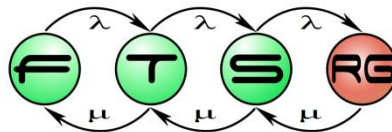


Model Queries

General Concepts
Graph Patterns
Incrementality
VIATRA QUERY

Model Driven Systems Development
Lecture 04



MOTIVATION

Motivation: early validation of design rules

SystemSignalGroup design rule (from AUTOSAR)

- A *SystemSignal* and its *SystemSignalGroup* must be mapped to the same IPDU
- Challenge: find violations

AUTOSAR:

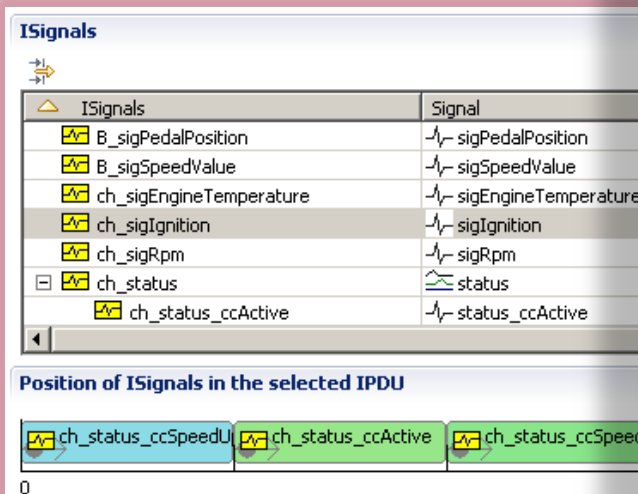
- standardized SW architecture of the automotive industry
- now supported by modern modeling tools

Design Rule/Well-formedness constraint:

- each valid car architecture needs to respect
- designers are immediately notified if violated

Challenge:

- >500 design rules in AUTOSAR tools
- >1 million elements in AUTOSAR models
- models constantly edited by designers



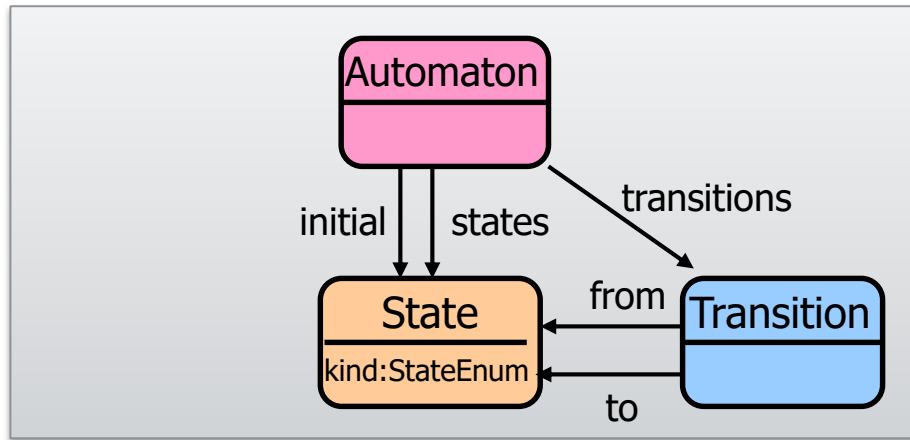
Model tree System editor: demoSystem

Element description Problems

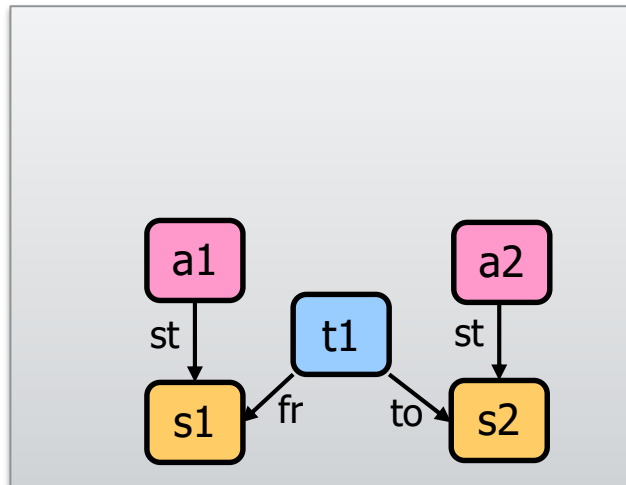
0 errors, 2 warnings, 0 others

Description	Resource	Path	Location	Type
✖ Errors (4 items)				
✖ ISignal of a grouped System Signal should be mapped to an IPdu along with the ISignal of the System Signal Group	demo_swc.xml	/alma	/rootP...	AUTOSAR P...
✖ ISignal of a grouped System Signal should be mapped to an IPdu along with the ISignal of the System Signal Group	demo_swc.xml	/alma	/rootP...	AUTOSAR P...
✖ ISignal of a grouped System Signal should be mapped to an IPdu along with the ISignal of the System Signal Group	demo_swc.xml	/alma	/rootP...	AUTOSAR P...
✖ Reference IPduTimingSpecification has invalid multiplicity! (Must be in: [1, 1])	demo_swc.xml	/alma	/rootP...	AUTOSAR P...

A simple example



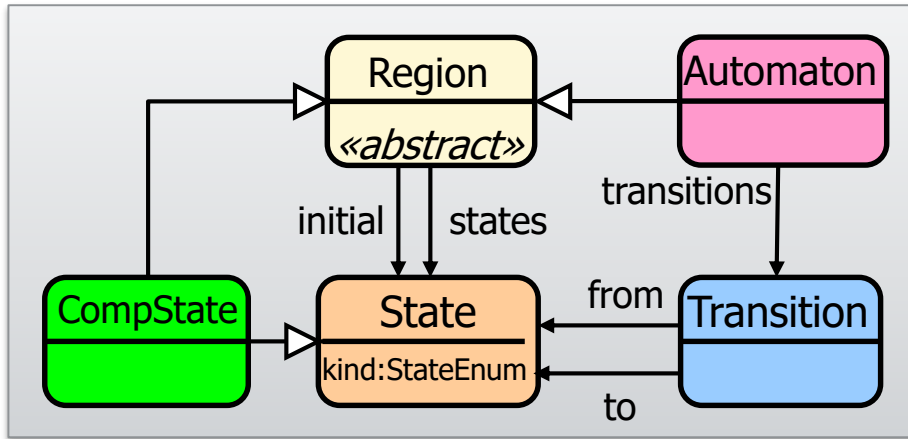
Metamodel



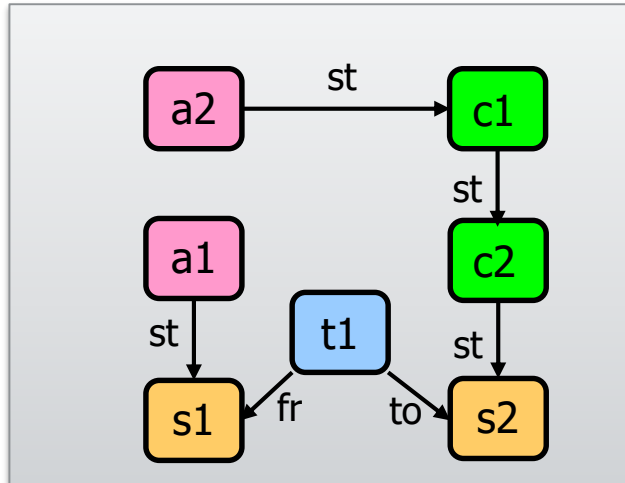
Violation example

- Well-formedness constraint:
 - Transition source & target states must be owned by same automaton
- Goal: to find violations...
 - A violation is a *Transition*, whose „*from*” link points to a *State* *x*, and „*to*” link points to a *State* *y*, where the automaton of *x* is not the automaton of *y*
 - How to check this?

A more complex example



Metamodel

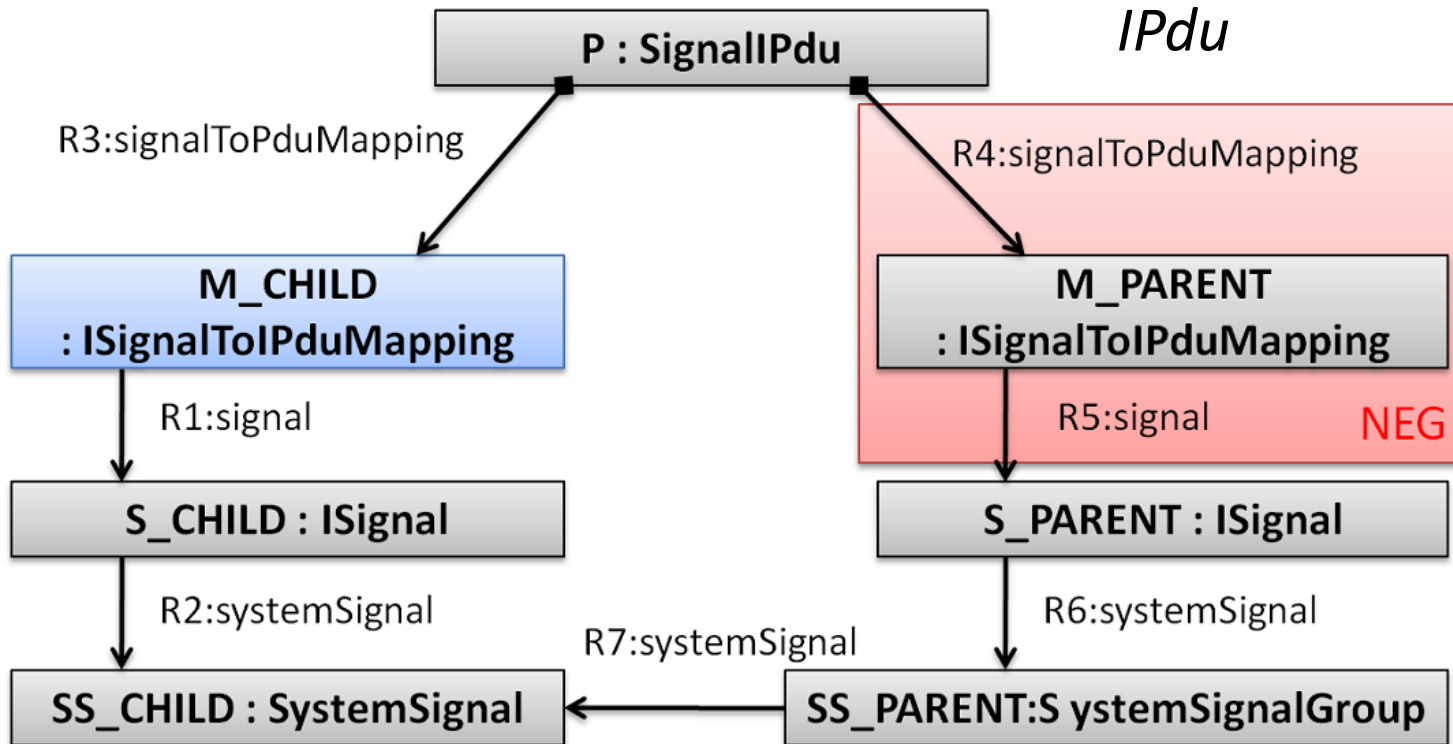


Violation example

- Well-formedness constraint:
 - Transition source & target states must be owned by regions belonging to same automaton
- Goal: to find violations...
 - A violation is a *Transition*, whose „from” link points to a *State* x, and „to” link points to a *State* y, where...
 - How to check this?

Another complex example

- Well-formedness constraint:
 - A *SystemSignal* and its group must be in the same *IPdu*



Programmatic traversal vs. queries

- Goal: find constraint violations in model
 - Traverse model in general-purpose language

```
for (Automaton automaton : automatons) {  
    for (Transition transition : automaton.getTransitions()) {  
        State sourceState = transition.from;  
        // which automaton defines this state?  
        Automaton sourceAutomaton = null;  
        for (Automaton candidate : automatons) {  
            if (candidate.getStates().contains(sourceState)) {  
                sourceAutomaton = candidate;  
                break;  
            }  
        }  
        // ... do the same for targetState, then  
        if (sourceAutomaton != targetAutomaton)  
            // report violation  
    }  
}
```

„simple
example”

(though much simpler when
bidirectional navigation is available)

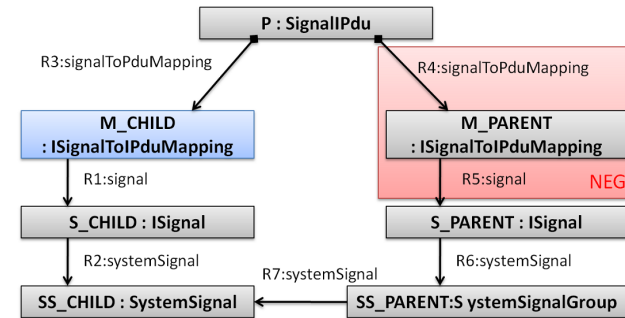
Programmatic traversal vs. queries

- Goal: find constraint violations in model
 - Traverse model in general-purpose language
 - Use a **Query DSL**
 - More concise
 - **Declarative** functional specification of the query
 - Freely interpreted by **query engine** (e.g. optimization)
 - Can be platform-independent
- Validation is just one use cases for **model queries**
 - Derived features
 - M2M/M2T Transformation, Simulation
 - ...

Query Language Styles

■ SQL-like (relational algebra)

- Example: EMF Query
- ☺ Good for attribute restrictions
- ☹ Not very concise for relationships (many joins)



■ Functional style

- Example: OCL
- Somewhat declarative

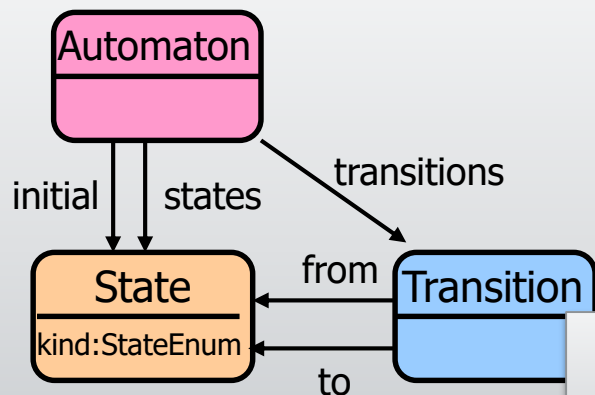
context Transition **inv**:

```
Automaton.allInstances()->forAll(a |  
    a.states->includes(self.from) =  
    a.states->includes(self.to)
```

■ Logic style

- Domain relational calculus / graph patterns / Datalog
- Even more declarative

Model Query as Logic



Metamodel

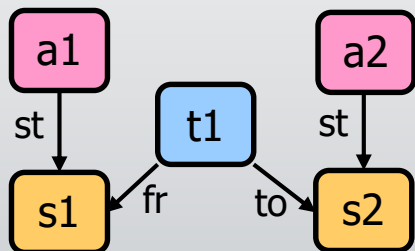
A violation is a *Transition*,
whose „from” link points to a *State* x
and „to” link points to a *State* y ,
where the automaton of x
is not the automaton of y

in formal logic
(Domain Relational
Calculus)

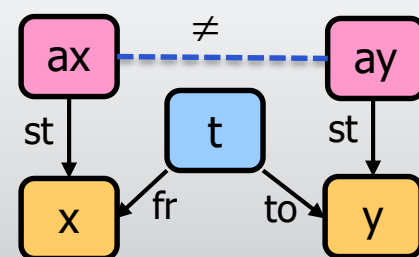
$$\{t \mid \text{Transition}(t) \wedge \exists x, y, ax, ay: \text{from}(t, x) \wedge \text{to}(t, y)$$

Datalog-like query languages

violates(t) :-
 Transition(t), from(t, x), to(t, y),
 states(ax, x), states(ay, y), ax \neq ay



Violation example



Graph pattern

MODEL QUERIES AND GRAPH PATTERN MATCHING

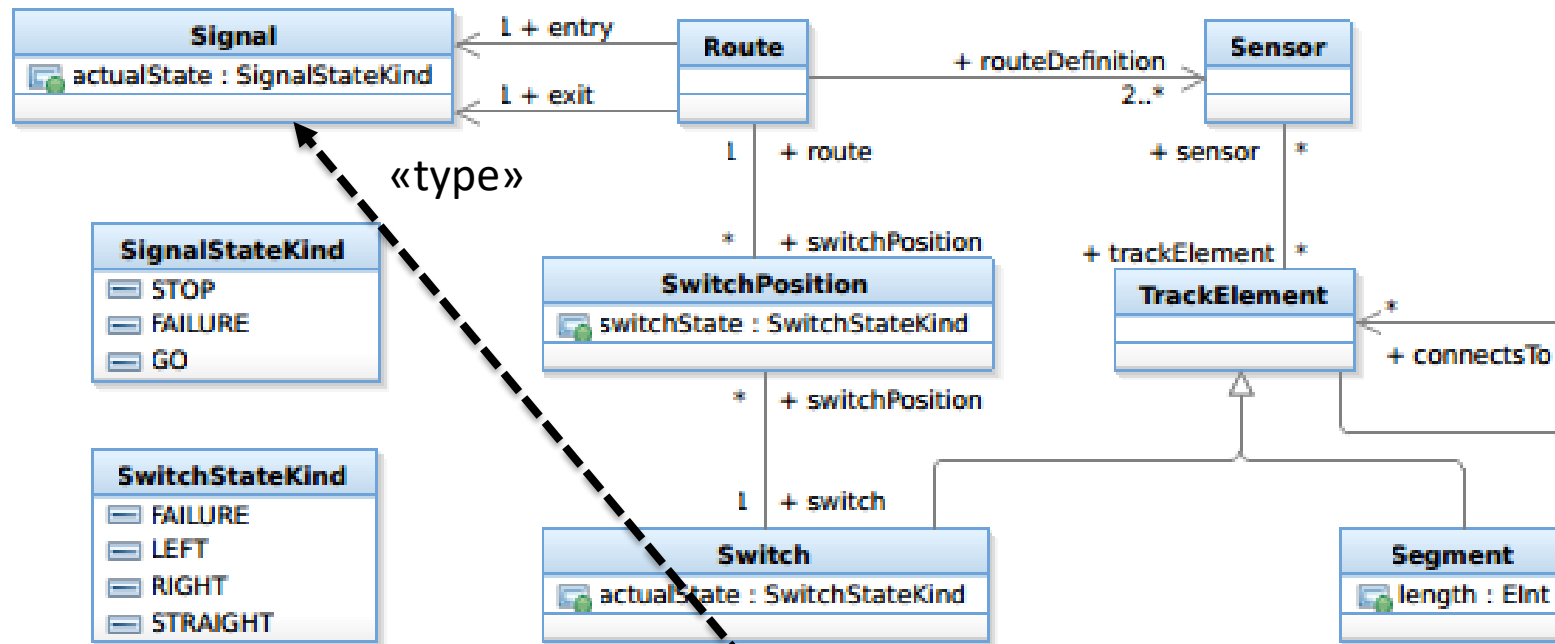
What is a model query?

- For a programmer:
 - A piece of code that searches for parts of the model
- For the scientist:
 - **Query** = set of constraints that have to be satisfied by (parts of) the (graph) model
 - **Result** = set of model element tuples that satisfy the constraints of the query
 - **Match** = bind constraint variables to model elements
- A query engine: Support
 - the definition&execution of model queries

Query(A,B) $\leftarrow \wedge \text{cond}_i(A_i, B_i)$

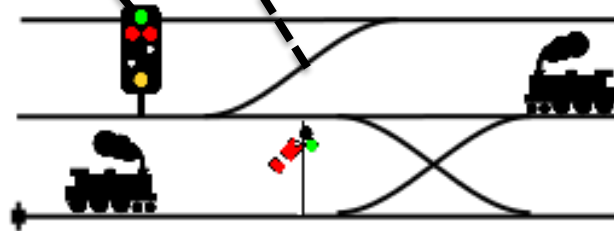
- all tuples of model elements a, b
- satisfying the query condition
- along the match $A=a$ and $B=b$
- parameters A,B can be input/ output

Motivating example

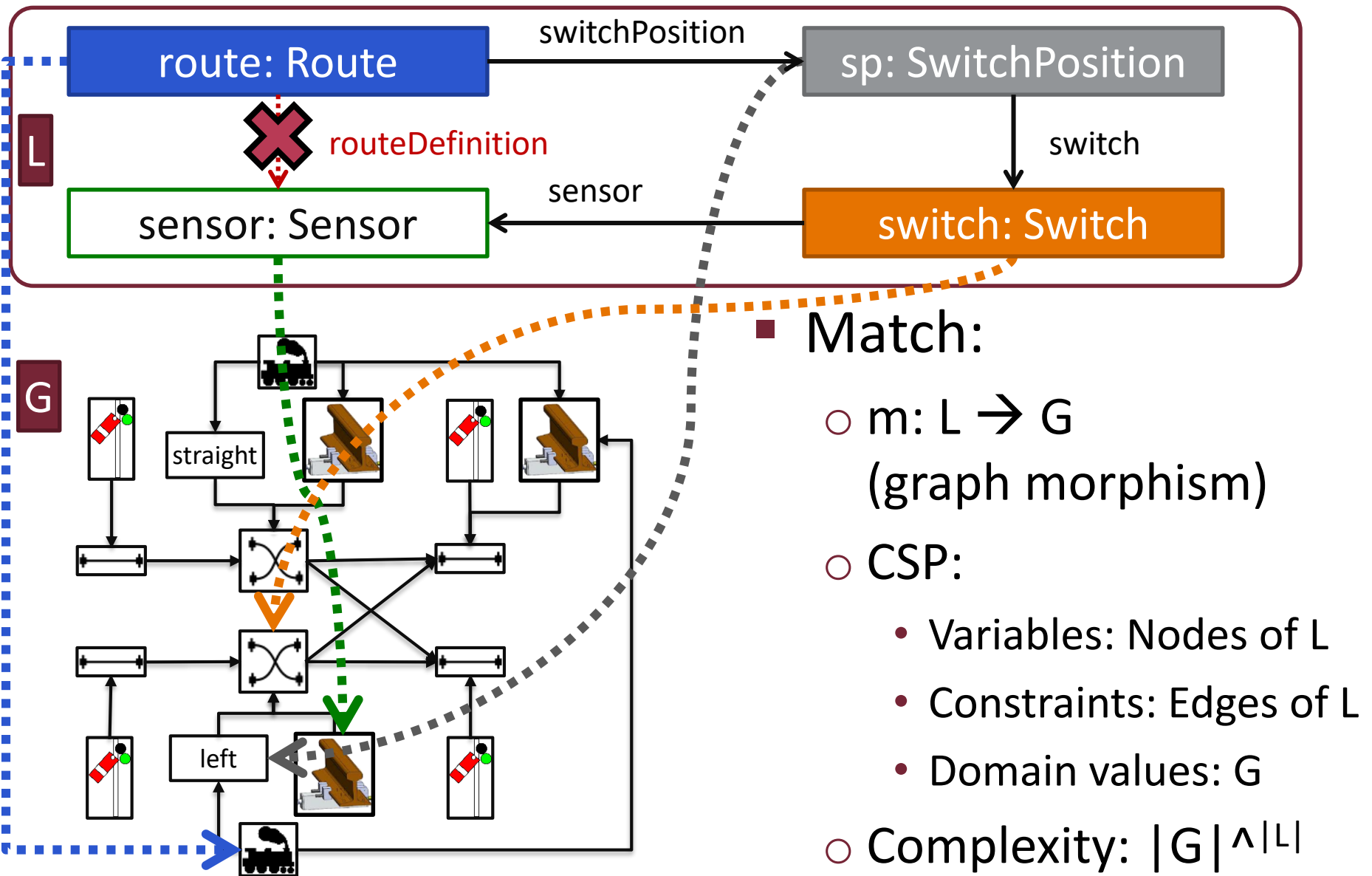


Meta-model

Model

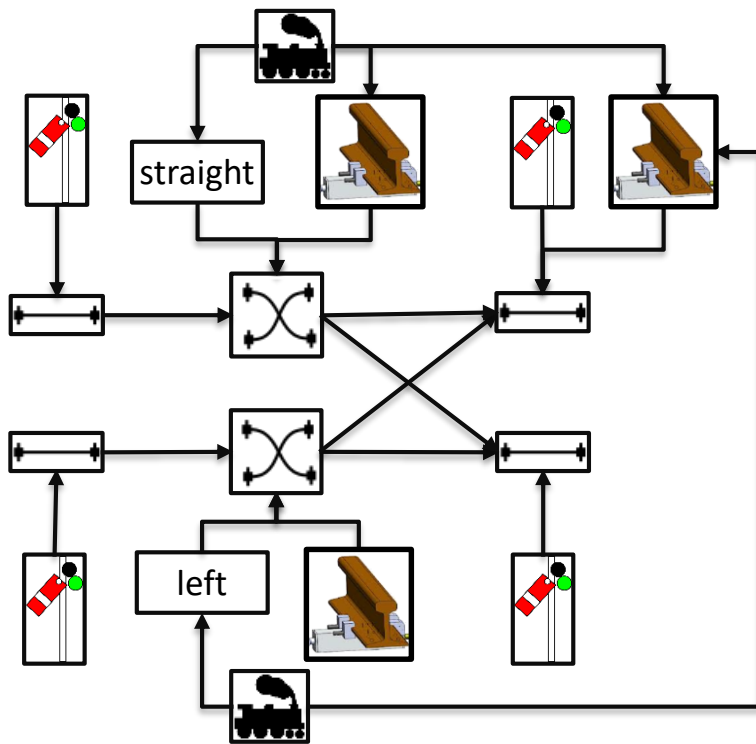
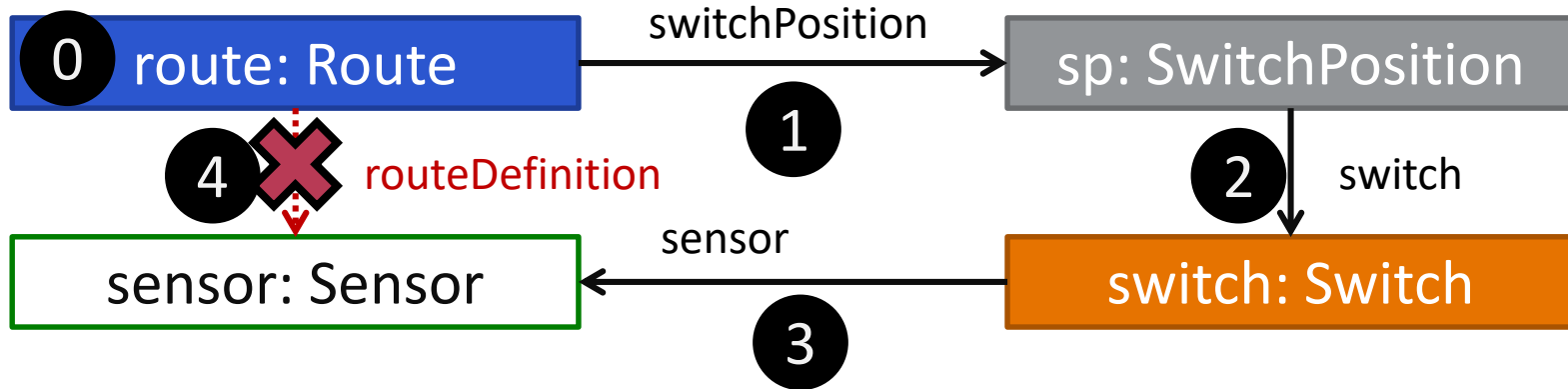


Graph Pattern Matching for Queries



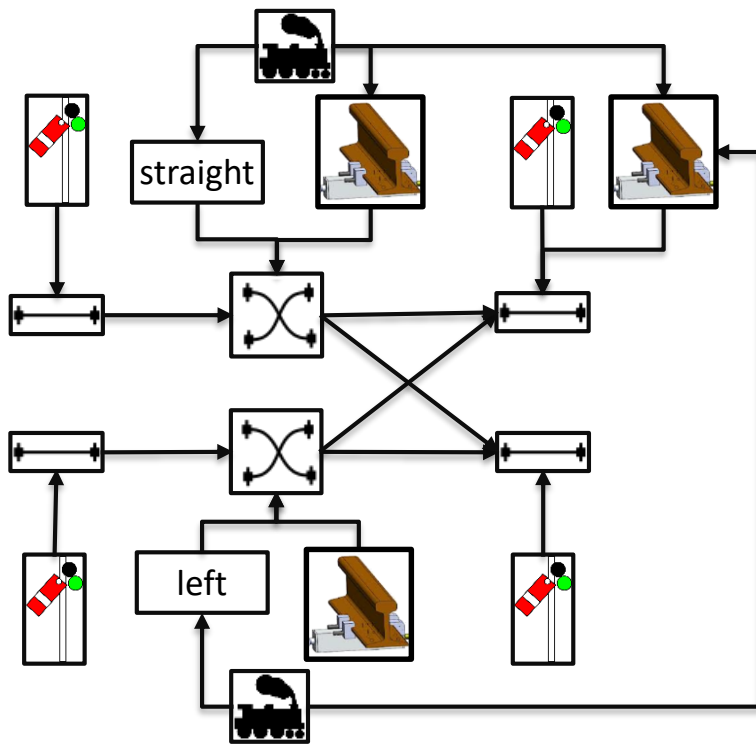
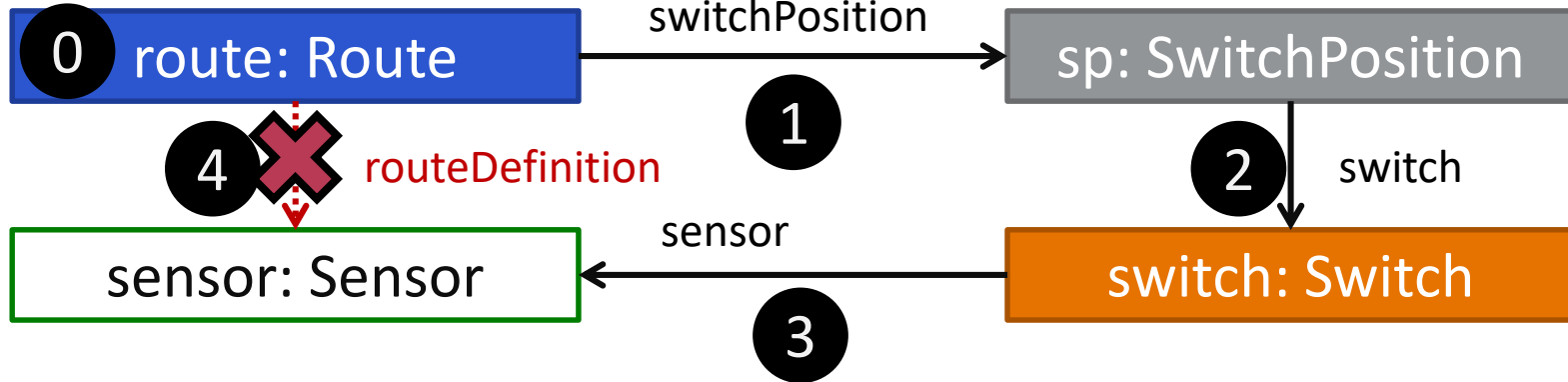
All sensors with a switch that belongs to a route must directly be linked to the same route.

Graph Pattern Matching (Local Search)

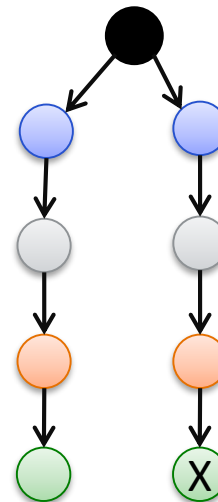


- Search Plan:
 - Select the first node to be matched
 - Define an ordering on graph pattern edges
- Search is restarted from scratch each time

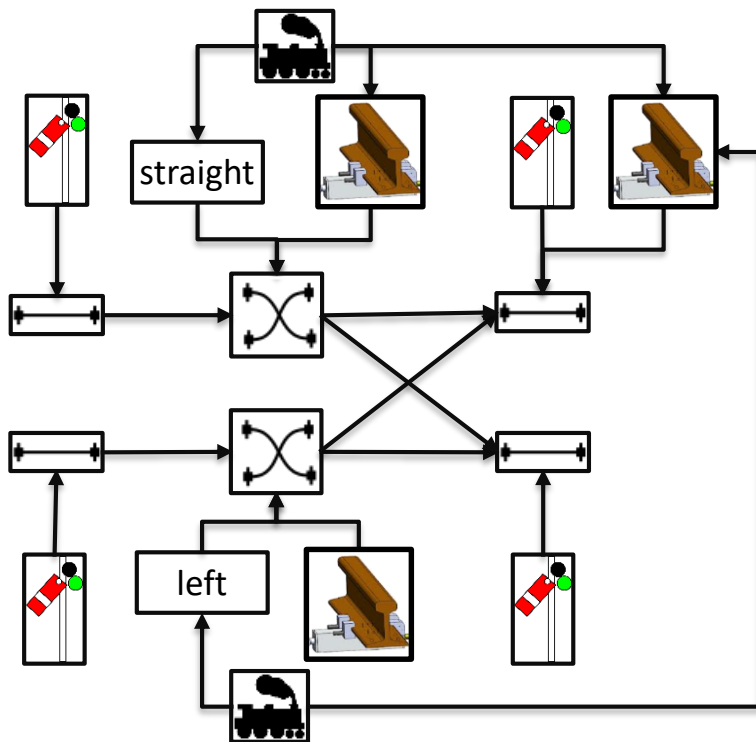
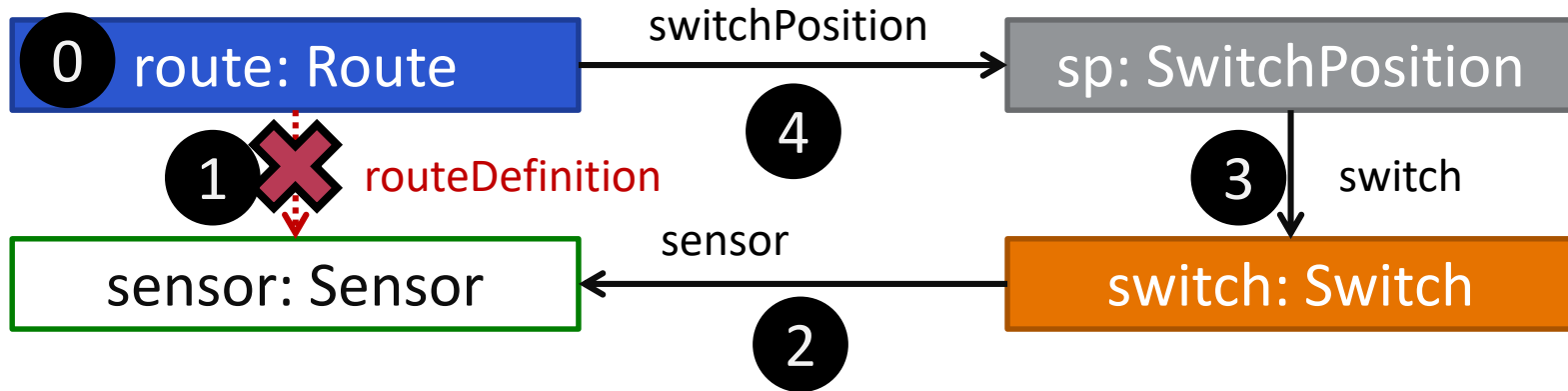
Graph Pattern Matching (Local Search)



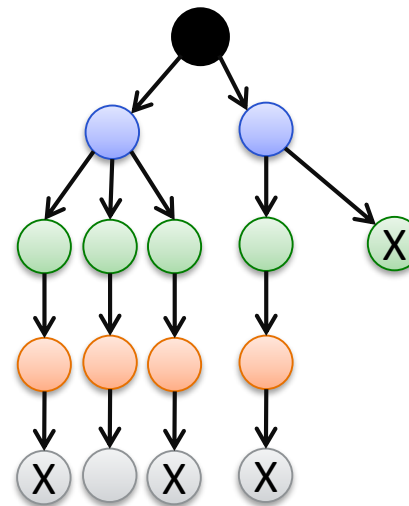
Search Tree:



Graph Pattern Matching (Local Search)



Alternate Search Tree:

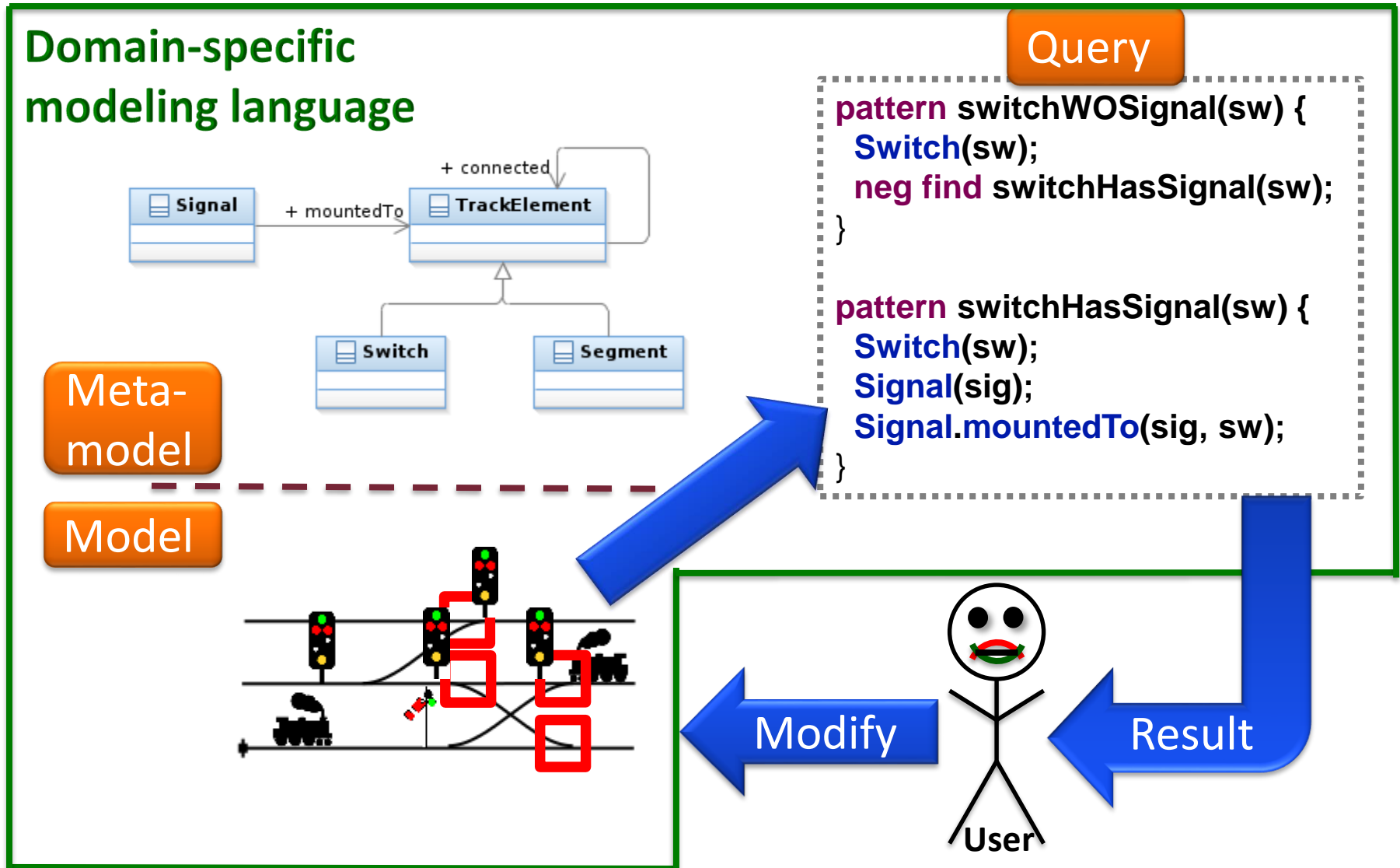


Local Search based PM

- Runtime depends on search plan
- Good search plan: narrow at root wide at leaves

INCREMENTALITY IN QUERIES AND TRANSFORMATIONS

Validation of Well-formedness Constraints



Model sizes in practice

- Models with 10M+ elements are common:
 - Car industry
 - Avionics
 - Source code analysis
- Models evolve and change continuously

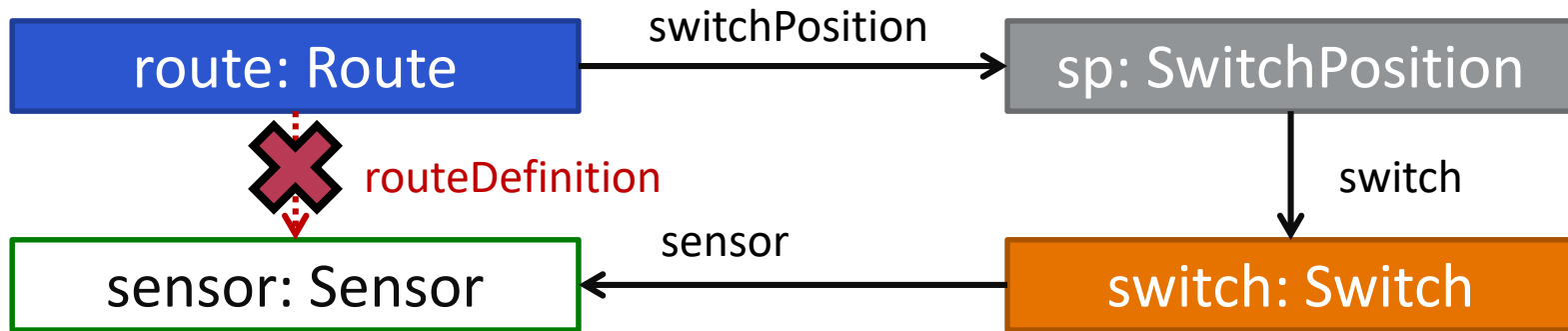
Application	Model size	Validation can take hours
System design models	10^8	
Sensor data	10^9	
Geospatial models	10^{12}	

Source: Markus Scheidgen, *How Big are Models – An Estimation*, 2012.

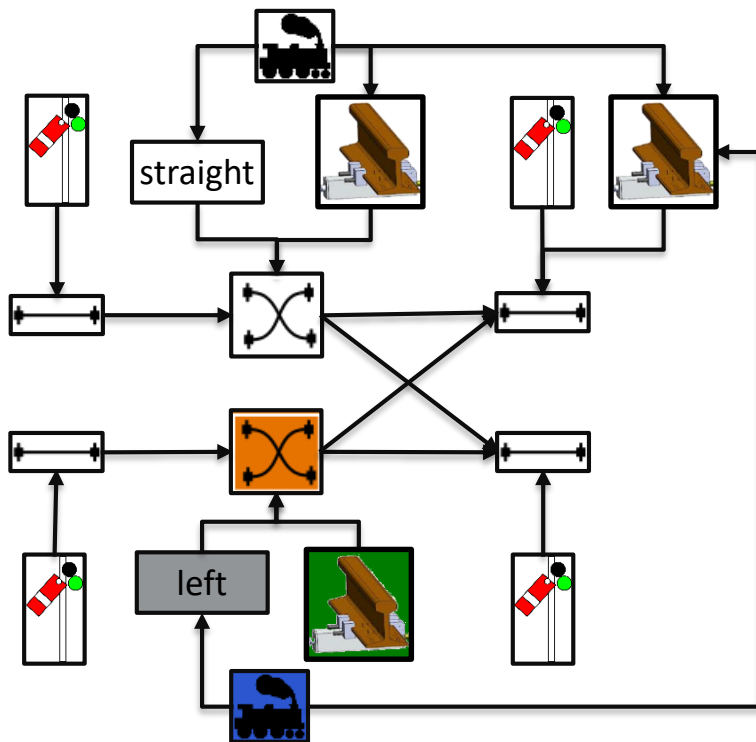
Performance of query evaluation

- Query performance = Execution time as a function of
 - Query complexity
 - Model size
 - Result set size
- Motivation for incrementality
 - Don't forget previously computed results!
 - Models changes are usually small, yet up-to-date query results are needed all the time.
 - Incremental evaluation is an essential, but not a well supported feature.

Incremental Graph Pattern Matching



route	sp	switch	sensor
r1	sp1	sw1	



■ Main idea: More space, less time

- Cache matches of patterns
- Instantly retrieve match (if valid)
- Update caches upon model changes
- Notify about relevant changes

■ Approaches:

- TREAT, LEAPS, RETE, ...
- Tools: VIATRA, GROOVE, MoTE, TCore

Batch vs. Live Query Scenarios

■ Batch query

(pull / request-driven):

1. Designer selects a query
2. One/All matches are calculated
3. Action is applied on one/all matches
4. All Steps 1-3 are redone if model changes

- Query results obtained upon designer demand

■ Live query

(push / event-driven):

1. Model is loaded
 2. Queries loaded
 3. Calculate full match set
 4. Model is changed
 5. Iterate Steps 3 and 4 until system is stopped
- Query results are always available for designer

VIATRA Query: An Open Source Eclipse Project

- **Declarative graph query language**
 - Transitive closure, Negative cond., etc.
 - Compositional, reusable

Definition



- **Incremental evaluation**
 - Cache result set
 - Maintain incrementally upon model change

Execution



- Derived features,
- On-the-fly validation
- View generation,
- Works out-of-the-box with EMF applications

Features

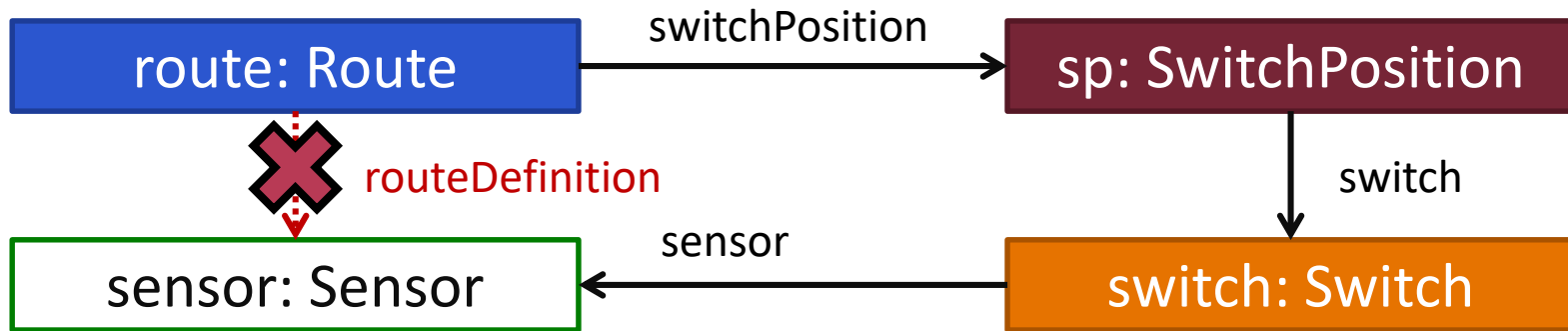


<http://eclipse.org/viatra>

Formerly known as
EMF-INCQUERY

GRAPH MODEL QUERIES: THE LANGUAGE

The VIATRA QUERY Language (VQL)



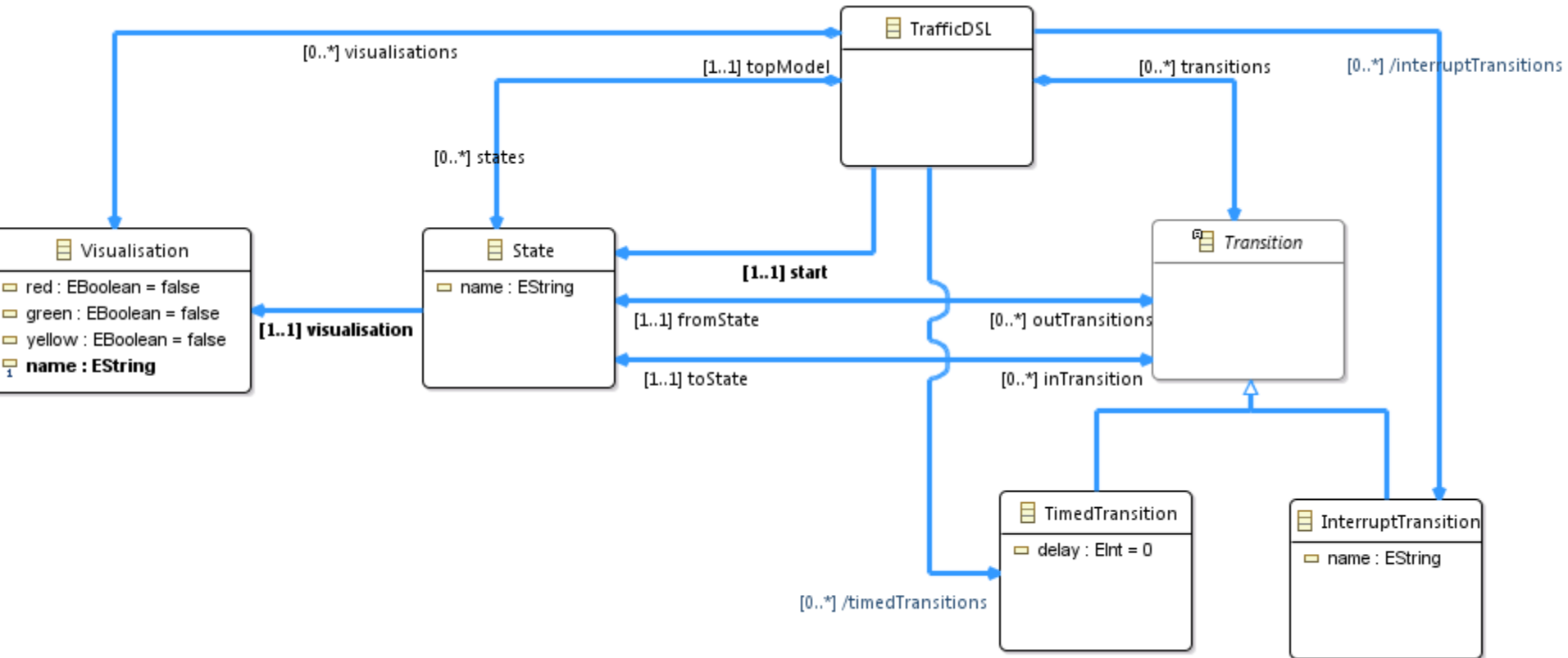
```
pattern routeSensor(sensor: Sensor) = {  
    TrackElement.sensor(switch,sensor);  
    Switch(switch);  
    SwitchPosition.switch(sp, switch);  
    SwitchPosition(sp);  
    Route.switchPosition(route, sp);  
    Route(route);  
    neg Route.routeDefinition(route, sensor);  
}
```

- VQL: declarative query language
 - Attribute constraints
 - Local + global queries
 - Compositionality+Reusability
 - Negation, Aggregations
 - Recursion, Transitive Closure over Regular Path Queries
 - Syntax: DATALOG style

Example

Statecharts metamodel

Other detailed examples



```
// s is a state of a statemachine with name n
pattern state(s: State, n: java String) {
    State.name(s,n);
}

// Old VIATRA style
pattern state(s,n) {
    State(s);
    NamedElement.name(s,n);
}

// Smart type inference
pattern state(s,n) {
    State.name(s,n);
}

// Checks if a state is red
pattern redState(s: State) {
    State.visualisation.red(s, true);
    State.visualisation.green(s, false);
    State.visualisation.yellow(s, false);
}
```

Simple queries

Optional parameter type

```
// s is a state of a statemachine with name n
pattern state(s: State, n: java String) {
    State.name(s,n);
}
```

```
// Old VIATRA style
```

```
pattern state(s,n) {
    State(s);
    NamedElement.name(s,n);
}
```

```
// Smart type inference
```

```
pattern state(s,n) {
    State.name(s,n);
}
```

```
// Checks if a state is red
```

```
pattern redState(s: State) {
    State.visualisation.red(s, true);
    State.visualisation.green(s, false);
    State.visualisation.yellow(s, false);
}
```

Support for EMF types and Java datatypes

Query parameters

Attribute navigation

Conjunction of constraints

Path expression

```
// t is an interrupt transition between a
// from state and a to state with event e
pattern interruptTransition(t,from,to,e) {
    Transition.fromState(t,from);
    Transition.toState(t,to);
    InterruptTransition.name(t,e);
}
```

```
// The result of event is non-deterministic in state
pattern nondeterministicState(state, event) {
    find interruptTransition(_,state,to1,event);
    find interruptTransition(_,state,to2,event);
    to1 != to2;
}
// No events handled by state
pattern noInterruptTransition(state) {
    State(state);
    neg find interruptTransition(_,state,_,_);
}
```

Negation
„no such“

Pattern composition / call

Anonymous variables „any“
(see Prolog/Datalog)

Note that:

- negative calls do not bind variables of header parameters
- patterns should be connected by edges (avoid Cartesian product)

```
pattern transition(from,to) {  
    Transition.fromState(t,from);  
    Transition.toState(t,to);  
}
```

```
pattern reachable(from:State,to:State) {  
    from == to;  
} or {  
    find transition+(from,to);  
}
```

Disjunction
(on pattern level)

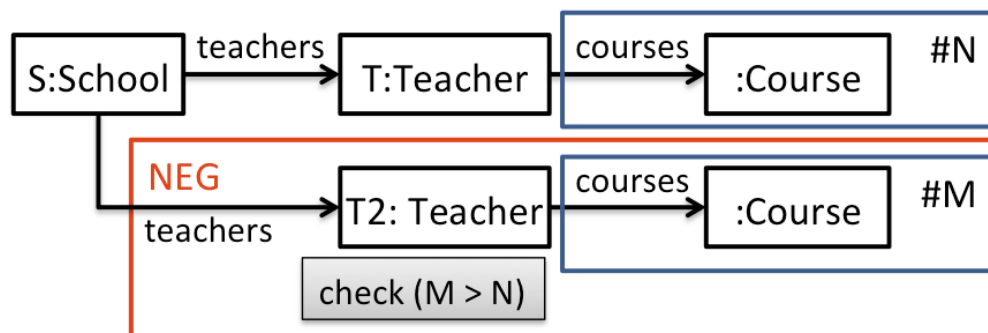
Transitive closure
over binary (2-param) patterns

```
pattern unreachableState(s:State) {  
    TrafficDSL.states(dsl,s);  
    TrafficDSL.start(dsl,start);  
    neg find reachable(start,s);  
}
```

Or equivalently:

```
neg find transition*(from,to);
```

teachersWithMostCourses(S,T)



Exercise 1:

most courses in a school or overall?

Exercise 2:

this would be even more efficient with **max find**... how and why?

```

pattern teachersWithMostCourses(school : School, teacher : Teacher) {
  School.teachers(school, teacher);
  neg find moreCourses(teacher);
}
  
```

Match counting

```

pattern moreCourses(teacher : Teacher) = {
  n == count find coursesOfTeacher(teacher, _course);
  m == count find coursesOfTeacher(teacher2, _course2);
  Teacher(teacher2); // In same school?
  teacher != teacher2;
  check(n < m);
}
  
```

Check expression
for attribute values
(pure!)


```

pattern reachableRec(from:State,to:State) {
  from == to;
} or {
  find transition(from, intermediate);
  find reachableRec(intermediate, to);
}

```

Query

from	to
s1	s1
s1	s2
s2	s2
...	...

Results

- **Set semantics** → query results form a **relation** (sets of tuples)
 - Order of tuples returned is undetermined
 - No tuples are duplicated (super important for aggregation!)
 - Not even if they differ in a hidden internal variable (e.g. **intermediate**)
 - Not even if they come from different **or**-connected pattern bodies (e.g. <s1,s1> via loop)
- (Partial) parameter binding/substitution
 - Find all states reachable from s1 ⇔ substitute s1 into **from**, filter relation
- Recursion semantics: least fixed point
 - (Runtime option switch required)

No need to pre-declare which parameter is input/output

Recursion is difficult, hence transitive closure support

Overview of VIATRA QUERY Language

- Features of the pattern language
 - Works with any (*pure*) EMF based DSL and application
 - Reusability by pattern composition
 - Recursion, negation
 - Generic and parameterized model queries
 - Bidirectional navigability of edges / references
 - Immediate access to all instances of a type
 - Complex change detection
- Benefits
 - Fully declarative + Scalable performance

VIATRA QUERY Development Tools

Loaded Model

example.traceability

Resource Set

- platform:/resource/example/example.traceability
 - CPS To Deployment
- platform:/resource/example/example.cyberphysicalsystem
 - Cyber Physical System
- platform:/resource/example/example.deployment
 - Deployment

Selection Parent List Tree Table Tree with Columns

Property	Value
Cps	Cyber Physical System
Deployment	Deployment

- Works with most EMF-based editors out-of-the-box
- Reveals matches as selection

Queries are applied & updates on-the-fly

```
// Pattern declaration
8 pattern hostIpAddress(host: HostInstance, ip : java String) {
9     //Type constraint stating that variables 'host' and 'ip' are
10     HostInstance.nodeIp(host,ip);
11 }
```

Pattern Editor

Query Results

ReteEngine

Engine details

- Resource Set
- Engine options: defaults
- Base index options: defaults

org.eclipse.viatra.examples.cps.queries.hostIpAddress - 6 matches

- host=Host Instance Aragorn, ip=152.66.102.6
- host=Host Instance Arwen, ip=152.66.102.2
- host=Host Instance Celeborn, ip=152.66.102.4
- host=Host Instance Cirdan, ip=152.66.102.1
- host=Host Instance Sauron, ip=152.66.102.3
- host=Host Instance Shelob, ip=152.66.102.5

Query Results

VIATRA QUERY VALIDATION FRAMEWORK

VIATRA QUERY Validation Framework

- Simple validation engine
 - Supports on-the-fly validation through incremental pattern matching and problem marker management
 - Uses VIATRA QUERY graph patterns to specify constraints
- Simulates EMF Validation markers
 - To ensure compatibility and easy integration with existing editors
 - Doesn't use EMF Validation directly
 - Execution model is different

Well-formedness rule specification by graph patterns

- WFRs: *Invariants* which must hold at all times
- Specification = set of elementary constraints + context
 - Elementary constraints: Query (pattern)
 - Location/context/key: a model element on which the problem marker will be placed
- Constraints by graph patterns
 - Define a pattern for the “bad case”
 - Either directly
 - Or by negating the definition of the “good case”
 - Assign one of the variables as the location/context

Match:
A violation
of the
invariant

EXAMPLE

Statechart validation constraint

- “All interrupt names on transitions going out of a single state must be distinct.”
- Capture the bad case as a query
 - There are two outgoing interrupt transitions triggered by the same event
- Add a @constraint annotation to derive an error/warning message

```
@Constraint(key = {a, event}, message = „State $a.name$ handles event  
$e.name$ ambiguously", severity = "warning" )  
pattern nondeterministicState(a, event) {  
    find interruptTransition(_,a,to1,event);  
    find interruptTransition(_,a,to2,event);  
    to1 != to2;  
}  
  
@Constraint(key = {state}, message = "There should be at least one timed  
transition going from a state", severity = "error")  
pattern noTimedTransition(state) {  
    State(state);  
    neg find timedTransition(_,state,_,_);  
}
```

EXAMPLE GUI – VIATRA Model Validation

- Works with most EMF-based tools out-of-the-box
- Manages error-warning markers on-the-fly as the user is editing the model = Instantaneous feedback

Markers in the Problems View

Validation lifecycle

■ Constraint violations

- Represented by Problem Markers (Problems view)
- Marker text is updated if affected elements are changed in the model
- Marker removed if violation is no longer present

■ Lifecycle

- Editor bound validation (markers removed when editor is closed)
- Incremental maintenance not practical outside of a running editor

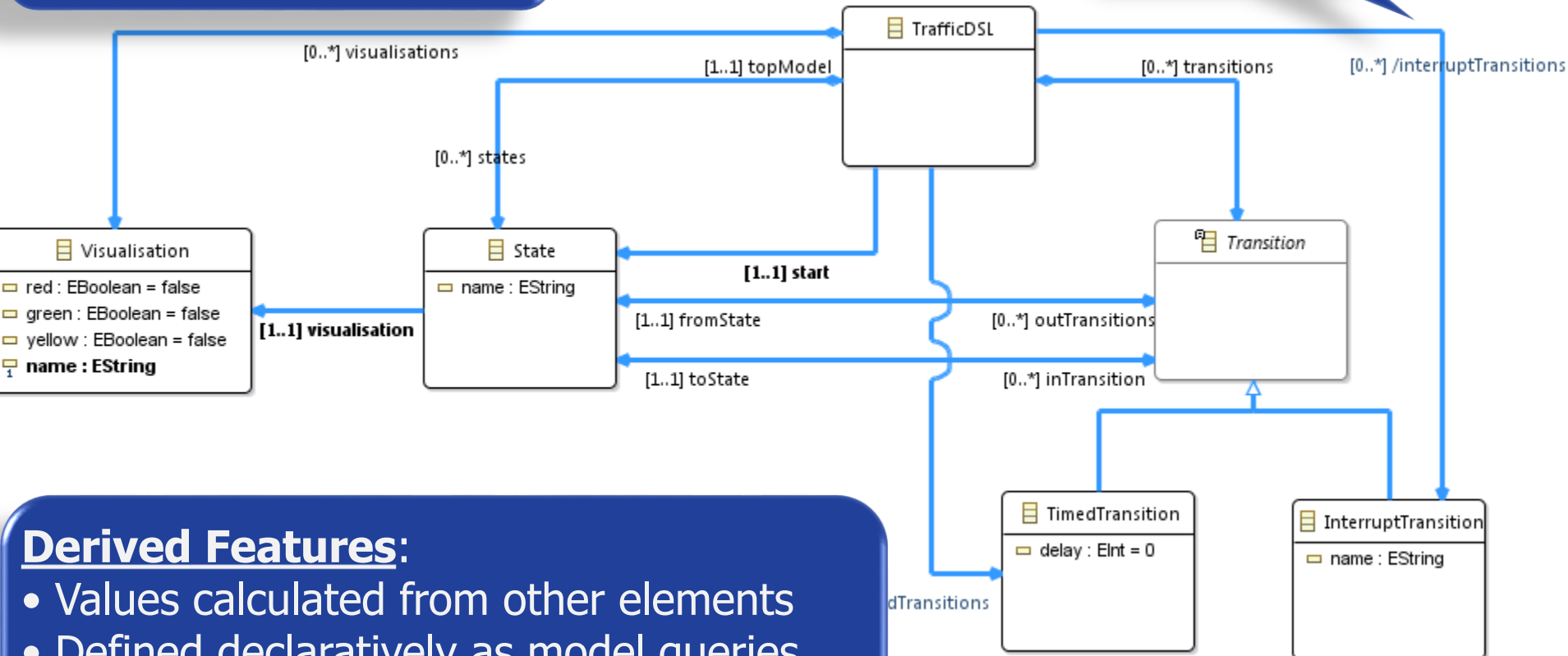
CALCULATING DERIVED FEATURES BY INCREMENTAL QUERIES

Metamodels with Derived Features

/interruptTransitions(A,B):

- B is an InterruptTransition
- B is a transition in A

Derived
Reference



Derived Features:

- Values calculated from other elements
- Defined declaratively as model queries (e.g. OCL, graph queries)
- Tooling: handle as regular EMF elements

Example

Handling Derived Features as Queries

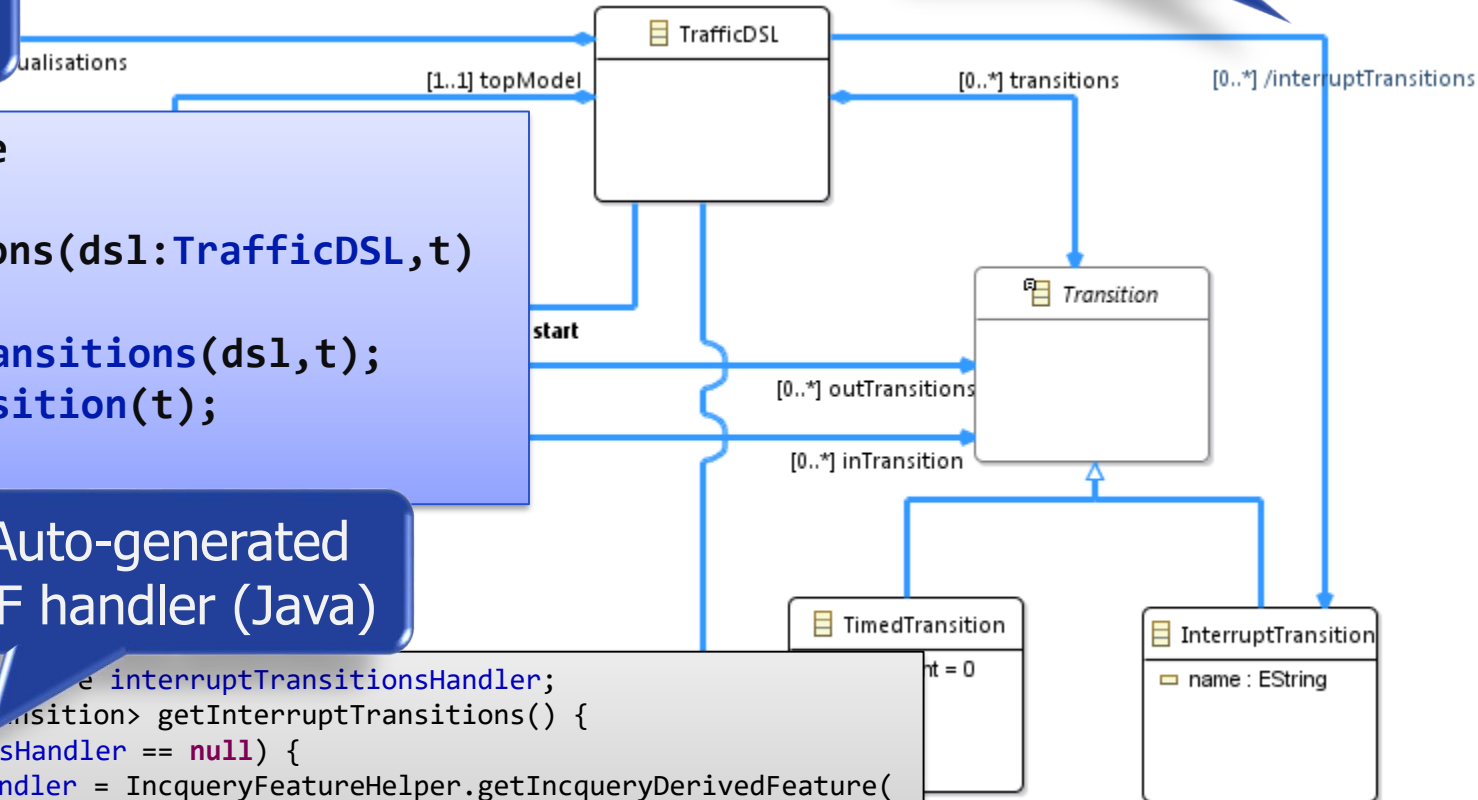
Derived
Reference

DF specification:
as a query

```
@QueryBasedFeature
pattern
interruptTransitions(dsl:TrafficDSL,t)
{
    TrafficDSL.transitions(dsl,t);
    InterruptTransition(t);
}
```

Auto-generated
DF handler (Java)

```
private IncqueryDerivedFeature interruptTransitionsHandler;
public EList<InterruptTransition> getInterruptTransitions() {
    if (interruptTransitionsHandler == null) {
        interruptTransitionsHandler = IncqueryFeatureHelper.getIncqueryDerivedFeature(
            this, SystemPackageImpl.Literals.DATA_READING_TASK,
            "system.queries.InterruptTransitions", "TrafficDSL", "InterruptTransition",
            FeatureKind.MANY_REFERENCE, true, false);
    }
    return interruptTransitionsHandler.getManyReferenceValueAsEList(this);
}
```



Query-based Derived Features: Why?

- Benefits of declarative derived specification
 - Guaranteed consistency of change notifications
 - Required e.g. for correct UI display (e.g. derived node label)
 - Provided by VIATRA QUERY incremental engine
 - Conciseness
 - As with implementing any model query
 - Especially if change notifications are relevant
- Exercise: how would you manually implement:

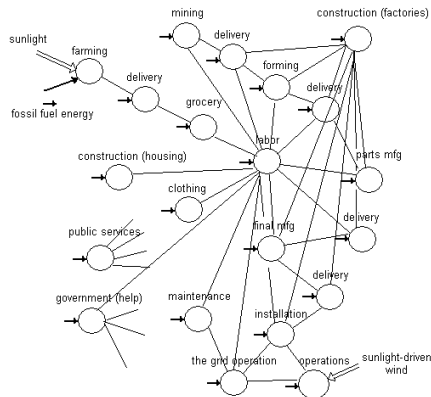
„A *Component X* **strongly depends** on another *Component Y* if it or any of its direct or indirect *child components Z* has a *feature F* typed with Y, but only mandatory (non-0 lower bound multiplicity) features and child components count”

How to manually react to:

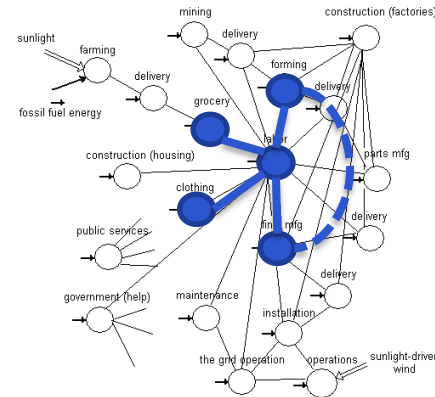
- Changing a multiplicity
- Moving a component

VIATRA VIEWERS

Live abstractions



Complex model



Computed overlay
aka. "View"

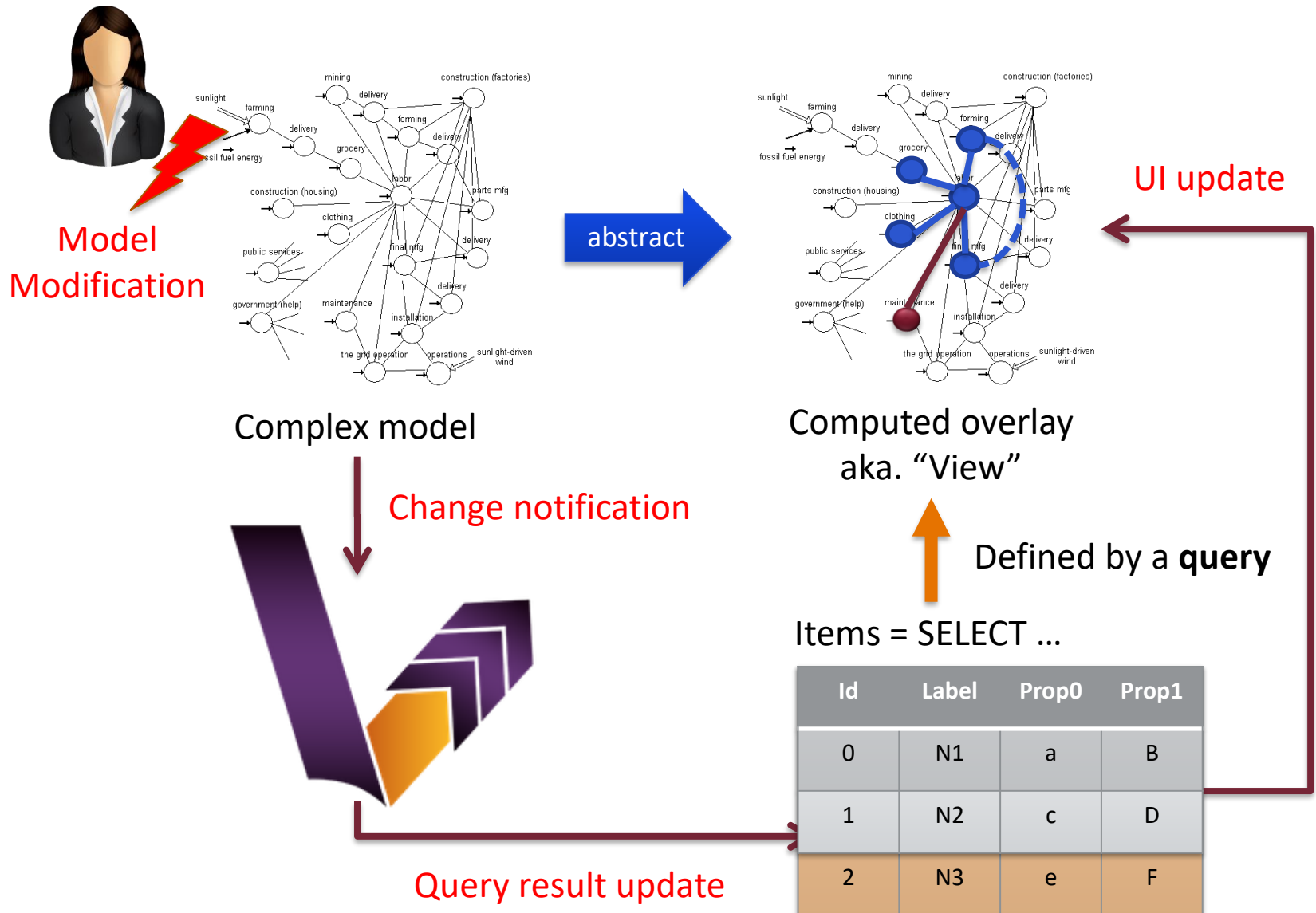


Defined by a **query**

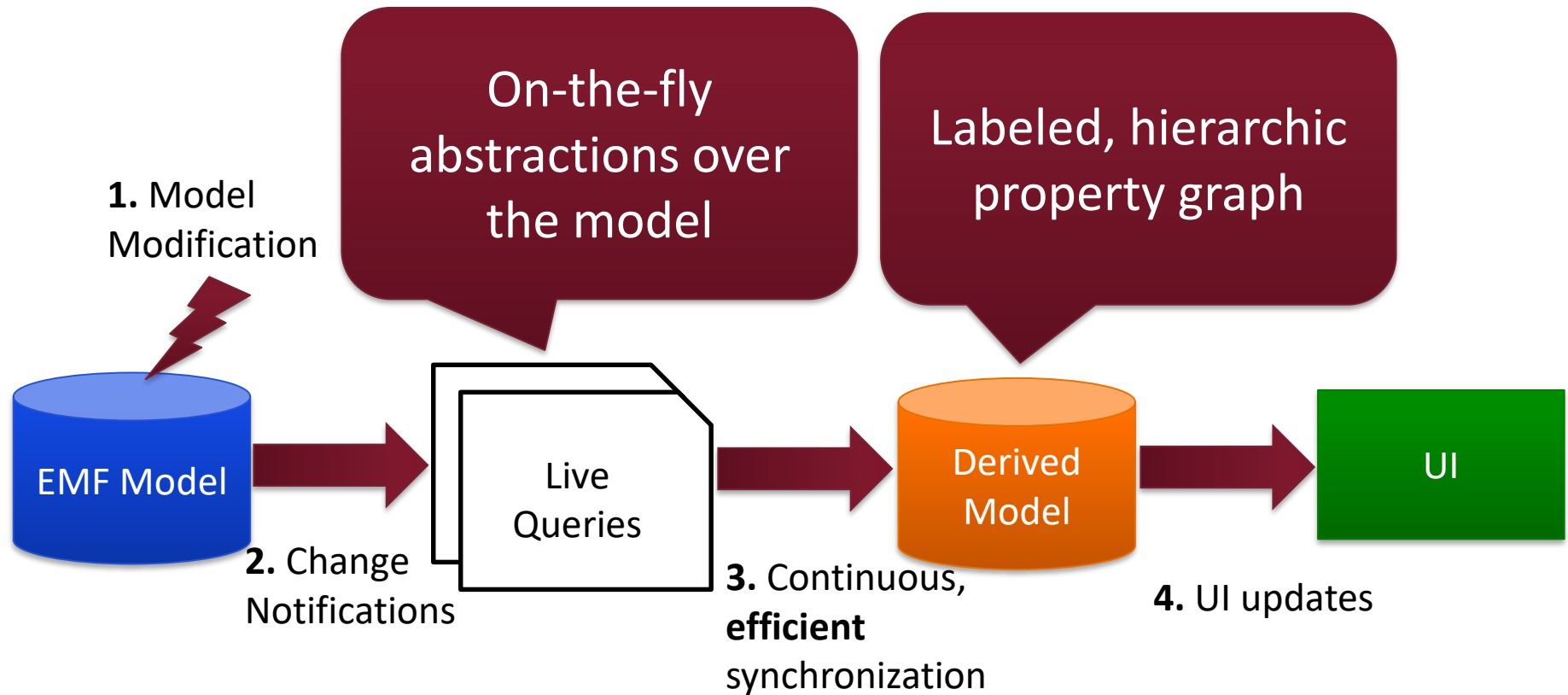
Items = SELECT ...

Id	Label	Prop0	Prop1
0	N1	a	B
1	N2	c	D

Live abstractions



VIATRA Viewers



- Visualize things that are not (directly) present in your model
- Provides an easy-to-use API for integration into your presentation layer
 - Eclipse Data Binding
 - Simple callbacks

Example

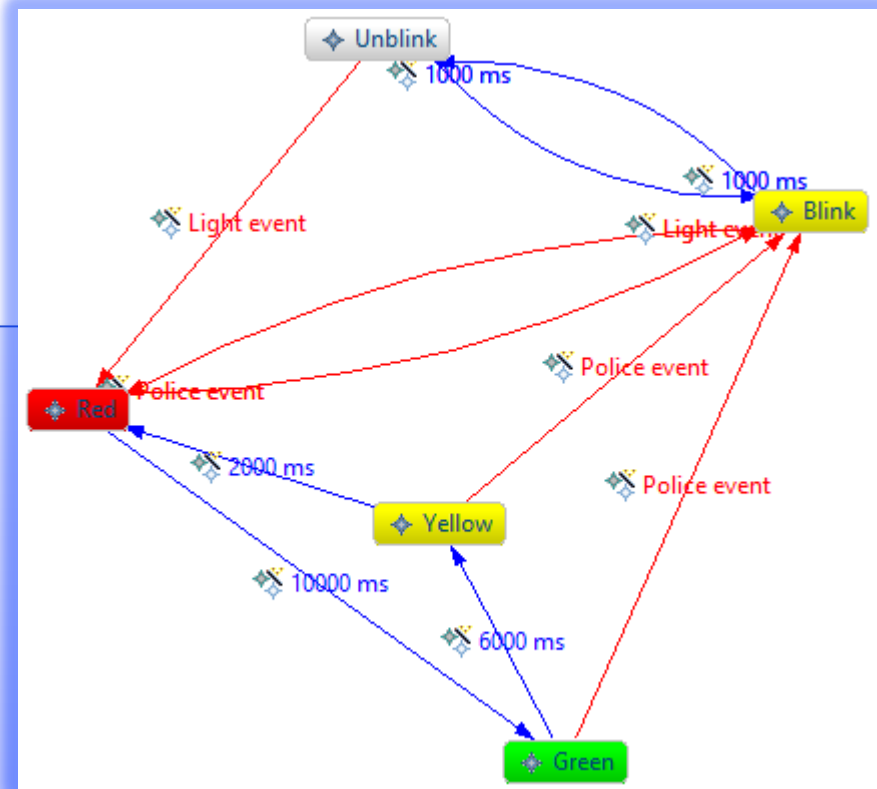
Query based view annotations

```
@Format(color = "#ff0000")
@Item(item = s, label = "$n$")
pattern redState(s: State,n) { ... }
```

```
@Item(item = s, label = "$n$")
pattern state(s,n) = { ... }
```

```
@Format(lineColor = "#0000ff")
@Edge(source = from, target = to, label = "$d$ ms")
pattern timedTransition(t,from,to,d) = { ... }
```

```
@Format(lineColor = "#ff0000")
@Edge(source = from, target = to, label = "$e$ event")
pattern interruptTransition(t,from,to,e) = { ... }
}
```



What can I do with all this? – query-based live abstractions

Syntax	Eclipse technology	Pros
Trees, tables, Properties (JFace viewers)	EMF.Edit	The real deal: doesn't hide abstract syntax
Diagrams	GEF, GMF, Graphiti	Easy to read and write for non-programmers
Textual DSLs	Xtext	Easy to read and write for programmers
JFace, Zest, yFiles Your tool!	VIATRA Viewers	Makes understanding and working with complex models a lot easier

PERFORMANCE BENCHMARKS

The Train Benchmark

■ Model validation workload:

- User edits the model
- Instant validation of well-formedness constraints
- Model is repaired accordingly

■ Scenario:

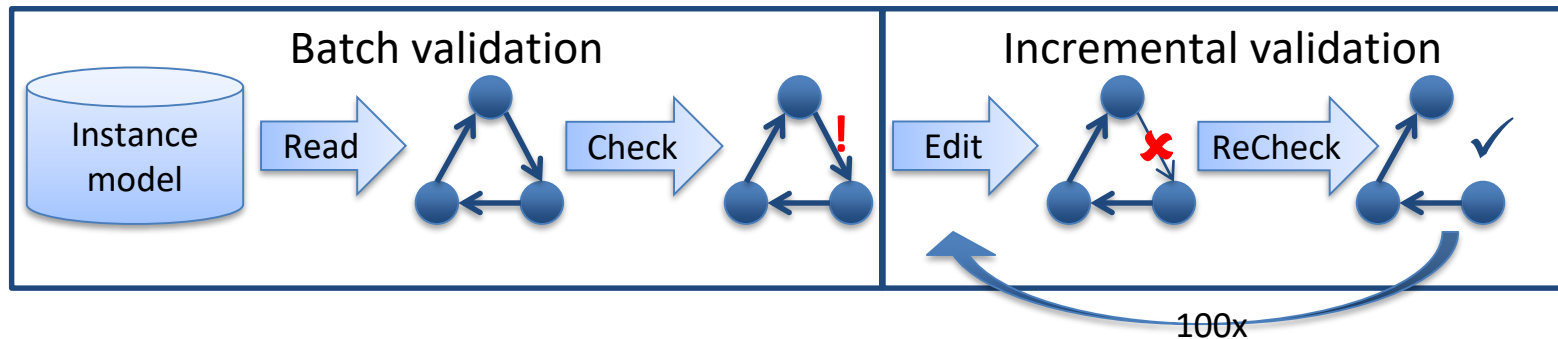
- Load
- Check
- Edit
- Re-Check

■ Models:

- Randomly generated
- Close to real world instances
- Following different metrics
- Customized distributions
- Low number of violations

■ Queries:

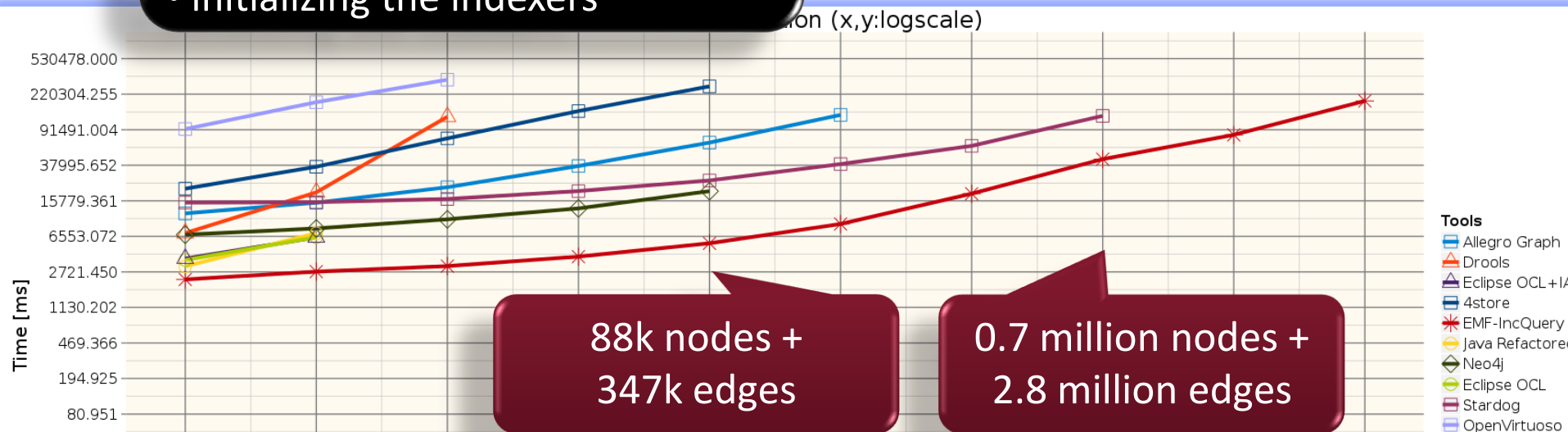
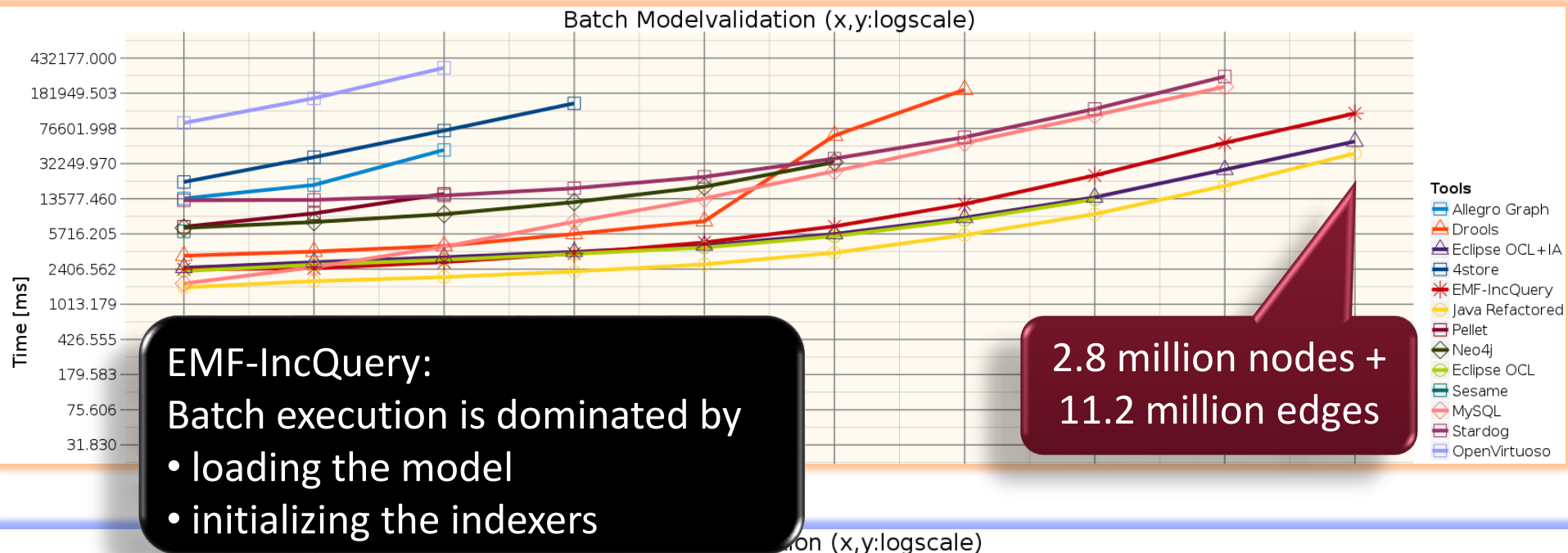
- Two simple queries (<2 objects, attributes)
- Two complex queries (4-7 joins, negation, etc.)
- Validated match sets



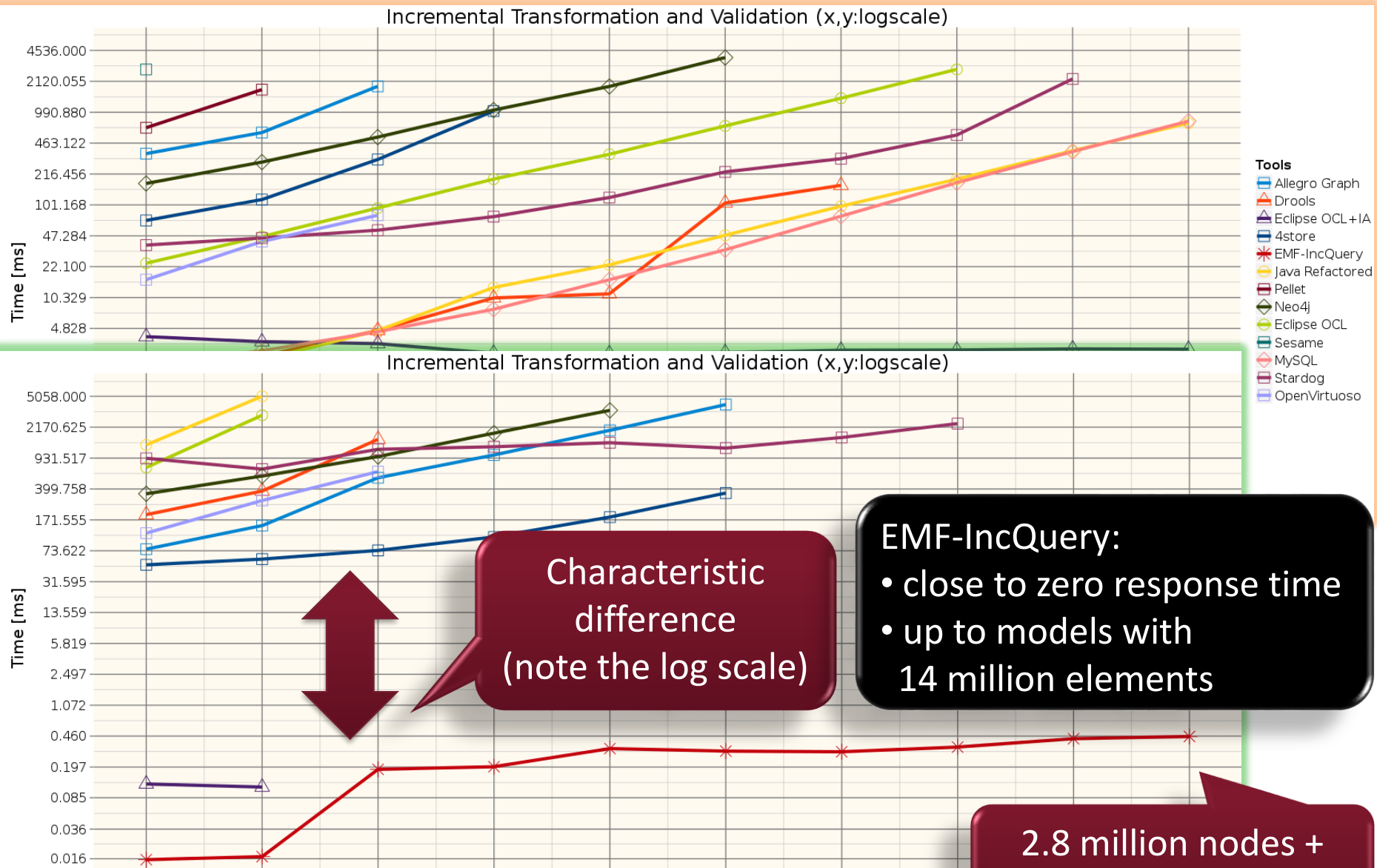
What Tools are Compared?



Batch validation runtime (complex queries)

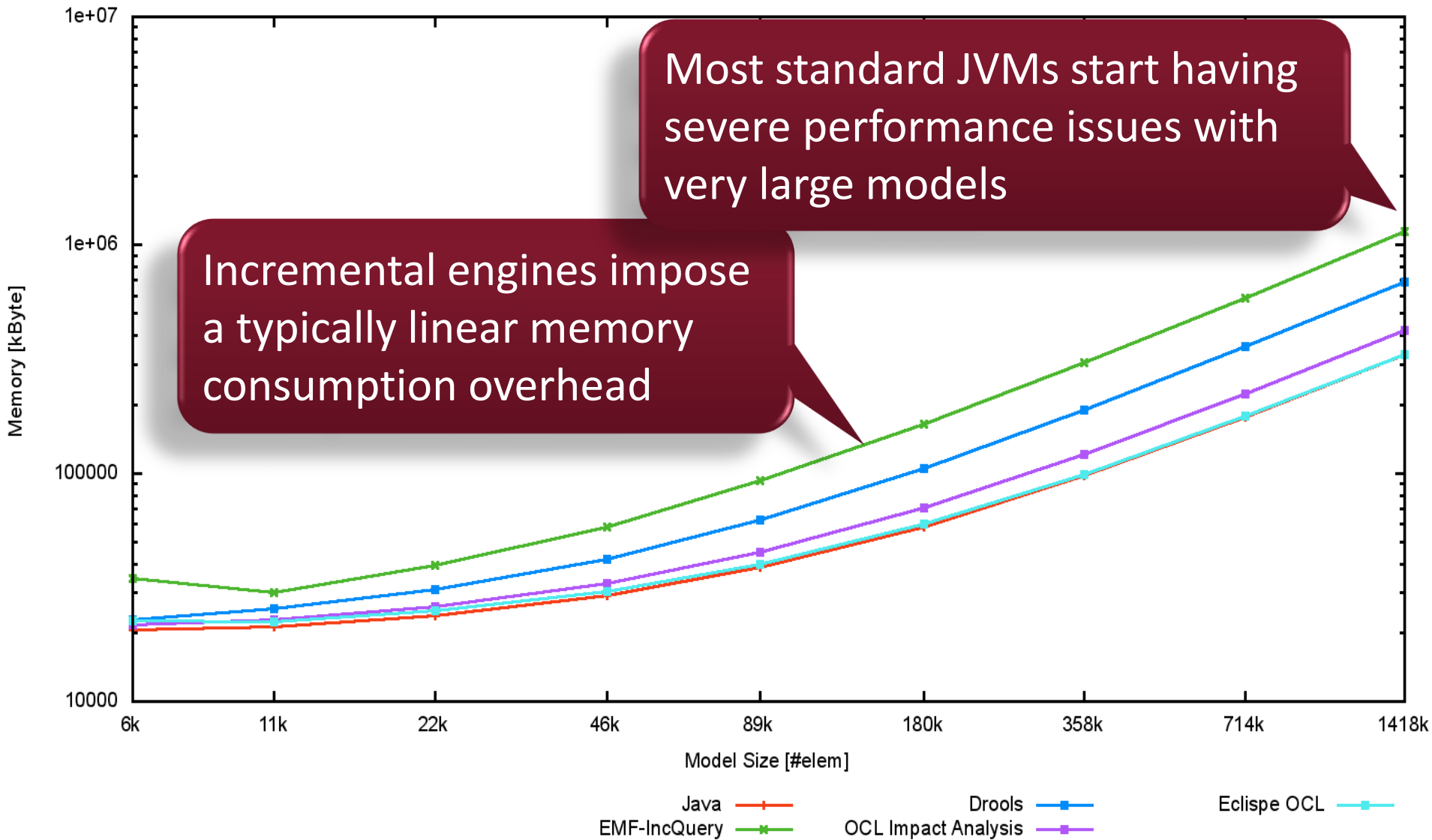


Re-validation time (complex queries)



Memory usage

AllTestCaseAvg Memory Usage



CONCLUSIONS

Selected Applications of VIATRA QUERY

- Complex traceability
- Query driven views
- Abstract models by derived objects

Toolchain for
IMA configs



- Connect to Matlab
Simulink model
- Export: Matlab2EMF
- Change model in EMF
- Re-import:
EMF2Matlab

MATLAB-EMF
Bridge



- Live models
(refreshed 25
frame/s)
- Complex event
processing

Gesture
recognition



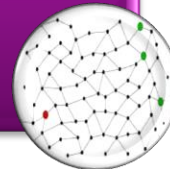
- Experiments on open
source Java projects
- Local search vs.
Incremental vs.
Native Java code

Detection of
bad code smells



- Rules for operations
- Complex structural
constraints (as GP)
- Hints and guidance
- Potentially infinite
state space

Design Space
Exploration



- Itemis (developer)
- Embraer
- Thales
- ThyssenKrupp
- CERN

Known Users

