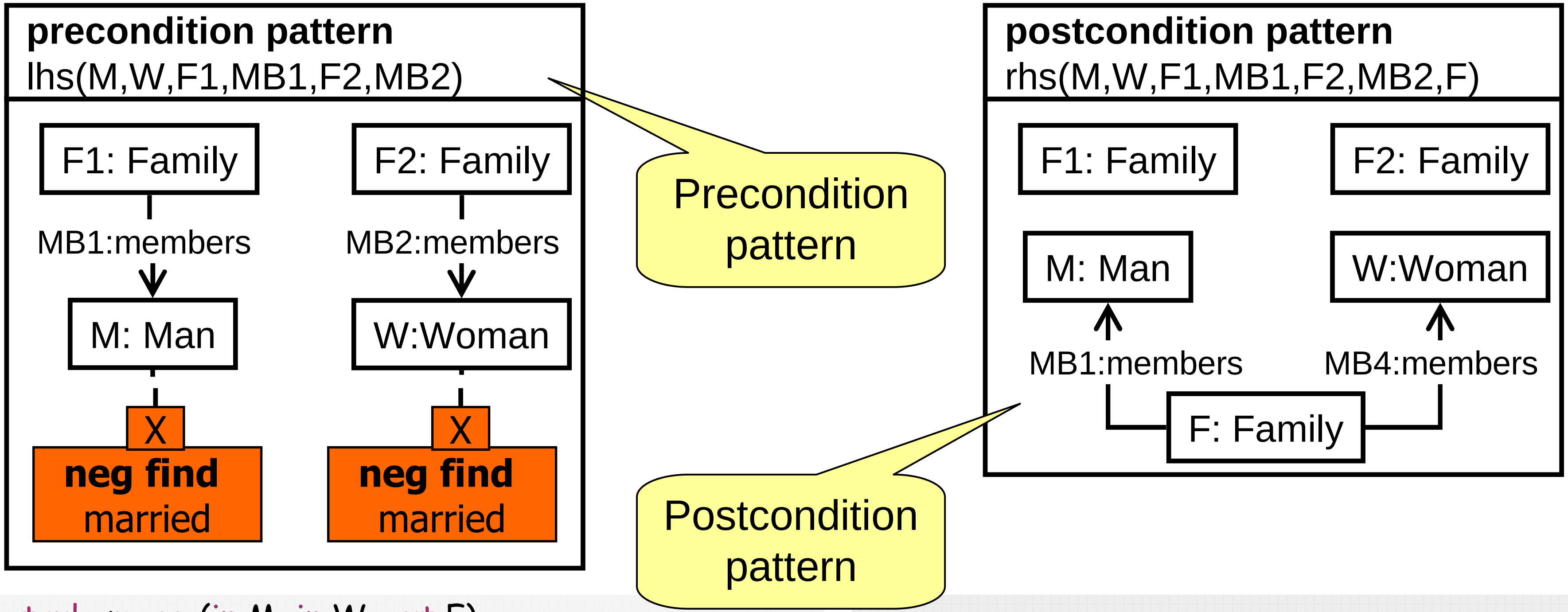
```
pattern mayMarry(X, D) = {
    man(M);
    family(F1);
    family.members(MB1, F1, M);
    woman(W);
    family(F2);
    family.members(MB2, F2, W);
    neg find married(M);
    neg find married(W);
    check (F1 != F2);
}

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

Recursive patterns (VTCL)




```
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);
 }
```

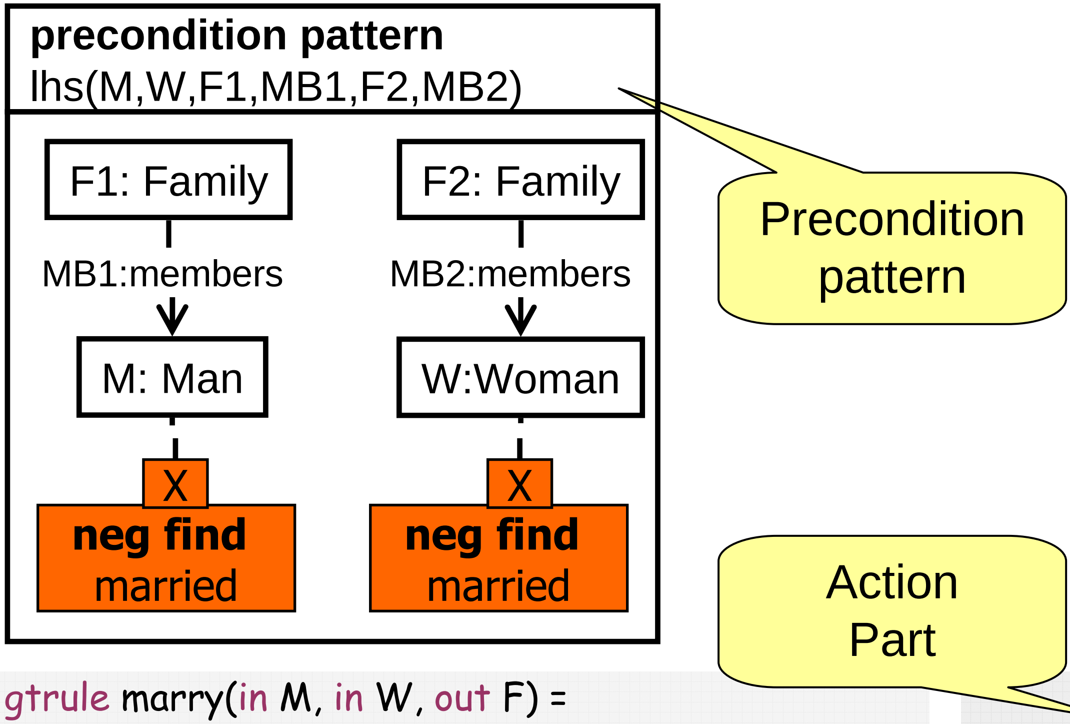
Graph transformation rules (VTCL)




```
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);
    neg find married(W);
}
```

```
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);
}
```

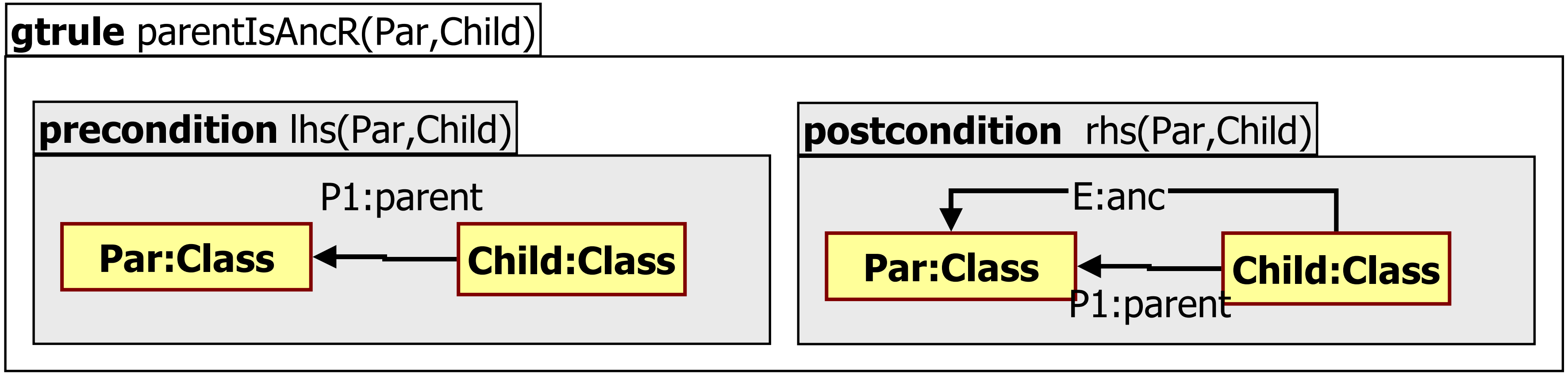
Graph transformation rules (VTCL)

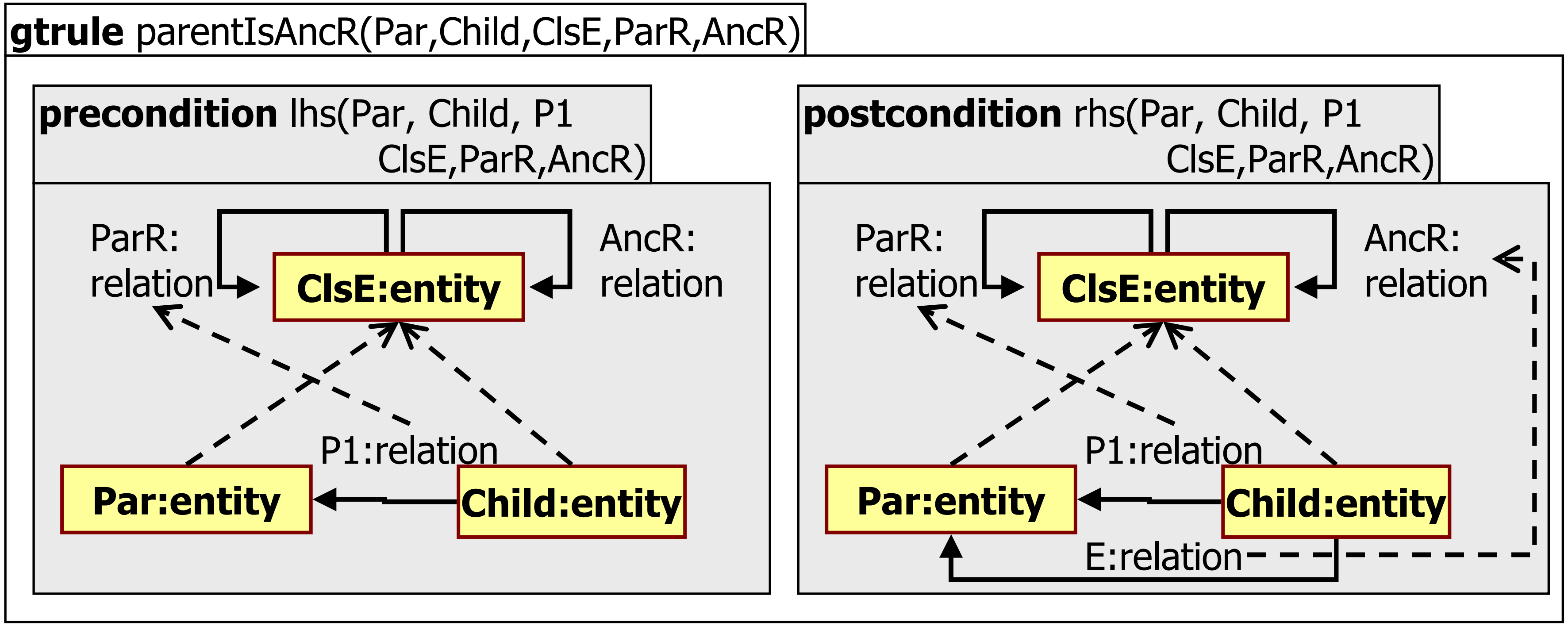



```
gtrule marry(in M, in W, out F) =
precondition pattern
lhs(M,W,F1,MB1,F2,MB2) = {
      family(F1);
      family.members(MB1, F1, M);
      man(M);
      family(F2);
      family.members(MB2, F2, W);
      woman(W);
      neg find married(M);
      neg find married(W);
}
```

```
action
  {
      delete(MB1);
      delete(MB2);
      new(family(F));
      new(family.members(MB3, F, M));
      new(family.members(MB4, F, W));
      new(man.wife(WF1, M, W));
  }
```

Generic GT rules (VTCL)





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
```

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

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

// Generated
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

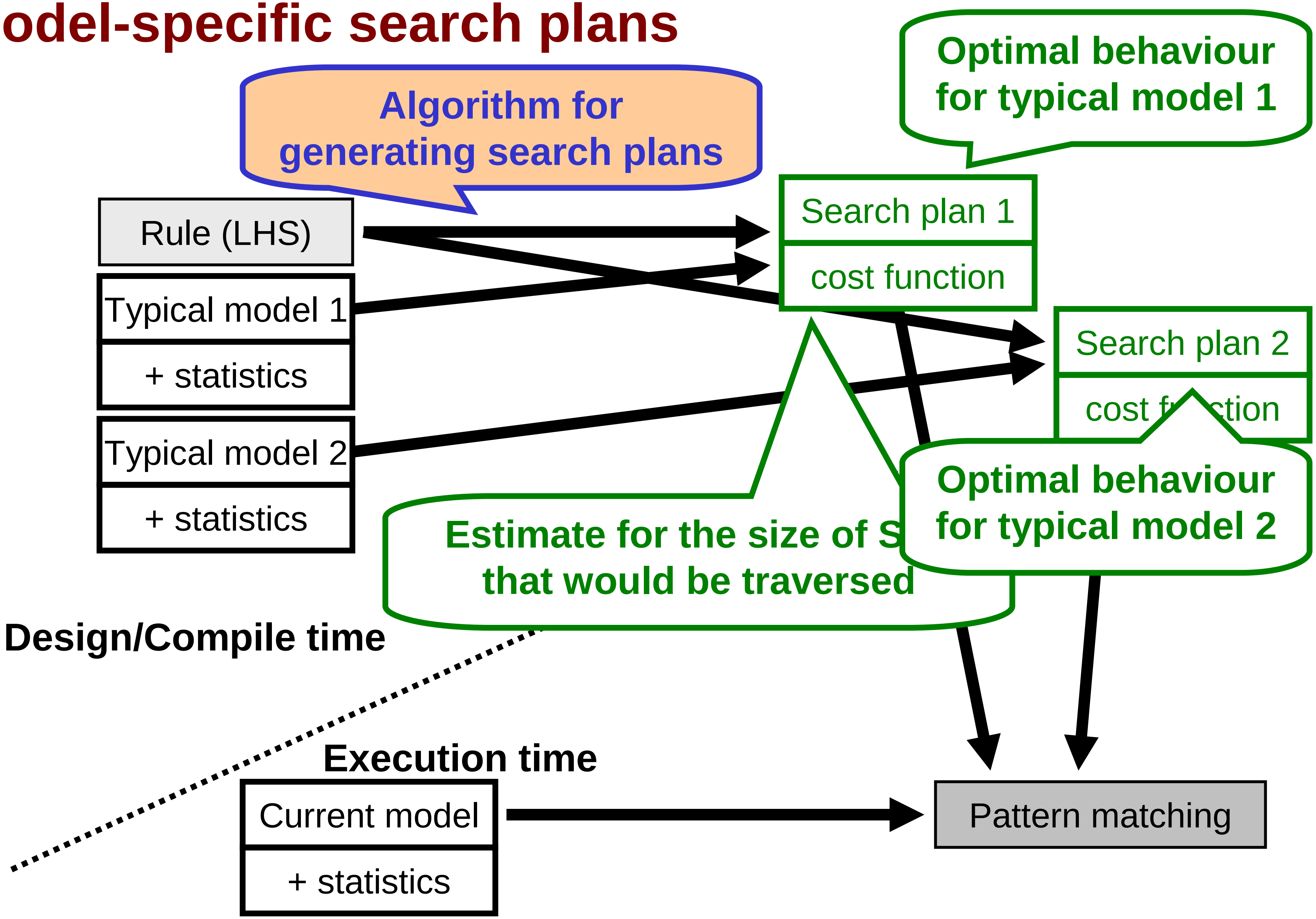
- Reusability
  - Pattern calls
  - Rule calls
- Traceability
  - Reference metamodels
  - Explicit storage of reference models
- External transformations
- Live transformations vs. Batch transformations

Transformation engine(s)

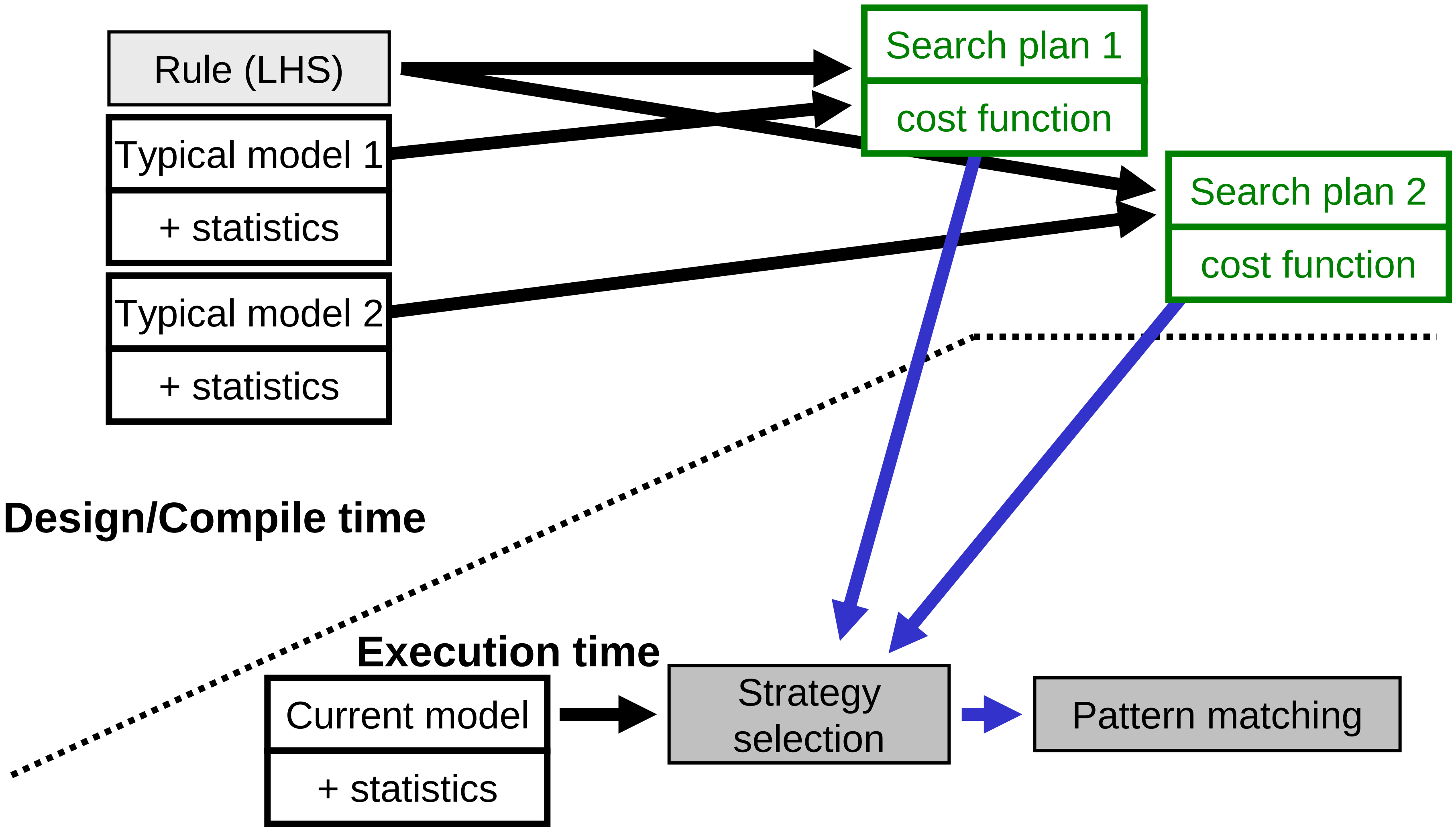
Why Not XSLT?

- 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
    - …

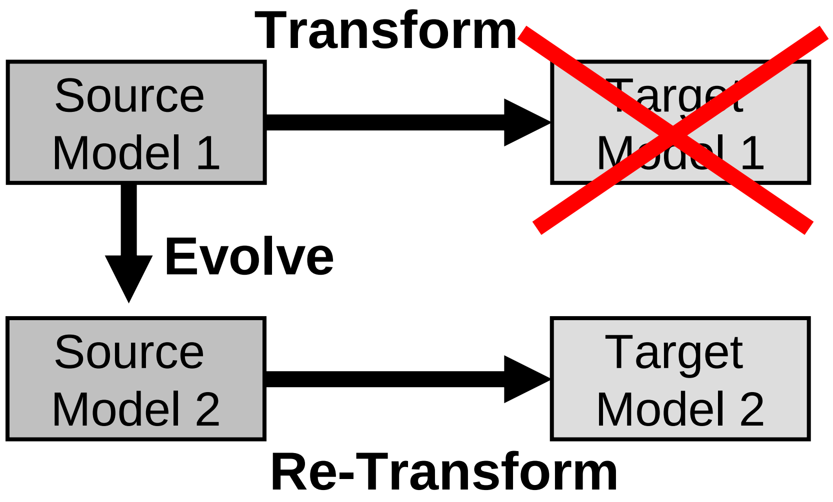
Model-specific search plans

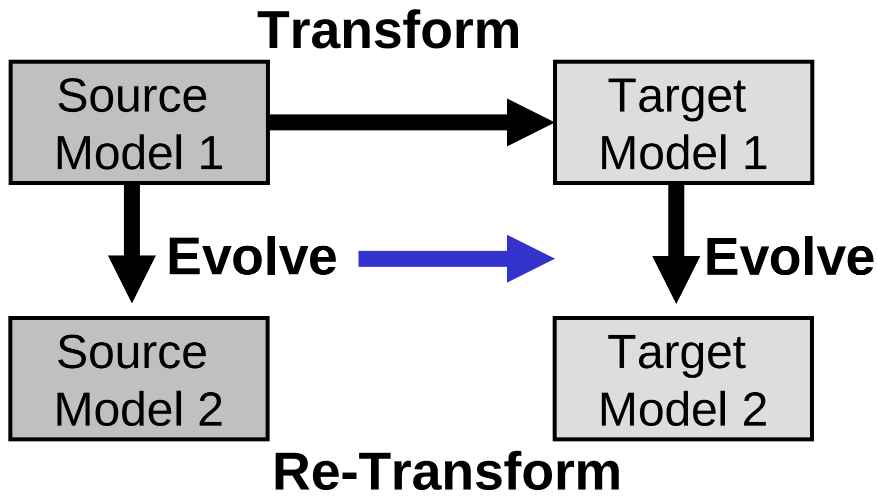


Adaptive PM approach



Model Synchronization Problem



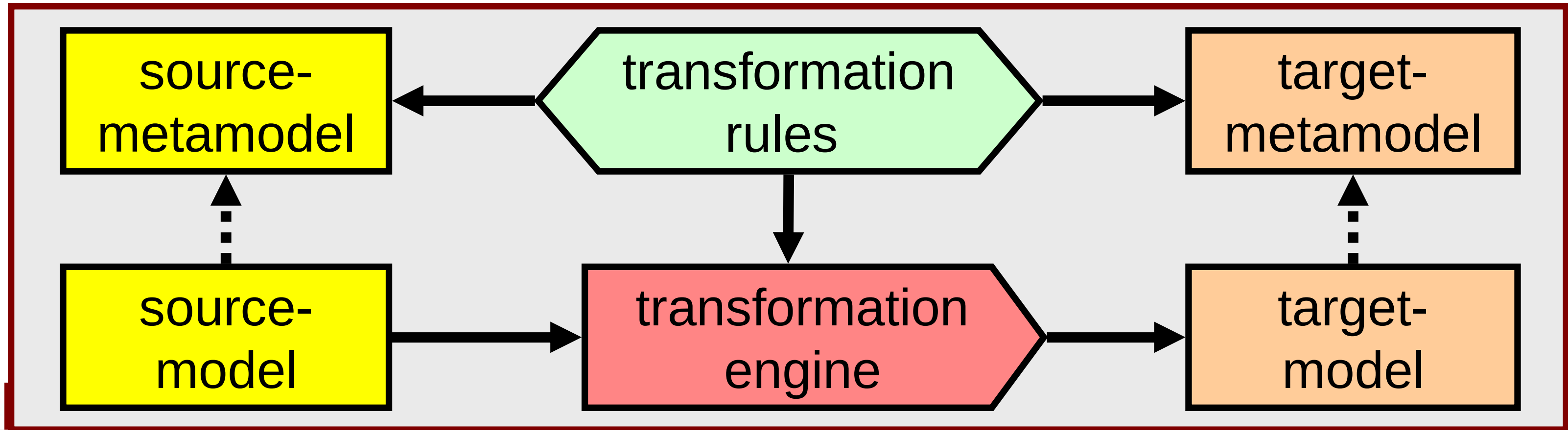


- Batch Approach
  - Restart tranformation from scratch each time
- Incremental Approach
  - Store partial matches of patterns
  - Propagate model changes
  - Apply xform for novel model parts
- Implemented by adapting RETE networks

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



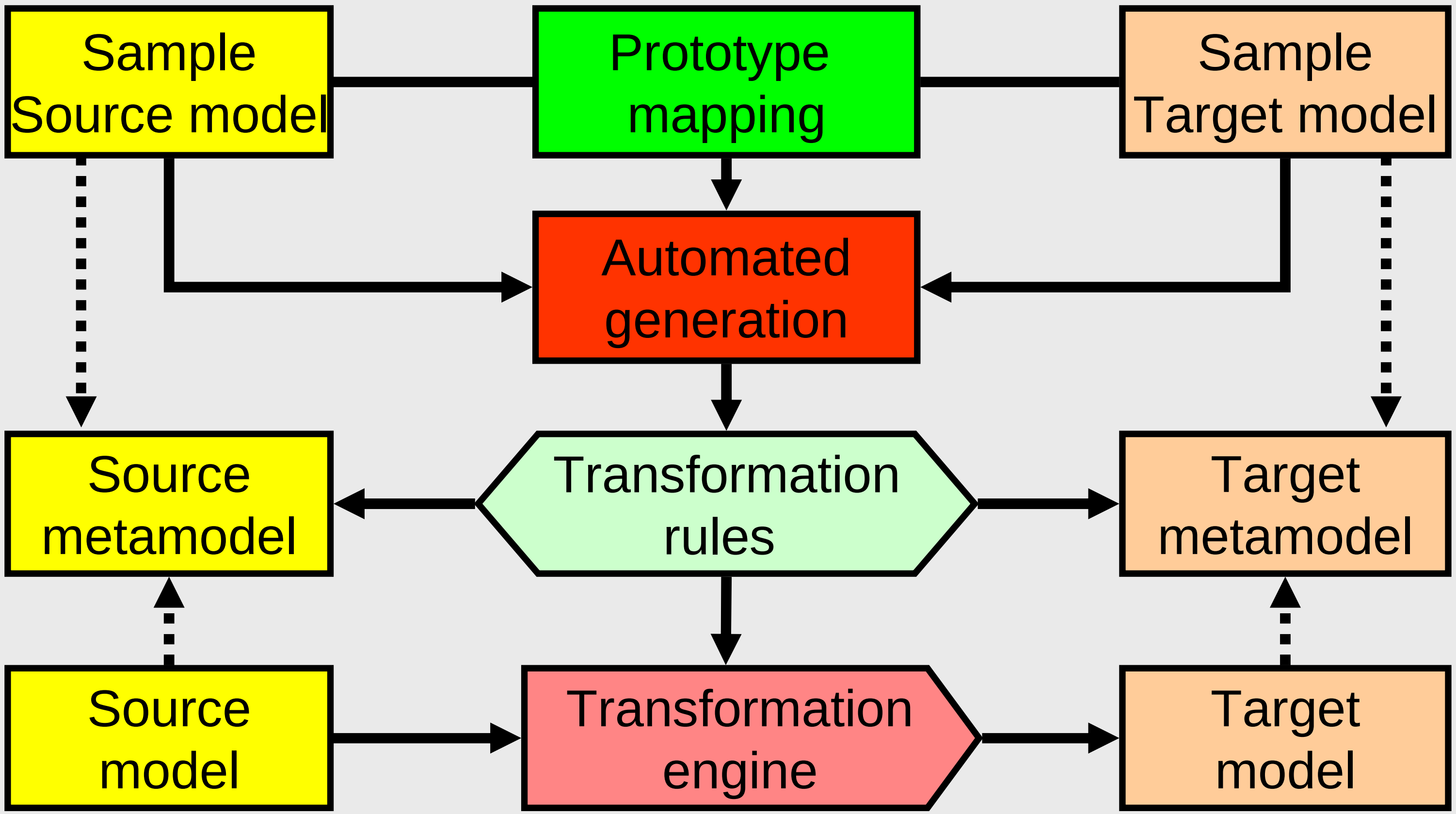
Model Transformation by Example

- Start with prototy
  - sample source a
- Automatically ge

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, TTTech, Fiat, Profactor, Hella, Liebherr, Thales, Esterel
  - 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

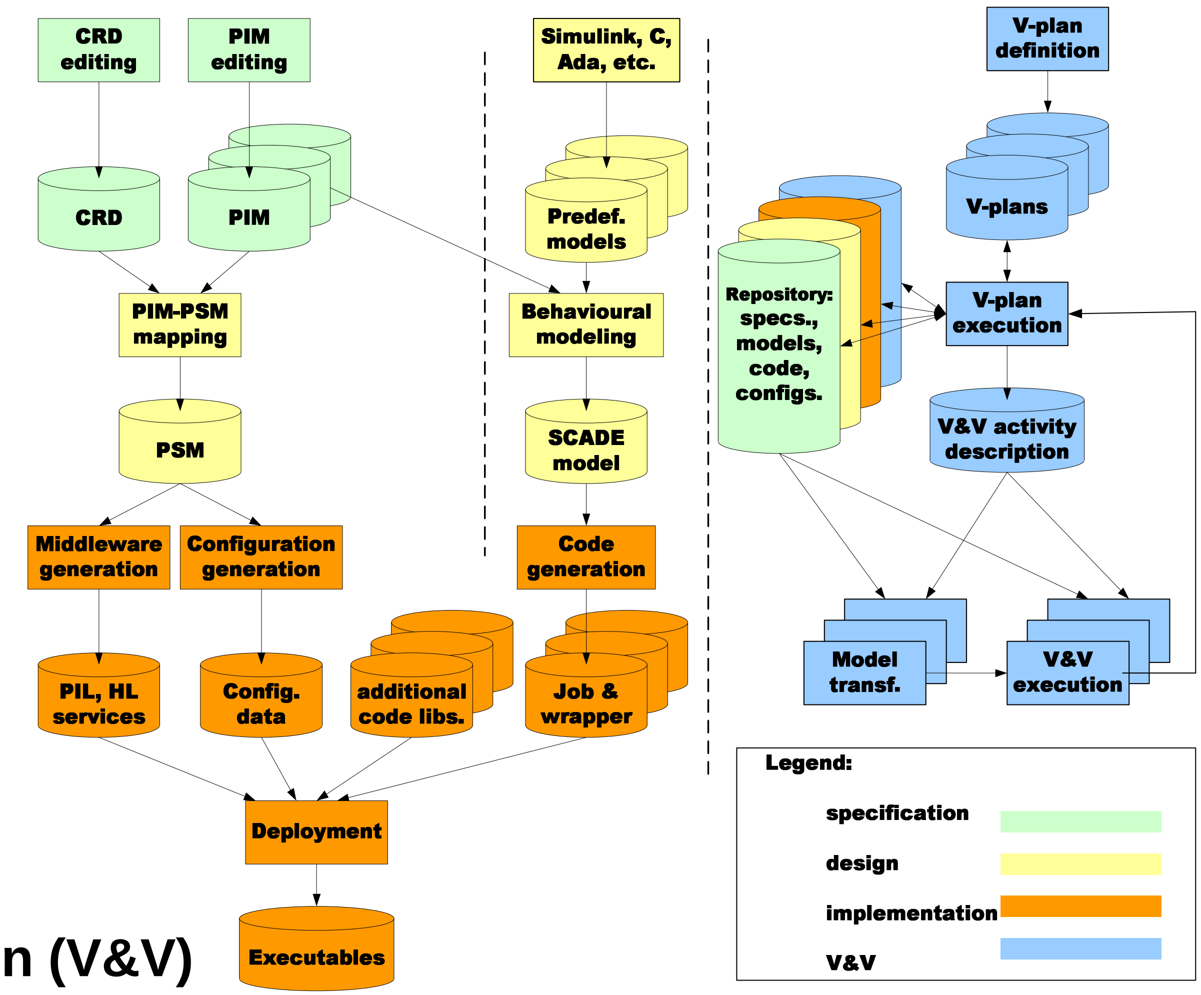
  ⇒ from *federated* to *integrated* systems

- DECOS tool chain
  - Supports
    - Specification
    - Design
    - Implementation
    - V&V of embedded systems

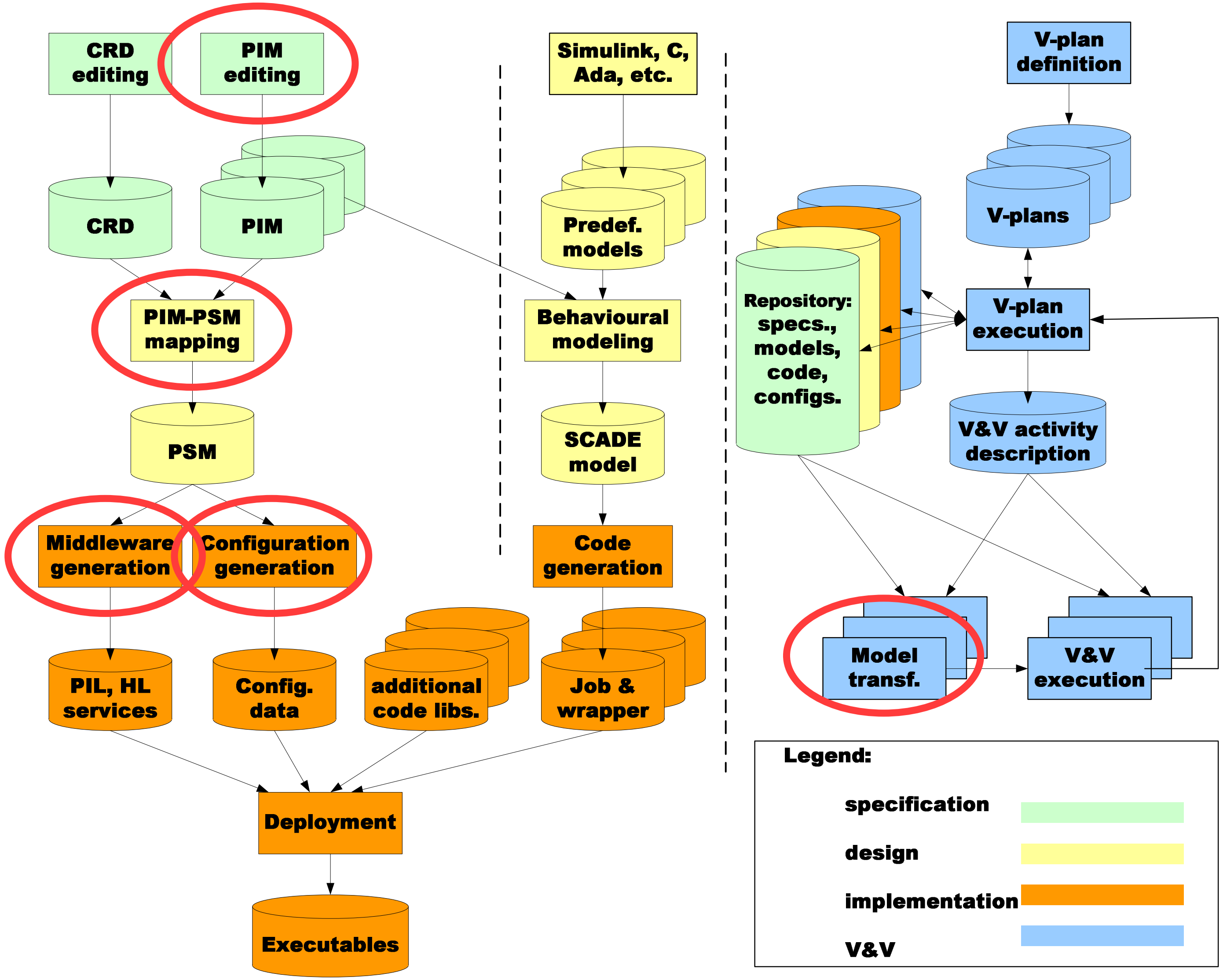
Tool Chain: Model-Based Integrated Development Support

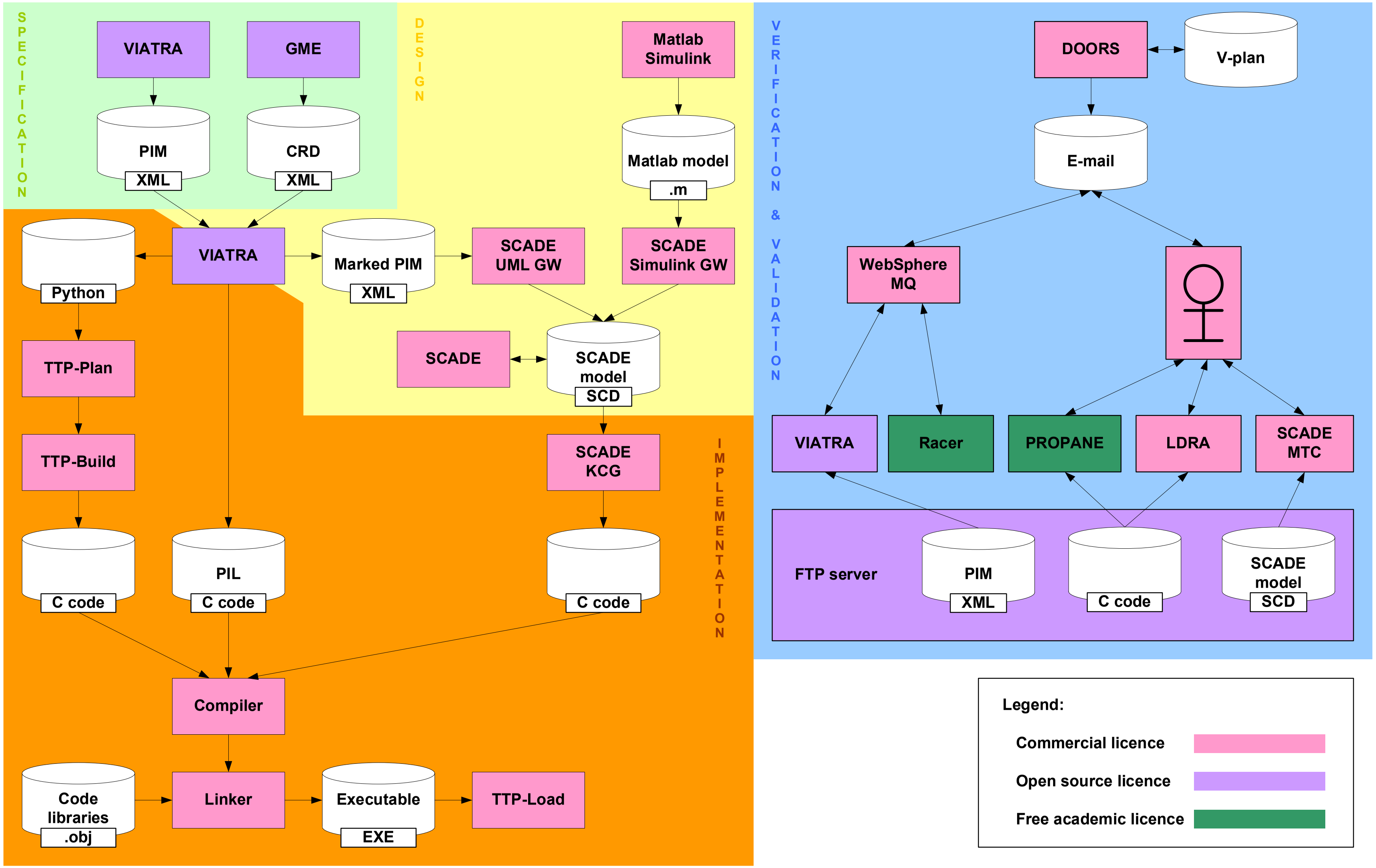
"From Requirements To Deployment"

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




SPECIFICATION
DESIGN
IMPLEMENTATION
VERIFICATION & VALIDATION
VIATRA
GME
PIM
XML
CRD
XML
Matlab Simulink
Matlab model
.m
Python
VIATRA
Marked PIM
XML
SCADE UML GW
SCADE Simulink GW
TTP-Plan
SCADE
SCADE model
SCD
TTP-Build
SCADE KCG
C code
PIL
C code
C code
Compiler
Code libraries
.obj
Linker
Executable
EXE
TTP-Load
DOORS
V-plan
E-mail
WebSphere MQ
VIATRA
Racer
PROPANE
LDRA
SCADE MTC
FTP server
PIM
XML
C code
SCADE model
SCD
Legend:
Commercial licence
Open source licence
Free academic licence

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
  - I. Ráth, A. Pataricza, L. Gönczy, A. Balogh, G. Varró, B. Polgár, Á. Horváth, D. Tóth, G. Bergmann, A. Ökrös, Zs. Déri, Z. Balogh
  - And many more students