EMF-IncQuery

Incremental evaluation of model queries over EMF models

Gábor Bergmann, Ákos Horváth,
Ábel Hegedüs, Zoltán Ujhelyi, Balázs Polgár,
István Ráth, **Dániel Varró**

Model Driven Software Development
Lecture 11

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

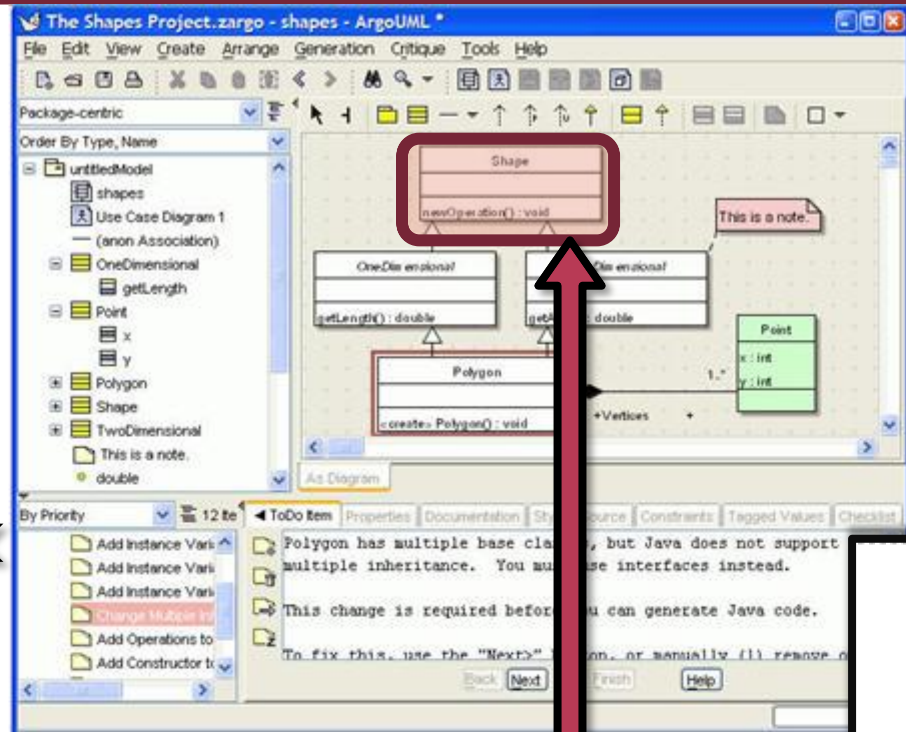
MOTIVATION

First of all…

- What is a model query?
  - A piece of code that looks for certain parts of the model.
- "Mathematically"
  - Query = set of constraints that have to be satisfied by (parts of) the model.
  - Result = set of model elements (element configurations) that satisfy the constraints of the query.
- A query engine?
  - Supports the definition/execution of model queries.

Hi Jane, what do you do at work?

Jane



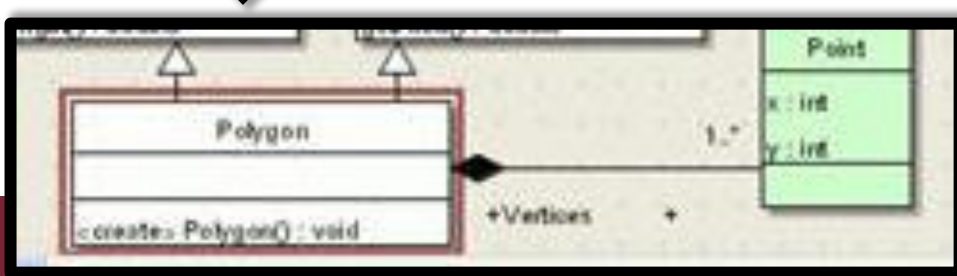
Gen

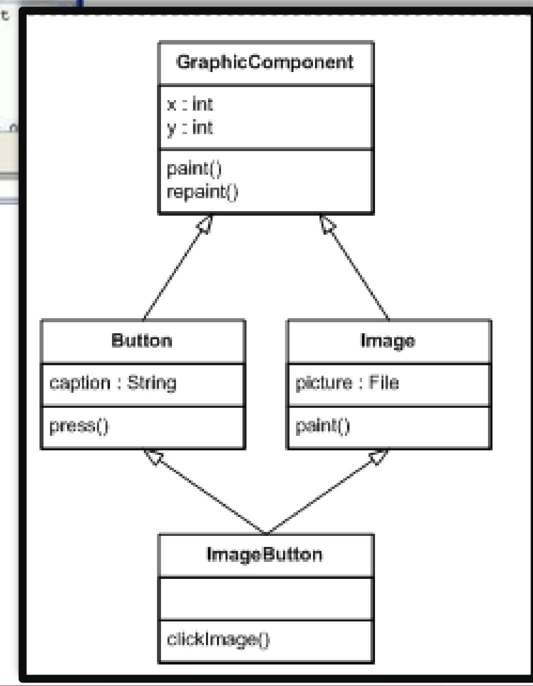
View

Report

Detect

Boss





Constraint

Model queries

- Queries are at the heart of MDD.
  - Views
  - Reports
  - Generators
  - Validators
  - …
- Development issues
  - Complex queries are hard to write

Issues with query development

- 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)

Issues with query execution

- Query performance
  - = Execution time, as a function of
    - Query complexity
    - Model size / contents
    - Result set size
- 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 very well supported feature.

Model query engine wish list

- Declarative query language
  - Easy to learn
  - Good bindings to the imperative world (Java)
  - Safe yet powerful
  - Reusable
- High performance
  - Fast execution for complex queries over large models
  - First-class support for incremental execution
- Technology
  - Works with EMF out-of-the-box

STATE OF THE ART

Problem 1: Expressiveness

- EMF Query (declarative)
  - Low expressiveness
  - Limited navigability
    - no „cycles”
- OCL (declarative)
  - Verbose
  - Lack of reusability support
  - Local constraints of a model element
  - Poor handling of recursion

→ Challenging to use

Problem 2: Incrementality

- Goal: Incremental evaluation of model queries
  - Incremental maintenance of result set
  - Avoid unnecessary re-computation
- Related work:
  - Constraint evaluation (by A. Egyed)
    - Arbitrary constraint description
      - Can be a bottleneck for complex constraints
      - Always local to a model element
    - Listen to model notifications
    - Calculate which constraints need to be reevaluated
  - No other related technology directly over EMF
  - Research MT tools: with varying degrees of support

Problem 3: Performance

- Native EMF queries (Java program code): Lack of
  - Reverse navigation along references
  - Enumeration of all instances by type
  - Smart Caching

- Scalability of (academic) MT tools
  - Queries over >300K model elements (several proofs): FUJABA, VIATRA2 (Java), GrGEN, VMTS (.NET), Egyed's tools

EMF-IncQuery

- **Expressive** declarative query language by graph patterns
- **Incremental** cache of matches (materialized view)
- **High performance** for large models

INCQUERY TECHNOLOGY OVERVIEW

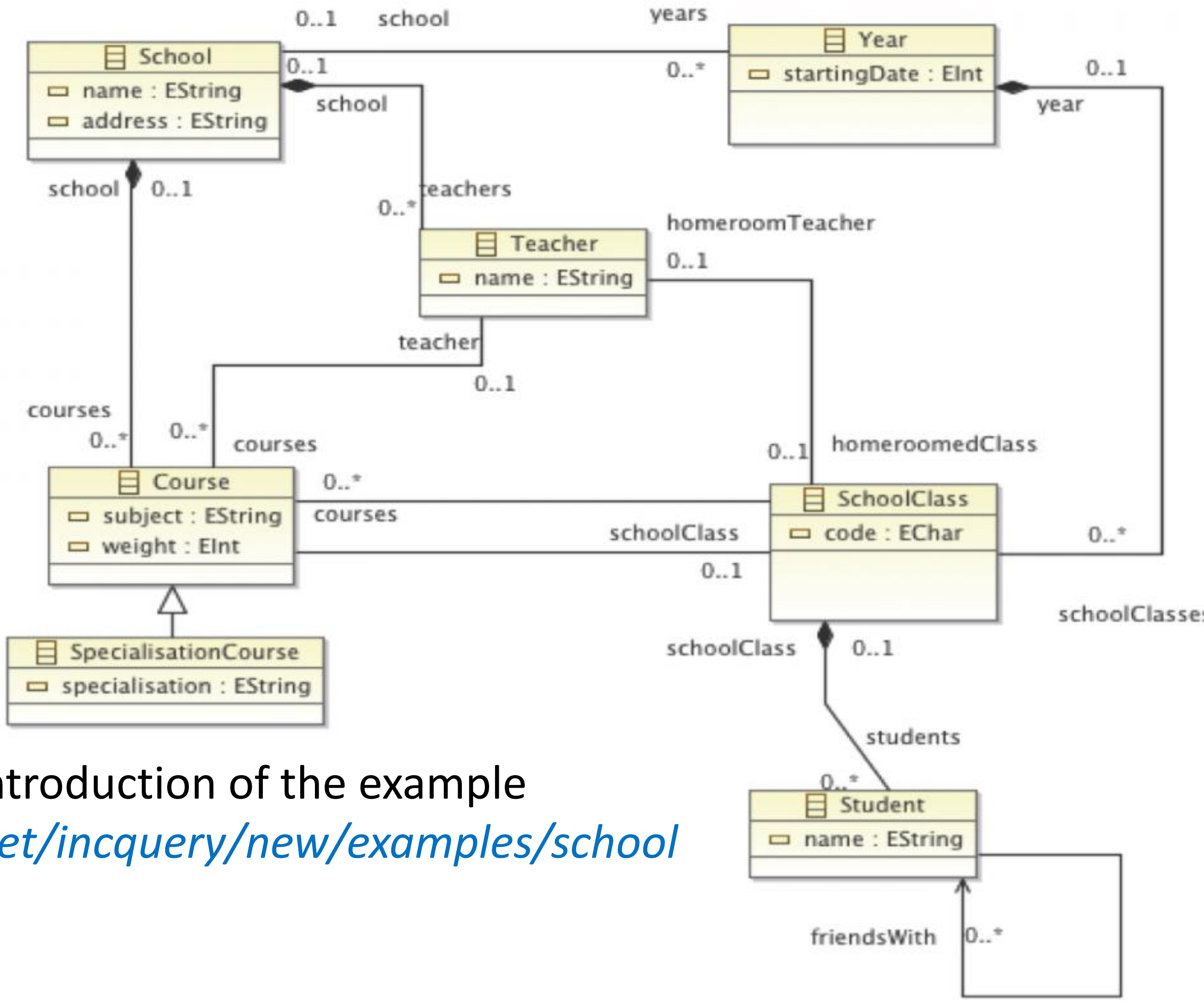
Technology Overview

- What is EMF-INCQuery?
  - Query language + incremental pattern matcher + development tools for EMF models
    - Works with any *(pure)* EMF application
    - Reusability by pattern composition
    - Arbitrary recursion, negation
    - Generic and parameterized model queries
    - Bidirectional navigability
    - Immediate access to all instances of a type
    - Complex change detection
- Benefits
  - Fully declarative + Scalable performance

Contributions

- **Expressive** declarative query language by graph patterns
  - Capture local + global queries
  - Compositionality + Reusabilility
  - Transitive closure, Negation
- **Incremental** cache of matches (materialized view)
- **High performance** for large models

Example: School metamodel



- Detailed introduction of the example
  incquery.net/incquery/new/examples/school

IQPL - Simple queries

```
pattern schools(Sch) = {
School(Sch);
}

pattern teachers(T) = {
Teacher(T);
}

pattern teachersOfSchool(T:Teacher,Sch:School) = {
School.teachers(Sch,T);
}
pattern studentOfSchool(S:Student,Sch:School) = {
Student.schoolClass.courses.school(S,Sch);
}
```

IQPL - Simple queries

Query definition

Query parameters

```
pattern schools(Sch) = {
School(Sch);
}

pattern teachers(T) = {
Teacher(T);
}

pattern teachersOfSchool(T:Teacher,Sch:School) = {
School.teachers(Sch,T);
}
pattern studentOfSchool(S:Student,Sch:School) = {
Student.schoolClass.courses.school(S,Sch);
}
```

Type constraints

Syntactic Sugar

Navigation – no restcitions on the navigation!

Path expression

IQPL – pattern composition and NAC

```
pattern coursesOfTeacher(T:Teacher, C:Course) = {
Teacher.courses(T,C);
}

pattern classesOfTeacher(T, SC) = {
find coursesOfTeacher(T,C);
Course.schoolClass(C,SC);
}
pattern teacherWithoutClass(T:Teacher) = {
neg find classesOfTeacher(T,SC);
}
```

IQPL – pattern composition and NAC

Pattern call

```
pattern coursesOfTeacher(T:Teacher, C:Course) = {
Teacher.courses(T,C);
}

pattern classesOfTeacher(T, SC) = {
find coursesOfTeacher(T,C);
Course.schoolClass(C,SC);
}
pattern teacherWithoutClass(T:Teacher) = {
neg find classesOfTeacher(T,SC);
}
```

Automatic type inference – type constraints can be omitted

Negative application call

IQPL – transitive closure and disjunction

```
pattern friendlyTo(S1:Student, S2:Student) = {
Student.friendsWith(S1,S2);
} or {
Student.friendsWith(S2,S1);
}

pattern inTheCircleOfFriends(S1:Student,Someone:Student) = {
find friendlyTo+(S1,Someone);
S1!=Someone; // we do not allow self loops
}
pattern moreFriendsThan(S1 : Student, S2 : Student) {
N == count find inTheCircleOfFriends(S1, _Sx1);
M == count find inTheCircleOfFriends(S2, _Sx2);
check(N > M);
}
pattern theOnesWithTheBiggestCircle(S:Student) = {
neg find moreFriendsThan(Sx,S);
}
```

IQPL – transitive closure and disjunction

```
pattern friendlyTo(S1:Student, S2:Student) = {
Student.friendsWith(S1,S2);
} or {
Student.friendsWith(S2,S1);
}

pattern inTheCircleOfFriends(S1:Student,Someone:Student) = {
find friendlyTo+(S1,Someone);
S1!=Someone; // we do not allow self loops
}
pattern moreFriendsThan(S1 : Student, S2 : Student) {
N == count find inTheCircleOfFriends(S1, _Sx1);
M == count find inTheCircleOfFriends(S2, _Sx2);
check(N > M);
}
pattern theOnesWithTheBiggestCircle(S:Student) = {
neg find moreFriendsThan(Sx,S);
}
```

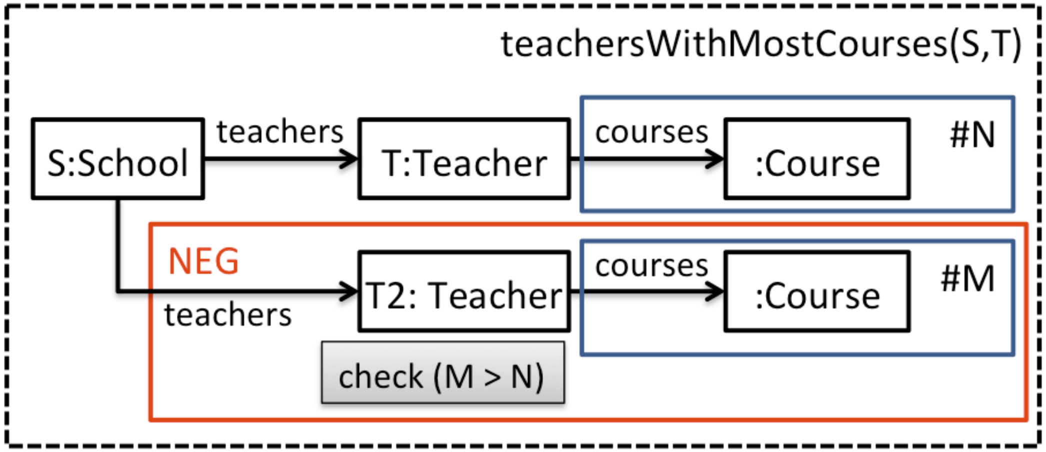
Disjunction

Transitive closure

Check expression

Example application of effective NAC application

IQPL - Advanced model query

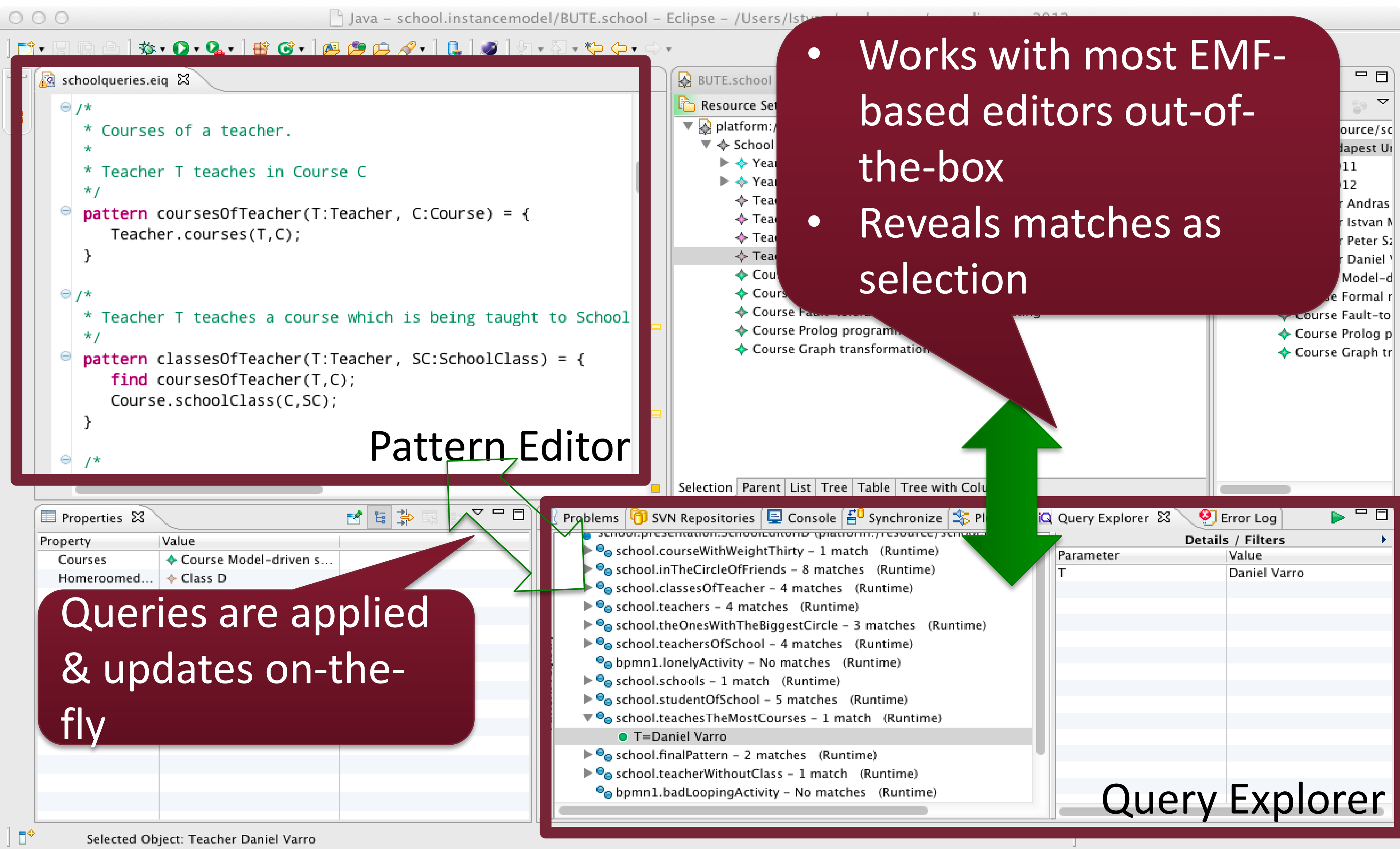



```
pattern teachersWithMostCourses(
 School : School, Teacher : Teacher) = {
   School.teachers(School,Teacher);
   neg find moreCourses(Teacher);}

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

INCQUERY Development Tools



Contributions

- **Expressive** declarative query language by graph patterns
  - Capture local + global queries
  - Compositionality + Reusabilility
  - Transitive closure, Negation
- **Incremental** cache of matches (materialized view)
  - Cheap maintenance of cache (only memory overhead)
  - Notify about relevant changes (new match – lost match)
  - Enable reactions to complex structural events
- **High performance** for large models

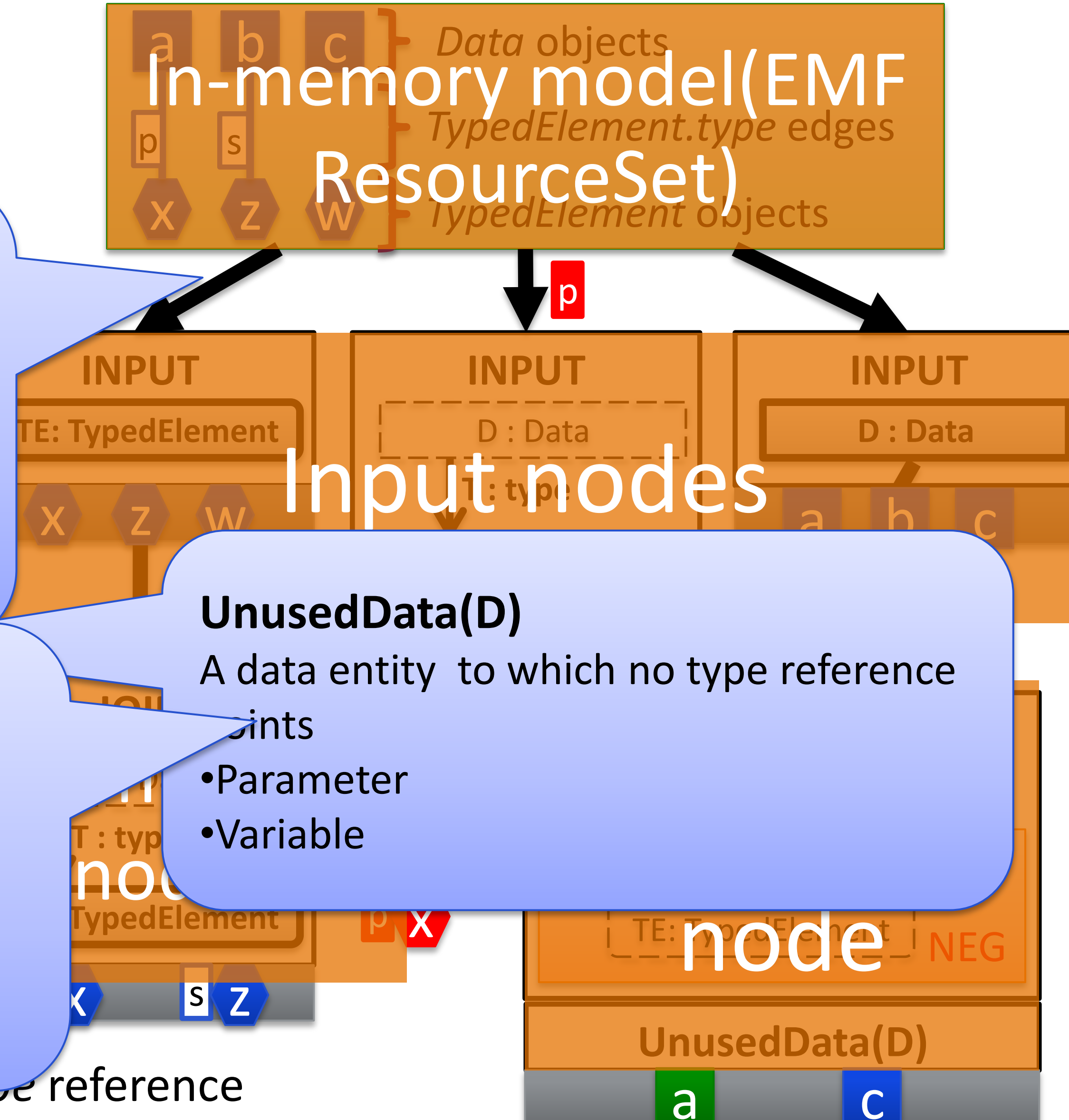
RETE nets

- RETE network
  - node: (partial) matches of a (sub)pattern

Notification

Transparent: user modification, model imports, results of a transformation, external modification, …
→ RETE is always updated!

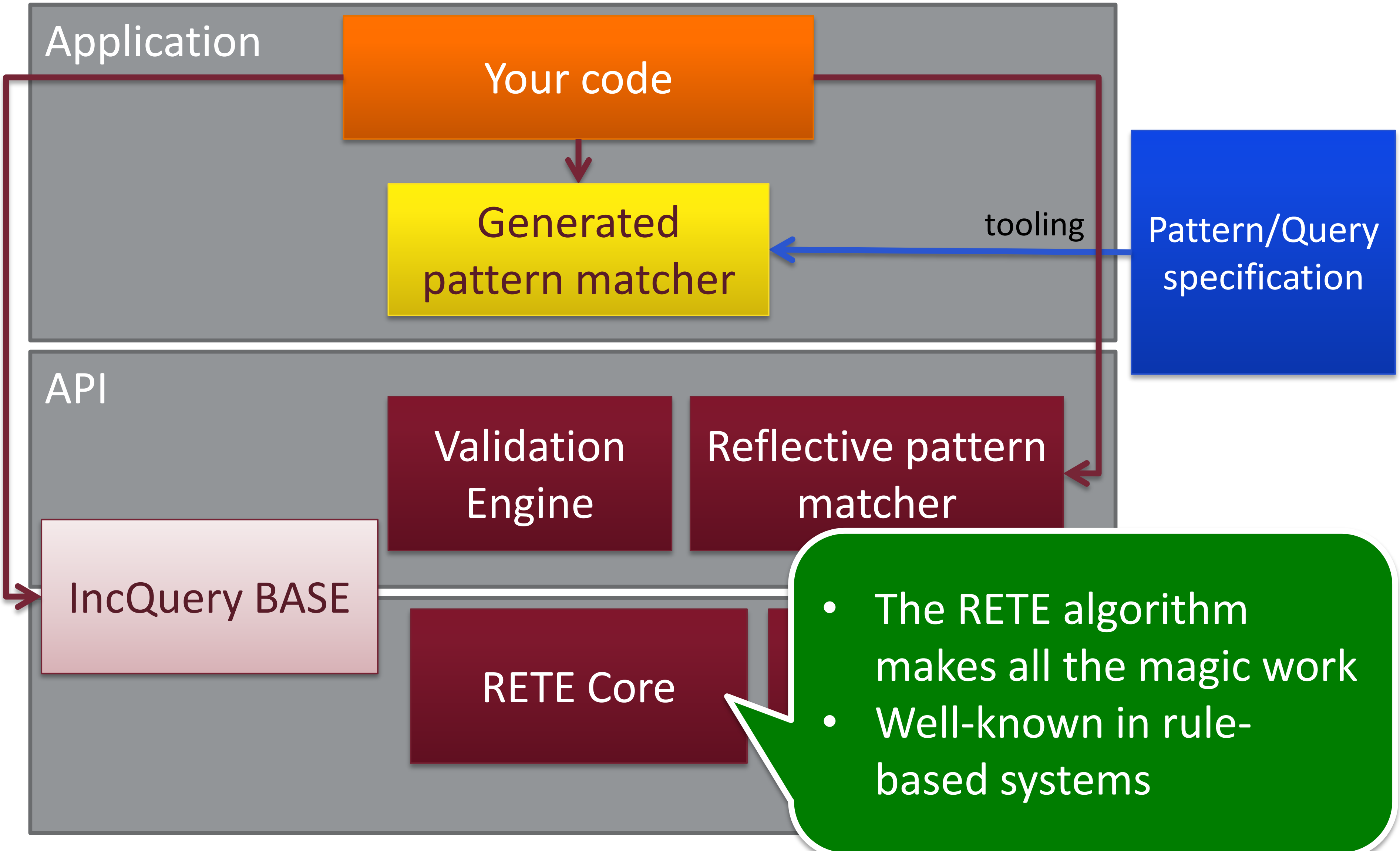
Experimental results: good, if…
- There is enough memory
- Transactional model manipulation



UnusedData(D)
A data entity to which no type reference points
•Parameter
•Variable

- change: delete/retarget *type* reference

EMF-IncQuery Architecture v0.7

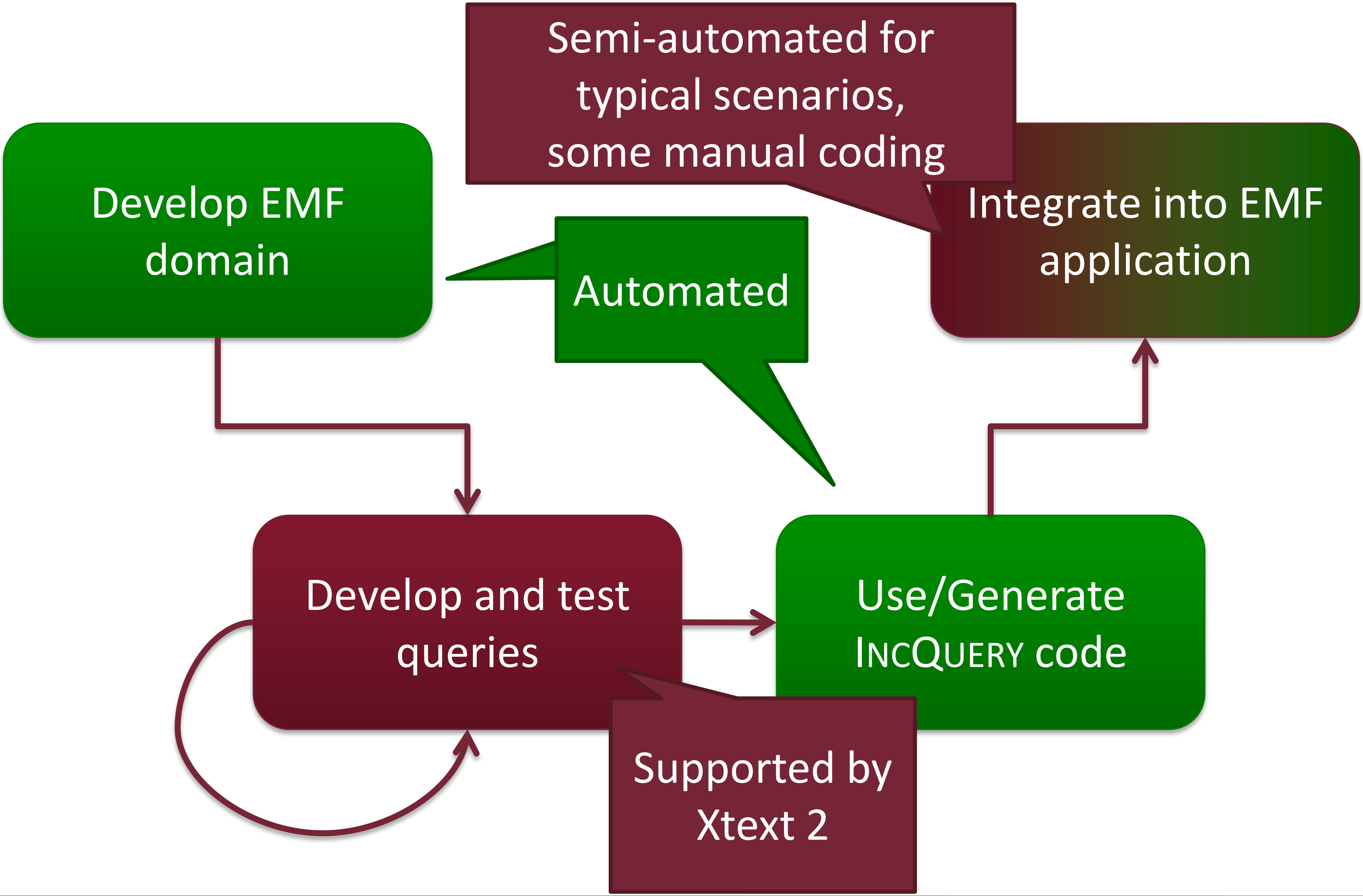


- The RETE algorithm makes all the magic work
- Well-known in rule-based systems

IncQuery BASE

- Light-weight Java library for simple (yet very powerful) EMF model 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

Development workflow



Generated pattern matchers

- INCQuery runtime
  - Eclipse plugin
    - Depends only on EMF and the INCQuery core
    - No VIATRA2 dependency!
  - Private code: pattern builders
    - Parameterize the RETE core and the generic EMF PM library
  - Public API: Pattern matcher access layer
    - Query interfaces
    - Data Transfer Objects (DTOs)
    - Used to integrate to EMF applications

Generated Sample UI

- Command contributions
  - Project explorer, Navigation, Package Explorer
  - Perform model loading and query execution
  - Display the results on the UI
    - List (default)
      - Pretty prints a list of matches
    - Counter
      - Prints the number of matches

IncQuery Runtime

Generic Query API

Generic Change API

Generated Query API

Generated Change API

Generated Query API

- Basic queries
  - Uses tuples (object arrays) corresponding to pattern parameters
  - `Object[] getOneMatch()`
  - `Collection<Object[]> getAllMatches()`
- Parameterized queries
  - `getOneMatch(Object X, Object Y, …)`
  - `getAllMatches(Object X, Object Y, …)`
  - Null input values = unbound input variables

Based on pattern signature

Query Signatures

- **D**ata **T**ransfer **O**bjects generated for pattern signatures

```
pattern classesOfTeacher(T, SC) =
{
find coursesOfTeacher(T,C);
Course.schoolClass(C,SC);
}
```

- Signature query methods
  - `classesOfTeacherSignature getOneMatch()`
  - `classesOfTeacherSignature getOneMatchAsSignature(Object T, Object SC)`
  - `Collection< classesOfTeacherSignature> getAllMatchesAsSignature()`
  - `Collection< classesOfTeacherSignature> getAllMatchesAsSignature(Object T, Object SC)`

```
public class
   classesOfTeacherSignature
{
   Object T;
   Object SC;
}
```

- T, SC: EObjects or datatype instances (String, Boolean, …)

Query Signatures

- **D**ata **T**ransfer **O**bjects generated fo
  pattern signatures

```
pattern classesOfTeacher(T, SC) =
{
find coursesOfTeacher(T,C);
Course.schoolClass(C,SC);
}
```

- Signature query methods
  - `classesOfTeacherSignature getOneMatch()`
  - `classesOfTeacherSignature getOneMatchAsSignature(Teacher T, SchoolClass SC)`
  - `Collection< classesOfTeacherSignature> getAllMatchesAsSignature()`
  - `Collection< classesOfTeacherSignature> getAllMatchesAsSignature(Teacher T, SchoolClass SC)`

```
public class
   classesOfTeacherSignature
{
   Teacher T;
   SchoolClass SC;
}
```

- T, SC: EObjects or datatype instances (String, Boolean, …)

Integrating into EMF applications

- Pattern matchers may be initialized for
  - Any EMF Notifier
    - e.g. Resources, ResourceSets
  - (TransactionalEditingDomains)
- Initialization
  - Possible at any time
  - Involves a **single** exhaustive model traversal (independent of the number of patterns, pattern contents etc.)

Typical programming patterns

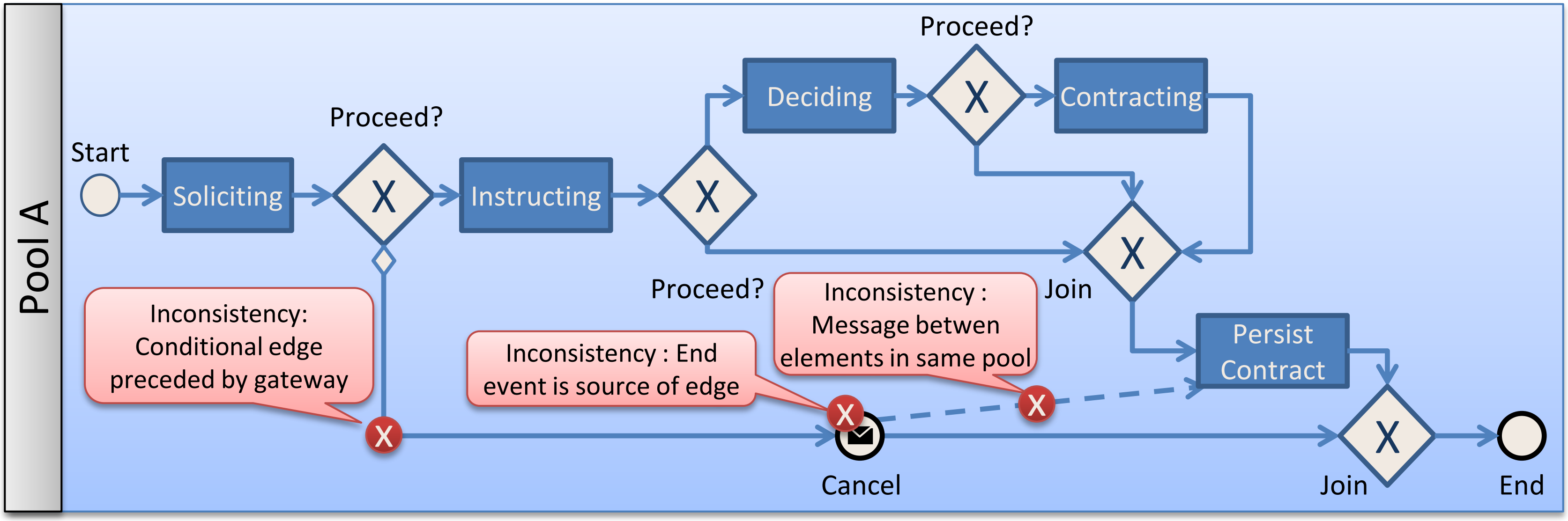
1. Initialize EMF model
   - Usually already done by your app ☺
2. Initialize incremental PM whenever necessary
   - Typically: at loading time
3. Use the incremental PM for queries
   - Model updates will be processed transparently, with minimal performance overhead
   - Delta monitors can be used to track complex changes
4. Dispose the PM when not needed anymore
   - + Frees memory
   - - Re-traversal will be necessary

BPMN well-formedness rules

- Traditionally specified by OCL constraints
  - OCL constraints can be attached to any EMF instance model via EMF Validation
- Rules specified by
  - **Tool developers**
  - (End users)

Well-formedness checking from a tool developer's perspective

- Well-formedness rules
  - Express constraints not (easily) possible by metamodeling techniques
  - Ensure "sane" modeling conventions & best practices
  - Aid code generation by design-time validation
- Example:



IncQuery Validation Engine

- 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

How it works – IncQuery Change API

- Track changes in the match set of patterns (new/lost)
- Delta monitors
  - May be initialized at any time
  - `DeltaMonitor.matchFoundEvents` / `DeltaMonitor.matchLostEvents`
    - Queues of matches (tuples/Signatures) that have appeared/disappeared since initialization
- Typical usage
  - Listen to model manipulation (transactions)
  - After transaction commits:
    - Evaluate delta monitor contents and process changes
    - Remove processed tuples/Signatures from .matchFound/LostEvents

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 A simple BPMN validation constraint

- "All Behaviors must have an Operation as their specification."
  - Otherwise they do not have any "interface" through which they could be accessed → "dead code"
- Bad case:

```
pattern loopingActivity(A : Activity)= {
Activity.looping(A, true);
}

@Constraint(location = "A", message = "$A.name$ is a bad looping
activity", severity = "warning" )
pattern badLoopingActivity(A : Activity)= {
find loopingActivity(A);
Activity.name(A, Name);
check(!(Name as String).startsWith("Loop"));
}
```

EXAMPLE A simple BPMN validation constraint

- "All Behaviors must have an Operation as their specification."
  - Otherwise they do not have any "interface" through which they could be accessed → "dead code"
- Bad case:

```
pattern loopingActivity(A : Activity)={
Activity.looping(A, true);
}

@Constraint(location = "A", message = "$A.name$ is a bad looping
activity", severity = "warning" )
pattern badLoopingActivity(A : Activity)= {
find loopingActivity(A);
Activity.name(A, Name);
check(!(Name as String).startsWith("Loop"));
}
```

Identify pattern variable "Activity" as the location

Path expression

Generated Sample Validation project

- Java classes: Constraint descendants
- Plugin.xml
  - Constraint registration
  - UI integration
    - Editor ID from genmodel

Validation lifecycle

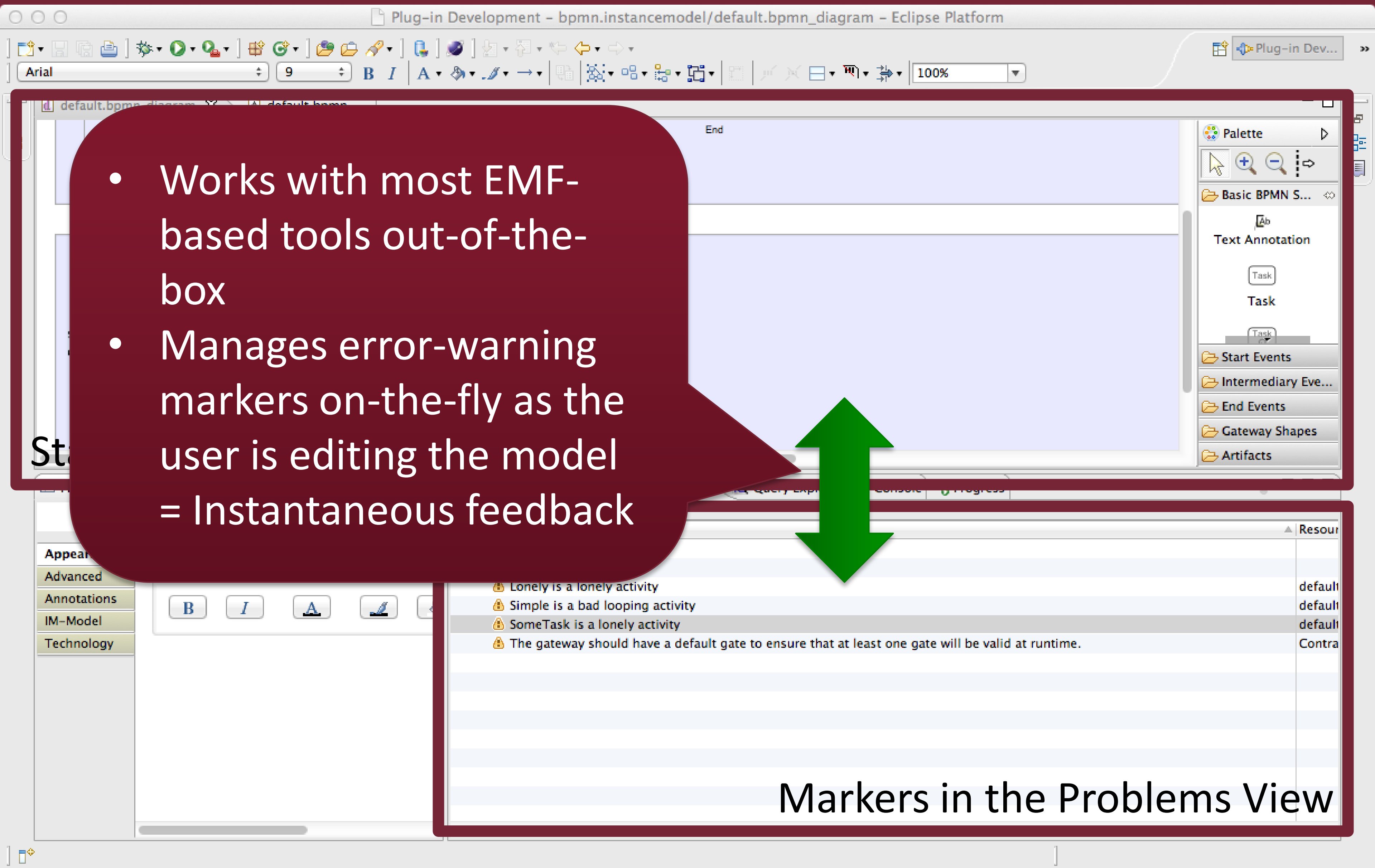
- Constraint violations
  - Represented by Problem Markers (Problems view)
  - Marker text is updated if affected elements are changed in the model
  - Marker removed if violation is no longer present
- Lifecycle
  - Editor bound validation (markers removed when editor is closed)
  - Incremental maintenance 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

EXAMPLE GUI - INCQUERY 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



PERFORMANCE BENCHMARKING

Challenges

- Performance evaluations are hard
  - Fair?
  - Reliable?
  - Reproducible?
  - Can the results be generalized?
- Benchmark example:
  on-the-fly constraint validation over AUTOSAR models
  - Conference presentation at MODELS 2010
  - Motivation: the Embedded Architect Tool of OptXware Ltd.
  - AUTOSAR models can be very large (>>1m elements)

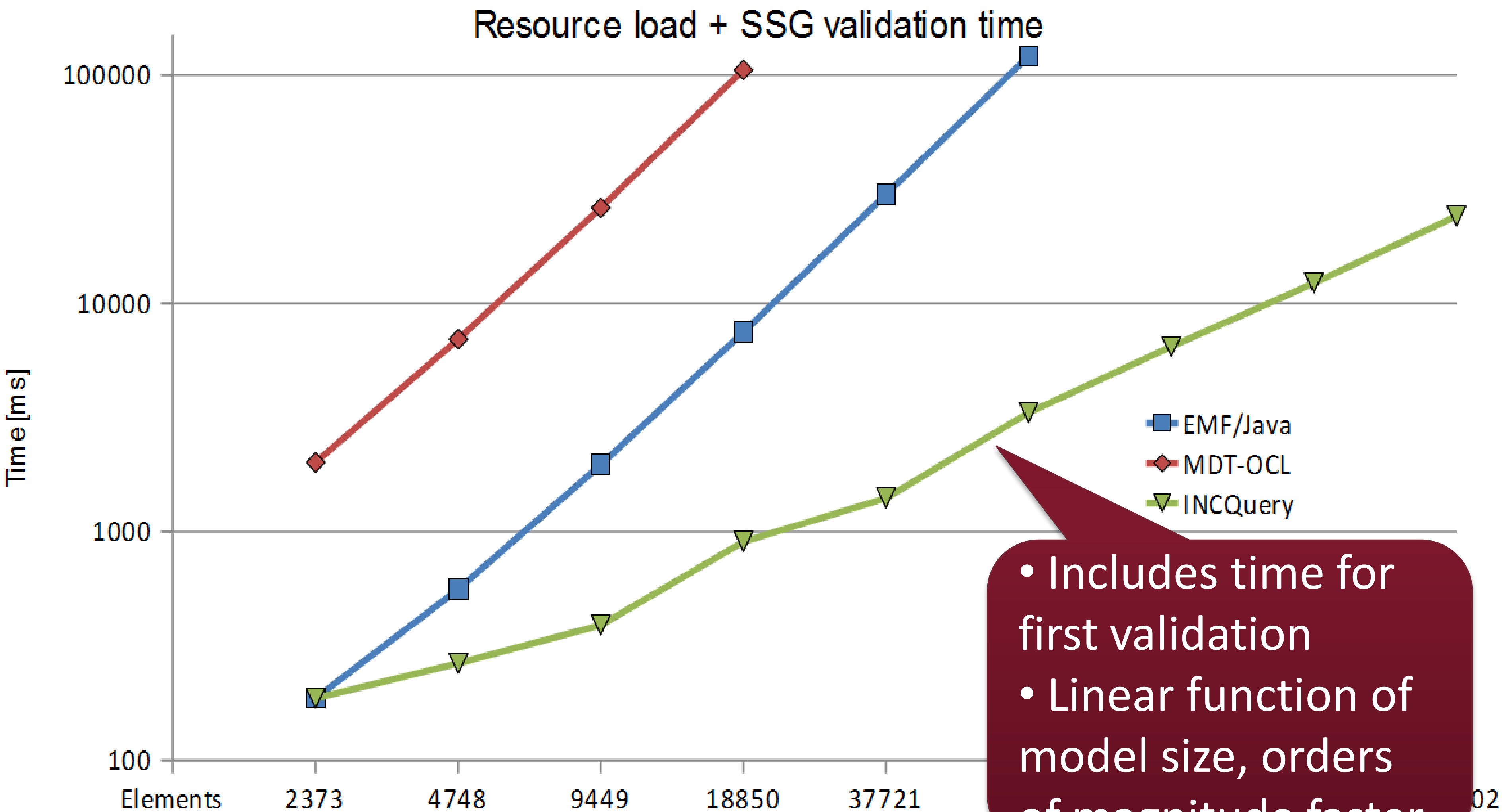
What is measured?

- Sample models were generated
  - matches are scarce relative to overall model size
- On-the-fly validation is modeled as follows:
  1. Compute initial validation results
  2. Apply randomly distributed, small changes
  3. Re-compute validation results
- Measured: execution times of
  - Initialization (model load + RETE construction)
  - Model manipulation operations (negligible)
  - Validation result (re)computation
- Compared technologies
  - MDT-OCL
  - Plain Java code that an average developer would write

IncQuery Results

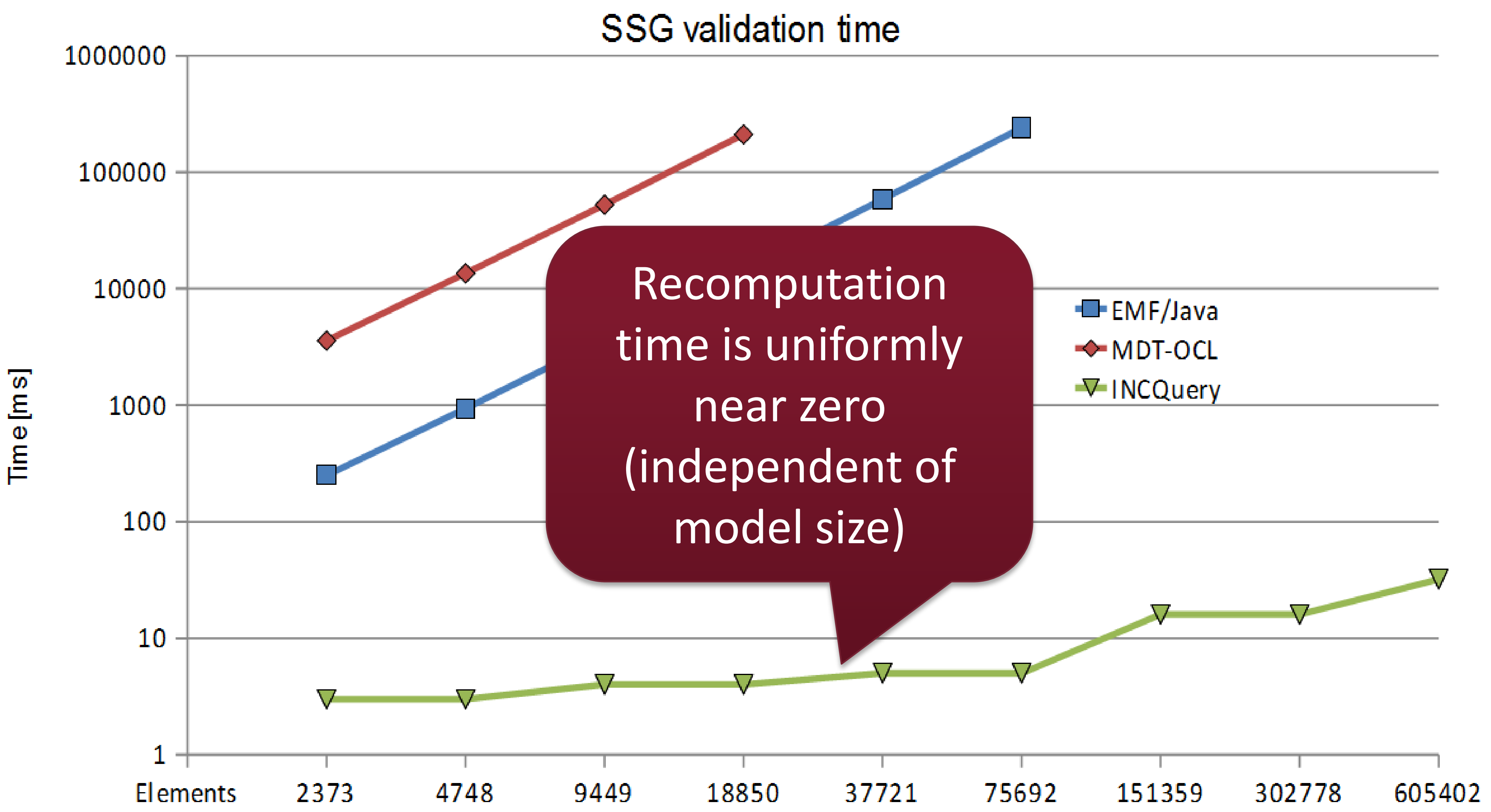
- Hardware: normal desktop PC (Core2, 4GB RAM)
- Model sizes up to 1.5m elements
- Initialization times (resource loading + first validation)
  - <1 sec for model sizes below 50000 elements
  - Up to 40 seconds for the largest model (grows linearly with the model size)
- Recomputation times
  - Within error of measurement (=0), **independent of model size**
  - **Retrieval of matches AND complex changes is instantaneous**
- Memory overhead
  - <50 MB for model sizes below 50000 elements
  - Up to 1GB for the largest model (grows linearly with model size)
- How does it compare to native code / OCL?

Initialization time



- Includes time for first validation
- Linear function of model size, orders of magnitude faster

Recomputation time



Performance overview

SSG and iSignal validation pattern in model family A

		EMF/Java			MDT-OCL			INCQuery			
Model Elements #	Model size [MB]	Res [s]	iSignal [s]	SSG [s]	Res [s]	iSignal [s]	SSG [s]	Res [s]	iSignal [s]	SSG [s]	Mem OH [MB]
2 373	30	0.06	0.00	0.25	0.13	0.16	3.58	0.17	0.00	0.00	3
4 748	31	0.08	0.00	0.94	0.16	0.17	13.53	0.22	0.00	0.00	6
9 449	32	0.13	0.01	3.67	0.20	0.19	52.48	0.30	0.00	0.00	12
18 850	33	0.22	0.01	14.52	0.30	0.22	210.48	0.45	0.01	0.00	22
37 721	37	0.42	0.01	58.56	0.47	0.27		0.75	0.01	0.01	45
75 692	43	0.78	0.02	239.53	0.86	0.33		1.58	0.01	0.01	92
151 359	55	1.81	0.03		1.84	0.53		3.22	0.02	0.02	187
302 778	81	3.63	0.06		3.64	0.88		6.19	0.02	0.02	373
605 402	135	7.14	0.09		7.48	1.63		12.00	0.02	0.03	746

Channel validation pattern in model family B

		EMF/Java		MDT-OCL		INCQuery		
Model Elements #	Model size [MB]	Res [s]	Channel [s]	Res [s]	Channel [s]	Res [s]	Channel [s]	Mem OH [MB]
2 972	30	0.06	0.00	0.14	0.17	0.19	0.00	2
6 237	31	0.09	0.02	0.16	0.22	0.27	0.00	4
12 708	32	0.16	0.00	0.25	0.31	0.38	0.00	8
24 885	34	0.28	0.03	0.34	0.33	0.89	0.00	14
47 228	38	0.49	0.06	0.53	0.48	1.28	0.00	28
90 586	44	1.13	0.09	1.20	0.80	2.41	0.00	55
180 389	58	1.94	0.19	2.05	1.41	4.56	0.00	111
370 660	91	4.06	0.39	4.08	2.50	9.00	0.00	225
752 172	156	8.09	0.80	8.11	5.00	20.38	0.00	456
1 558 100	295	17.28	1.59	17.39	10.13	40.22	0.00	943

Legend: Res – resource loading time
Mem OH – memory overhead

Assessment of the benchmark

- High performance complex queries are hard to write manually in Java.
- IncQuery can do the trick nicely as long as you have enough RAM.
- Metamodel structure has huge impact on performance when using “conventional” query technologies such as OCL, due to
  - Lack of reverse navigation
  - Lack of type enumeration (.allInstances())

Contributions

- **Expressive** declarative query language by graph patterns
  - Capture local + global queries
  - Compositionality + Reusabilility
  - Transitive closure, Negation
- **Incremental** cache of matches (materialized view)
  - Cheap maintenance of cache (only memory overhead)
  - Notify about relevant changes
  - Enable reactions to complex structural events
- **High performance** for large models
  - Linear overhead for loading
  - Instant response for queries
  - > 1 million model elements (on a desktop PC)

CONCLUSION

Closing thoughts

- On-the-fly validation is *only one scenario*
  - Early model-based analysis
  - Language engineering in graphical DSMLs
  - Soft-inter connections
  - Model execution/analysis (stochastic GT)
  - Tool integration
  - Model optimization / constraint solving
  - Design Space Exploration
  - …
- The tutorial examples
  - Do not make use of advanced features such as parameterized queries or complex structural constraints (recursion)
  - Are meant only as a starting point
  - The project website has many more examples!

Model transformations based on IncQuery

- High performance model transformations
  - Hybrid query approach
    - Use IncQuery for
      - Complex queries
      - Frequently used queries
      - Parameterized queries
    - Plain Java for simple queries (saves RAM)
  - Java for control structure & model manipulation
- High-level transformation languages (VIATRA2, ATL, Epsilon, …) could be “compiled” to run on this infrastructure
- Ongoing research: automatic mapping of SPARQL, OCL & others to the IncQuery language

~~Wish list~~ IncQuery features ☺

- Declarative query language
  - Easy to learn
  - Good bindings to the imperative world (Java)
  - Safe yet powerful
  - Reusable
- High performance
  - Fast execution for complex queries over large models
  - First-class support for incremental execution
- Technology
  - Works with EMF out-of-the-box

Pointers

- Pointers
  - Eclipse webpage:
    - http://www.eclipse.org/incquery/
  - „Official webpage”
    - https://incquery.net/
      - Documentation, language reference
      - Tutorials
      - Examples
      - Source code
      - …