EMF-IncQuery
Incremental evaluation of model queries

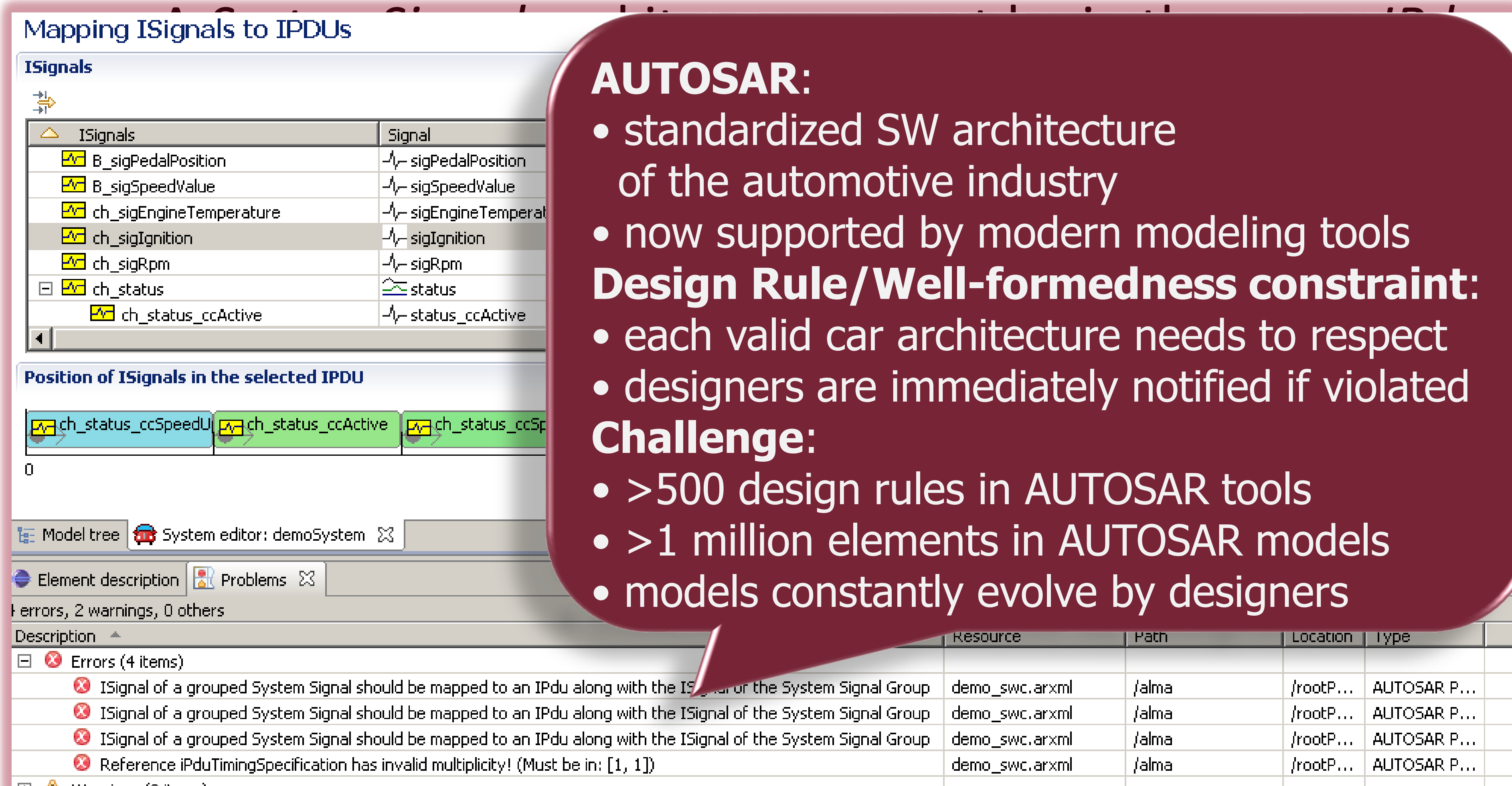
Model Driven Systems Development
Lecture 04

Budapest University of Technology and Economics
Department of Measurement and Information Systems

MOTIVATION

Motivation: Early validation of design rules

SystemSignalGroup design rule (from AUTOSAR)

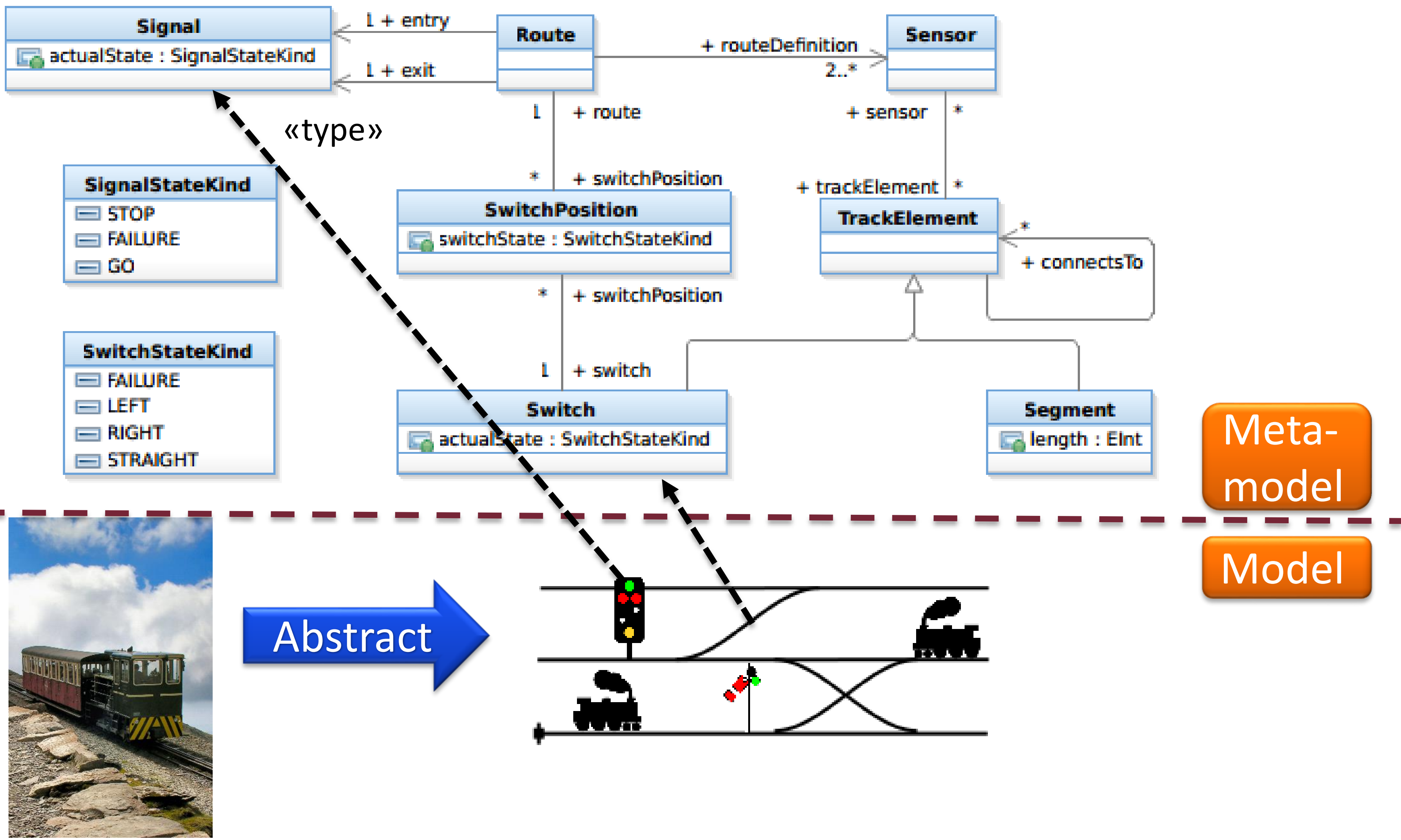


AUTOSAR:
- standardized SW architecture of the automotive industry
- now supported by modern modeling tools

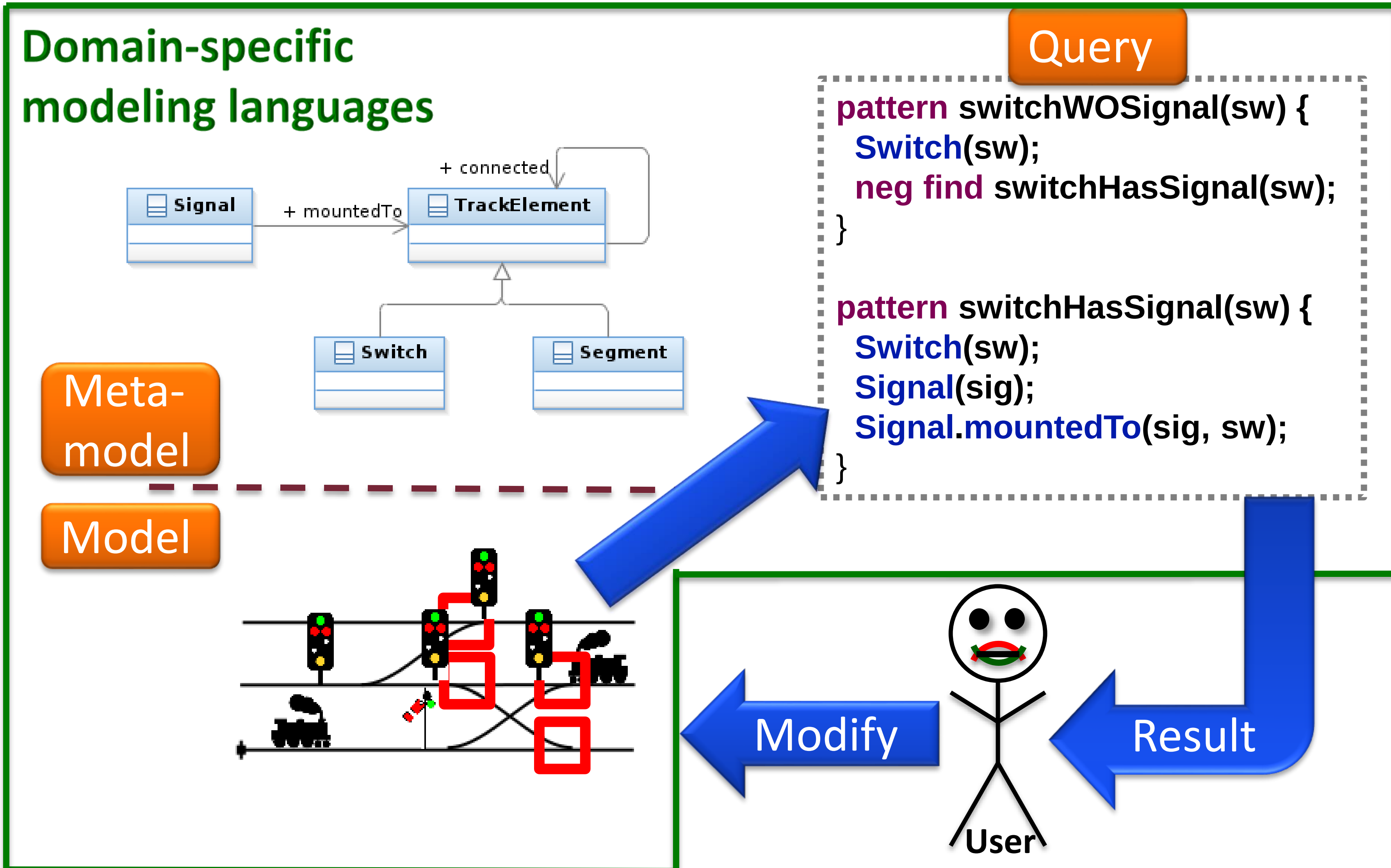
Design Rule/Well-formedness constraint:
- each valid car architecture needs to respect
- designers are immediately notified if violated

Challenge:
- >500 design rules in AUTOSAR tools
- >1 million elements in AUTOSAR models
- models constantly evolve by designers

Domain-Specific Modeling Languages



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

Validation can take hours

Application	Mod
System models	10^8
Sensor data	10^9
Geospatial models	10^{12}

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

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

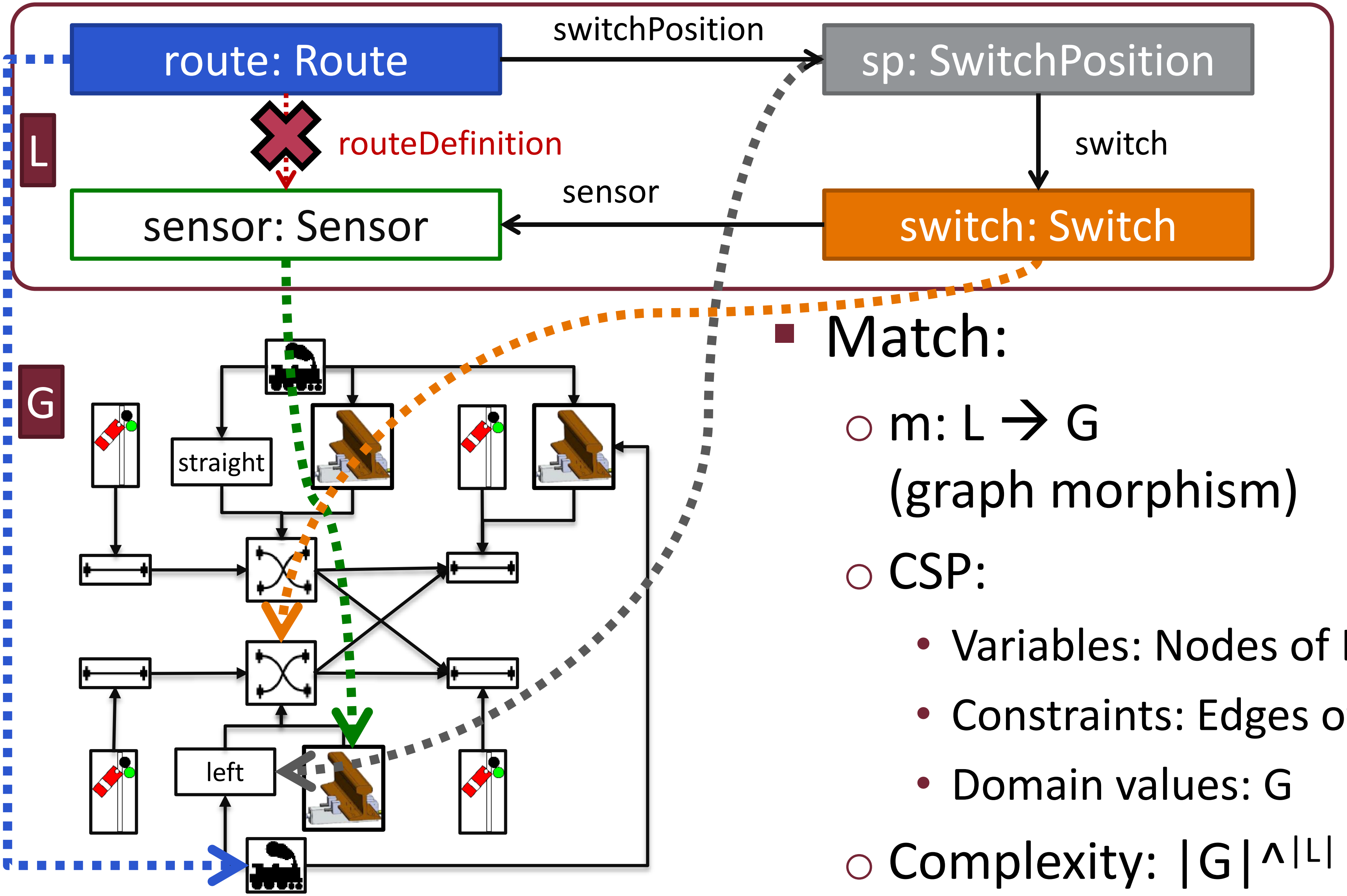
Query(A,B) $\leftarrow \wedge cond_i(A_i,B_i)$
- all tuples of model elements *a,b*
- satisfying the query condition
- along the match *A=a* and *B=b*
- parameters A,B can be input/ output

Categorization of Query Languages

- Hard to write?
- Your options
  - Java (or C/C++, C#, …)
  - Declarative languages (OCL, EMF Query 1-2, …)

	Imperative query languages	Declarative query languages
Expressive power	☹ (you write lots of code)	☺ (very concise)
Safety	☺☺ (precise control over what happens at execution)	☺☹ (unintended side-effects)
Learning curve	☺ (you already know it)	☹ (may be difficult to learn)
Reusability	☺ (standard OO practices)	☹☹ (???)
Performance	☹☺ (considerable manual optimization necessary)	☺☹ (depends on various factors)

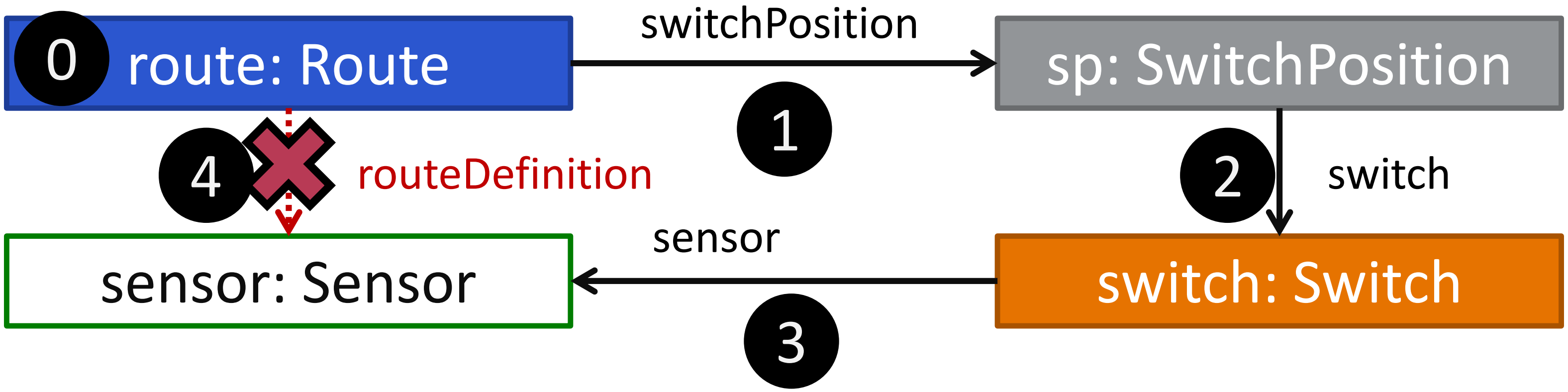
Graph Pattern Matching for Queries

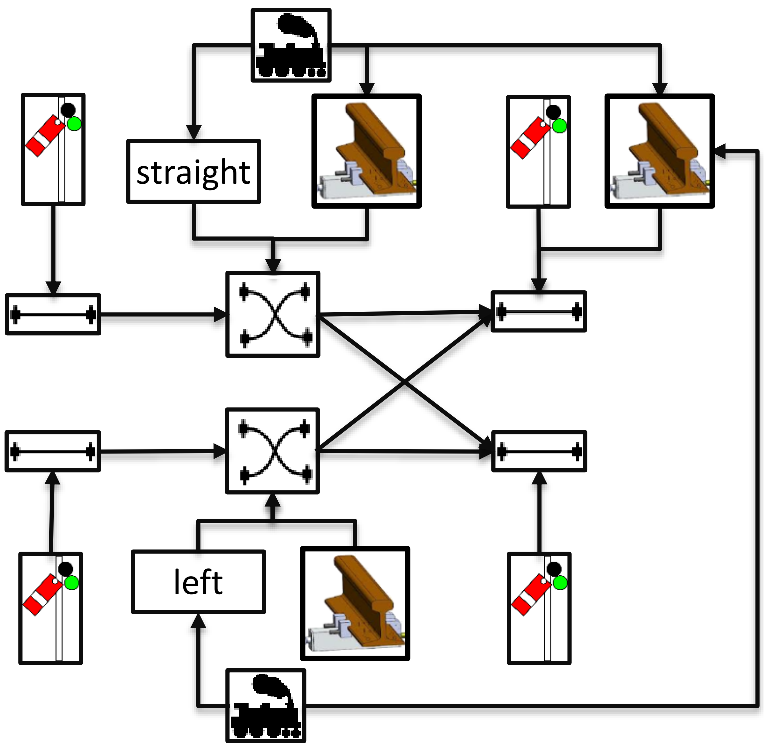


- Match:
  - m: L → G (graph morphism)
  - CSP:
    - Variables: Nodes of L
    - Constraints: Edges of L
    - Domain values: G
  - Complexity: $|G|^{|L|}$

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

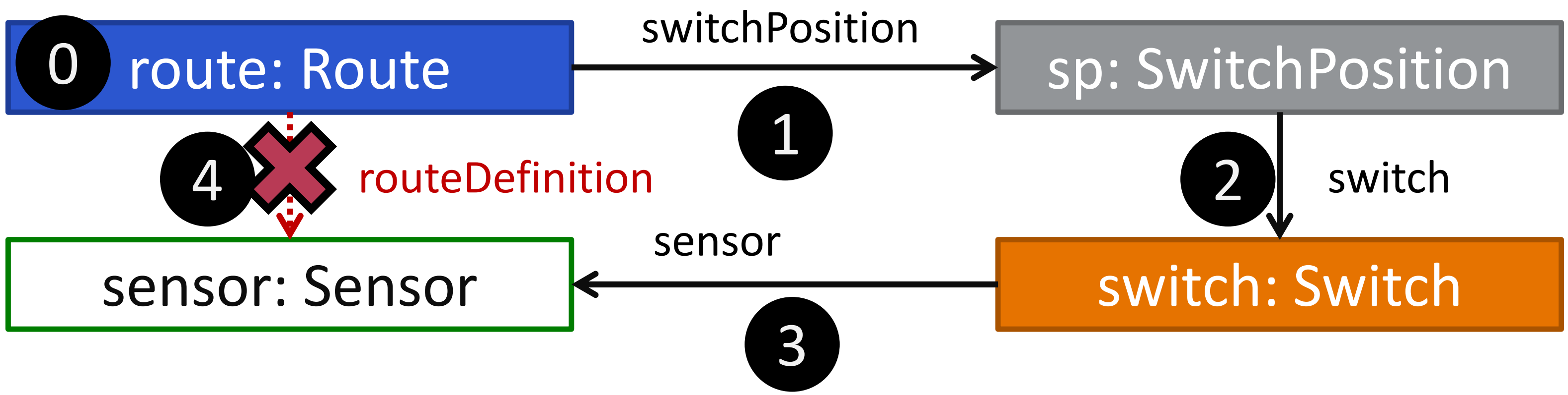
Graph Pattern Matching (Local Search)

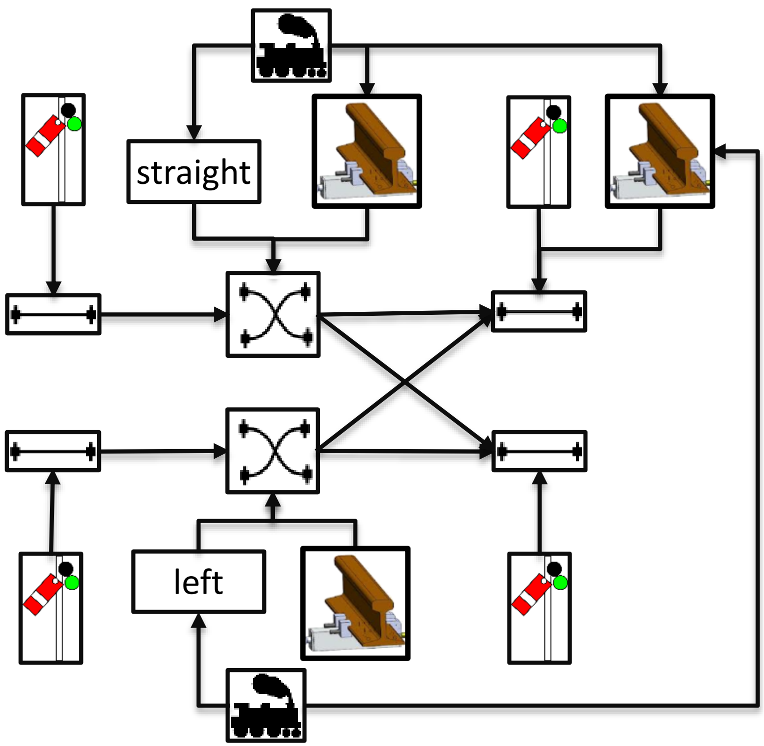




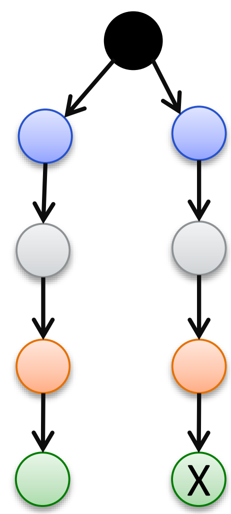
- Search Plan:
  - Select the first node to be matched
  - Define an ordering on graph pattern edges
- Search is restarted from scratch each time

Graph Pattern Matching (Local Search)

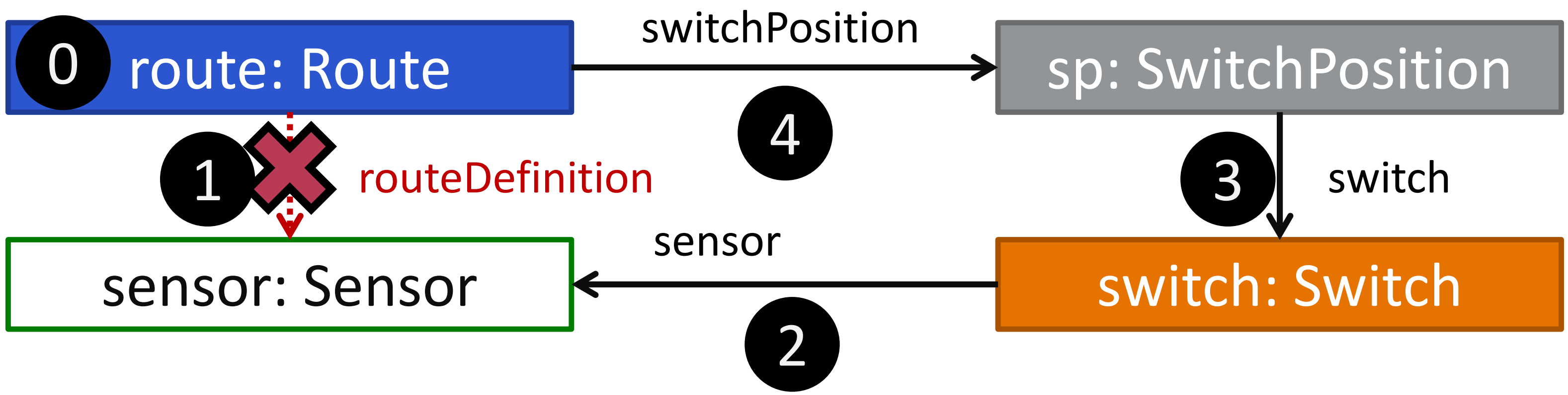


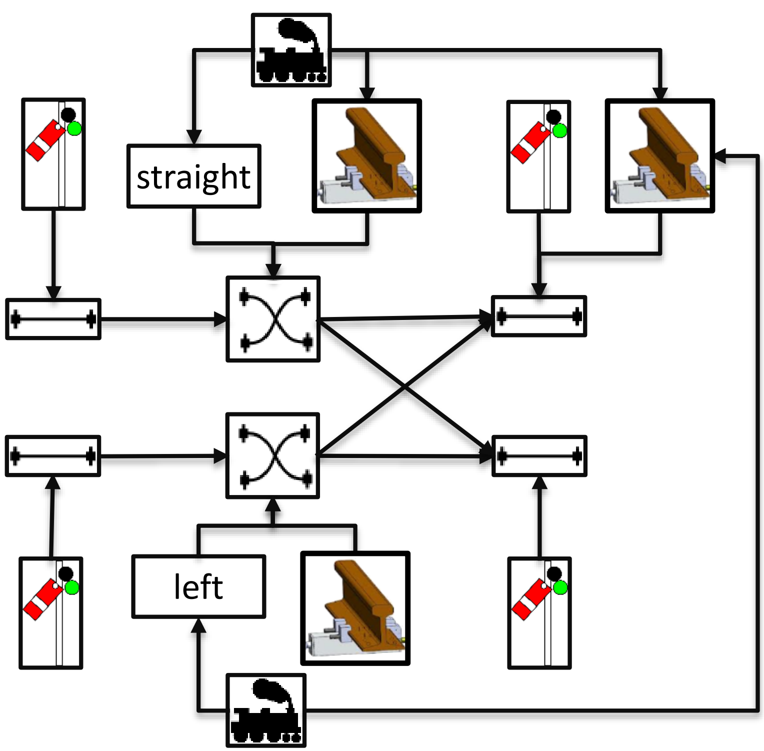


- Search Tree:

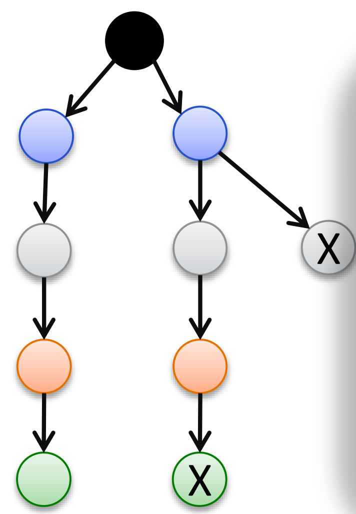


Graph Pattern Matching (Local Search)





- Alternate Search Tree:



Local Search based PM

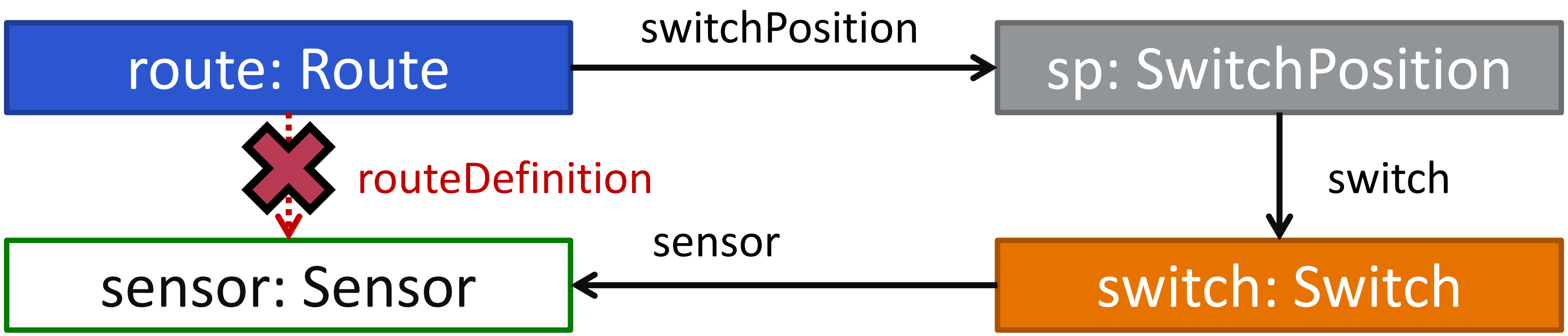
- Runtime depends on search plan
- Good search plan: narrow at root wide at leaves

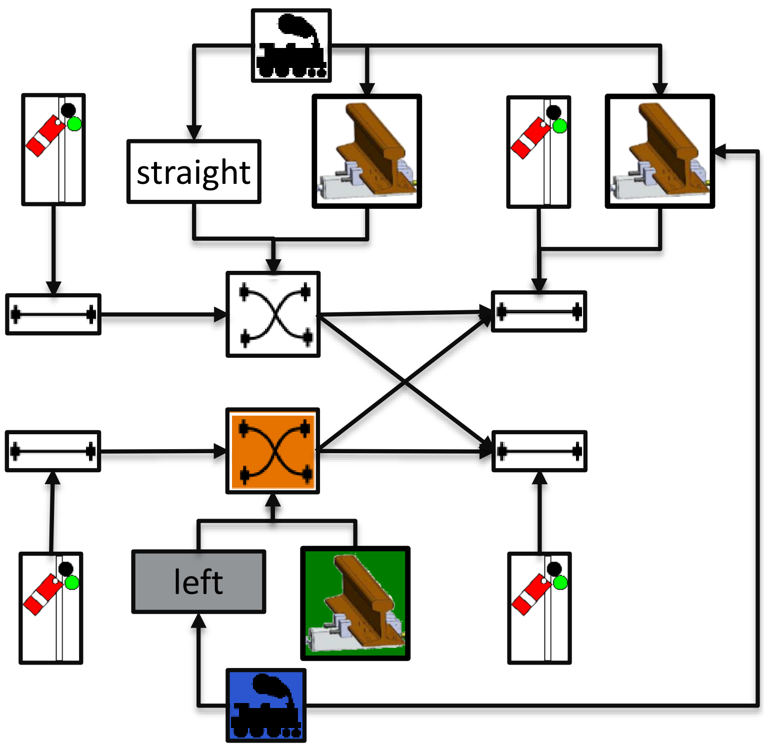
INCREMENTALITY IN QUERIES AND TRANSFORMATIONS

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 to 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. Rule 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. Rule system is loaded
  3. Calculate full match set
  4. Model is changed (rules fired or designer updates)
  5. Iterate Steps 3 and 4 until rule system is stopped
- Query results are always available for designer

EMF-IncQuery: 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

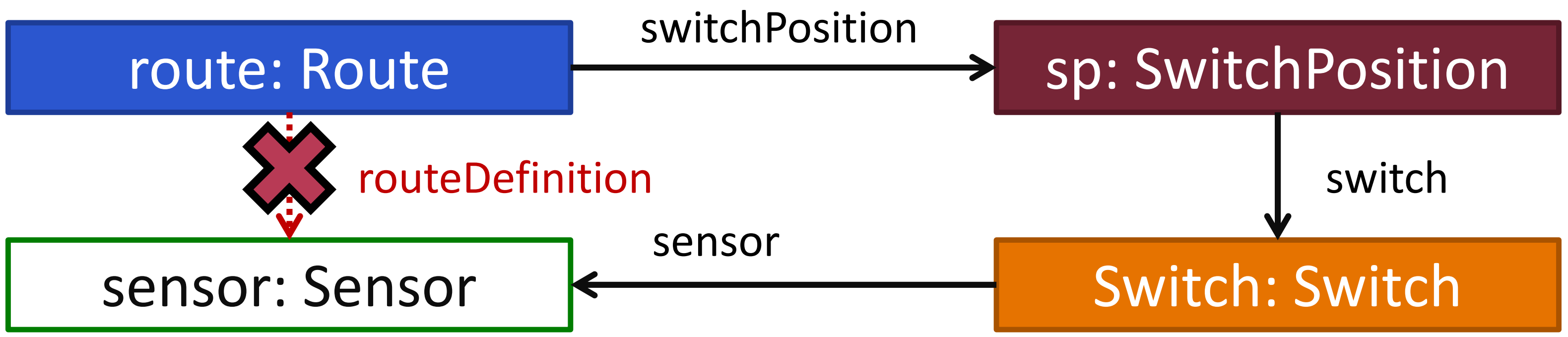
- Derived features,
- On-the-fly validation
- View generation,
- Works out-of-the-box with EMF applications

Features

http://eclipse.org/incquery

INCREMENTAL MODEL QUERIES: THE LANGUAGE

The IncQuery (IQ) Graph Query Language

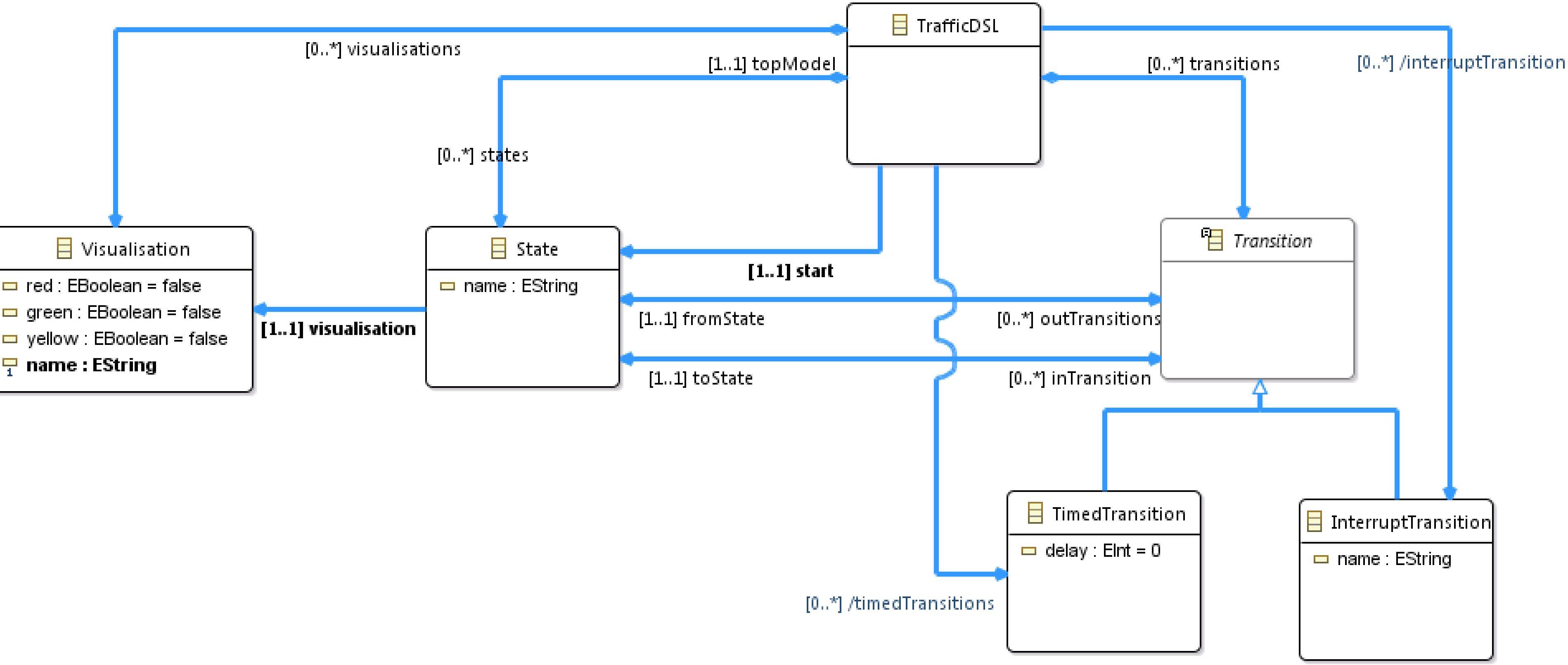



```
pattern routeSensor(sensor: Sensor) = {
      TrackElement.sensor(switch,sensor);
      Switch(switch);
      SwitchPosition. switch(sp, switch);
      SwitchPosition(sp);
      Route.switchPosition(route, sp);
      Route(route);
      neg find head(route, sensor);
}
pattern head(R, Sen) = {
  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

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

Example Statecharts metamodel

- Other detailed examples



IQPL Simple queries

```
// S is a state of a statemachine with name N
pattern state(S:State, N) {
    State.name(S,N);
}
// Old VIATRA style
pattern state(S,N) {
    State(S);
    State.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

```
// S is a state of a statemachine with name N
pattern state(S:State, N) {
    State.name(S,N);
}
// Old VIATRA style
pattern state(S,N) {
    State(S);
    State.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);
}
```

Query definition

Syntactic sugar

Query parameter

Type constraint

Attribute navigation

Path expression

IQPL Simple queries

Support for built-in EMF datatypes: Strings, integers, etc.

```
// S is a state of a statemachine with name N
pattern state(S:State, N) {
    State.name(S,N);
}
// Old VIATRA style
pattern state(S,N) {
    State(S);
    State.name(S,N);
}
// Smart type inference
pattern state(S,N) {
    State.name(S,N);
}
// Checks if a state is r
pattern redState(S: State
    State.visualisation.r
    State.visualisation.gr
    State.visualisation.y
}
```

```
// T is a timed transition between a
// from state and a to state with delay D
pattern timedTransition(T,from,to,D) {
    Transition.fromState(T,from);
    Transition.toState(T,to);
    TimedTransition(T);
    TimedTransition.delay(T,D);
}
// T is an interrupt transition between a
// from state and a to state with delay D
pattern interruptTransition(T,from,to,E) {
    Transition.fromState(T,from);
    Transition.toState(T,to);
    InterruptTransition(T);
    InterruptTransition.name(T,E);
}
```

IQPL Pattern composition and NAC

Pattern composition / call

```
// 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 timed transition going out of a State
pattern noTimedTransition(State) {
    State(State);
    neg find timedTransition(_,State,_,_);
}
```

Negative application condition

Anonymous variables (see Prolog)

Transitive closure and disjunction

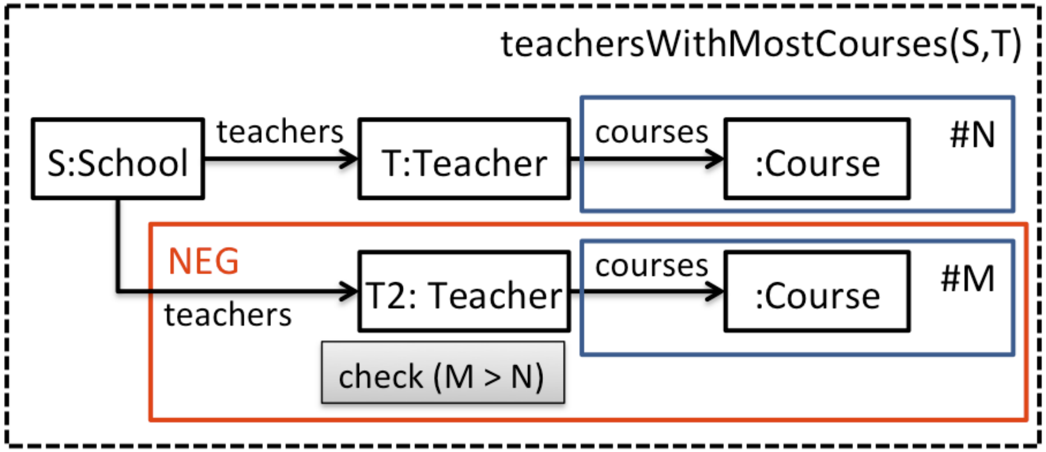
```
pattern transition(from,to) {
    Transition.fromState(T,from);
    Transition.toState(T,to);
}

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

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

Disjunction
(on pattern level)

Transitive closure
(over 2 param patterns)

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

IQPL Check expression & Match count




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

pattern moreCourses(Teacher : Teacher) = {
    N == count find coursesOfTeacher(Teacher,_Course);
    M == count find coursesOfTeacher(Teacher2,_Course2);
    Teacher(Teacher2);
    Teacher != Teacher2;
    check(N < M);}
```

Match counting

Check expression for attribute values (pure!)

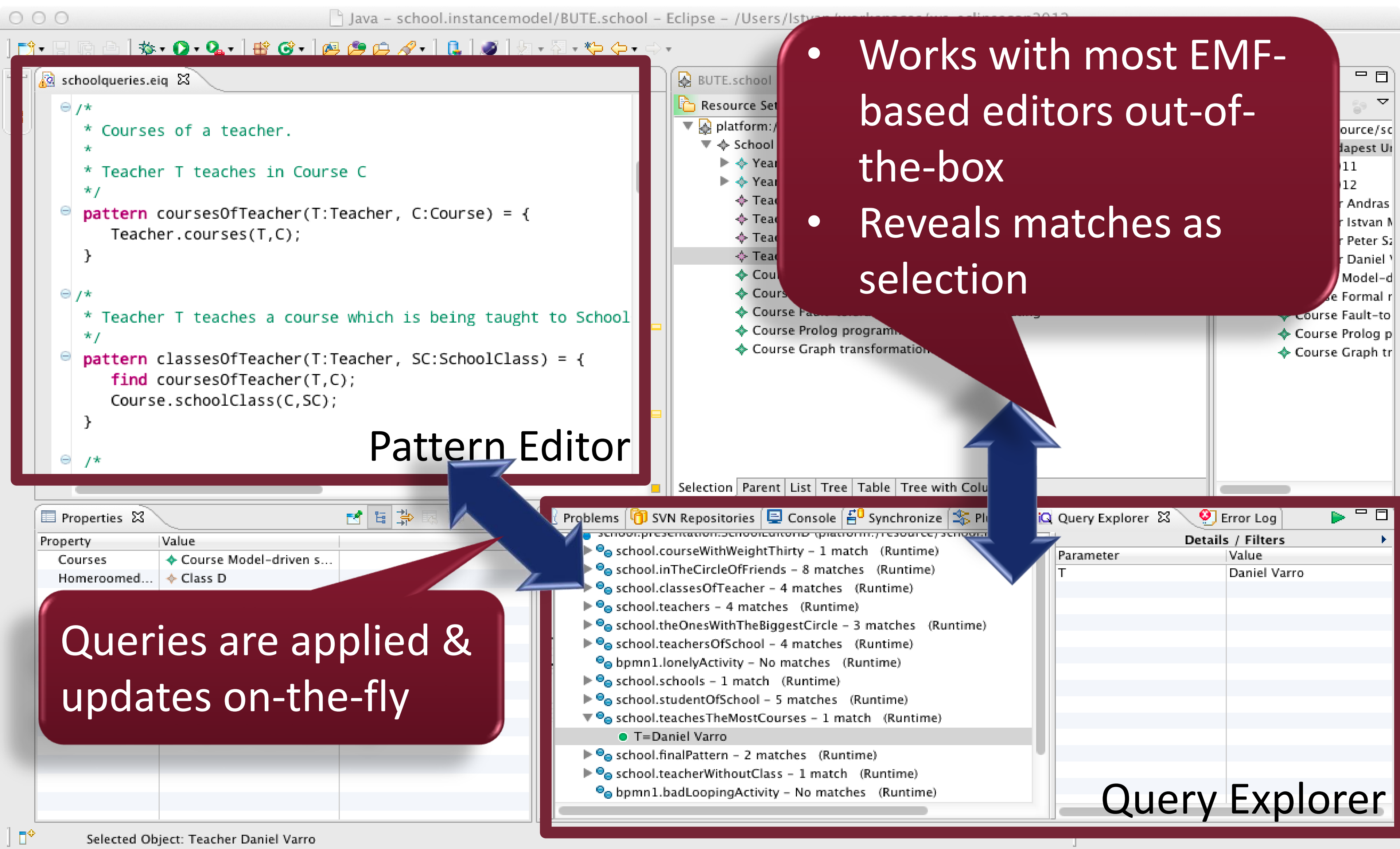
Overview of IncQuery Pattern Language

- Features of the pattern language
  - Works with any *(pure)* EMF based DSL and application
  - Reusability by pattern composition
  - Arbitrary 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

IncQuery Development Tools

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

Queries are applied & updates on-the-fly



EMF-IncQuery: An Open Source Eclipse Project

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

Definition

IncQuery

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

Execution

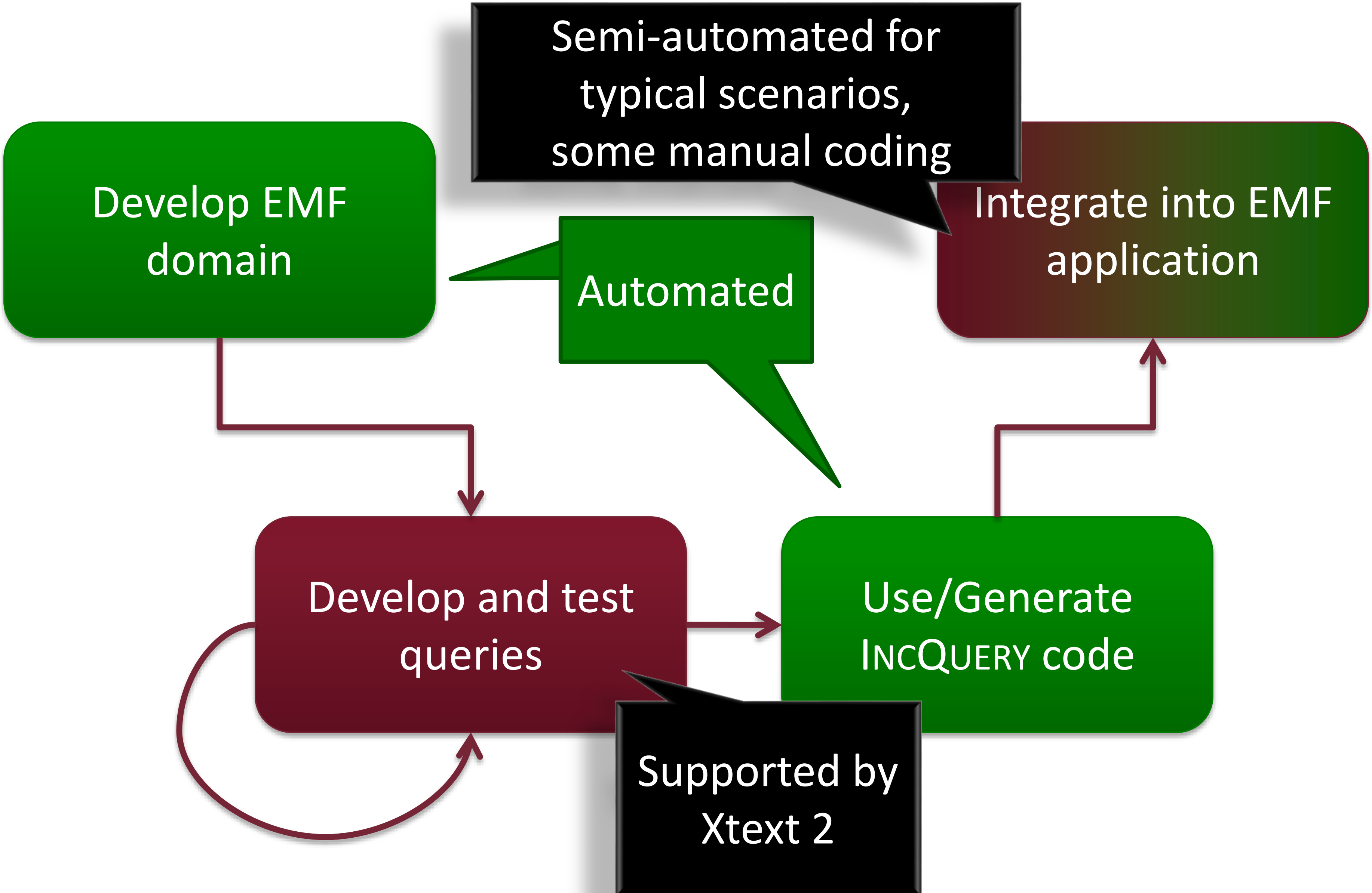
- Derived features,
- On-the-fly validation
- View generation,
- Works out-of-the-box with EMF applications

Tooling

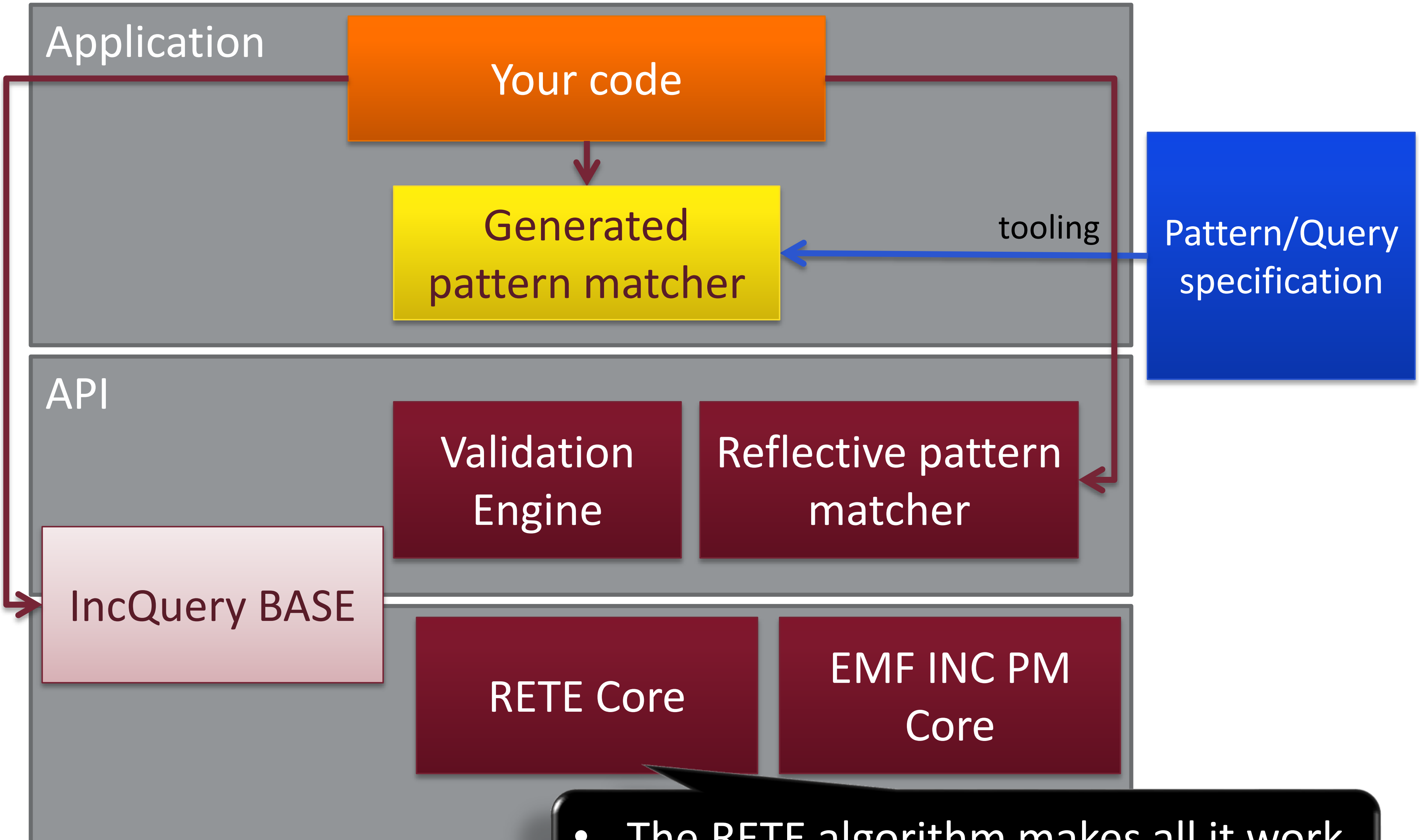
http://eclipse.org/incquery

OVERVIEW OF INCREMENTAL QUERY EVALUATION

Development workflow



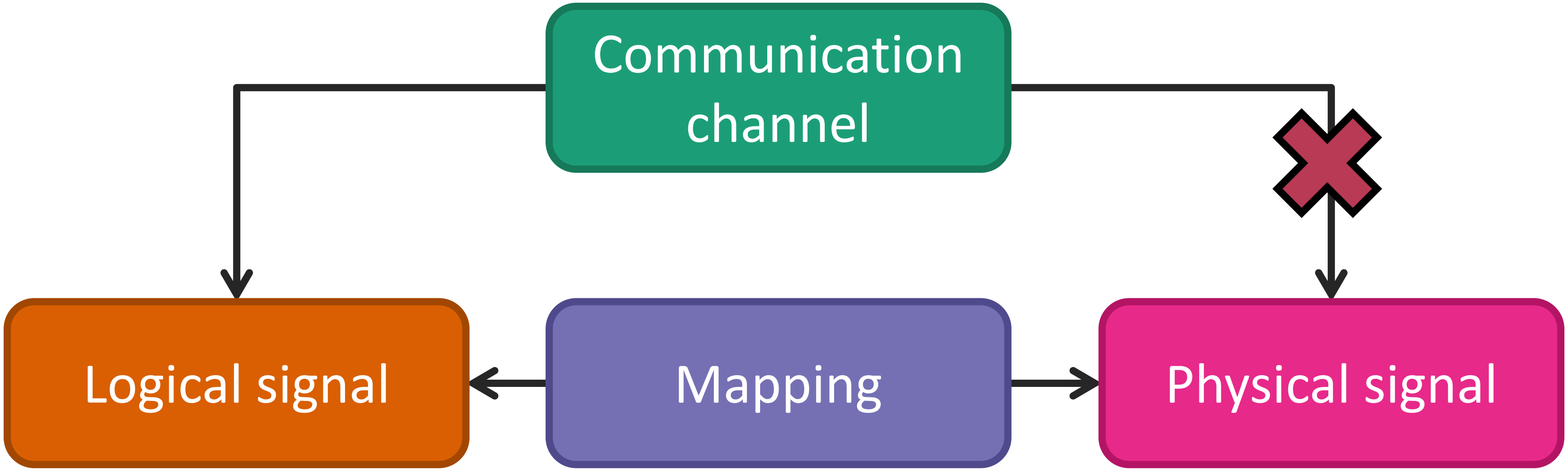
EMF-IncQuery Architecture v0.8



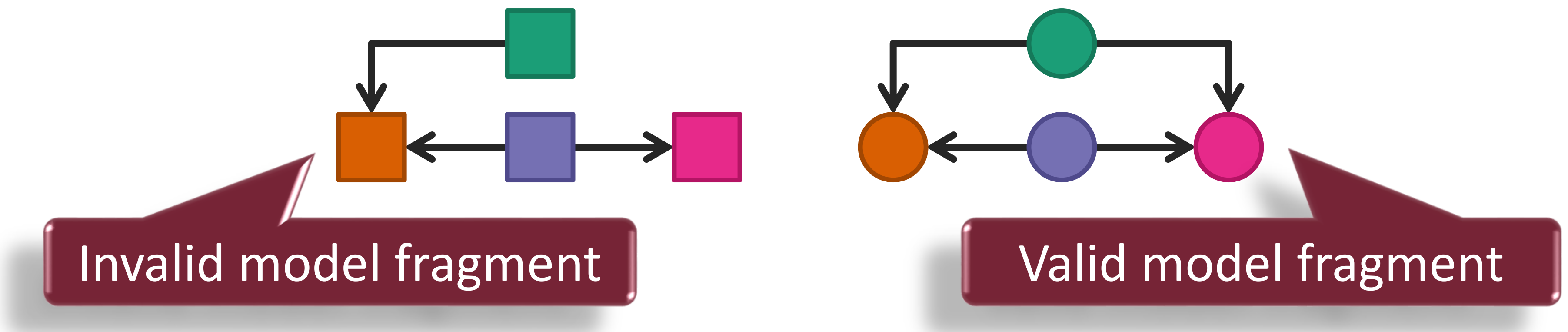
- The RETE algorithm makes all it work
- Well-known in rule-based systems

Incremental Query Evaluation by RETE

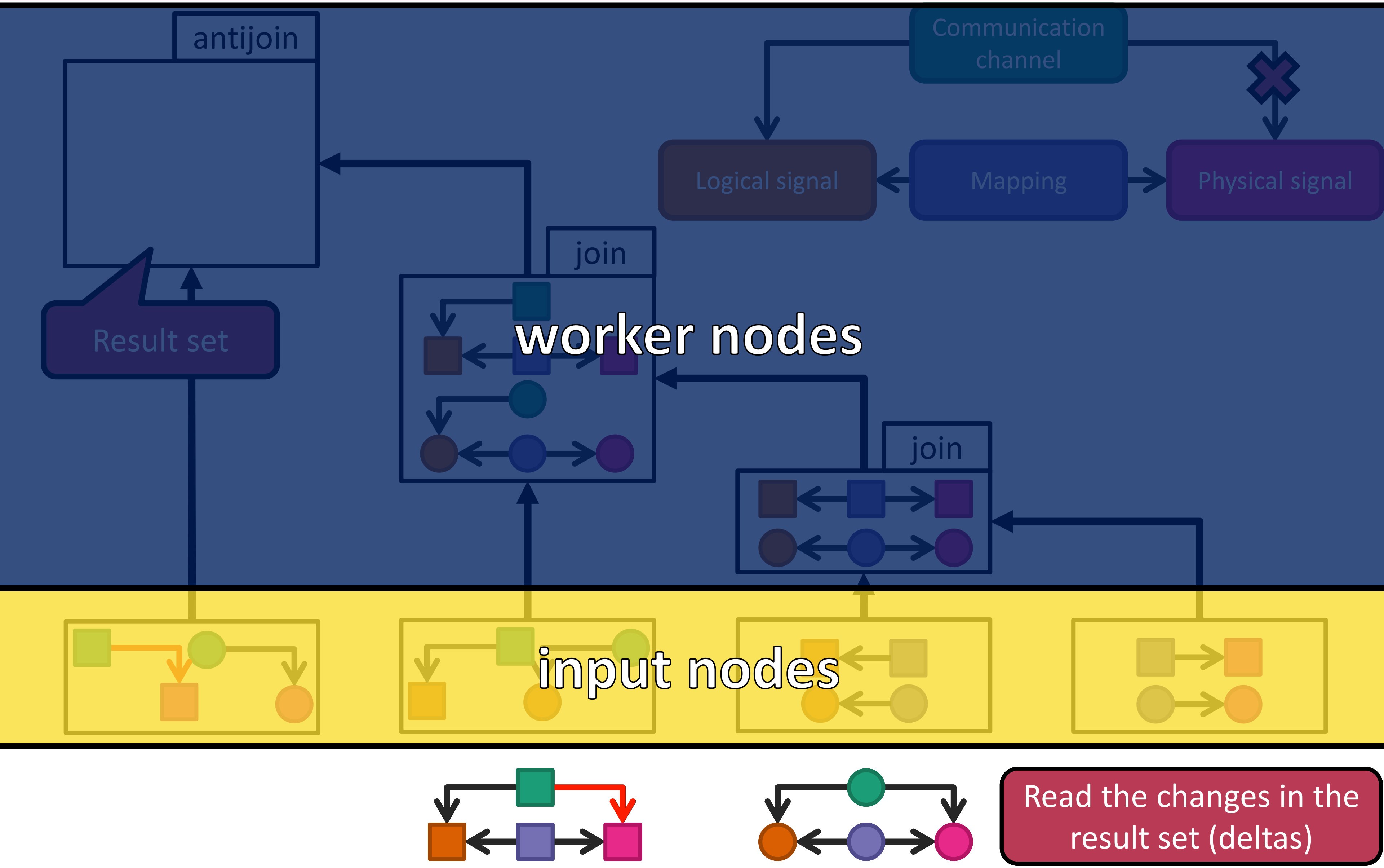
- AUTOSAR well-formedness validation rule



- Instance model

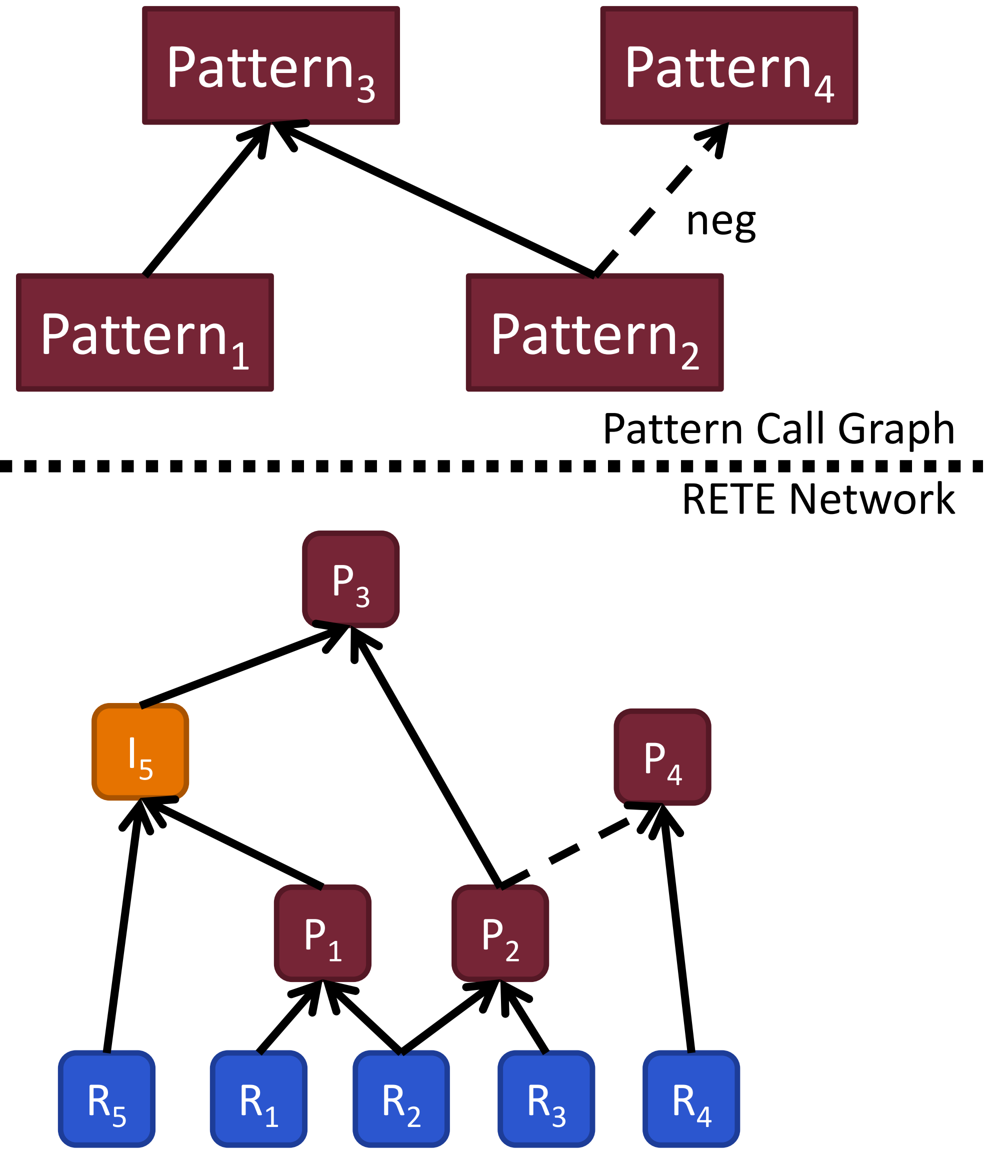


Incremental Query Evaluation by RETE



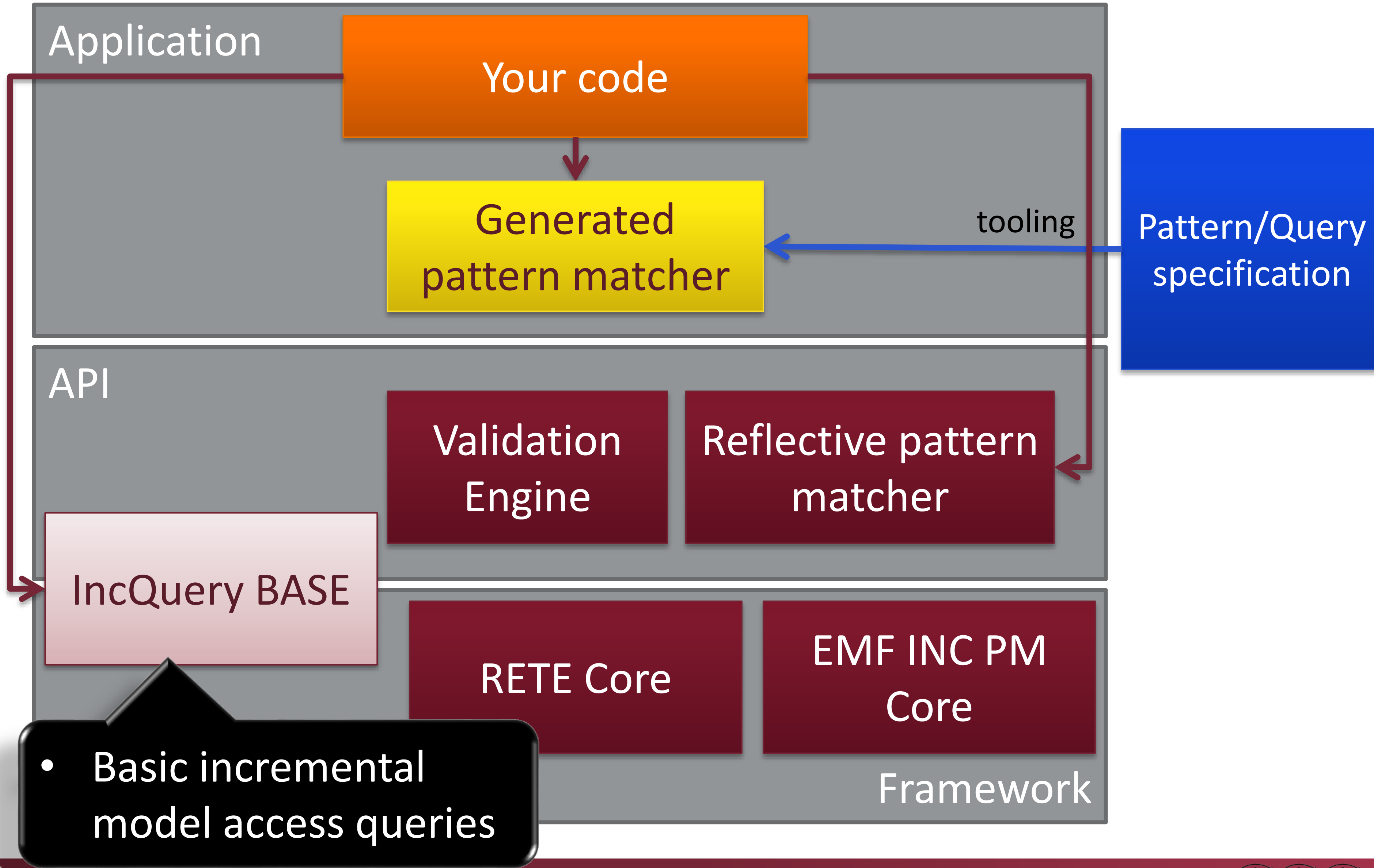
Read the changes in the result set (deltas)

Construction of RETE network



- **Single network for all patterns**
- **Node sharing**: controlled by the developer (pattern call graph)
- RETE visualization
- Advanced construction algorithm by dynamic programming: G. Varró et. al (ICMT 2013)

EMF-INCQUERY Architecture v0.8



IncQuery Base

- Light-weight Java library for basic (yet very powerful) EMF model access queries with **incremental evaluation**
- Supports
  - Get all instance elements by type
  - Reverse navigation along references
  - Get model elements by attribute value/type
- Very easy to integrate into any EMF tool (pure Java) – **standalone!**
- Same high performance and scalability as IncQuery
- Incremental transitive closure
  - Computation of e.g. reachability regions, connected model partitions, …
  - Innovative new algorithm for general graphs

EMF-IncQuery: An Open Source Eclipse Project

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

Definition

IncQuery

- **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

Tooling

http://eclipse.org/incquery

INCQUERY VALIDATION FRAMEWORK

IncQuery Validation Framework

- Simple validation engine
  - Supports on-the-fly validation through incremental pattern matching and problem marker management
  - Uses IncQuery 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: 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

```
// The result of Event is non-deterministic in State
@Constraint(location = A, message = "$A.name$ is a bad looping activity",
severity = "warning" )
pattern nondeterministicState(A, Event) {
    find interruptTransition(_,A,To1,Event);
    find interruptTransition(_,A,To2,Event);
    To1 != To2;
}
// No timed transition going out of a State
@Constraint(location = State, message = "There should be at most one timed
transition going from a state", severity = "error")
pattern noTimedTransition(State) {
    State(State);
    neg find timedTransition(_,State,_,_);
}
```

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

Validation UI integration

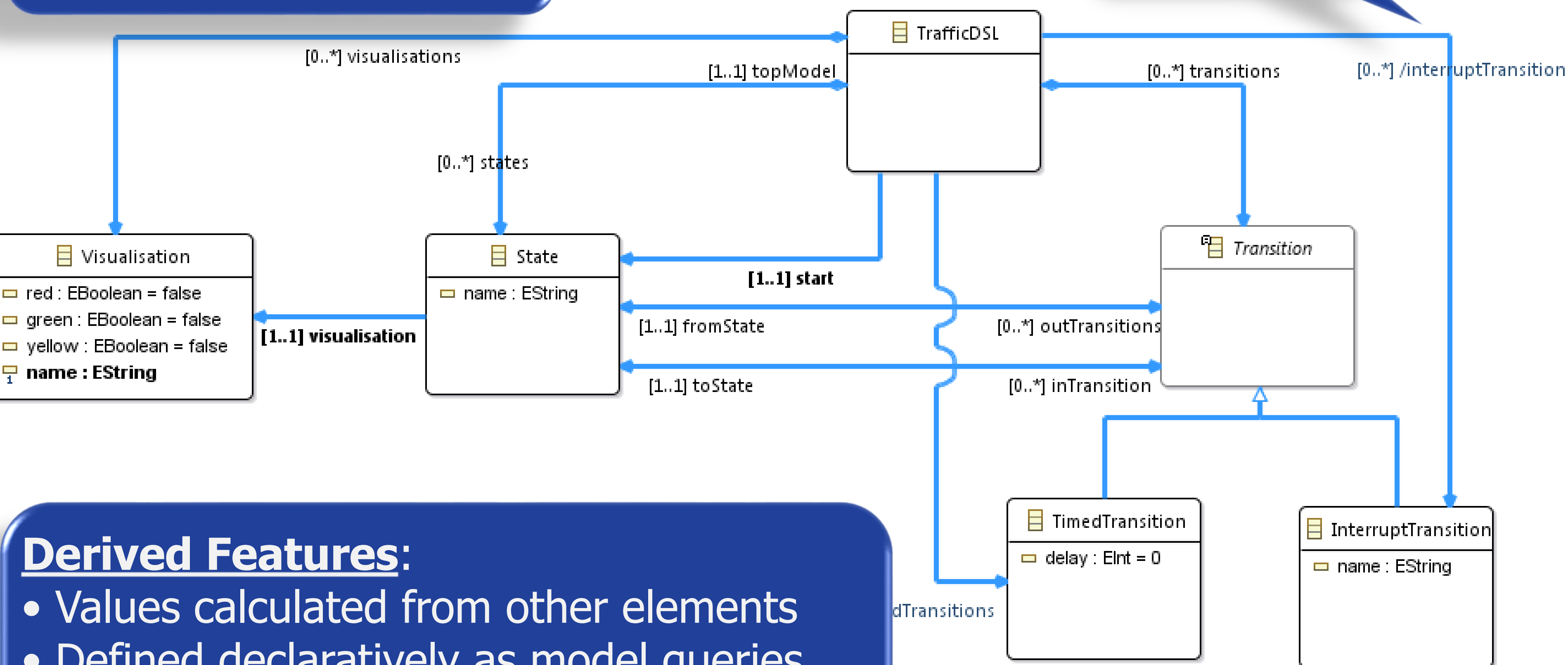
- A menu item (command) to start the validation engine
- Generic (part of the IncQuery Validation framework)
  - GMF editor command
    - Appears in all GMF-based editor's context menu
  - Sample Reflective Editor command
    - Appears on the toolbar
- Generated
  - EMF generated tree editor command
    - Appears on the toolbar

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

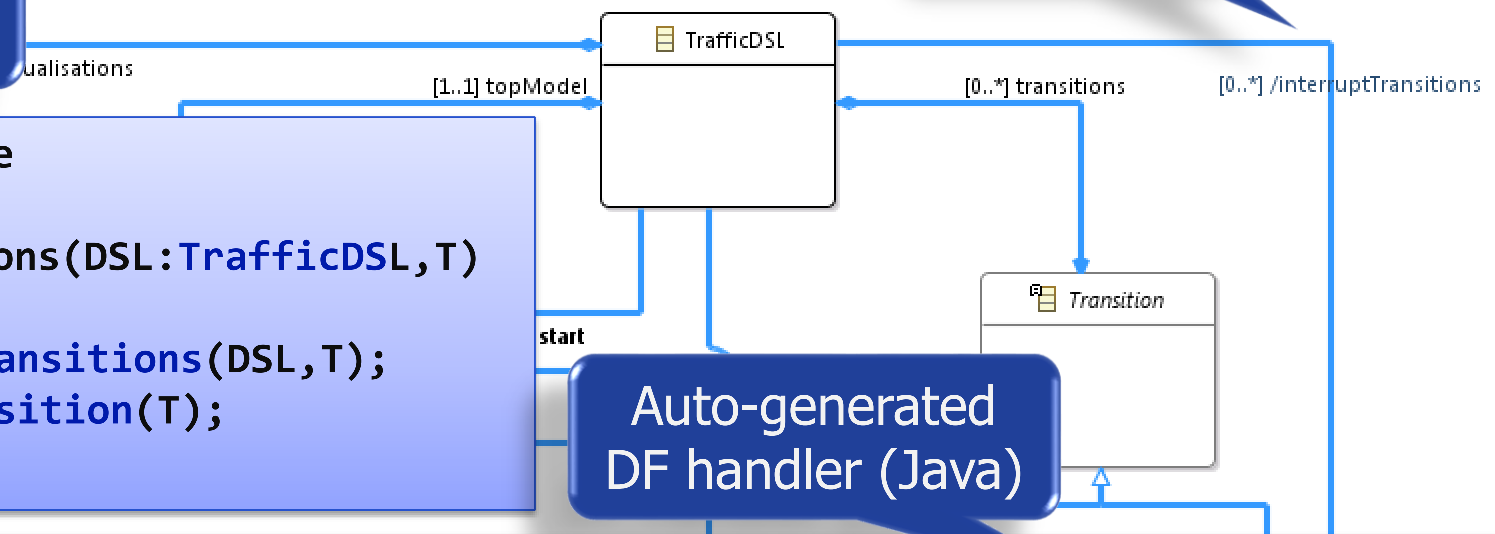


Derived Features:
- Values calculated from other elements
- Defined declaratively as model queries (e.g. OCL, graph queries)
- Tooling: handle as regular EMF elements

Example Handling Derived Features as Queries

Derived Reference

DF specification: as a query



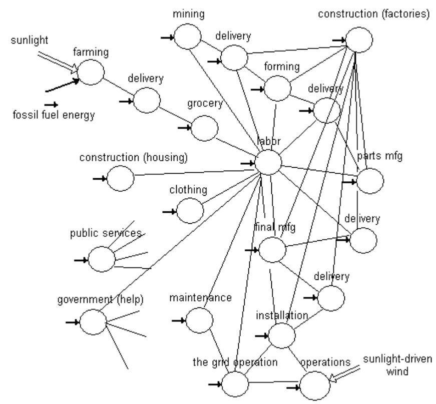

```
@QueryBasedFeature
pattern
interruptTransitions(DSL:TrafficDSL,T)
{
    TrafficDSL.transitions(DSL,T);
    InterruptTransition(T);
}
```

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);}
```

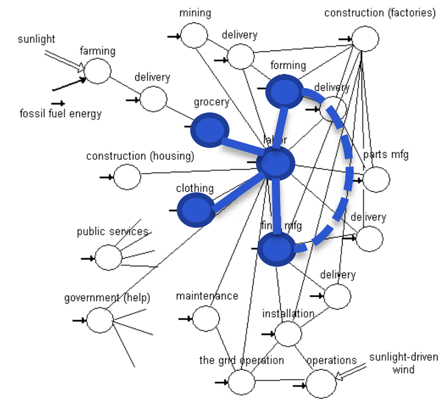
INCQUERY VIEWERS

Live abstractions



Complex model

abstract



Computed overlay
aka. “View”

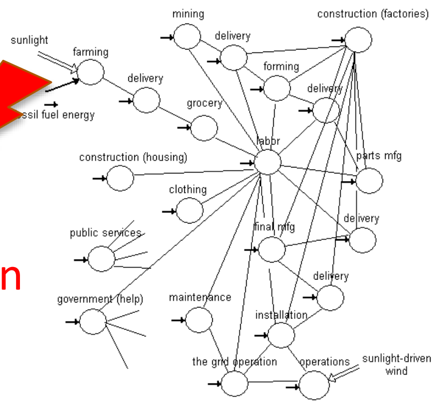
Defined by a **query**

Items = SELECT …

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

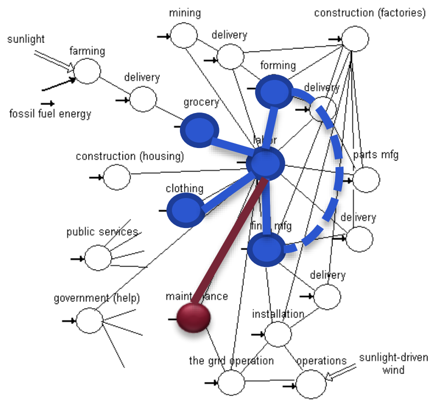
Live abstractions

Model Modification



Complex model

abstract



Computed overlay
aka. “View”

UI update

Change notification

Defined by a **query**

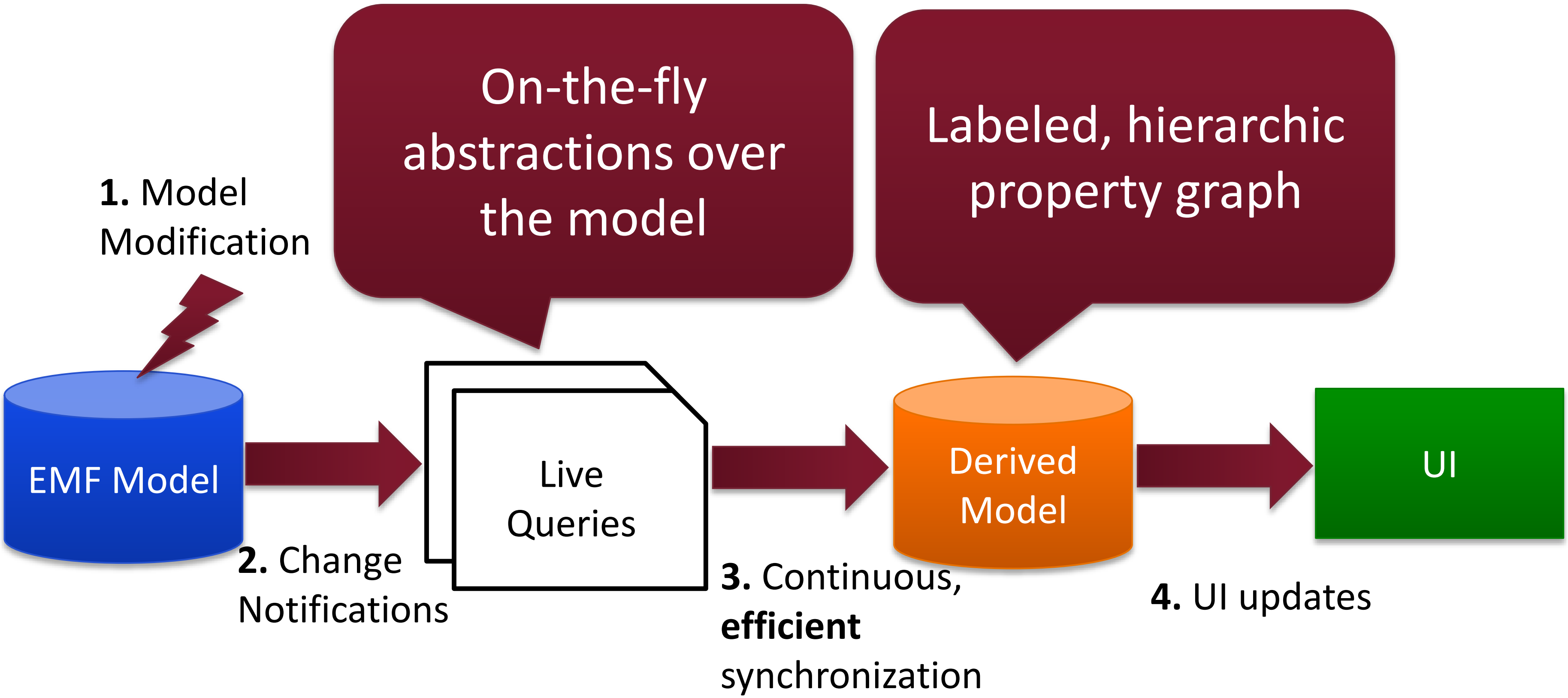
IncQuery

Items = SELECT …

Id	Label	Prop0	Prop1
0	N1	a	B
1	N2	c	D
2	N3	e	F

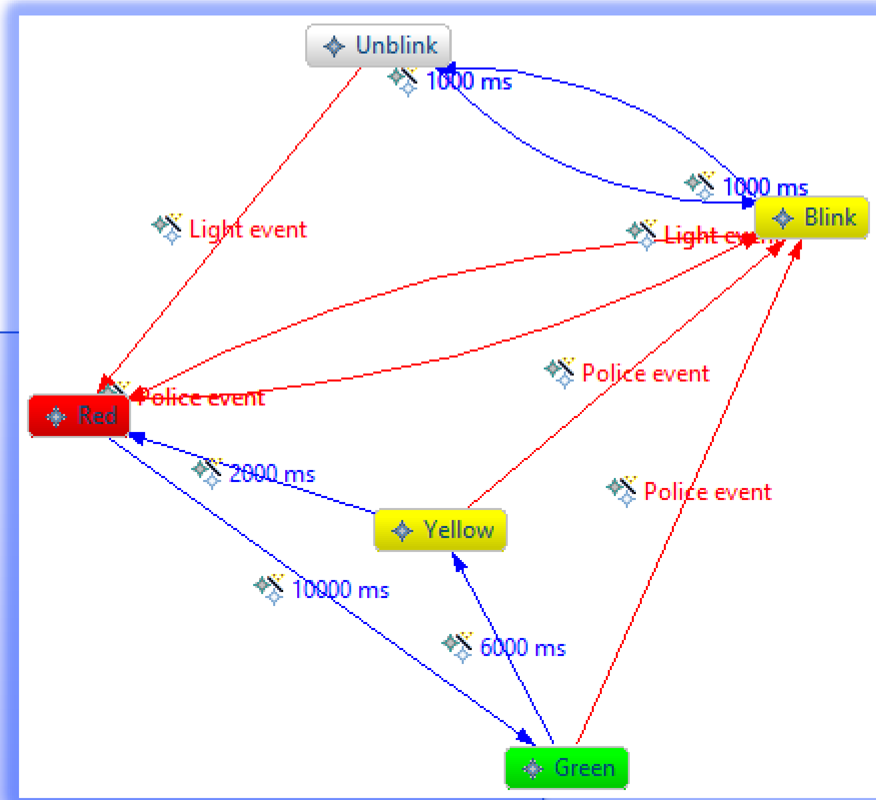
Query result update

INCQUERY 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) = { … }
}
```

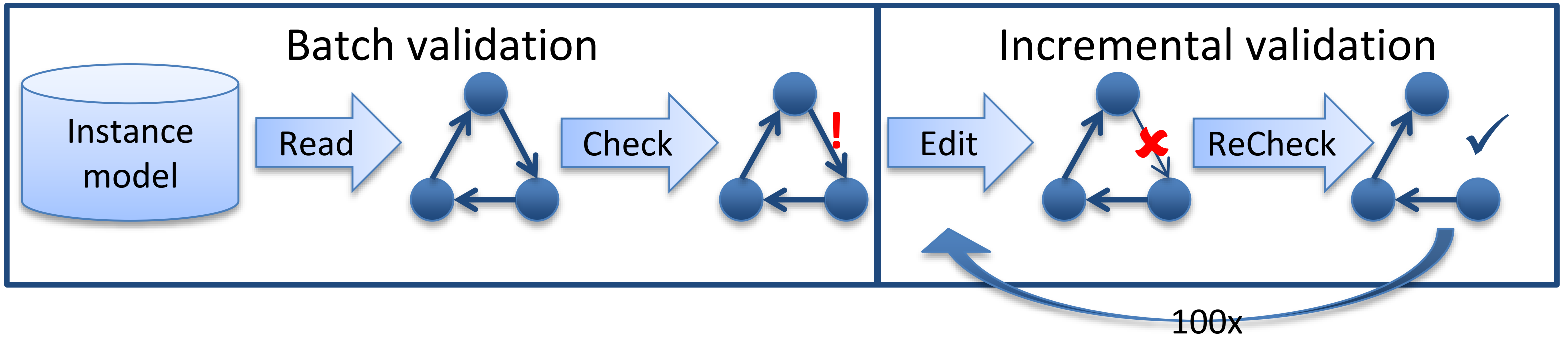
What can I do with all this? – query-based live abstractions

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!	**IncQuery 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?

Drools

Java

Neo4j
the graph database

IncQuery

FRANZ INC.
"Web 3.0's Database"

openRDF.org
Sesame

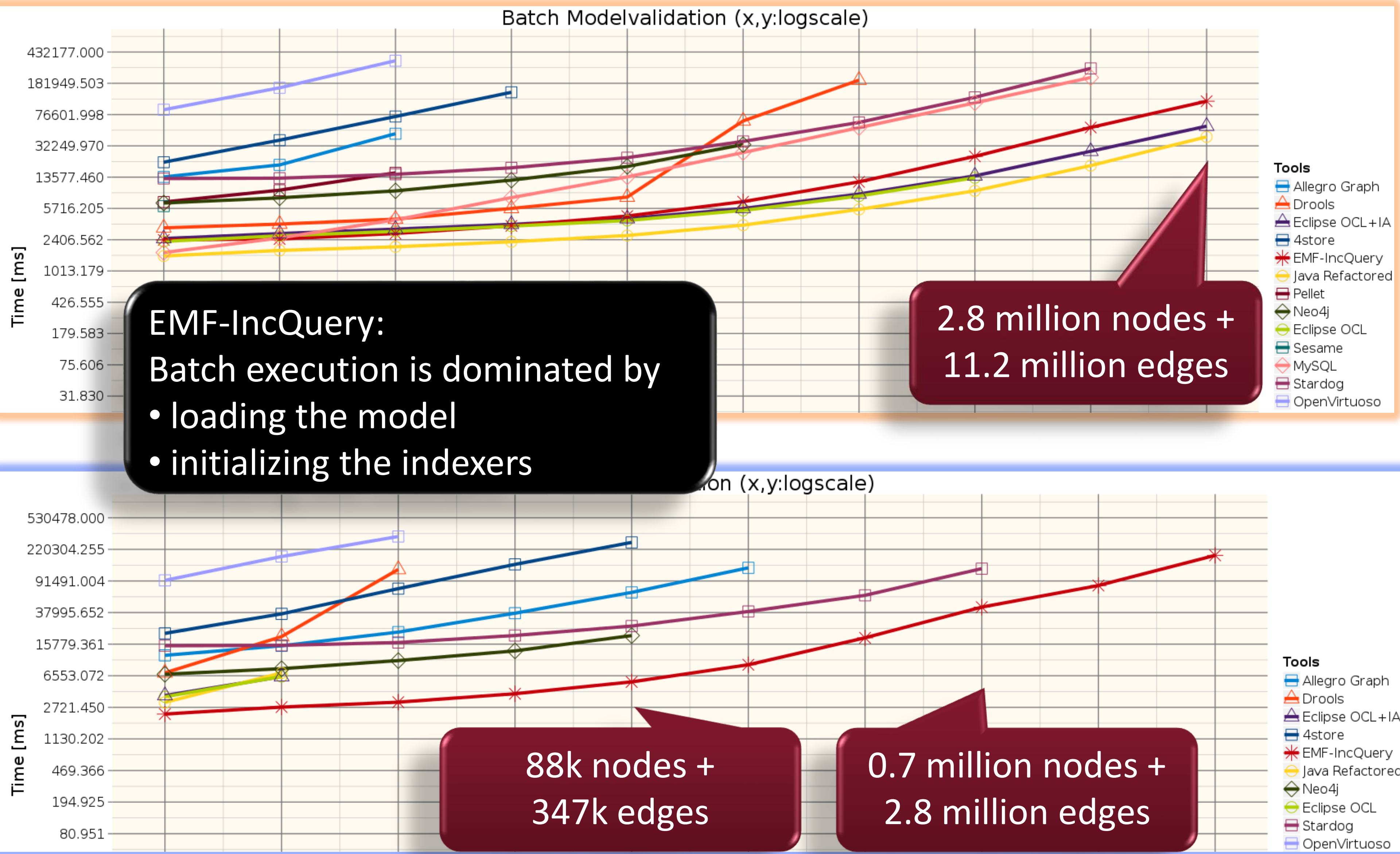
MySQL®

clark&parsia

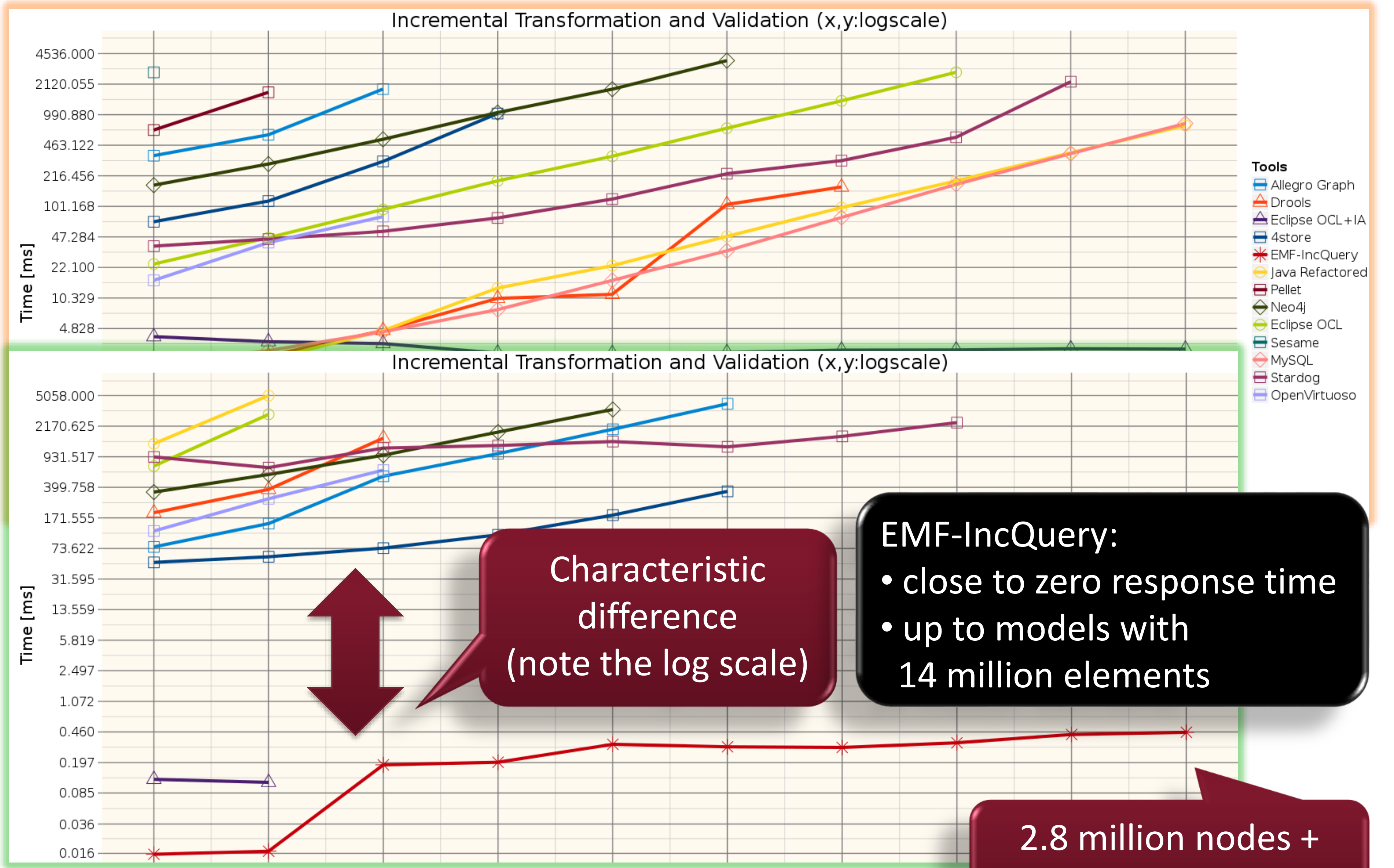
4store

OPENLINK®
VIRTUOSO
UNIVERSAL SERVER

Batch validation runtime (complex queries)

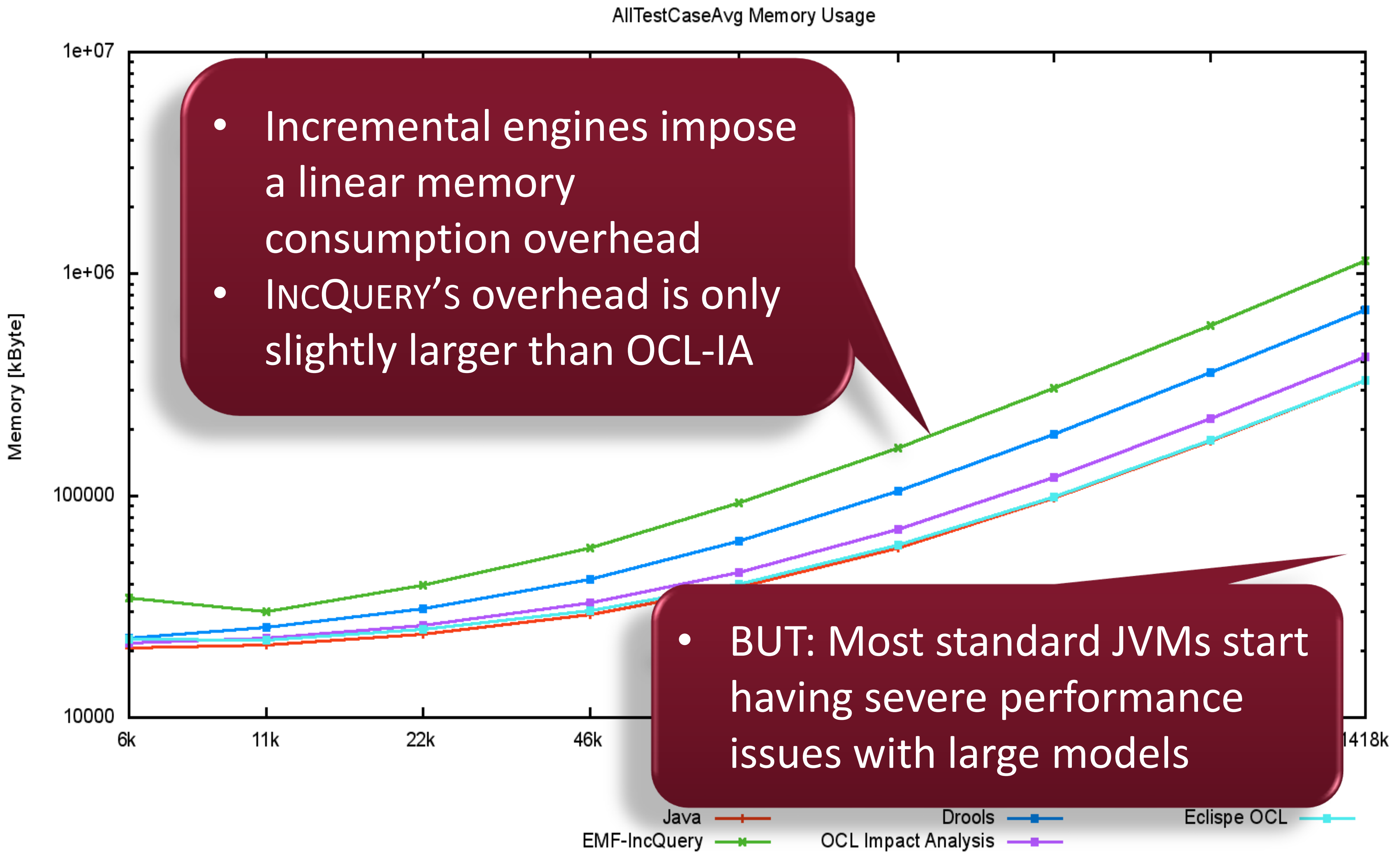


Re-validation time (complex queries)



http://incquery.net/publications/trainbenchmark for more details

Memory usage



- Incremental engines impose a linear memory consumption overhead
- INCQUERY's overhead is only slightly larger than OCL-IA

- BUT: Most standard JVMs start having severe performance issues with large models

CONCLUSIONS

Selected Applications of EMF-IncQuery

- 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

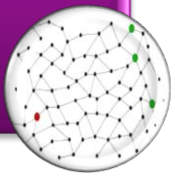
Gesture recognition

- 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

EMF-IncQuery: An Open Source Eclipse Project

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

Definition

IncQuery

- **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

Tooling

http://eclipse.org/incquery