Science of Computer Programming 98 (2015) 80–99

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

EMF-IncQuery: An integrated development environment for live model queries

Zoltán Ujhelyi *, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, Dániel Varró

Department of Measurement and Information Systems, Budapest University of Technology and Economics, H-1117 Magyar tudósok krt. 2, Budapest, Hungary

ARTICLE INFO

Article history:
Received 16 November 2012
Received in revised form 22 November 2013
Accepted 7 January 2014
Available online 9 January 2014

Keywords:
Live model query
EMF
Integrated development environment

ABSTRACT

As model management platforms are gaining industrial attention, the importance of automated model querying techniques is also increasing. Several important engineering tasks supported by model-based tools – such as well-formedness constraint validation or model transformations – rely on efficiently evaluating model queries. If the models change rapidly or frequently, it is beneficial to provide live and incrementally evaluated queries that automatically propagate model changes to keep query results consistent.

The current paper reports on the of EMF-IncQuery framework focusing on new features of its *integrated development environment* (such as query validation and visualization) and its support for *integrating queries to existing applications* (e.g. by auto-generated data bindings) built on top of the industry standard Eclipse Modeling Framework (EMF). Our approach is illustrated on a case study integrating well-formedness constraints to the Papyrus UML tool by live model queries of EMF-IncQuery with negligible additional manual programming effort.

1. Introduction

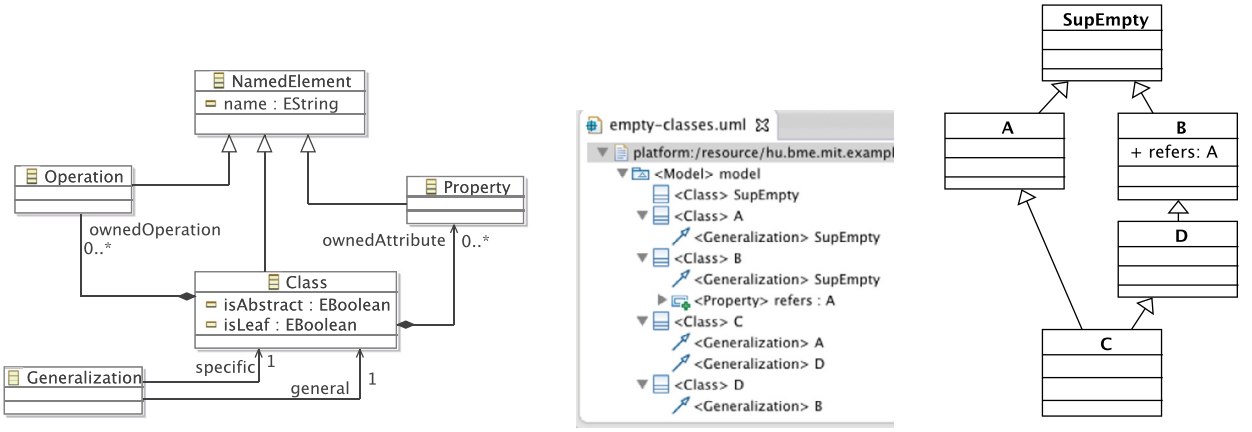
As model management platforms are gaining more and more industrial attention, the importance of automated model querying techniques is also increasing. Queries form the underpinning of various technologies such as model transformation, code generation, domain-specific model execution and well-formedness validation that are all essential in state-of-the-art modeling tools and toolchains.

The leading industrial modeling ecosystem, the Eclipse Modeling Framework (EMF [1]), provides different ways for querying the contents of models. These approaches range from manually coded model traversal to high-level declarative constraint languages such as Eclipse OCL [2]. However, industrial experience [3] shows strong evidence of scalability problems in complex query evaluation over large EMF models, and manual query optimization is time consuming to implement on a case-by-case basis.

In order to overcome this limitation, the EMF-IncQuery framework (first introduced in [3]) proposes to use declaratively specified queries over EMF models, executing them efficiently *without manual coding* using incremental graph pattern matching techniques [4]. The benefits of EMF-IncQuery with respect to the state-of-the-art of querying EMF models [2,5] include: (i) a high-level and powerful declarative graph pattern based *query language* [6], (ii) a highly efficient *query engine* capable of evaluating queries over models with millions of elements [3], and (iii) an advanced *integrated development environment* [7]

* Corresponding author. Tel.: +36 1 463 3579.
E-mail addresses: ujhelyiz@mit.bme.hu (Z. Ujhelyi), bergmann@mit.bme.hu (G. Bergmann), hegedusa@mit.bme.hu (Á. Hegedüs), ahorvath@mit.bme.hu (Á. Horváth), izso@mit.bme.hu (B. Izsó), rath@mit.bme.hu (I. Ráth), szatmari@mit.bme.hu (Z. Szatmári), varro@mit.bme.hu (D. Varró).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2014.01.004



(a) Metamodel (b) Abstract Syntax (c) Concrete Syntax

Fig. 1. A Simplified presentation of UML class diagrams.

to ease the construction and validation of model queries. In addition, (iv) the EMF-IncQuery efficiently *addresses several shortcomings of the EMF API* (such as instance enumeration of a certain type and backward navigation). Furthermore, (v) its modular architecture enables *easy integration with existing EMF-based modeling tools* [8]. Finally, (vi) its underlying implementation supports *multi-threaded and parallel processing*, node sharing and other low-level optimization strategies [9].

In the current paper, we present the state-of-the-art EMF-IncQuery framework by focusing on the integrated development environment and integration aspects. As a novel contribution with respect to previous papers [7,8] we present novel visualizations of queries, new validations for queries and the new data binding features for EMF integration. Furthermore, we significantly extend the description of the underlying tool architecture. Additionally, we compare the query evaluation performance of the Eclipse OCL project [2] and the EMF-IncQuery framework using a new, incremental query evaluation benchmark. As a new case study, we illustrate how EMF-IncQuery can be integrated into the Papyrus UML modeling tool [10] to provide advanced querying, visualization and on-the-fly model validation over UML models.

Structure of the paper The rest of the paper is structured as follows. In Section 2 we give a brief overview of the query language and a high level overview of the EMF-IncQuery framework. In Section 3, we present the novel features of the development environment targeted at the specification, visualization and debugging of live model queries. As a follow-up, in Section 4 we elaborate the case study to highlight the most important integration features of EMF-IncQuery. Section 5 features an incremental query evaluation benchmark that is used to compare the query performance of EMF-IncQuery and the Eclipse OCL project. Section 6 summarizes related work, and Section 7 concludes the paper discussing directions for future work.

2. Background: incremental model queries

In this section we give a quick overview of the EMF-IncQuery framework. After a short presentation of the UML Class Diagram formalism, we describe the query language together with the basics of graph patterns in Section 2.1. Then we introduce the query engine, including the change reporting facilities, in Section 2.2, and finally the development environment for model queries in Section 2.3.

Example 1. In this paper we rely on UML class diagrams to introduce the technicalities of our approach. The corresponding subset of the UML metamodel is depicted in Fig. 1(a) using the representation of the EMF Ecore language. The central element is an EClass named *Class* that has two EAttributes *isAbstract* and *isLeaf*. Additionally, it can contain any number of *Operation* and *Property* objects along two containment EReferences. Each of these inherits a *name* attribute from the *NamedElement* class. Finally, two classes can be in a *Generalization* relation expressed by an EClass *Generalization* leading from a child Class (*specific*) to a parent Class (*general*).

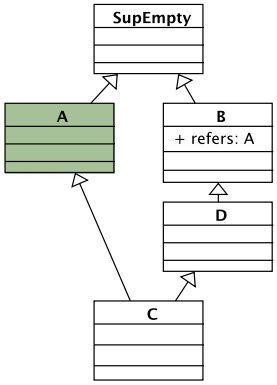
Fig. 1(b) presents a UML class diagram conforming to the metamodel definition in Fig. 1(a) based on an Eclipse EMF-based tree editor with five classes providing a multiple inheritance hierarchy. Each model object is represented by a tree item, attributes are either displayed in the labels, or only visible in a dedicated *Properties view*. Fig. 1(c) shows the same instance model in a graphical concrete syntax of UML.

2.1. Defining model queries with graph patterns

The EMF-IncQuery framework aims at the efficient definition and evaluation of incremental model queries over EMF-based model, building on the idea of *graph patterns* [6]. A graph pattern is a graph-like structure representing a condition

```
1  pattern superClass(sub : Class, sup : Class) {
2    Generalization.specific(gen, sub);
3    Generalization(gen);
4    Generalization.general(gen, sup);
5  }
6
7  pattern hasOperation(cl : Class, op : Operation) {
8    Class.ownedOperation(cl, op);
9  } or {
10   find superClass+(cl, owner);
11   Class.ownedOperation(owner, op);
12 }
13
14 pattern emptyClass(cl : Class) {
15   neg find hasOperation(cl, _op);
16   neg find hasProperty(cl, _pr);
17   Class.name(cl, n);
18   check(!(n.endsWith("Empty")));
19 }
```

(a) Graph Patterns Defining Empty Classes



(b) Empty Classes

Fig. 2. Graph patterns for detecting empty UML classes.

(or constraint) matched against a large instance model graph. The graph pattern formalism is usable for various purposes in model-driven development, such as defining declarative graph transformation rules or capturing general-purpose model queries including model validation constraints [11].

A basic graph pattern consists of *structural constraints* prescribing the interconnection of nodes and edges of a given type, as well as *expressions* to define *attribute constraints*. A *negative application condition* (NAC) defines cases when the original pattern is *not* valid (even if all other constraints are met), in the form of a negative sub-pattern.

The nodes and attributes used in the constraints are referenced using *pattern variables*. The *parameter variables* of a graph pattern are a subset of the pattern variables that represent the model elements and attributes that are interesting from the perspective of the query user. Pattern variables that are not parameters are called *local variables*.

It is possible to check for only the existence (or non-existence in case of negative subpatterns) of a reference or sub-pattern by the use of *single use variables*. As such local variables are only used once in the pattern body, commonly as a parameter of a relation definition or a pattern call, their values are not interesting for any other constraint.

It is also possible to extend the capabilities of the declarative pattern definitions by the addition of imperative, Java-like *check constraints*. However, to support efficient incremental evaluation, it is expected that these check expressions are pure, deterministic functional methods (no side-effects), and model traversal is disallowed. These constraints are partially enforced by the language by only allowing to refer to variables storing EAttributes values from the pattern body.

A *match* of a graph pattern is a tuple of parameter variables that fulfills all the following three conditions: (1) have the same structure as the pattern; (2) satisfy all structural and attribute constraints; and (3) does not satisfy any NAC.

By default, the result of a model query expressed as a graph pattern is the set of all matches with different bindings for the pattern parameter variables. However, by *binding* parameter variables to model elements or attribute values it is possible to filter the returned values. This binding process allows the use of the same pattern for getting all possible matches and for checking whether a selected match is present in the result set.

The query language of EMF-IncQuery is a textual language describing graph patterns as a set of constraints. The entire language was specified in [6], in this paper we only give short example related to UML class diagrams.

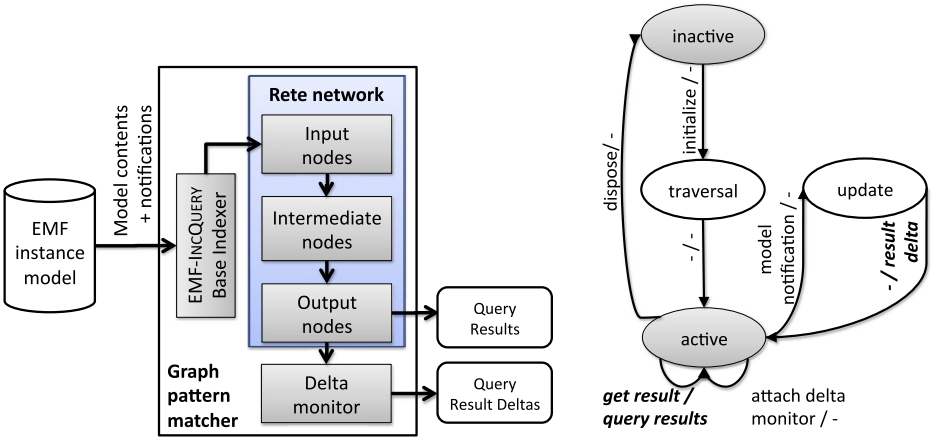
Example 2. To illustrate the pattern language of EMF-IncQuery we present a set of patterns in Fig. 2(a) for identifying *empty classes* in UML: classes that do not have operations or properties (not even in their superclasses), suggesting an incomplete model. However, if the name of the class is postfixed with the string “Empty”, we consider the class empty by design, so it is not returned.

Fig. 2(b) shows an instance model, where the single empty class `A` is emphasized with a different background color. The `SupEmpty` class is not considered empty because of its name, while the classes `B`, `C` and `D` either define or inherit the property called `refers`.

The pattern `superClass` in Line 1 consists only of structural constraints: it describes the direct superclass relation by a generalization node (local variable `gen`) that is connected both to the classes referenced as `sub` and `sup`.

The pattern `hasOperation` in Line 7 consists of the disjunction of two bodies: one represents the fact that the selected class `cl` holds an `Operation`. The second body uses the transitive closure of the relation defined by the `superClass` pattern in Line 10 to select the indirect superclasses of a selected class, and then declares that the superclass `owner` holds an Operation.

Finally, the pattern `emptyClass` in Line 14 selects classes without operations and properties by evaluating two corresponding NACs (the `hasProperty` pattern is omitted as it works exactly the same as the presented `hasOperation`). The



(a) Architecture (b) Life Cycle

Fig. 3. Incremental graph pattern matchers in EMF-IncQuery.

second parameters of the pattern calls are single-use variables (starting with the '_' symbol), so these NACs are simple non-existence checks. The *check* expression in the Line 18 reuses the `String.endsWith` Java method on a local variable.

2.2. The runtime components of the EMF-IncQuery framework

The EMF-IncQuery framework provides an efficient incremental query engine based on Rete networks [12]. At its core, the engine manages the incremental evaluation and lifecycle management of queries based on the Rete engine implementation originally developed for the VIATRA2 model transformation framework [4].

Fig. 3 gives an architectural overview of the EMF-IncQuery runtime components. In Fig. 3(a) the subcomponents of pattern matchers and their data flow are depicted, while Fig. 3(b) presents the internal states of the matcher. In the following, we give a brief overview of the entire matching process.

Incremental graph pattern matching by Rete Rete-based pattern matching relies on a network of nodes storing *partial matches* of a graph pattern. A partial match enumerates the model elements that satisfy a subset of the constraints of the graph pattern. Pattern matches are readily available at any time as they are updated incrementally at model changes.

Input nodes are used to represent the underlying model elements (e.g., EClasses, EReferences or EAttributes). *Intermediate nodes* are used to execute some basic operations – such as filtering, projection or join – on the outputs of connected Rete nodes (either input or other intermediate ones), and store and output the results. Finally, the match set of the entire pattern is available as an *output (or production) node*.

When a pattern matcher is initialized in EMF-IncQuery, the entire input model is *traversed*, setting up the Rete network. After the network is initialized, the pattern matcher becomes *active*, and returns the current *query results* on demand. Additionally, after a matcher is not required it can be *disposed*, removing the Rete network and freeing up used memory.

The EMF-IncQuery base indexer The contents of EMF instance models (and the corresponding change notifications) are connected to the query engine using a model indexer component called EMF-IncQuery Base.[1]

The indexer creates and maintains caches of frequently used low-level incremental queries such as instant enumeration of all instances of a given EClass, or reverse navigation along unidirectional EReferences together with advanced features such as calculating the transitive closure of elements reachable using a set of EReferences.

The separation of the indexer from the Rete network is useful for two different reasons: (1) the indexer can be reused without the main query engine component in EMF applications, and (2) the indexer can extend the capabilities provided by the query engine with features cumbersome to implement inside Rete networks, such as the transitive closure.

Processing updates in an EMF context Upon creation, the indexer is set up to receive notifications about all changes affecting the opened EMF models, such as the creation or deletion of model elements. When receiving a notification, at first the Base index is updated, then the corresponding input nodes of the Rete network are notified. Each time an input node receives notifications, an update token is released on each of their outgoing edges. Upon receiving an update token, a Rete node determines how (or whether) the set of stored tuples will change, and release update tokens accordingly. This way, the effects of an update will propagate through the network, eventually influencing the result set stored in production nodes.

[1] http://wiki.eclipse.org/EMFIncQuery/UserDocumentation/API/BaseIndexer.

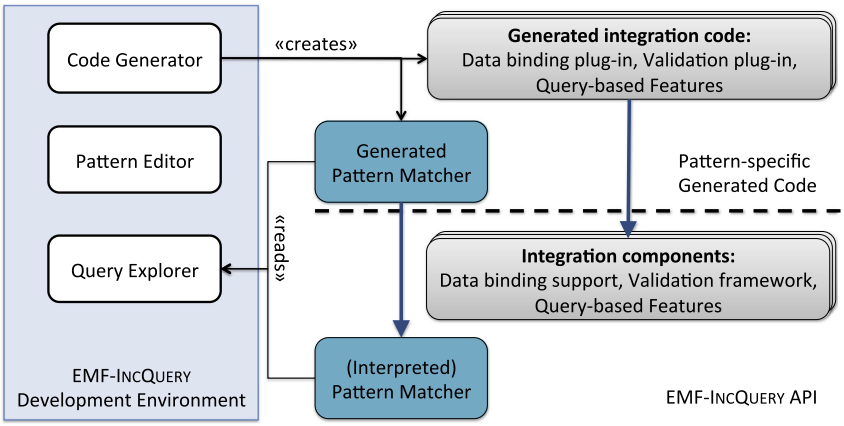


Fig. 4. Overview of EMF-IncQuery development environment.

To receive notifications of result changes, *Delta monitors* can be *attached* to output nodes that provide *Query Result Deltas* representing the added or removed matches.

2.3. *The* EMF-IncQuery *development environment*

The development workflow of the EMF-IncQuery framework focuses on the specification and evaluation of queries and the automatic generation of integration code for plugging into existing EMF-based applications. As depicted in Fig. 4, the development environment offers three major components: (1) the Pattern Editor, (2) the Query Explorer and (3) the Code Generator.

Pattern editor The EMF-IncQuery development environment provides an Xtext-based [13] editor for the pattern language with syntax highlighting, code completion and well-formedness validation. The editor is tightly integrated with the other components: the code generator is integrated into the Eclipse builder framework, and is executed after changes in pattern definitions are saved (unless Eclipse automatic builders are turned off), while the Query Explorer updates the displayed query results. The well-formedness validation of queries is described in more detail in Section 3.1.

Query explorer In order to evaluate complex model queries the EMF-IncQuery development environment provides the Query Explorer. This component visualizes live query results of both interpretative and generated pattern matchers in a generic view, and provides a quick feedback cycle during transformation development. The Query Explorer is presented in more detail in Section 3.2.

Code generator The environment also helps the integration of queries into a Java application by maintaining a project with pattern-specific generated matcher code. The generated matcher is semantically equivalent of the interpretative one, but provides an easy-to-integrate type-safe Java API, and some performance optimizations are also executed.

Furthermore, the generator may also produce code for various integration components, such as the *data binding support*, *validation framework* or *query-based features*. These integration scenarios are detailed in Section 4.

Example 3. Fig. 5 shows the EMF-IncQuery development environment while developing the case study. On the left side the used model and plug-in projects are shown. As EMF-IncQuery projects are plug-in projects, their management relies on already existing Eclipse features (a). In the center, the *Query Editor* (b) is open next to the *Papyrus UML editor* (c) that contains a sample model for evaluating the queries currently developed. Finally, in the bottom of the screen the *Query Explorer* (d) has already loaded the model and the queries from the editors, and reacts on changes in any of the editors.

3. Developing live model queries with EMF-IncQuery

The EMF-IncQuery development environment supports the development of live model queries by (i) providing early feedback for possible errors in query definitions and (ii) providing various visualizations for queries and query results to help to understand and debug live model queries.

In this section, we introduce the main features of the EMF-IncQuery development environment targeting at the validation and understandability of live queries. At first, we present well-formedness constraint validation rules for graph patterns (Section 3.1), then the query explorer components for manual query evaluation (Section 3.2) and finally we present some graph visualizations for the various models used in EMF-IncQuery (Section 3.3).

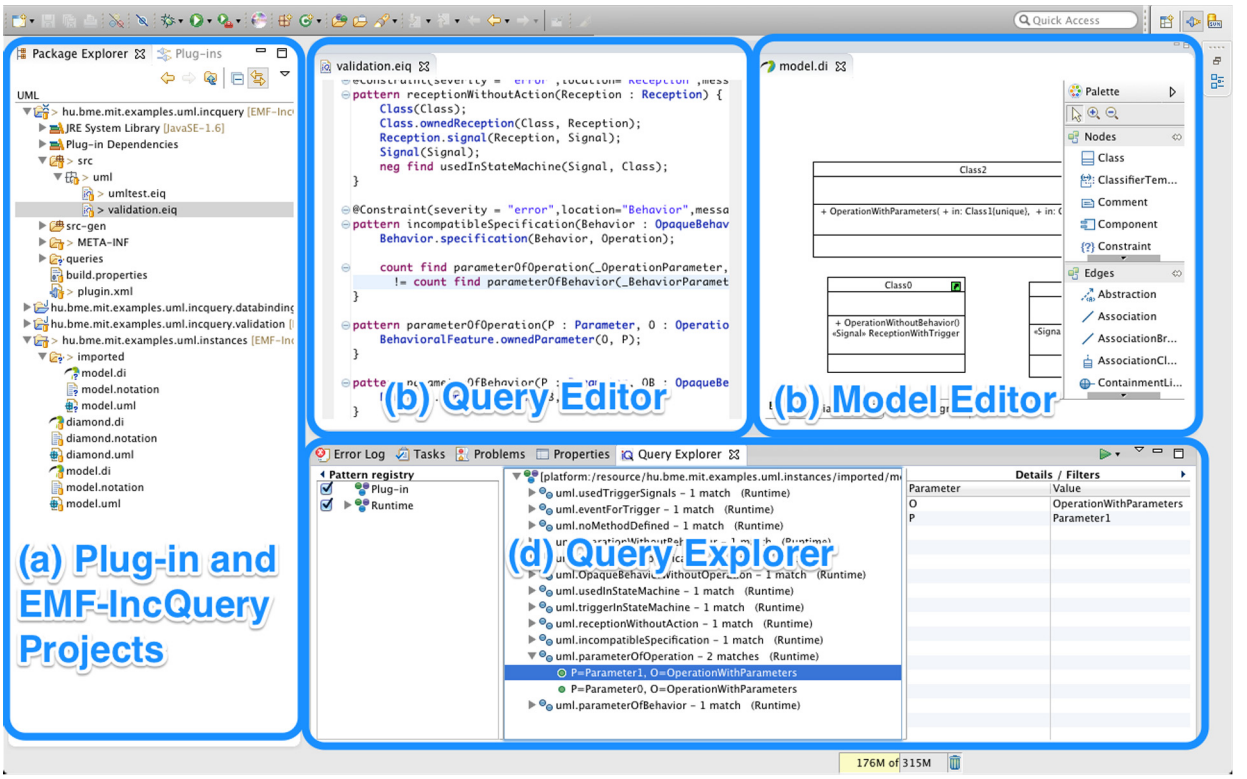


Fig. 5. The EMF-IncQuery development environment.

3.1. Validation of model queries

The high-level, declarative nature of graph patterns combined with complex structures (e.g. transitive closures or pattern composition) still makes it possible to write erroneous queries that may lead to unexpected runtime behavior. Some of these flaws can be detected by static validators resulting in early feedback. In this paper, we present three useful well-formedness rules the environment validates that were implemented using as Xtext validators.

Type inference Type inference and type checking is used to identify pattern variables with inconsistent type constraints (e.g. stating that a variable has multiple incompatible EMF types). These issues are often introduced by a misparameterized pattern composition, such as incorrect ordering of parameters. In Fig. 6(a) the parameters of the `hasOperation` call are swapped, resulting in ambiguous type constraints that is reported as an error.

As patterns with inconsistent type constraints always evaluate to empty result sets, these issues are marked as errors.

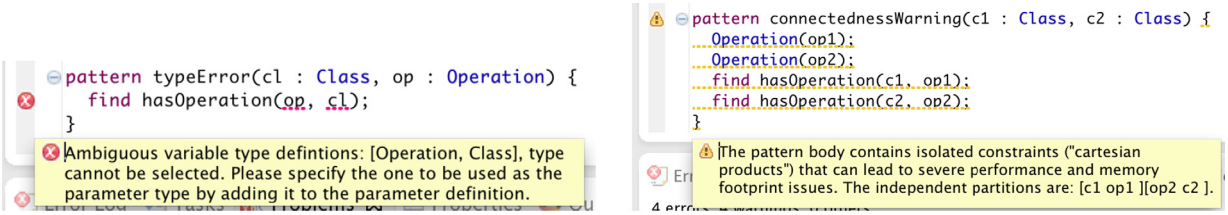
Connectedness The defined patterns are also checked for connectedness: if there are independent constraints in the pattern, the size of the result set and the memory consumption of the pattern matching increases as the Cartesian product. Such issues are often caused by a missing constraint. E.g., in Fig. 6(b), two operations are selected from two classes, but neither the classes nor the operations are connected by any constraint.

In most cases, such connectedness problems represent an undesired behavior, however, as in some cases this may be intended, only a warning is reported containing the connected subsets of variables.

Detecting unused variables As a variable used only a single time frequently indicates a misspelled variable, a *variable usage counter* is implemented that checks the number of uses of variables in all pattern bodies. In Fig. 6(c), a new single-use variable `c1` is introduced in the pattern body instead of the very similarly named parameter variable `cl`, altering the pattern to check existence whether the Operation is contained in any class.

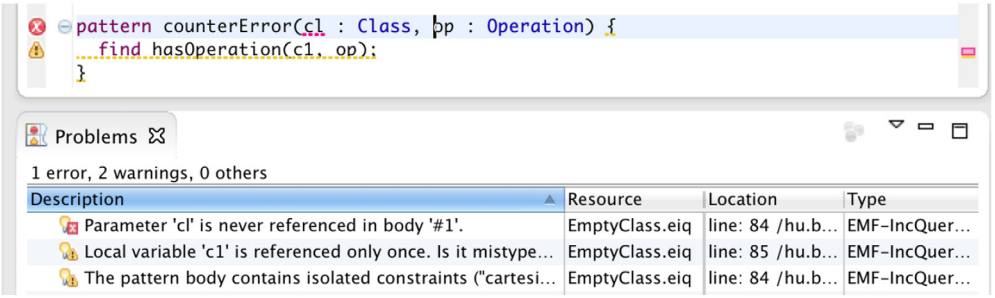
As single-use variables are often useful, this issue is reported only as a warning with the suggestion that names single-use variables should be prefixed with an '_' symbol. If the '_' prefix is used, the validator works differently: if the variable is used only once, nothing is reported, but if it used again, an error is reported.

Although unused variables are also reported by the connectedness validator, the variable counter is still useful as its more specific error messages of the usage counters guide the pattern developers how to fix these issues.



(a) Type Inference

(b) Connectedness



(c) Unused Variable Detection

Fig. 6. Well-formedness validation of model queries.

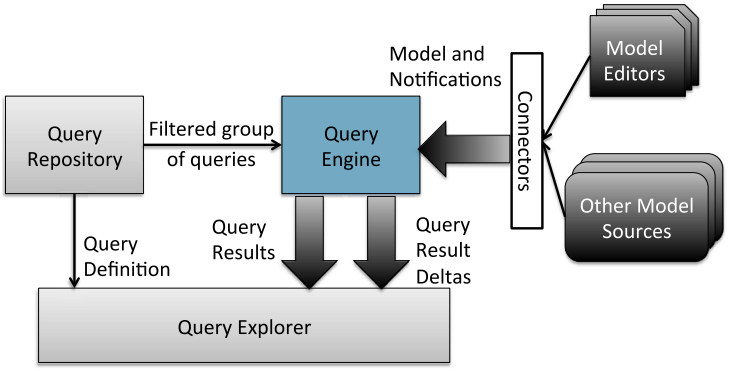


Fig. 7. The architecture of the query explorer.

3.2. Applying queries to instance models

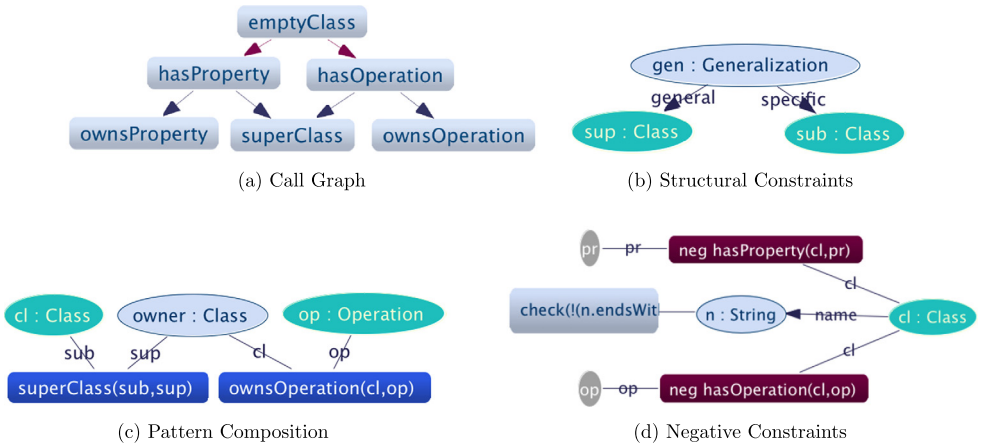
In addition to static analysis of the defined queries, the EMF-IncQuery development environment also supports the manual evaluation of query results by the *Query Explorer*. The component is already presented in greater detail in [7], now we only describe its architecture briefly, and discuss its integration of the query development workflow.

The Query Explorer is capable of loading input models from various sources, and execute the queries on them. The component reacts to both model and query changes, providing instant feedback during query development.

Architecture To support all these cases, we implemented the architecture depicted in Fig. 7. The defined queries are loaded to a *Query Repository* that creates filtered groups of them. These definitions are referenced in the user interface and added to the *Query Engine* that also loads models. To load these models from various model sources, model source connectors were introduced that load models from existing editors, and allows navigating back to the model element definitions as needed.

Development workflow To evaluate a query in the Query Explorer, both the query definitions and some sample models are to be loaded. The Query Explorer provides a single-button interface for loading both the query definitions and models from the currently open editor (regardless of their type). *Traceability links* are also maintained to support navigation to the pattern definition or model element in their corresponding editors when they are double-clicked in the Query Explorer.

When both queries and instance models are loaded, the Query Explorer *groups* matches by model source and query definition, and allows result *filtering* by binding query parameters (by default, all parameters are unbound).



(a) Call Graph (b) Structural Constraints

(c) Pattern Composition (d) Negative Constraints

Fig. 8. Visualizing graph patterns (screenshots). (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

In case of model and query updates, the Query Explorer updates the displayed matches *incrementally* to have an always up-to-date results without blocking the user interface. Upon model changes the *query result deltas* provide sufficient information to update the displayed matches. However, in case of query updates, the created Rete network has to be re-created that might include a re-traversal of the input models. To avoid such re-traversals, the Query Explorer uses specialized Base indexers referring all possible model elements. The drawback of this approach is that it results in higher memory footprint for the EMF-IncQuery framework than standard execution, where query-specific indexers are created only from elements interesting for the current set of queries. For more details about incremental update procedure of the component please consult [7].

This incremental update feature allows updating queries in the pattern editor, saving the changes and the match results are instantly updated. This results in a short feedback cycle, making the Query Explorer a key component of our query definition workflow.

3.3. Visualizing graph patterns

In order to provide a graphical overview of the structure of defined queries, two graph-based visualizations were designed to depict the structure of the patterns: the connections between interdependent patterns and the internal constraint graph of single patterns.

All visualizations are created using the Zest graph visualization framework [14] that simplifies the definitions of Eclipse-integrated visualizations. The framework comes with a set of graph layout algorithms that can be extended by custom algorithms. Visualized graphs can also be rearranged by the query developer manually to ease understanding of the more complex cases.

3.3.1. Call graph visualization

To visualize the connection between the interdependent patterns, a simple *call graph* is created: its nodes are the patterns, and two patterns are connected by a directed edges if the first pattern contains a reference to the second one (both positive and negative constraints are considered).

Example 4. The call graph of the patterns mentioned in Fig. 2(a) is presented in Fig. 8(a). The outgoing NACs of the `emptyClass` pattern are presented with red lines, while other connections with blue ones.

3.3.2. Constraint graph visualization

A different way to visualize graph patterns is to create a *constraint graph* [15] similar to the one used to visualize constraint satisfaction problems. A constraint graph is a hypergraph, whose nodes are the pattern variables and constant values used inside a graph pattern; hyperedges between nodes represent a constraint referring all corresponding variables. Unary (type) constraints are included inside the label of the variable node.

Example 5. The constraint graph of the `superClass` pattern (displayed in Fig. 8(b)) consists of three nodes: the parameter variables `sub` and `sup`, and the local variable `gen` that are connected with two references (namely `Generalization.general` and `Generalization.specific`).

Pattern composition constraints could refer to any number of variables, so the hyperedge needs to connect several nodes. This is displayed in the constraint graph of the `hasOperation` pattern in Fig. 8(c): both the `ownsProperty` and the `superClass` patterns are connected to two variables – one in common and one unique.

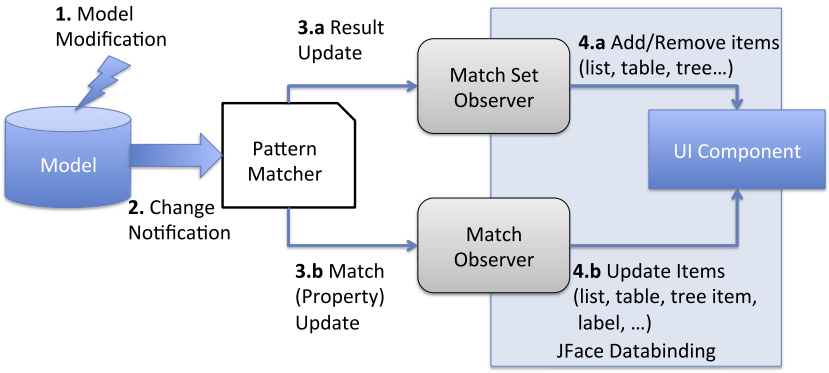


Fig. 9. Data binding of model queries.

Negative pattern composition constraints are displayed in a different color (in this case red), as seen in the visualization of the `emptyClass` pattern in Fig. 8(d). Additionally, that pattern uses anonym variables as well, that are also represented using a lighter grey color.

Such constraint graphs are useful for giving a quick overview of how the various variables interact while calculating the results of a graph query.

4. Integrating EMF-IncQuery into EMF-based applications

When developing EMF-based applications, there are some common tasks that are easy to express with live query evaluation. In order to support these cases, EMF-IncQuery provides *integration components*. An integration component may extend the language with a new annotation (and some corresponding validation) and adds a new code generator fragment that generates the required integration code for the selected framework.

EMF-IncQuery already includes three different integration frameworks: (i) *data binding support* eases user interface development for EMF applications by supporting the display of model elements in forms; (ii) the *validation framework* provides live well-formedness constraint validation for custom EMF editors; and (iii) *query-based derived features* allows enriching EMF models by new attribute values or references calculated (and updated) by model queries.

4.1. Data binding support

Data binding overview Data binding, supported in Eclipse applications by the JFace Data binding [16] framework, is a generic, declarative technique for binding and synchronizing data between data sources called *Observables*. An Observable has a set of *Observable (property) values* representing its current state. A *Binding* synchronizes two Observable values, either uni- or bidirectionally.

The JFace Data Binding framework already supports the creation of Observables from EMF model elements. For each model element, the list of observable values include all declared attributes and references, and it can be extended using simple, path-based traversals. However, creating a filtered observable list is not supported directly.

Queries as data sources The result deltas of model queries allows an efficient implementation of bindings, as it is possible to get notified about only the relevant model changes. There are two ways to bind query results: (1) binding the entire *match set* into a table or list component, or (2) binding a single *match* into any selected UI component (e.g., a label or a tree item). It is also possible to combine these approaches, e.g., in the "*master-detail data binding*" scenario the selection of a selected list or table (*master*) acts as the source element of some other UI components (*detail*) bindings.

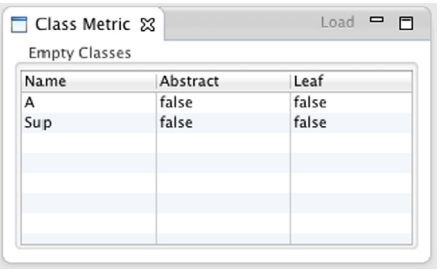
Fig. 9 depicts the architectural overview of query-based data bindings. When setting up the bindings, a *Match Set Observer* is created from the result set of the selected pattern, together with *Match Observer* for every match.

The update process goes as follows: in case of *model modifications* (1) the pattern matcher is *notified* (2). If a new match appears or disappears during the processing of the changes, the Match Set Observer is notified (3.a) to propagate the changes using the data binding framework into the UI by *adding or removing displayed items* (4.a). Similarly, the created match observers listen to property updates of existing matches (3.b). These changes are also transferred into the UI by the data binding framework, updating existing UI components (4.b).

Example 6. To illustrate the concept of observable matchers, we will collect all the *empty classes* of the model. Additionally, we created two observable values storing whether the selected class is *abstract* or a *leaf* class. The required annotations are presented in Fig. 10(a).

```
1 @ObservableValue(name = "abstract",
2   expression = "cl.isAbstract")
3 @ObservableValue(name = "leaf",
4   expression = "cl.isLeaf")
5
6 pattern emptyClass(cl : Class) {
7   ...
8 }
```

(a) Data binding annotations



(b) A Table Filled using Data binding

Fig. 10. Data binding support.

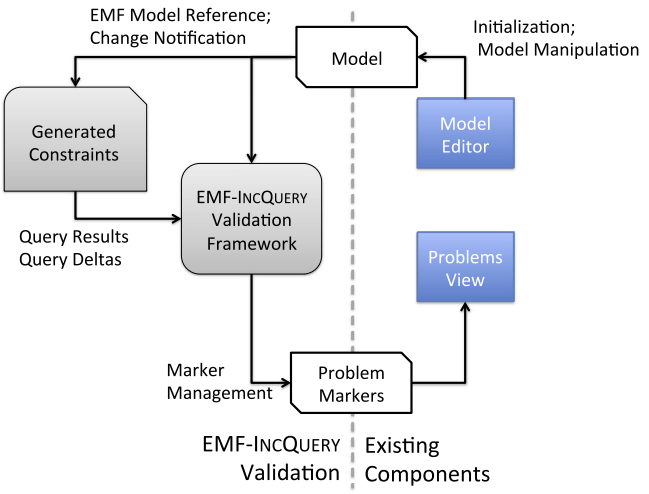


Fig. 11. The architecture of validation framework projects.

Based on the annotated patterns the data binding generator creates pattern-specific Observables that can be bound to the user interface using the standard JFace Data Binding API. This allows user interface programmers to bind the results of the pattern matcher of the `emptyClass` pattern to a table or list widget. Fig. 10(b) depicts a table viewer filled with this observable.

4.2. Validation framework

Well-formedness constraint validation is important for domain-specific language editors, as it allows correcting problems before they cause problems in the developed applications. The EMF Validation Framework project [17] supports the definition of well-formedness constraints that can be executed manually. However, an incremental query engine could provide *live validation* by reporting errors during editing, instantly when they are introduced.

EMF-IncQuery allows the definition of such live model validation [11] as depicted in Fig. 11. The EMF-IncQuery Validation Framework consists of a dedicated service that manages the defined constraints, loads models and creates (and updates) *problem markers* of the constraint violations. Although EMF-IncQuery does not depend on the EMF Validation Framework, the created problem markers are the same. This means, if an editor, such as the Papyrus UML editor [10], displays violations graphically, it will also work with the markers of EMF-IncQuery.

Constraints are generated from annotated patterns from the query language to describe the erroneous model element, the error message and its severity.

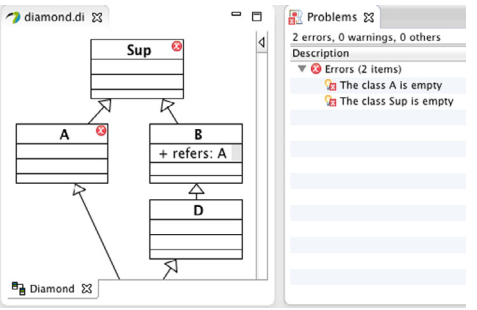
Example 7. The validation annotations for empty classes is presented in Fig. 12(a).

To associate error markers to *empty classes*, a `Constraint` annotation needs to be added. The annotation parameters specify the *severity* of the issue, the model element that the marker needs to be attached to (*location*), the error *message*. Additionally, setting the optional *editor identifier* parameter generates the initialization code for the selected editor.

During runtime, after running the generated initialization code (available from the menu), the found markers are displayed, as presented in Fig. 12(b). The *Problems view* on the right lists all errors (including the ones of the Validation Framework), while based on the support of Papyrus UML red icons are used to visually present the error locations.

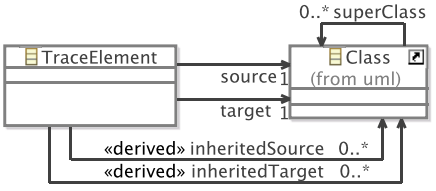
```
@Constraint(
 severity = "error",
 location = "cl",
 message="The class $c.name$ is empty",
 targetEditorId = "...papyrusEditor" )

pattern emptyClass(cl : Class) {
  ...
}
```

(a) The Validation Annotation



(b) The Validation Framework (Integrated to Papyrus UML)

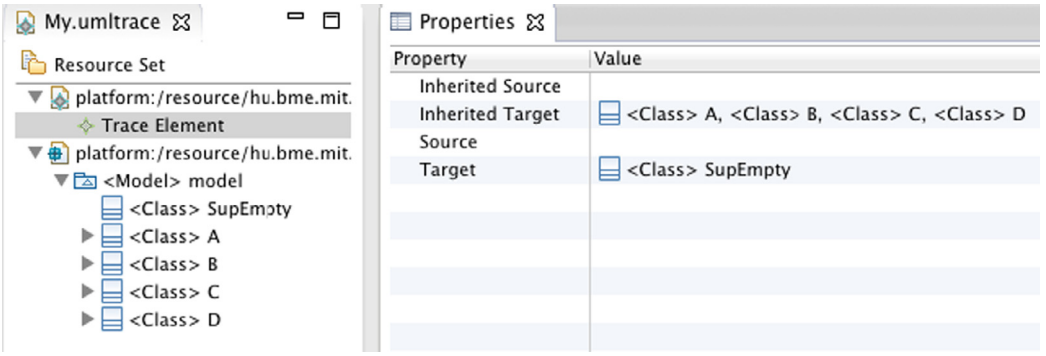
Fig. 12. The validation framework. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)



(a) A Trace Metamodel

```
@QueryBasedFeature(
  feature = "inheritedTarget",
  source = trace, target = cl)
pattern inheritedTarget(
  trace : TraceElement,
      cl : Class) {
  Class(trg);
  TraceElement.target(trace, trg);
  find superClass+(cl, trg);
}
```

(b) Defining Query-based Features



(c) The Calculated Query-based Feature

Fig. 13. Query-based feature support.

4.3. Query-based features

A third integration scenario we considered is enriching EMF models with *derived features*, that is representing information that can be calculated from other elements. EMF allows defining such derived features via Java code, however, it is challenging to provide an implementation with transparent EMF notification mechanism and reasonable performance. EMF-IncQuery queries were proposed as an implementation for the features in [18,19], as the use of generic adapter code (provided by EMF-IncQuery) makes it possible to return the results of an arbitrary query as an EMF derived feature.

Example 8. To illustrate query-based feature support we defined a traceability metamodel between UML classes in Fig. 13(a). The metamodel consists of only a single *TraceElement* class that refers to a *source* and a *target* class. However, as the traceability relation is related to the subtypes of the referred classes, we introduce two *derived features* that list all classes that extend the source and target classes respectively.

Fig. 13(b) describes the pattern we use to define the `inheritedTarget` feature. The pattern finds the Class at the `target` reference of the `TraceElement` and returns all its subclasses (using the previously defined `superClass` pattern). Then the `QueryBasedFeature` annotation is used to select the EMF derived feature to provide the adapter.

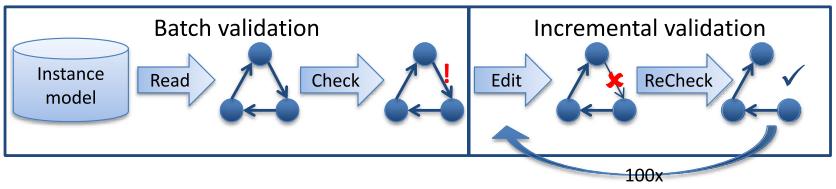


Fig. 14. The benchmark process.

The generator fragment for query-based features is capable of finding and modifying the Java code generated from the metamodel, making the integration seamless from the model users perspective, while change notifications rely on the query result delta feature of EMF-IncQuery.

Finally, Fig. 13(c) shows the derived feature in a UML context: by setting the `Target` reference in a `Trace Element`, all `Inherited Targets` are calculated automatically.

The support for query-based features complements data binding support. In both cases, existing models are extended with automatically calculated features together with change notification. However, data binding support is used outside the boundaries of the metamodel, while query-based features extend existing ones transparently.

5. Query performance benchmark

When integrating model queries into off-the-shelf applications, a very important aspect is the scalability of the approach. For this purpose, we first introduce a benchmark that is motivated by well-formedness constraint validation executed on user-performed editing steps in Section 5.1. Then we present the comparison results between EMF-IncQuery and the Eclipse OCL in Section 5.2. In this paper only the main results are published, for detailed measurement values and installation artefacts please refer to our benchmark website [20].

5.1. Benchmark specification

Our macrobenchmark addresses on-the-fly model validation in (domain-specific) model editors: at first, the entire model is validated, then after each model manipulation (e.g., the deletion of a reference) is followed by an immediate re-validation.

Instead of relying on the UML example used throughout the paper, a new metamodel and corresponding model queries motivated by our industrial experiences [21] are defined for benchmarking purposes. This domain enables the definition of both simple and more complex model queries while it is uncomplicated enough to incorporate solutions from other technological spaces (e.g. ontologies, relational databases and RDF). This allows the comparison of the performance aspects of wider range of query tools from a constraint validation viewpoint.

As opposed to database query measurements, such as [22–24], our benchmark focuses more on response time than throughput based on predefined queries on evolving models.

5.1.1. Benchmark process

To measure performance also when the underlying model is changing, four benchmark *phases* were defined, as illustrated in Fig. 14.

1. During the *read* phase, the previously generated instance model is loaded from hard drive to memory. This includes the parsing of the input as well as initializing data structures of the tool. The time for tool initialization normally increases with the amount of caching used in a specific tool.
2. In the *check* phase, the instance model is queried to identify invalid elements. This can be as simple as reading the results from cache, or the model can be traversed based on some index. By the end of this phase, erroneous objects need to made available in a list.
3. In the *edit* phase, the model is modified to simulate effects of manual user edits. Here the change set is small, compared to modifications by model transformations tool.
4. The re-validation of the model is carried out in the *re-check* phase similarly to the *check* phase.

When executing the benchmark, the time of each phases is recorded separately, but form the perspective of user experience, two aggregated values are interesting. The first time called *batch validation* is the time that the user must wait to start to use the tool, represented by the sum of the *read* and *check* phases. The *incremental validation* is consists of the *edit* and *re-check* phases performed 100 times, expressing the time that the user spent waiting for the tool validation.

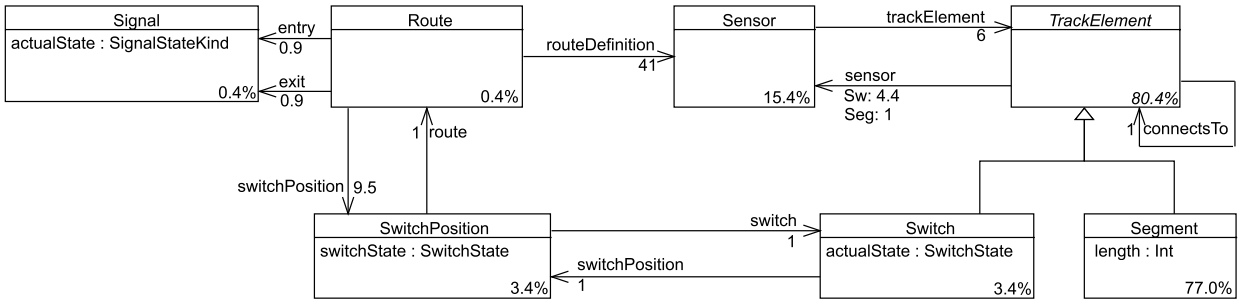


Fig. 15. Train metamodel and instance model characteristics.

5.1.2. Railway domain

The *railway domain* is presented in Fig. 15 with a metamodel (enhanced with statistics). *TrackElements* can be connected to each other, which are *Segments* (with length) or *Switches* (characterized by their actual state). A *Route* is defined by a series of *Sensors* (installed next to *TrackElements*), with an entry and exit *Signal*.

To show some characteristics of the generated instance models, the distribution of the object types and the average number of edges for each object is also presented in Fig. 15. In case of classes the percentage of instances is shown: e.g., 3.4% of the model elements are instance of the class *Switch*, 77.0% is *Segment*, thus 80.4% of the model is TrackElement. The average number of the given relation for an instance is displayed for associations: e.g., there are 9.5 *switchPosition* relations in average for every instance of the *Route* class.

This railway domain (and defined queries) developed for this benchmark were also tested and used previously by our academic partner [25] for evaluating a local search based pattern matching algorithm.

5.1.3. Instance model generation

In the first phase of the benchmark, a previously generated *instance model* is loaded from the file system. These models are systematically generated based on the metamodel and the defined complex model queries: small instance model fragments are generated based on the queries, and then they are placed, randomized and connected to each other. The methodology takes care of controlling the number of matches of all defined model queries.

To break symmetry, the exact number of elements and cardinalities are randomized. This brings artificially generated models *closer to real world instances*, and *prevents query tools from efficiently storing* or caching of instance models. During the generation of the railway system model, errors are injected at random positions. These errors can be found in the check phase of the benchmark, which are reported, and can be corrected during the edit phase.

The initial number of constraint violating elements are low (e.g. 0.3% of model elements for the RouteSensor case). The exact match count of each generated instance model and query is presented in the benchmark website [20].

5.1.4. Queries

In the validation and re-validation phase of the benchmark, elements violating well-formedness constraints are returned by the queries. These constraints are first defined informally in plain text (in a tool independent way) and then formalized using the standard OCL language as a reference implementation (available on the benchmark website [20]). Finally, the functionally equivalent variants of these queries are formalized using the query language of different tools applying tool based optimizations. As a result, all query implementation must return (the same set of) invalid instance model elements.

Two simple queries (involving at most two objects) and two complex queries (involving 4–7 variables and joins) were defined and the graphical representation of each queries can be seen in Fig. 16. More detailed, technical-level query definitions are available at the benchmark website [20].

- The constraint *PosLength* checks, *whether Segments have non-negative length*. Violation can occur, for example, when the length remained on a default (zero or negative) value since the object's creation. This query realizes a simple integer attribute check.
- The following *SwitchSensor* query is a safety requirement, and the textual definition is: *Every switch must have at least one sensor connected to it*. This validation query checks for a missing *sensor* reference, represented by a negative application condition (depicted as NEG).
- The *RouteSensor* constraint is used to check for broken cycles as follows: *All sensors that are associated with a switch that belongs to a route* (*along a predefined switch position*), *must also be associated directly with the same route*. This is enforced by describing the missing route Definition edges (NEG).
- The most complex query in the benchmark is called *SignalNeighbor*, which specifies the following constraint: *A route is incorrect, if it has an exit signal, and a sensor connected to another sensor* (*which is in a definition of another route*) *by two track elements, but there is no other route that connects the same signal and the other sensor* (*NEG*).

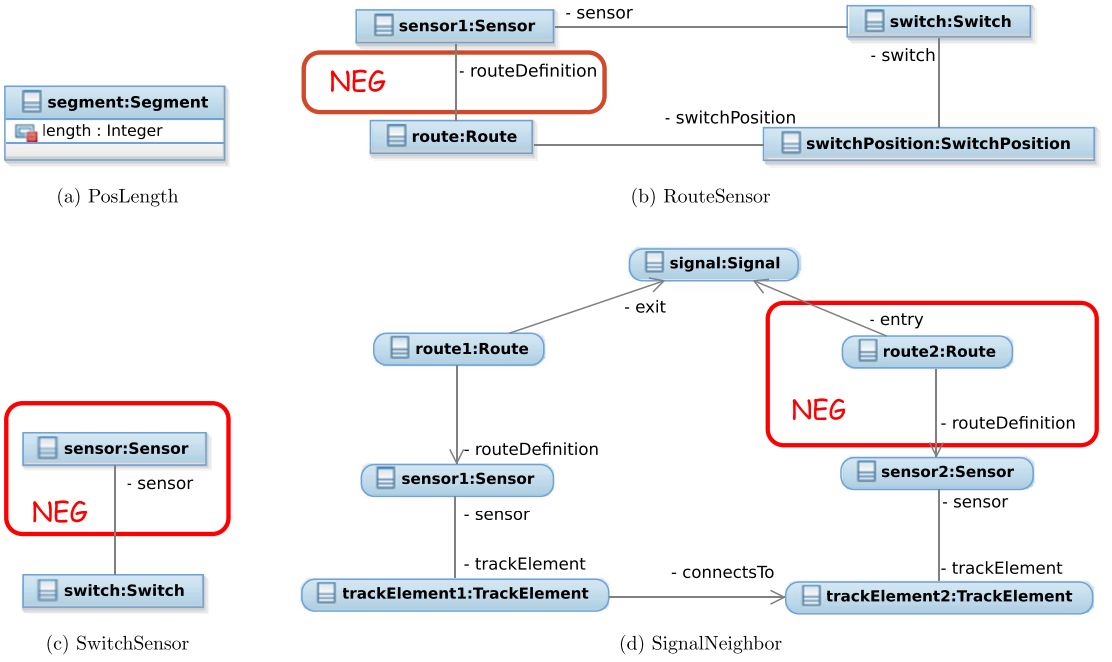


Fig. 16. Queries in graphical representation.

```
1 context Switch:
2 self.sensor->isEmpty()
```

(a) SwitchSensor in OCL

```
1 context Switch:
2 Sensor.allInstances()->forAll(s|
3 not s.switch->includes(self))
```

(b) SwitchSensor in OCL using allInstances()

```
1 pattern switchSensor(sw) = {
2   Switch(sw);
3   neg find hasSensor(sw);
4 }
5 pattern hasSensor(te) = {
6   TrackElement(te);
7   Sensor(sen);
8   TrackElement.sensor(te, sen);
9 }
```

(c) SwitchSensor in IncQuery Pattern Language

Fig. 17. SwitchSensor query in various languages.

5.1.5. Query implementation

These queries are implemented in OCL (for Eclipse OCL and Impact Analyzer) and in EMF-IncQuery, detailed on the benchmark website [20]. Although, some important aspects are illustrated in Fig. 17 by the *SwitchSensor* query.

In this case, the OCL implementation described in Fig. 17(a) only checks the attribute for all switches. On the other hand, when backward navigation is needed (imagine that the *Switch* class has no *sensor* relation, but only the *Sensor* class has a *switch* relation), only target individuals can be enumerated with the (resource intensive) *allInstances()* operation, and navigation can start from these target objects. Such OCL query (with the same meaning) is depicted in Fig. 17(b), but the same construct was needed for the *SignalNeighbor* query (Fig. 16(d)), because all objects cannot be navigated starting from a Route context object.

For the EMF-IncQuery query implementation such enumerations are indexed by default (which means that it is a cheap operation), and the language enables navigation in both ways (see Line 8, where named relations are used, and variables are written in the correct order). An important feature is that commonly used subpatterns can be extracted (by giving them a name), and referred from other patterns, like the pattern *switchSensor* which calls *hasSensor* (negatively). Subpatterns can be called positively also, which is extensively used in the SignalNeighbor pattern (Fig. 16(d)), as e.g. the Route-Sensor-TrackElement subpattern occurs twice. Such subpattern reuse is resulted in lower memory usage.

5.1.6. Model modification

In the *edit* phase the model is modified to change the result set to be returned by the query in the re-check phase. During the modification the simulated user always performs hundred random edits (fixed low constant) which increases the number of erroneous elements. An edit operation only modify single model elements at once – more complex model manipulation is modeled as a series of edits.

In the *PosLength* query's modify phase a randomly selected segment's length is updated to 0, which means that an error is injected. In the case of *SwitchSensor* query, errors are injected by deleting sensor relations of randomly selected

switches which became invalid. In the case of the *RouteSensor* query, the route definition connection between the randomly selected routes and their first connected sensors are removed. In the fourth, the *SignalNeighbor* case, errors are introduced by unconnecting the entry edge of the selected routes.

5.2. Performance comparison of EMF-IncQuery and Eclipse OCL

In this section, we compare the performance of the Eclipse OCL tool with EMF-IncQuery focusing on the execution times of *batch validation* and *incremental validation*. This approach focuses on the user-visible aspects of the benchmark, thus demonstrating the usefulness of the query tools in the integration use-case presented in Section 4.

5.2.1. An overview of the Eclipse OCL Project

The OCL [26] language is commonly used for querying EMF model instances in validation frameworks. It is a standardized navigation-based query language, applicable over a range of modeling formalisms. Taking advantage of the expressive features and wide-spread adoption of this query language, the project Eclipse OCL [2] provides a powerful query interface that evaluates such expressions over EMF models.

The project also supports incremental evaluation by including an Impact Analyzer (IA) [27] that calculates the constraints to be reevaluated based on a model change. During EMF modifications it looks for possible context objects that could change the match set, and re-evaluation can be executed only for those objects. As it is intended only for incremental use, Eclipse OCL is used for calculating the first result set (batch mode).

5.2.2. Environment and measurement methodology

In order to measure tool efficiency instead of some bottlenecks (e.g. the lack of memory) or transient effects (like random CPU hogs), we payed attention to the hardware-software environment. The benchmark was executed on a physical machine that contains two quad core Intel Xeon L5420 CPU (2.50 GHz), 32 GBs of RAM. 64 bit Ubuntu 12.04 OS with OpenJDK JVM version 1.6.0_24 was used. To avoid external influences, such as swapping, trashing or parallel software execution, swap support, and unnecessary operating system services (like cron) were turned off, and disk caches were cleared between executions. Similarly, to avoid Java *garbage collection*, an extra large heap limit (15 GB) was set.

Before acquiring memory usage (free heap space) from the JVM, GC calls were triggered five times to sweep unfreed objects from the RAM. The time of each phase was recorded with millisec precision and recorded to CSV format for later offline evaluation.

To measure the scalability of the tools the implemented queries were executed on a set of generated models between sizes 30 k and 14 M model elements. Each total time was limited to 12 minutes.

To make the performance measurements of a tool for a given query-model pair was independent from the others, every measurement was run in a different JVM. This large degree of isolation was required to reduce the effects of complex interference between queries and instance models.

5.2.3. Measurement results

The measurement results of the benchmark is displayed in Fig. 18. These diagrams show the batch query performance, incremental evaluation time, and memory usage of each tools, for different model sizes. Additionally, the initial and the updated result set size is displayed under the model sizes in the batch and incremental queries, respectively.

The left column shows charts of the *RouteSensor* query, while the more complex *SignalNeighbor* is presented in the right column. The remaining *PosLength* and *SwitchSensor* queries are only presented at the benchmark website [20], as their results are very similar to the *RouteSensor* case.

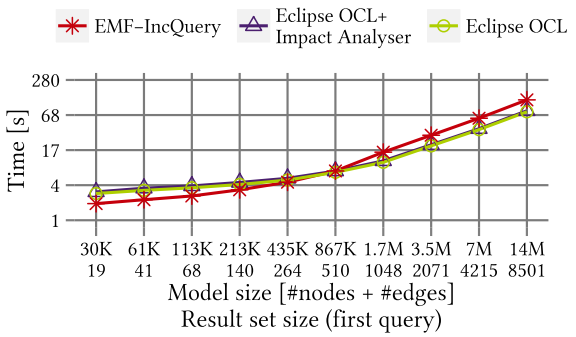
Batch query evaluation In case of batch query evaluation, both OCL implementations use the same algorithm, thus their execution time is roughly the same. The roughly negligible differences are due to the initialization of the OCL Impact Analyzer.

For the *batch query* evaluation of the *RouteSensor* query Fig. 18(a) shows that EMF-IncQuery performs similarly to Eclipse OCL. It is slightly faster for small models (2 s and 3 s respectively), but is slower for large models (up to 125 s and 78 s), where this 50% slowdown (once in the whole scenario) can be attributed to the initial (Rete) cache build.

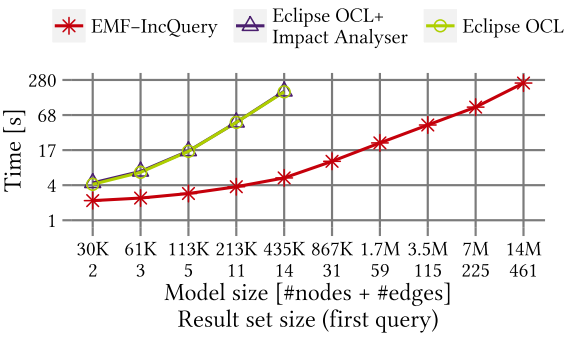
For the more complex *SignalNeighbor* query Fig. 18(b) depicts that EMF-IncQuery (somewhat surprisingly) outperforms OCL solutions: it is noticeably faster for small models (2 s and 4 s), and over 435 k model elements OCL did not finish with the initial analysis in 12 minutes. This performance gain might be attributed to the more efficient (cached) enumeration of instances, and the possibility of backward navigation (with the help of auxiliary structures) on unidirectional references used by this query.

Incremental query evaluation In the *incremental case*, Eclipse OCL evaluates the query on each issue (i.e.: hundred times) from scratch, its execution time increases linearly with model size, resulting slow overall evaluation.

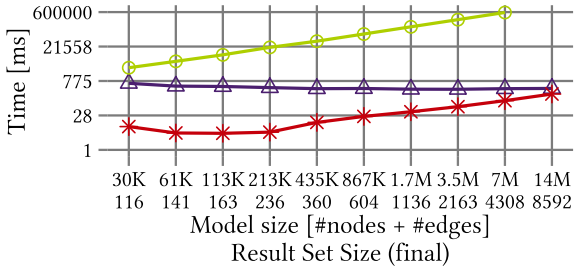
For the *RouteSensor* query (Fig. 18(c)), the Impact Analyzer performs the 100 modifications in 50 ms regardless of the model size. On the same query, EMF-IncQuery starts much faster, but its speed reduces on the larger models (from 9 to 220 ms). On the other hand, the Impact Analyzer is an order of magnitude slower on the *SignalNeighbor* query (Fig. 18(d))



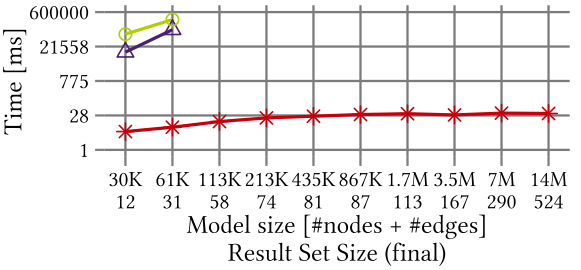
(a) Batch Query Evaluation Time - RouteSensor



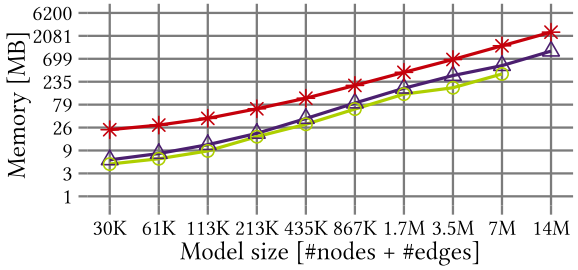
(b) Batch Query Evaluation Time - SignalNeighbor



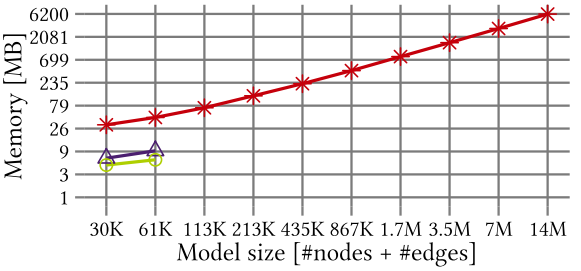
(c) Incremental Query Evaluation Time - RouteSensor



(d) Incremental Query Evaluation Time - SignalNeighbor



(e) Memory Usage - RouteSensor



(f) Memory Usage - SignalNeighbor

Fig. 18. Benchmark results.

query: it does not finish in 12 minutes for models over 61 k model elements, while EMF-IncQuery handles every model regardless of size under 40 ms.

The performance of the Impact Analyzer is most likely affected by the previously mentioned unidirectional references. The slowdown of EMF-IncQuery is probably caused by the increased number of matches (from 116 to 8592), as query results are always available in the output nodes of Rete networks, and only a linear traversal of these stored matches is needed to return them.

Memory usage Fig. 18(e) and Fig. 18(f) show the memory usage of the measured tools. As batch OCL evaluation does not need extra data structures (e.g.: caches), its memory measurements show basically the amount of memory needed to load the models, from 4 MB up to 660 MB in case of the largest model. Impact Analyzer needed up to 50% more memory, from 6 MB to 1 GB. The size of Rete networks created by EMF-IncQuery are largely query-dependant, increasing memory usage between a factor of 3 and 5, loading the largest models between 1.5 and 6 GB of RAM.

5.2.4. Threats to validity

We tried to mitigate *internal validity* threats by reducing the number of uncontrolled variables during measurements: a physical machine was used with all unnecessary software components turned off and every execution was isolated into a freshly initialized JVM.

The queries are semantically equivalent in the different query languages and the result sets are the same for every model. Additionally, to ensure comparable results the created high-quality query implementations were reviewed: the OCL

implementation by Ed Willink from the Eclipse OCL project, the usage of Impact Analyzer by Axel Uhl from the Impact Analyzer developer team. The graph patterns were written by the developers of EMF-IncQuery.

Considering *external validity*, the generalization of the results largely depends on how representative the metamodels, the models and the queries are compared to real use cases. In this paper, only a limited query feature set was measured, however the follow-up paper [28] details how to define the complexity of various query-instance model combinations.

The metamodel and the query specifications were motivated by an industrial case study, and the selected queries feature commonly used validation tasks such as attribute and reference checks or cycle detection. We tried to ensure that the instance models have a similar structure and distribution to other models by parameterizing the generation process based on our experience with other domains. To summarize, we believe that the train domain and the generated instance models represent other domain-specific languages and available instance models well.

Our current measurements only loaded and executed a single query in each run. When loading multiple queries, query interference may change the results greatly. A more detailed evaluation of this issue is planned for the future.

Considering resource-constrained environments, we believe that limiting available memory will alter the results the most, as the memory management overhead will reduce the performance of EMF-IncQuery.

It is important to note that heap usage were measured after executing a garbage collection, so these measurements do not contain memory usage of temporary constructs. This means that maximum heap usage might have been larger. Furthermore, limiting heap space by the maximum usage results in excessive garbage collection and thus an increased runtime. However, in our experience setting the limit to two times the measured values, such issues do not occur.

In case of the benchmark queries, we measured a 1.5 GB heap size in case of a model with 3.5 M model elements that we believe is manageable in a developer machine with 4–8 GB of RAM. On the other hand, when handling such large models the existing user interface itself could become a bottleneck. Thus we believe, our measurement results hold also in the integrated development environments.

5.2.5. Summary

To summarize our results, by comparing the query performance of EMF-IncQuery and OCL we have demonstrated that EMF-IncQuery is capable of maintaining query results efficiently, even on models with 14 million model elements.

In case of simpler queries, the batch query time is comparable to OCL-based solution, while in case of complex queries it performs better. After incremental updates the Rete algorithm makes the updated results available quickly, clearly outperforming the OCL-based solutions.

From resource usage it can be said that while EMF-IncQuery uses more memory, it is clearly not an issue for models smaller than 100 K elements. Although larger (a few hundred MBs) of memory is required for models from 100 K to 1 M elements, it can be handled with careful query implementations (e.g., by avoiding unconnected patterns that increase the memory usage). However, this way the queries are executed faster than in case of OCL. For models larger than 1 M elements, several GBs of heap space is required, but the response times of EMF-IncQuery scale better for complex queries and model sizes.

To summarize, batch query response times are comparable for all implementations. However, for incremental cases (and complex queries), EMF-IncQuery performed faster at the cost of higher memory usage.

6. Related work

General purpose database tools In the database community, several development environments were proposed for SQL queries, such as the MySQL Workbench [29], the InfoSphere Data Architect [30] or the Oracle Enterprise Manager [31]. The tools have various capabilities, but in general they provide query editing and evaluation support, often including static and dynamic validation to detect performance bottlenecks. Additionally, the Enterprise Manager tool can also generate a Graphical Explain Plans to give an insight to the performance of queries. For the comparison of MySQL and EMF-IncQuery see our preliminary results at our benchmark webpage [20].

In case of graph databases, similar environments are available, such as the Neoclipse environment [32] for the Neo4j database. The environment allows to edit and evaluate queries, and additionally provides a graph visualization tool for the underlying models. The first performance results of this tool in context of our benchmark is available online [20].

A simultaneous visualization for multiple query results were proposed in [33]. The query structures can also be presented as proposed in [34].

Furthermore, these solutions require significant additional integration effort to embed into existing EMF applications compared to EMF-IncQuery.

Model queries over EMF There are several technologies for providing declarative model queries over EMF. Here we give a brief summary of the mainstream techniques, none of which support incremental behavior.

EMF Model Query 2 [5] provides query primitives for selecting model elements that satisfy a set of conditions; these conditions range from type and attribute checks to enforcing similar condition checks on model elements reachable through references. Unfortunately, the limited expressive power of Model Query 2 permits only simple queries. For example, more complex patterns involving circles of references or attribute comparisons between nodes cannot be detected. However, Query 2 can also evaluate queries on instance models that have not been loaded into memory (using indices).

EMF Search [35] is a framework for searching over EMF resources, with controllable scope, several extension facilities, and GUI integration. Unfortunately, only simple textual search (for model element name/label) is available by default; advanced search engines can be provided manually in a metamodel-specific way.

Both Query 2 and Search rely on simple, standard Eclipse UI features to invoke queries and present the results to the user. Incremental evaluation is not supported, and thus presentation features that would rely on dynamically updated query results are not feasible.

OCL-based approaches The OCL development environment of the Eclipse OCL project [2] provides different ways to edit OCL constraints: an Xtext-based editor for file-based editing, an embedded editor inside Ecore model editors. Additionally, it provides an OCL Console that allows quick specification and (batch) evaluation of constraints.

Cabot et al. [36] present an advanced three-step optimization algorithm for incremental runtime validation of OCL constraints that ensures that constraints are re-evaluated only if changes may induce their violation and only on elements that caused this violation. However, the approach only works on boolean constraints, and as such it is less expressive than our technique.

An interesting model validator over UML models is presented in [37], which incrementally re-evaluates constraint instances whenever they are affected by changes using an evaluation network. A limitation of this approach is that it only supports a subset of OCL and only permits constraints with a single free variable; therefore, general-purpose model querying is not viable.

To summarize, OCL tools only focus on pure query functionality and only provide simple development aids for detecting syntax errors. Advanced validation and visualization features have, to our best knowledge, not yet been developed. Additionally, while some OCL tools do support incremental evaluation, their current update processing API requires a significant manual coding effort to build data binding and visualization on top of them.

EMF-based model transformation tools For the sake of completeness, we also briefly overview model transformation tools that can be used to provide model queries (even though this is not their primary use case).

EMF-IncQuery is not the first tool to apply graph pattern matching to EMF [38,39], but its incremental (graph) pattern matching feature is unique in an EMF context.

The development environment of *EMF-based model transformation tools* such as ATL [40], Henshin [38], QVTo [41] or eMoflon [42] provide support for specifying, executing and evaluation of transformations. While the sophistication and quality of the development tools provided for model transformation frameworks varies greatly, it can be noted as they focus rather on the more general transformation problem than query development support.

For the ATL (ATL Transformation Language) an incremental transformation approach was published in [43] using OCL impact analysis techniques. Although the solution extends non-incremental ATL, many restrictions were applied: queries and in-place transformations are not supported. Additionally, some OCL constructs were excluded, like the predefined operation `allInstances` used in the benchmark.

7. Conclusion and future work

In previous work, we presented various aspects of EMF-IncQuery technology, focusing on query evaluation efficiency [3], the query language [6], integration of queries using derived features [8] and well-formedness validation rules [11].

In the current paper, building on these previous contributions we presented an evolved and mature integrated development environment that augments the core features with a powerful integrated development environment that leverages various Eclipse technologies to provide advanced query development, integration and visualization features.

The core contributions of the current paper with regard to query development are (i) advanced query validation features that provide instant validation feedback for the most common challenges encountered by EMF-IncQuery users, and (ii) several advanced visualization of query structures and query result that aid developers in fine-tuning query performance.

Additionally, using a case study based on the Papyrus UML tool [10], we highlighted how tool developers can use standard interfaces such as JFace data binding, Ecore derived features and EMF Validation to integrate efficient queries based on EMF-IncQuery into their applications.

Finally, we compared the query evaluation performance of the EMF-IncQuery framework with the Eclipse OCL project, and found that the Rete-based query evaluation of EMF-IncQuery results in very fast incremental query evaluation at a higher (but manageable) memory overhead.

Future work As a main direction for future development work on the existing EMF-IncQuery features, we are planning to support the *query-based abstract visualization* of instance models. Two use cases of this idea would be the parameterization of tree and/or graph viewer so that tuples in the query result may correspond to tree elements, parent-child relationships, or graph nodes and edges, respectively. This technique, analogously with the data binding concept, allows to create abstract visualization of instance models where only important aspects of the model are shown (as defined by a query) and the rest is hidden to improve clarity.

Acknowledgements

The authors would like to thank Ed Willink of the Eclipse OCL project for his help in validating the OCL queries of the published train benchmark and Axel Uhl from the Impact Analyzer developer team for assessing our use of their tool.

This work was partially supported by the CERTIMOT project (ERC_HU-09-1-2010-0003), by the grant TÁMOP – 4.2.2.B-10/1-2010-0009 and the János Bolyai Scholarship. This research was partially realized in the frames of TÁMOP 4.2.4. A/1-11-1-2012-0001 "National Excellence Program – Elaborating and operating an inland student and researcher personal support system". The project was subsidized by the European Union and co-financed by the European Social Fund.

Appendix A. Installation guide

The EMF-IncQuery development environment requires an Eclipse installation with EMF and Xtext installed. To support existing installed domain editors (such as graphical editors based on the GMF or Graphiti projects) further integration options are available. The visualization feature also require the Zest library to be installed.

EMF-IncQuery can be installed using the usual installation methods from the update site http://incquery.net/update/incquery-etsdemo.

Example projects for EMF-IncQuery, including an extended version of the UML example presented here are available in the form of source projects. These projects can be imported into an EMF-IncQuery installation where they can be evaluated without modification.

For more detailed, technical-level guide please consult the web site created to supplement this paper.[2] Further documentation is available in the EMF-IncQuery website,[3] including query examples with detailed description.[4]

Appendix B. Supplementary material

Supplementary material related to this article can be found online at http://dx.doi.org/10.1016/j.scico.2014.01.004.

References

[1] The Eclipse Project, Eclipse modeling framework, http://www.eclipse.org/emf, 2012.
[2] MDT OCL, http://www.eclipse.org/modeling/mdt/?project=ocl, 2012.
[3] G. Bergmann, A. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, A. Ökrös, Incremental evaluation of model queries over EMF models, in: D. Petriu, N. Rouquette, Ø. Haugen (Eds.), Model Driven Engineering Languages and Systems, in: Lect. Notes Comput. Sci., vol. 6394, Springer, Berlin, Heidelberg, 2010, pp. 76–90.
[4] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, G. Varró, Incremental pattern matching in the VIATRA transformation system, in: GRaMoT'08, 3rd International Workshop on Graph and Model Transformation, 30th International Conference on Software Engineering.
[5] The Eclipse Project, EMF model query 2, http://wiki.eclipse.org/EMF/Query2, 2012.
[6] G. Bergmann, Z. Ujhelyi, I. Ráth, D. Varró, A graph query language for EMF models, in: J. Cabot, E. Visser (Eds.), Theory and Practice of Model Transformations, in: Lect. Notes Comput. Sci., vol. 6707, Springer, Berlin, Heidelberg, 2011, pp. 167–182.
[7] Z. Ujhelyi, T. Szabó, I. Ráth, D. Varró, Developing and visualizing live model queries, in: Proceedings of the 1st Workshop on the Analysis of Model Transformations (AMT) @ MoDELS'12, AMT'12, ACM, New York, NY, USA, 2012.
[8] G. Bergmann, A. Hegedüs, A. Horváth, I. Ráth, Z. Ujhelyi, D. Varró, Integrating efficient model queries in state-of-the-art EMF tools, in: C. Furia, S. Nanz (Eds.), Objects, Models, Components, Patterns, in: Lect. Notes Comput. Sci., vol. 7304, Springer, Berlin, Heidelberg, 2012, pp. 1–8.
[9] G. Bergmann, I. Ráth, D. Varró, Parallelization of graph transformation based on incremental pattern matching, in: Proceedings of GT-VMT 2009, in: ECEASST, vol. 18, 2009.
[10] The Eclipse Project, MDT papyrus, http://www.eclipse.org/modeling/mdt/papyrus/, 2012.
[11] G. Bergmann, A. Hegedüs, A. Horváth, I. Ráth, Z. Ujhelyi, D. Varró, Implementing efficient model validation in EMF tools, in: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2011, pp. 580–583.
[12] C.L. Forgy, Rete: A fast algorithm for the many pattern/many object pattern match problem, Artif. Intell. 19 (1982) 17–37.
[13] The Eclipse Project, Xtext, http://www.eclipse.org/Xtext, 2012.
[14] I. Bull, Model driven visualization: Towards a model driven engineering approach for information visualization, Ph.D. thesis, University of Victoria, Victoria, BC, Canada, 2008.
[15] R. Dechter, J. Pearl, Network-based heuristics for constraint-satisfaction problems, Artif. Intell. 34 (1987) 1–38.
[16] The Eclipse Project, JFace data binding, http://wiki.eclipse.org/JFace_Data_Binding, 2012.
[17] The Eclipse Project, EMF validation framework, http://eclipse.org/modeling/emf/?project=validation, 2012.
[18] I. Ráth, A. Hegedüs, D. Varró, Derived features for EMF by integrating advanced model queries, in: A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, D. Kolovos (Eds.), Modelling Foundations and Applications, in: Lect. Notes Comput. Sci., vol. 7349, Springer, Berlin, Heidelberg, 2012, pp. 102–117.
[19] Á. Hegedüs, Á. Horváth, I. Ráth, D. Varró, Query-driven soft interconnection of EMF models, in: 15th ACM/IEEE International Conference on Model Driven Engineering Languages & Systems, in: Lect. Notes Comput. Sci., vol. 7590, Springer, Innsbruck, 2012, acceptance rate: 23%.
[20] The train benchmark website, https://incquery.net/publications/trainbenchmark/full-results, 2013.
[21] Model-based generation of tests for dependable embedded systems, 7th EU framework programme, http://www.mogentes.eu/, 2011.
[22] C. Bizer, A. Schultz, The Berlin SPARQL benchmark, Int. J. Semantic Web Inf. Syst. 5 (2009) 1–24.
[23] M. Schmidt, T. Hornung, G. Lausen, C. Pinkel, SP2Bench: A SPARQL performance benchmark, in: Proc. of the 25th International Conference on Data Engineering, IEEE, Shanghai, China, 2009, pp. 222–233.

[2] http://incquery.net/publications/incquery-development-environment.
[3] http://eclipse.org/incquery.
[4] http://wiki.eclipse.org/EMFIncQuery/UserDocumentation/Examples.

[24] Transaction Processing Performance Council (TPC), TPC-C benchmark, 2010.
[25] G. Varró, F. Deckwerth, M. Wieber, A. Schürr, An algorithm for generating model-sensitive search plans for EMF models, in: Z. Hu, J. Lara (Eds.), Theory and Practice of Model Transformations, in: Lect. Notes Comput. Sci., vol. 7307, Springer, Berlin, Heidelberg, 2012, pp. 224–239.
[26] The Object Management Group, Object constraint language, v2.0, http://www.omg.org/spec/OCL/2.0/, 2006.
[27] A. Uhl, T. Goldschmidt, M. Holzleitner, Using an OCL impact analysis algorithm for view-based textual modelling, ECEASST 44 (2011).
[28] B. Izsó, Z. Szatmári, G. Bergmann, Á. Horváth, I. Ráth, Towards precise metrics for predicting graph query performance, in: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), 11–15 Nov. 2013, pp. 421–431.
[29] MySQL workbench database design. Development. Administration. Migration, http://www.mysql.com/why-mysql/white-papers/mysql-wp-workbench.php, 2012.
[30] InfoSphere data architect, http://www-01.ibm.com/software/data/optim/data-architect/, 2012.
[31] Enterprise Manager, http://www.oracle.com/technetwork/oem/enterprise-manager/overview/index.html, 2012.
[32] Neoclipse, http://neo4j.org/, 2012.
[33] S. Havre, E. Hetzler, K. Perrine, E. Jurrus, N. Miller, Interactive visualization of multiple query results, in: Proceedings of the IEEE Symposium on Information Visualization 2001, INFOVIS'01, IEEE Computer Society, Washington, DC, USA, 2001, p. 105.
[34] L. Hu, K.A. Ross, Y.-C. Chang, C.A. Lang, D. Zhang, Queryscope: visualizing queries for repeatable database tuning, Proc. VLDB Endow. 1 (2008) 1488–1491.
[35] The Eclipse Project, EMFT search, http://www.eclipse.org/modeling/emft/?project=search, 2012.
[36] J. Cabot, E. Teniente, Incremental integrity checking of UML/OCL conceptual schemas, J. Syst. Softw. 82 (2009) 1459–1478.
[37] A. Reder, A. Egyed, Incremental consistency checking for complex design rules and larger model changes, in: R. France, J. Kazmeier, R. Breu, C. Atkinson (Eds.), Model Driven Engineering Languages and Systems, in: Lect. Notes Comput. Sci., vol. 7590, Springer, Berlin, Heidelberg, 2012, pp. 202–218.
[38] E. Biermann, C. Ermel, G. Taentzer, Precise semantics of EMF model transformations by graph transformation, in: MoDELS'08, Springer, 2008.
[39] H. Giese, S. Hildebrandt, A. Seibel, Improved flexibility and scalability by interpreting story diagrams, in: Proceedings of GT-VMT 2009, in: ECEASST, vol. 18, 2009.
[40] The Eclipse Project, ATL – A model transformation technology, http://www.eclipse.org/atl/, 2012.
[41] The Eclipse Project, Model to model project, http://www.eclipse.org/m2m/, 2012.
[42] eMoflon, http://www.moflon.org/, 2012.
[43] F. Jouault, M. Tisi, Towards incremental execution of ATL transformations, in: L. Tratt, M. Gogolla (Eds.), Theory and Practice of Model Transformations, in: Lect. Notes Comput. Sci., vol. 6142, Springer, Berlin, Heidelberg, 2010, pp. 123–137.