EMF-IncQUERY
Incremental evaluation of model queries

Model Driven Systems Development
Lecture 04
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

Abstract

- `type`

Meta-model
- `Signal`
  - `actualState : SignalStateKind`
  - `1 + entry`
  - `1 + exit`
- `Route`
  - `1 + route`
  - `+ routeDefinition 2..*`
- `Sensor`
  - `+ sensor`
  - `+ trackElement`
  - `+ connectsTo`
- `SignalStateKind`
  - `STOP`
  - `FAILURE`
  - `GO`
- `SwitchStateKind`
  - `FAILURE`
  - `LEFT`
  - `RIGHT`
  - `STRAIGHT`
- `SwitchPosition`
  - `switchState : SwitchStateKind`
  - `+ switchPosition`
- `Switch`
  - `actualState : SwitchStateKind`
  - `1 + switch`
- `Segment`
  - `length : Elnt`

Model

- `Abstract`
Domain-specific modeling languages

Meta-model

Model

Query

pattern switchWOSignal(sw) {
    Switch(sw);
    neg find switchHasSignal(sw);
}

pattern switchHasSignal(sw) {
    Switch(sw);
    Signal(sig);
    Signal.mountedTo(sig, sw);
}
Model sizes in practice

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

- Models evolve and change continuously

<table>
<thead>
<tr>
<th>Application</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>System models</td>
<td>$10^8$</td>
</tr>
<tr>
<td>Sensor data</td>
<td>$10^9$</td>
</tr>
<tr>
<td>Geospatial models</td>
<td>$10^{12}$</td>
</tr>
</tbody>
</table>

Validation can take hours

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: Supports
  - the definition & execution of model queries

\[
\text{Query}(A,B) \leftarrow \land_{i} \text{cond}_i(A_i,B_i)
\]
- 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
Categorization of Query Languages

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

<table>
<thead>
<tr>
<th></th>
<th>Imperative query languages</th>
<th>Declarative query languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressive power</td>
<td>☹ (you write lots of code)</td>
<td>☺ (very concise)</td>
</tr>
<tr>
<td>Safety</td>
<td>☺☺ (precise control over what happens at execution)</td>
<td>☺☺ (unintended side-effects)</td>
</tr>
<tr>
<td>Learning curve</td>
<td>☺ (you already know it)</td>
<td>☹ (may be difficult to learn)</td>
</tr>
<tr>
<td>Reusability</td>
<td>☺ (standard OO practices)</td>
<td>☹☺ (???)</td>
</tr>
<tr>
<td>Performance</td>
<td>☺☺ (considerable manual optimization necessary)</td>
<td>☺☺ (depends on various factors)</td>
</tr>
</tbody>
</table>
Graph Pattern Matching for Queries

- **Match:**
  - $m : L \rightarrow G$ (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:**

  ```
  ▶️ 
  1. switchPosition
  2. switch
  3. sensor
  4. routeDefinition
  5. route: Route
  ```

- **Graph Pattern Matching (Local Search):**

  - **Switch Position:**
  - **Sensor:**
  - **Route:**
  - **Switch:**

- **Search Tree:**

  ```
  ▶️ 
  1. switchPosition
  2. switch
  3. sensor
  4. routeDefinition
  5. route: Route
  ```
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

- **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

![Diagram with nodes and edges representing the relationships between route, switchPosition, switch, and sensor, with a table showing an example of route, sp, switch, and sensor assignments.](image)
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

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

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

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

**Website**
http://eclipse.org/incquery
INCREMENTAL MODEL QUERIES: THE LANGUAGE
The IncQuery (IQ) Graph Query Language

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

```plaintext
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);
}
```
Example: Statecharts metamodel

- Other detailed examples
// 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);
}
// 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);
}
// 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,
    State.visualisation.green(S,
    State.visualisation.yellow(S,
})
// 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);
}
Pattern composition and NAC

// 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, _, _, _);
}
Transitive closure and disjunction

```java
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)
老师们课程最多：

```
pattern teachersWithMostCourses(S, T) = {
  S.teachers(T);
  neg find moreCourses(T);
}
```

更多课程的模式：

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

检查表达式，为属性值（纯！）
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

**Pattern Editor**

```java
/*
 * Courses of a teacher.
 * Teacher T teaches in Course C
 */

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

/*
 * Teacher T teaches a course which is being taught to School
 */

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

/***
 * SchoolCourses of a teacher.
 * Teacher T teaches in School S
 */

pattern schoolCoursesOfTeacher(T:Teacher, S:School) = {
    find coursesOfTeacher(T, C);
    School.schoolCourses(T, S, C);
}
```
EMF-IncQuery: An Open Source Eclipse Project

Definition

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

Execution

• Incremental evaluation
  • Cache result set
  • Maintain incrementally upon model change

Tooling

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

http://eclipse.org/incquery
OVERVIEW OF INCREMENTAL QUERY EVALUATION
Development workflow

1. Develop EMF domain
2. Develop and test queries
3. Use/Generate INQUERY code
4. Integrate into EMF application

Automated

Semi-automated for typical scenarios, some manual coding

Supported by Xtext 2
EMF-IncQUERY Architecture v0.8

- The RETE algorithm makes it all work
- Well-known in rule-based systems
Incremental Query Evaluation by RETE

- AUTOSAR well-formedness validation rule

- Instance model

Invalid model fragment

Valid model fragment
Incremental Query Evaluation by RETE

- **antijoin**
- **Result set**
- **Communication channel**
- **Logical signal**
- **Mapping**
- **Physical signal**
- **worker nodes**
- **input nodes**

**Join**
- **R**esult set
- **Join**

**Physical signal**

**Logical signal**

**Map**

**Communication channel**

**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

Application

Your code

Generated pattern matcher

Pattern/Query specification

tooling

API

Validation Engine

Reflective pattern matcher

IncQuery BASE

RETE Core

EMF INC PM Core

Framework

• Basic incremental model access queries
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

http://eclipse.org/incquery

- Declarative graph query language
- Compositional, reusable
- Transitive closure, Negative cond, etc.
- Incremental evaluation
- Cache result set
- Maintain incrementally upon model change
- Derived features, On-the-fly validation
- Works out-of-the-box with EMF applications
- View generation, IncQuery with Eclipse EMF applications

Eclipse EMF-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

```plaintext
// 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 Features:
  • Values calculated from other elements
  • Defined declaratively as model queries (e.g. OCL, graph queries)
  • Tooling: handle as regular EMF elements
Handling Derived Features as Queries

DF specification: as a query

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

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

Computed overlay
aka. “View”

Defined by a query

Items = SELECT ...

<table>
<thead>
<tr>
<th>Id</th>
<th>Label</th>
<th>Prop0</th>
<th>Prop1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N1</td>
<td>a</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>N2</td>
<td>c</td>
<td>D</td>
</tr>
</tbody>
</table>
Live abstractions

Complex model

Change notification

Computed overlay aka. “View”
Defined by a query
Items = SELECT ...

Model Modification

abstract

UI update

Query result update

<table>
<thead>
<tr>
<th>Id</th>
<th>Label</th>
<th>Prop0</th>
<th>Prop1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N1</td>
<td>a</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>N2</td>
<td>c</td>
<td>D</td>
</tr>
<tr>
<td>2</td>
<td>N3</td>
<td>e</td>
<td>F</td>
</tr>
</tbody>
</table>
**INCQUERY Viewers**

1. **Model Modification**
   - EMF Model

2. **Change Notifications**
   - Live Queries

3. **Continuous, efficient synchronization**
   - Derived Model

4. **UI updates**
   - UI

- 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

```java
@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) = { ... }
```
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Eclipse technology</th>
<th>Pros</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trees, tables, Properties</td>
<td>EMF.Edit</td>
<td>The real deal: doesn’t hide abstract syntax</td>
</tr>
<tr>
<td>(JFace viewers)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diagrams</td>
<td>GEF, GMF, Graphiti</td>
<td>Easy to read and write for non-programmers</td>
</tr>
<tr>
<td>Textual DSLs</td>
<td>Xtext</td>
<td>Easy to read and write for programmers</td>
</tr>
<tr>
<td>JFace, Zest, yFiles</td>
<td>INCQUERY Viewers</td>
<td>Makes understanding and working with complex models a lot easier</td>
</tr>
<tr>
<td>Your tool!</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

---

[Diagram showing batch and incremental validation processes with corresponding steps: Read, Check, Edit, ReCheck, and an indication of 100x improvement.]
What Tools are Compared?

- Drools
- Neo4j (the graph database)
- Java
- IncQuery
- openRDF.org (Sesame)
- Franz Inc.
- clark&parsia
- Virtuoso
- 4store
**Batch validation runtime (complex queries)**

**Batch ModelVALIDATION (x,y:logscale)**

**EMF-IncQuery:**
Batch execution is dominated by
- loading the model
- initializing the indexers

- 2.8 million nodes + 11.2 million edges
- 88k nodes + 347k edges
- 0.7 million nodes + 2.8 million edges
Re-validation time (complex queries)

Characteristic difference (note the log scale)

EMF-IncQuery:
• close to zero response time
• up to models with 14 million elements

2.8 million nodes + 11.2 million edges

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

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

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

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

http://eclipse.org/incquery