Visualizing Results of Live Model Queries

Zoltán Ujhelyi
ujhelyiz@mit.bme.hu

Tamás Szabó
szabta89@gmail.com

István Ráth
rath@mit.bme.hu

Dániel Varró
varro@mit.bme.hu
Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1117 Magyar tudósok krt. 2., Budapest, Hungary

ABSTRACT

Several important tasks performed by model driven development tools — such as well-formedness constraint validation or model transformations — rely on evaluating model queries. If the model changes rapidly or frequently, it is beneficial to provide live queries that automatically propagate these model changes into the query results. To ease the development and debugging of live queries, the development environment should provide a way to evaluate the query results continuously, helping to understand how the created query works.

This paper presents a generic live model query visualizer that displays and updates the query results depending on their source models. It has been implemented for the EMF-IncQuery framework and presented here for validating BPMN models.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.2 [**Software Engineering**]: Computer-aided software engineering (CASE)

Keywords

Live model queries, graphical user interface, integrated development environment

1. INTRODUCTION

Model queries play a central role in many scenarios in model driven engineering ranging from well-formedness constraint evaluation to model synchronization and calculating dependencies or metrics between system components. By definition, a model query retrieves those elements from a model that satisfy a query condition, which can be a complex composition of subconditions.

During the evaluation of these model queries, one can distinguish between on-demand and live evaluation strategies. In case of an on-demand strategy, query re-evaluation is initiated explicitly by the user or a transformation program, and the result set is typically calculated from scratch. In case of live model queries, the query results are updated automatically when the underlying model changes (without explicit user request). This is a useful feature for validation in model editors [3] as well as for live model transformations in model synchronization scenarios [12].

The main motivation of our current work is to provide an integrated development environment (IDE) for designing and debugging live model queries. On the one hand, such an IDE should include a powerful query editor (built upon a domain-specific language framework such as Xtext or GMF) to support the specification of queries. On the other hand, a query IDE should also support the evaluation and debugging of queries, which is the main focus of the current paper.

This paper presents an Eclipse IDE for visualizing the results of live model queries. Our solution is (i) generic in the sense that it accepts models developed by arbitrary model editors. Thanks to the incremental evaluation strategy of the underlying EMF-IncQuery framework, (ii) it provides live view of the query results (for all queries over all model instances). Furthermore, to improve usability by avoiding information overload, it also provides various filtering strategies to control the information presented to the query developers. Our approach is exemplified by visualizing live queries in the industry standard EMF models, but its underlying architecture allows easy adaptation for other modeling frameworks.

In the rest of the paper, Section 2 discusses the challenges of providing a powerful IDE for query development using BPMN models as a motivating example. Then, our proposal is outlined in Section 3 including its requirements, architecture, and user interface issues. Related work on debugging queries and transformations is provided in Section 4, while Section 5 concludes our paper.

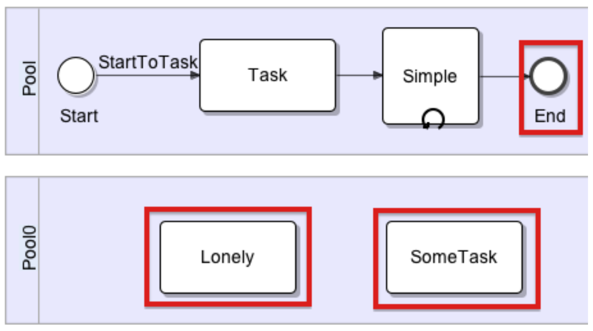


Figure 1: A simple BPMN workflow with sink activities

2. DEVELOPING LIVE MODEL QUERIES

2.1 Case Study: Validation of BPMN Models

Our approach will be exemplified using simple model queries for the Business Process Model and Notation (BPMN [11]) modeling language. BPMN is a widely used standard for specifying business processes featuring a flowchart-like notation.

BPMN models both control and data flow explicitly; for control flow, activities, events and gateways (e.g. conditional decisions) are available, while data flow may be specified between activities and artifacts. The elements of the process are organized into pools and swimlanes to represent e.g. the organizational structure.

When managing BPMN models, additional structural well-formedness constraints should be validated, for example *sink activities* – activities with no outgoing control or data flow edge – should only be used as final activities of a workflow.

EXAMPLE 1. *To illustrate BPMN models, Figure 1 depicts a simple BPMN workflow with two pools and six activities (one start, one end and four regular activities). Additionally, the sink activities are framed.*

2.2 Incremental Model Queries Using EMF-IncQuery

The EMF-IncQuery framework [3] provides a development environment for incremental model queries based on the formalism of graph patterns [4]. Graph patterns represent conditions or constraints that are to be fulfilled by a part of an instance model. These conditions are either *structural constraints* prescribing the existence of nodes and edges of a given type or *expressions* to define attribute constraints. Additionally, graph patterns can refer to other patterns using the *find* construct and *negative application conditions* define cases where the original pattern is *not* valid even if all other constraints are met.

The EMF-IncQuery framework features a query engine based on the Rete rule evaluation network [6], an incremental pattern matching technique [12], that (1) performs well even in the range of millions of elements and (2) provides notifications for query result changes as well.

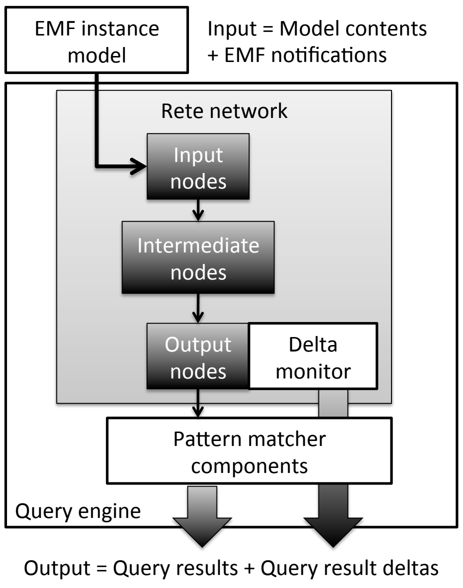

EMF instance model
Input = Model contents + EMF notifications
Rete network
Input nodes
Intermediate nodes
Output nodes
Delta monitor
Pattern matcher components
Query engine
Output = Query results + Query result deltas

Figure 2: The EMF-IncQuery Architecture

Figure 2 gives an overview of the architecture of EMF-IncQuery. The query engine uses EMF instance models as input, and initializes a Rete network based on the model contents. Additionally, the engine registers itself to receive notifications for model changes to update the Rete network accordingly. The pattern matcher components package the results stored in the network as query results, and makes result deltas available using attached delta monitors.

During query evaluation query parameters might be left unbound, returning all possible matches. However, binding some parameters to a concrete value (most commonly model elements) effectively filters the match results by removing matches without the selected elements.

EXAMPLE 2. *Figure 3 specifies how to identify sink activities using graph patterns as defined in the query language of* EMF-IncQuery.

The pattern `sinkActivityNames` *consists of three constraints: (1) there is an* `Activity` *marked* `A` *(2) with a defined name* `Name`*, that (3) has no outgoing edges – this constraint is defined with a negative pattern call.*

The pattern `hasOutEdge` *is used to represent the activities that have either an outgoing sequence flow edge or an outgoing message flow edge. Both of these edges are captured as a corresponding pattern that represents the connection between the selected activity and another, non-specified one.*

The pattern `sequenceFlowEdge` *(and the omitted, but very similar* `messageFlowEdge` *pattern) describes an edge whose source and target activities are set according to the parameters.*

2.3 Debugging Incremental Model Queries

```
/*Calculates the Name of a sync activity*/
pattern sinkActivityNames(Name) {
 Activity(A);
 Activity.name(A, Name);
 neg find hasOutEdge(A);
}

pattern hasOutEdge(A: Activity) {
 find sequenceFlowEdge(_From, A, _Other);
} or {
 find messageFlowEdge(_From, A, _Other);
}

pattern sequenceFlowEdge(
           Flow:SequenceEdge,
           Src:Activity, Dst:Activity) {
 SequenceEdge.source(Flow, Src);
 SequenceEdge.target(Flow, Dst);
}
```

Figure 3: Example Graph Patterns

The high-level, declarative nature of query definitions in EMF-IncQuery can make it hard to understand all corner cases of complex patterns without evaluating them (possibly using multiple models). Additionally, creating composite patterns can be even more error-prone, as the corner cases are often distributed in different pattern definitions. In our experience this means some manual experimentation and debugging is needed to identify inconsistencies in query definitions. In this section, we exemplify the main challenges based on the evaluation of the Sink Activities query.

To evaluate the newly written queries, it is important to create experimental BPMN models – for this reason, the existing BPMN editors should be reused, in our case, the one included in the Eclipse SOA Tools Platform suite [16].

The BPMN editor may also be used to display the query results on top of the original model (for example, by coloring the result set). However, in case of large result sets it becomes hard to differentiate between different matches, and there is no generic, editor-independent way to implement this functionality.

The evaluation of live queries is impractical using interactive consoles, such as the Interactive OCL Console of Eclipse [1], because in case of model changes it is required to manually re-execute the query to obtain and compare the modified results.

For these reasons, we propose to use a *Query result viewer* that displays the results of a set of queries, and automatically updates them when either the instance models or the query definitions are changed. To achieve this, the viewer listens to query result deltas coming from the Query engine.

As the viewer is a separate user interface component, for traceability it needs also allow navigating to the referenced model elements directly in the already opened editor.

Additionally, filtering the visible set of queries is highly beneficial. E.g., when evaluating the `sinkActivityNames` pattern from Figure 3, only the `hasOutEdge`, `sequenceFlowEdge` and `messageFlowEdge` patterns are interesting, all the others are to be hidden.

Finally, during the evaluation of the `sinkActivityNames` pattern we are often interested in whether a selected Activity is included in the results or not. For this reason, the viewer needs to support filtering the results by binding input parameters for queries.

3. A DEVELOPMENT ENVIRONMENT FOR LIVE MODEL QUERIES

In this section, we propose a live model query visualizer extension to integrated domain-specific modeling environments. We assume the environment already has some editors for the selected instance models and the query definitions, and design the visualizer to extend their functionality with live model query visualization.

3.1 Requirements

Based on our experience with developing incremental model queries we defined the following list of requirements:

R1. **Genericity**: The created visualizer should be able to load the input models from the selected model editor (regardless of its type), and also from any specific model viewer based on the current selection.

    This way, it is not required for the query developer to provide the test instance models using an editor known in advance, while providing an additional viewer for the query results.

R2. **Incrementality**: The visualizer should react automatically to changes both in the query (the developer edits an already loaded query) and the model (executing a model manipulation step), while keeping the display of query results consistent with the actual model.

    This requirement relates to the live nature of the model queries by providing an always up-to-date display of the query results.

R3. **Traceability**: The visualizer needs to maintain its model and query sources, and it should allow the user to navigate back to the selected model element in the input model and the query definitions.

    This requirement helps establish a connection between the displayed query results and the corresponding model elements, thus assisting the developer to put the results in context.

R4. **Presentation**: The visualizer should provide a focused user interface using filtering and grouping mechanisms. Additionally, the default filtering must be defined to match common uses cases to reduce the need for manual filtering.

Requirements R1 and R2 provide the base live model query viewer functionality. On the other hand, requirements R3 and R4 are needed to present the features to the query developer in an understandable and easy-to-use way.

3.2 Architecture

To fulfill the requirements, we propose the architecture depicted in Figure 4. The user interface collects its data from two components: the *Query Repository* and the *Result Viewer*.

The *Query Repository* component collects and manages all available *queries*. The Query Repository also allows the definition of *query groups* – a set of queries that can be managed together. These groups can be created automatically based on the storage (for example the file or project that defines the query). Furthermore, it is possible to define custom groups - either manually, or by analyzing the dependencies or similarities between various queries.

The *Result Viewer* component reads a set of queries and query groups from the Query Repository – either relying on the default filtering rules, or allowing manual filtering. Then the Result Viewer applies the filters to a set of available models, most commonly an already existing *model editor*. It is possible to use different sources, like the current selection of the IDE, or a custom developed data source – such as the models of a currently executed model transformation.

As these model sources (even existing editor technologies) provide different ways of reading their model (and possibly setting up traceability on the user interface for Requirement R3), we define a *Model source connector* interface that handles all sources in a generic way. A model source connector has two responsibilities: (1) it has to know how to attach itself into the model source, and return the defined model as requested, and (2) provide notifications of model changes, including the inavailability of the model source (e.g. after the closing of a model editor).

When the models and the queries are both available, the Result Viewer initializes query evaluators for each query and model – including a single traversal of the input model and the setup phase for change notification handling. Additionally, if *result filtering* is initialized in the user interface, the query parameters are bound to the selected model element.

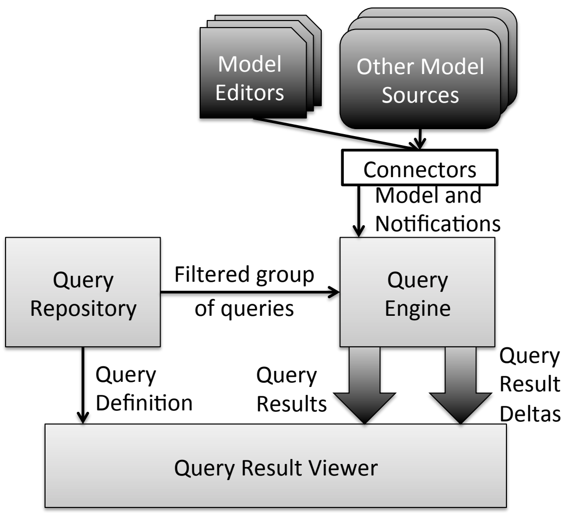


Figure 4: The Proposed Architecture

3.3 Incremental User Interface Updates

An engine for Live Model Queries, such as EMF-IncQuery, already reacts on *instance model changes* and updates the modified query results incrementally. By subscribing to result change notifications, the user interface is capable of reflecting the changes in the instance model in a performant way. Additionally, if the connected model source becomes unavailable (such as when closing the model editor) a cleanup phase is invoked inside the query engine to ensure consistency.

However, in case of *pattern definition changes*, more complex model-view reconciliation steps are to be taken, since the query engine maintains a query-specific view of the input model that is typically invalidated when the definition of the query is changed. Instead of a complete re-initialization, our system uses a more efficient strategy whereby additional model traversals are avoided. The key idea is to use generic model indexers (provided by a low-level incremental query library called EMF-IncQuery Base[1]) that allow the pattern matcher to track elementary changes *for all model elements* (regardless of the query definition). This way, at the cost of a slight runtime memory usage increase, the system is capable of incrementally updating query results when query definitions change, without having to perform model traversals.

While this mode of operation (called *wildcard mode* in EMF-IncQuery) is on by default, the system also allows the user to turn it off (in which case model retraversals will be performed as a fallback option) to allow the user to reduce the memory overhead to a minimum, when working with large instance models.

3.4 Selecting Visible Queries

As the number of queries (especially if counting the subqueries) increases, the amount of information presented by the visualizer may exceed the comfortable usability limit. To overcome this issue, we propose to use a query filtering mechanism.

By our approach, queries can be annotated with visibility information by the developer: by marking a pattern invisible it is possible to avoid displaying some internal patterns. If this annotation is not set, the filter defines different rules for *workspace* and *packaged* queries. As the latter ones represent already tested queries, they are filtered out by default. On the other hand, as queries in the *workspace* are under development, they are displayed unless the developer states otherwise.

An additional way to filter displayed queries is to filter by corresponding metamodels: if a model read from the model source is not based on the same metamodel the query refers to, the result will be always empty, making the query safe to filter out. However, this filter necessitates the gathering of the referenced metamodels both from the query definition and the underlying model that is hard to implement in some corner cases using EMF technologies.

[1] `http://viatra.inf.mit.bme.hu/incquery/base`

Similarly, the environment can rely on hints from the query developer to restrict what kind of model sources the selected query is compatible with. For example, the developer can select an editor type or file extension for the query to activate on.

3.5 Implementation and Evaluation

The proposed live query visualizer has been implemented for EMF-IncQuery, extending its Eclipse-based user interface (as depicted in Fig. 5).

The visualizer features a three-pane design: (1) the queries stored in the *Query Repository* are displayed in a collapsible pane on the left, (2) the *Result Viewer* is displayed as a tree hierarchy that presents the loaded models and the corresponding queries in the middle, and (3) a *Detail pane* is used to display the values of the single result parameters, and optionally a place to set up result filtering.

For loading queries and instance models from currently open editors, a single toolbar button is used that automatically selects the best applicable *Model source connector*. Additional load options (such as loading only a subset of the input model) are available from a dropdown menu (Requirement R1).

Instance models (and the corresponding queries) are unloaded automatically if the model becomes unavailable (for example while closing the editor). It is also possible to manually unload a selected model by using the pop-up menu in the tree hierarchy.

The displayed query results are updated during model changes, and in case of query modifications the modified queries are reloaded as discussed in subsection 3.3 (Requirement R2).

By double clicking on a query in the *Result Viewer* the defining query is opened in its corresponding editor Similarly, by double clicking on a model element in the *Detail Pane*, the corresponding model element is revealed and selected in its editor (Requirement R3).

Finally, we carefully selected the default filtering rules to provide a useful default query selection for most uses, while remaining versatile enough to evaluate queries efficiently in more complex cases (Requirement R4).

Example 3. *The visualizer is illustrated in Figure 5a while displaying the previously defined query* `sinkActivityName` *and its used helper queries. The* Query Repository *is closed, while the detail pane shows a selected result.*

After editing the input model by entering an additional sequence edge, one activity (Lonely) is no longer a sink activity, and the result set is updated automatically as shown in Figure 5b.

This visualizer is already used by the developers of EMF-IncQuery, and we are planning to run a more detailed evaluation based on the feedback of the users of the framework. For now, we argue that this implementation fulfills all our requirements and is useful during the evaluation of developed queries.

4. RELATED WORK

Debugging of model transformations. Certain debugging support is provided in many model transformation tools including ATL, GReAT, VIATRA, FUJABA, Tefkat, and many more. The authors of [5] propose a dynamic tainting technique for debugging failures of model transformations, and propose automated techniques to repair input model faults [10]. Colored Petri nets are used for underlying formal support for debugging transformations in [13] extended in the subsequent PhD thesis [14]. The debugging of triple graph grammar transformations is discussed in [15].

A forensic debugging approach of model transformations was introduced in [8] by using the trace information of model transformation executions in order to determine the interconnections of source and target elements with transformation logic. Dynamic backward slicing of model transformations was proposed recently in [17] for debugging purposes.

Visualizing query results. Several Eclipse-based model query tools (such as [1, 2]) allow to compute the result set of a query, but they do not follow a live update approach, thus re-computation is initiated only upon user's demand.

In the database community, the results of multiple queries were visualized simultaneously in [7]. The structure of queries can also be visualized as proposed in [9] However, none of these results are directly adaptable in querying domain-specific models captured in EMF.

5. CONCLUSIONS AND FUTURE WORK

In the paper, we presented an Eclipse-based environment for visualizing results of live model queries. Query results are updated immediately and automatically after the underlying model or query changes. Query results can be simultaneously observed for multiple queries and multiple models. Our approach allows direct traceability from query results to arbitrary model editors over EMF models. Finally, to improve usability, we developed several strategies for filtering the result set.

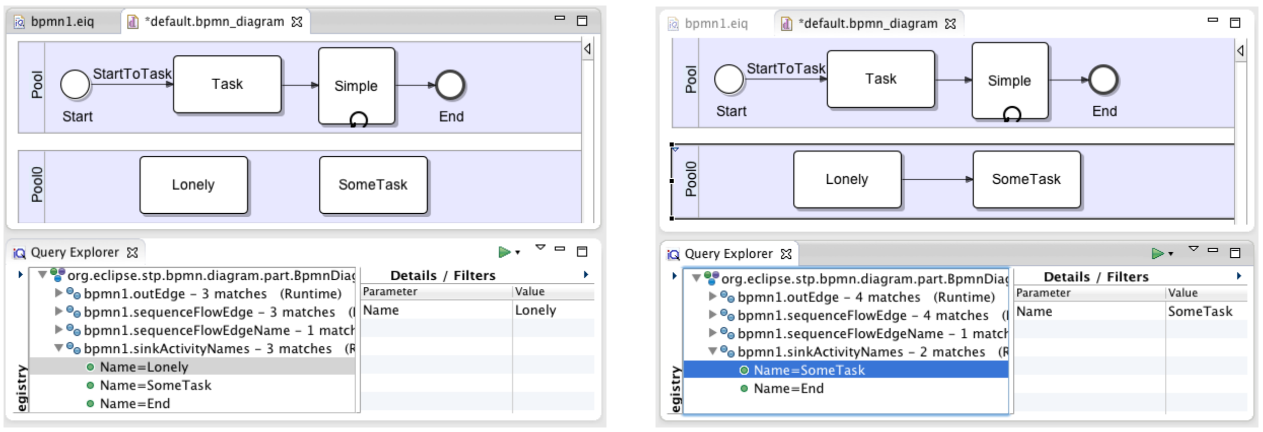
In the future, we plan to improve our approach by dynamic grouping of graph patterns based on dependencies and similarities between them. This helps query developers to observe the differences between the result sets of similar patterns more easily.

6. ACKNOWLEDGMENTS

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 János Bolyai Scholarship.

7. REFERENCES

[1] Eclipse MDT OCL project. `http://www.eclipse.org/modeling/mdt/?project=ocl`.


bpmn1.eiq
*default.bpmn_diagram
Pool
StartToTask
Task
Simple
Start
End
Pool0
Lonely
SomeTask
Query Explorer
org.eclipse.stp.bpmn.diagram.part.BpmnDiag
bpmn1.outEdge – 3 matches (Runtime)
bpmn1.sequenceFlowEdge – 3 matches
bpmn1.sequenceFlowEdgeName – 1 match
bpmn1.sinkActivityNames – 3 matches
Name=Lonely
Name=SomeTask
Name=End
Details / Filters
Parameter
Value
Name
Lonely
egistry
bpmn1.eiq
*default.bpmn_diagram
Pool
StartToTask
Task
Simple
Start
End
Pool0
Lonely
SomeTask
Query Explorer
org.eclipse.stp.bpmn.diagram.part.BpmnDiag
bpmn1.outEdge – 4 matches (Runtime)
bpmn1.sequenceFlowEdge – 4 matches
bpmn1.sequenceFlowEdgeName – 1 match
bpmn1.sinkActivityNames – 2 matches
Name=SomeTask
Name=End
Details / Filters
Parameter
Value
Name
SomeTask
egistry

(a) Displaying Query Results (b) Live Result Update

Figure 5: The Implemented Query Result Browser

[2] MOMENT OCL project. `http://moment.dsic.upv.es/infocenter/index.jsp?topic=/es.upv.dsic.issi.moment.ocl.help/html/intro/intro.htmll`.

[3] G. Bergmann, A. Hegedüs, A. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró. Integrating efficient model queries in state-of-the-art EMF tools. In C. Furia and S. Nanz, editors, *Objects, Models, Components, Patterns*, volume 7304 of *Lecture Notes in Computer Science*, pages 1–8. Springer Berlin / Heidelberg, 2012. 10.1007/978-3-642-30561-0_1.

[4] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró. A graph query language for EMF models. In J. Cabot and E. Visser, editors, *Theory and Practice of Model Transformations*, volume 6707 of *Lecture Notes in Computer Science*, pages 167–182. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-21732-6_12.

[5] P. Dhoolia, S. Mani, V. S. Sinha, and S. Sinha. Debugging model-transformation failures using dynamic tainting. In *24th European conference on Object-oriented programming*, pages 26–51. Springer, 2010.

[6] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, Sept. 1982.

[7] S. Havre, E. Hetzler, K. Perrine, E. Jurrus, and N. Miller. Interactive visualization of multiple query results. In *Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*, INFOVIS '01, pages 105–, Washington, DC, USA, 2001. IEEE Computer Society.

[8] M. Hibberd, M. Lawley, and K. Raymond. Forensic debugging of model transformations. In *Model Driven Engineering Languages and Systems, 10th Int. Conf.*, volume 4735 of *LNCS*, pages 589–604. Springer, 2007.

[9] L. Hu, K. A. Ross, Y.-C. Chang, C. A. Lang, and D. Zhang. Queryscope: visualizing queries for repeatable database tuning. *Proc. VLDB Endow.*, 1(2):1488–1491, Aug. 2008.

[10] S. Mani, V. S. Sinha, P. Dhoolia, and S. Sinha. Automated support for repairing input-model faults. In *25th IEEE/ACM Int. Conf. on Automated Software Engineering*. ACM, 2010.

[11] Object Management Group. Business Process Model and Notation (BPMN) Version 1.2. `http://www.omg.org/spec/BPMN/1.2/`.

[12] I. Ráth, G. Bergmann, A. Ökrös, and D. Varró. Live model transformations driven by incremental pattern matching. In *Theory and Practice of Model Transformations*, volume 5063/2008 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2008.

[13] J. Schoenboeck, G. Kappel, A. Kusel, W. Retschitzegger, W. Schwinger, and M. Wimmer. Catch me if you can - debugging support for model transformations. In *Model Driven Engineering Languages and Systems, 13th Int. Conf.* Springer, 2010. LNCS 6002.

[14] J. Schönböck. *Testing and Debugging of Model Transformations*. PhD thesis, 2012.

[15] M. Seifert and S. Katscher. Debugging triple graph grammar-based model transformations. In *Fujaba Days*, pages 19–25, 2008.

[16] SOA Tools Platform. Eclipse BPMN Modeler. `http://www.eclipse.org/bpmn/`.

[17] Z. Ujhelyi, Á. Horváth, and D. Varró. Dynamic backward slicing of model transformations. In *International Conference on Software Testing and Validation (ICST 2012)*. IEEE, 04/2012 2012.