Local Search-Based Pattern Matching Features in EMF-IncQuery

Márton Búr[1,2], Zoltán Ujhelyi[1,2(✉)], Ákos Horváth[1,2], and Dániel Varró[1]

[1] Department of Measurement and Information Systems,
Budapest University of Technology and Economics, Magyar Tudósok krt. 2,
Budapest 1117, Hungary
marton.bur@inf.mit.bme.hu, varro@mit.bme.hu
[2] IncQuery Labs Ltd., Bocskai út 77-79, Budapest 1113, Hungary
{ujhelyi,horvath}@incquerylabs.com

Abstract. Graph patterns provide a declarative formalism to describe model queries used for several important engineering tasks, such as well-formedness constraint validation or model transformations. As different pattern matching approaches, such as local search or incremental evaluation, have different performance characteristics (smaller memory footprint vs. smaller runtime), a wider range of practical problems can be addressed. The current paper reports on a novel feature of the EMF-IncQuery framework supporting local search-based pattern matching strategy to complement the existing incremental pattern matching capabilities. The reuse of the existing pattern language and query development environment of EMF-IncQuery enables to select the most appropriate strategy separately for each pattern without any modifications to the definitions of existing patterns. Furthermore, a graphical debugger component is introduced that visualizes the execution of the search process, helping to understand how complex patterns behave. This tool paper presents the new pattern matching feature from an end users viewpoint while the scientific details of the pattern matching strategy itself are omitted. The approach is illustrated on a case study of automated identification of anti-patterns over program models created from Java source code.

Keywords: Local search-based pattern matching · EMF-IncQuery · Integrated development environment

1 Introduction

Model queries form the underpinning of various engineering tasks, such as model transformation, code generation or well-formedness validation. Declarative query formalisms (such as graph patterns or OCL constraints) define queries at a high level of abstraction allowing the use of different execution strategies such as local search-based or incremental pattern matching.

This work was partially supported by the MONDO (EU ICT-611125) project.

© Springer International Publishing Switzerland 2015
F. Parisi-Presicce and B. Westfechtel (Eds.): ICGT 2015, LNCS 9151, pp. 275–282, 2015.
DOI: 10.1007/978-3-319-21145-9_18

Experimental evaluations of the two strategies (like in [1]) demonstrated that incremental approaches, which rely on caching the result sets of queries, provide an order of magnitude faster re-evaluation time, but they also result in larger memory footprint and longer initialization phase compared to local search-based pattern matching. These different performance characteristics makes various strategies or approaches most useful for different kinds of problems.

While EMF-IncQuery has traditionally been tailored to provide incremental evaluation over graphs captured as EMF models, these experimental findings have triggered us to extend the EMF-IncQuery framework with a new feature to support local search based evaluation for queries integrated to the query development environment, which is reported in the current paper. The reuse of the existing pattern language and query development environment of EMF-IncQuery enables to select the most appropriate strategy separately for each pattern without any modifications to the definitions of existing patterns. In addition, we also report on a prototype graphical debugger to trace local search based evaluation. The novel features will be presented in the context of a case study aiming to detect anti-patterns in Java programs [1].

The rest of the paper is structured as follows. Sect. 2 gives a brief overview of graph patterns and EMF-IncQuery that is followed in Sect. 3 by an overview of local search based pattern matching. Then Sect. 4 presents the graphical debugger for pattern matching. Sect. 5 summarizes related work, and Sect. 6 concludes the paper discussing directions for future work.

2 Model Queries with EMF-IncQuery

2.1 Running Example: Anti-pattern Detection in Java Programs

In the current paper we will use the automated detection of coding anti-patterns over Java programs to demonstrate the local search support. As metamodel, the Java metamodel of the Columbus framework is used, together with a set of anti-patterns introduced in [1].

Example 1. Figure 1a presents a Java code snippet. The code consists of a public method called `equals` with a single parameter and a call of this method using a Java variable `srcVar`. This snippet shows an anti-pattern: the call `equals` can result in an exception if the variable `srcVar` is `null`. However, by swapping the literal parameter with the variable operand, no such exception could occur.

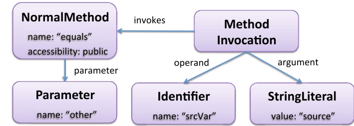
The model representation of this snippet is depicted in Fig. 1b as a typed graph. Each node represents an element of the abstract syntax graph of the Java model. To ease readability, several attribute values were omitted which are not required to understand the contributions and examples in this paper (such as the final flag of parameter definitions).

2.2 Graph Patterns

A *graph pattern* consists of *structural constraints* prescribing the interconnection between nodes and edges of given types and *expressions* to define *attribute*

```
public boolean equals(Object other) {...}
...
// Code inside another method
// The variable 'srcVar' is defined locally
srcVar.equals("source");
...
```

(a) Java Code Snippet



(b) Model Representation

Fig. 1. ASG representation of Java code

constraints. Both constraints can be illustrated as a graph where the nodes are typed as classes from the metamodel, while the edges prescribe the required connections of selected types between them. *Pattern parameters* are a subset of nodes and attributes interfacing the model elements interesting from the perspective of the pattern user. A *match* of a pattern in a model M is a binding of all pattern parameters to model elements of M that satisfies all constraints expressed by the pattern.

Complex patterns may reuse other patterns by different types of *pattern composition constraints*. A *(positive) pattern call* identifies a subpattern (or called pattern) that is embedded as an additional set of constraints while a *negative pattern call* invalidates cases when a match of the referred pattern is found.

Example 2. Figure 2 captures the "String Literal as Compare Parameter" problem as a graph pattern using the textual syntax of EMF-IncQuery that describes a case when a String literal is used as the argument of an `equals` call.

```
1  pattern stringLiteralCompare(
2    inv : MethodInvocation) {
3    StringLiteral(arg);
4    Expression(op);
5    NormalMethod(m);
6    MethodInvocation.invokes(inv, m);
7    MethodInvocation.operand(inv, op);
8    NormalMethod.name(m, "equals");
9    neg find literal(op);
10   find argument(inv, arg);
11   1 == count find argument(inv, _);
12 }
```

```
13 /** 'lit' is a Literal. */
14 pattern literal(lit : Literal){
15   Literal(lit);
16 }
17
18 /** 'arg' is an argument of
19     the invocation 'inv' */
20 pattern argument(
21   inv : MethodInvocation,
22   arg : Expression) {
23   MethodInvocation.arguments(inv, arg);
24 }
```

Fig. 2. Graph pattern representation of the string literal compare anti-pattern

The pattern consists of four variables : `inv` (of type `MethodInvocation`), `m` (of type `NormalMethod`), `op` (of type `Expression`) and `arg` (of type `StringLiteral`). The constraint in Line 6 represents a typed reference `invokes` between the model elements selected by `inv` and `m`, and a similar `operand` reference is required between variables `m` and `op`. Variable `m` is part of an attribute constraint in Line 8: its name attribute has to be the literal `"equals"`. To ensure that the operand `op` of the method invocation is not a `Literal`, a negative pattern call is used in Line 9. Finally, to confirm that the invoked method has only a single parameter, the

number of arguments are counted in Line 11 by counting the number of matches of the subpattern `argument` and checking if it equals to 1.

2.3 The Query Development Environment of IncQuery

EMF-IncQuery provides an integrated development environment where graph patterns can be created and debugged [2]. The environment consists of three major components: (1) a pattern editor to create queries, (2) the Query Explorer to display the results of various queries, and (3) a code generator creating a pattern matcher that can be integrated into existing Java (EMF) applications.

The Xtext-based pattern editor helps query development with advanced features such as syntax highlighting, code completion and well-formedness validation rules that check for common developer mistakes.

The *Query Explorer* is the main debugging component of EMF-IncQuery as it continuously evaluates the developed queries with changes of the model from a model editor, it is possible to find problematic cases of complex queries by modifying the models in the existing model editors, and watching for the expected query result changes. The Query Explorer relies on the pattern interpreter support of EMF-IncQuery instead of the generated code itself. This eases the development and debugging of graph patterns, as changes in the patterns can be evaluated in the development environment directly.

3 Local-Search Pattern Matching in IncQuery

3.1 Executing a Local Search Based Matching Strategy

Local search based pattern matching (LS) is commonly used in graph transformation tools [3,4] starting the match process from a single node and extending it step-by-step with the neighboring nodes and edges following a *search plan*.

Local search based pattern matching consists of four steps. (1) At first, in a preprocessing step the patterns are *normalized*: the constraint set is minimized, variables that are always equal are unified and positive pattern calls are flattened. These normalized patterns are evaluated by (2) the *query planner*, using a specified cost estimation function to provide search plans [5]: totally ordered lists of search operations used to ensure that the constraints from the pattern definition hold. From a single pattern specification multiple search plans can be derived, thus pattern matching includes (3) *plan selection* based on the input parameter binding and model-specific metrics. Finally, (4) *the search plan is executed* by a plan interpreter evaluating the different operations of the plans. If an operation fails, the interpreter backtracks; if all operations are executed successfully, a match is found.

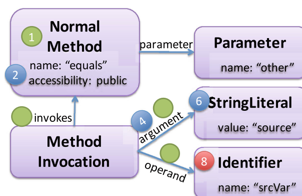
Example 3. To evaluate the String Literal Compare pattern from Fig. 2, a possible 8-step search plan is presented in Fig. 3a. First, (1) all `NormalMethod` instances are iterated over to (2) check for their name. Then a (3) backward navigation operation is executed to find all corresponding method invocations

to check (4–6) its argument and (7) operand references. At the last step, (8) a negative pattern call is executed by starting a new plan execution for the negative subplan, but only looking for a single solution. Note that the positive pattern call from Line 10 is flattened, resulting in operation (5), while the match counter and negative pattern calls from Line 11 and Line 9 are represented by pattern calls in operations (4) and (8), respectively.

Figure 3b illustrates the execution of the search plan on the simple instance model introduced previously. First, the `NormalMethod` is selected, then its `name` attribute is validated, followed by the search for the `MethodInvocation`. At this point, following the `argument` reference made it sure that only a single element is available, then the `StringLiteral` is found and checked. Finally, the `operand` reference is followed, and a NAC check is executed using a different search plan.

Operation	Note
1: Find all `m` that `m` ∈ `NormalMethod`	
2: Attribute test: `m.name=="equals"`	
3: Find `inv` that `inv.invokes` → `m`	
4: Count of `inv.argument` → `arg` is 1	Count
5: Find `arg` that `inv.argument` → `arg`	Flattened
6: Instance test: `arg` ∈ `StringLiteral`	
7: Find `op` that `inv.operand` → `op`	
8: NEG: `op` ∉ `Literal`	Negative

(a) A Possible Search Plan



(b) Search Plan Execution

Fig. 3. A search plan for the string literal compare pattern

3.2 Local Search Support in IncQuery

The local search feature of EMF-IncQuery relies on the existing features created for incremental pattern matching as much as possible. This includes the reuse of both the pattern language (together with its editor), pattern interpreter and the code generator framework itself. Furthermore, the generated local search based matchers provide code that is a drop-in replacement for existing, incremental ones (with the notable exception of not providing change notifications for the result set). The reuse of the pattern language with a common runtime API allows to specify the patterns once, while being able to select the corresponding strategy later based on the constraints of the created applications.

The search planner component relies on a cost function that estimates how expensive is to evaluate a selected constraint based on the already bound variables. Currently, only a simplistic cost function is implemented, but it was designed to be extensible with additional strategies, such as the dynamic programming based approach in [5].

Additionally, the local search-based pattern matcher can optionally reuse the model indexer of EMF-IncQuery for iterating over all instances of model

elements or traverse model edges backwards. This option allows fine-grained performance tuning of pattern matching, as reusing model indexes can greatly reduce search time, while requiring much less memory than Rete-based incremental matcher. In [1] we have evaluated the performance of search-based and incremental approaches, and found that incremental graph pattern matching can outperform other approaches in case of repeated execution of the same pattern, as search times are an order of magnitude smaller, at the cost of a longer initialization period and additional increase in memory cost by a factor of $10 - 15$.

4 Debugging Model Queries with Local Search

The high-level, declarative nature of graph patterns sometimes results in hard to understand corner cases. In such cases simply looking at match results, as supported by the Query Explorer, does not provide enough details to locate the source of the problem. To support this use case, the development environment of EMF-IncQuery has been extended with a *Local Search Debugger* view that follows through the execution of a search plan created for a pattern over a model.

As constraints of graph patterns are often not evaluated in the order of their definitions, it is hard to see which constraints are already evaluated. On the other hand, the ordered search operations visualize the status of pattern matching, and can be traced back to the source query. The view can also be used for query optimization, similar to explain plans [6] used for optimizing SQL queries.

As Fig. 4 depicts, the view has three distinct parts to display information about the execution. At the upper left corner (a) the search plan itself is shown, including the plans created for called patterns. Each line represents a search operation; child nodes are operations of a called pattern. The current status of the execution is depicted with a set of icons: check marks are assigned to executed operations, question marks are assigned to operations not yet started, while the current operation is denoted with the 'Run' symbol.

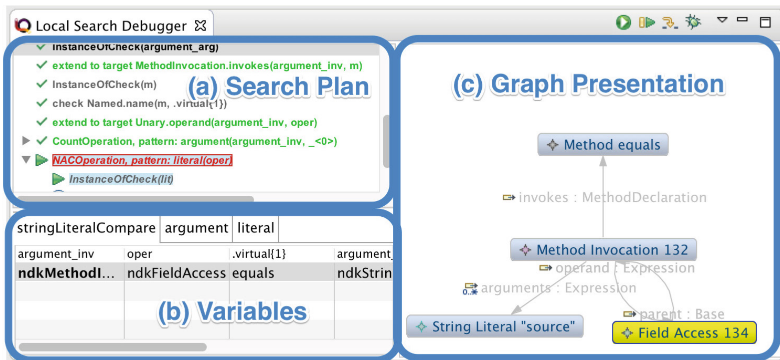


Fig. 4. The local search debugger

In the bottom left corner (b) a set of tables is presented summarizing the found matches. The tables include the found matches of all patterns in different tables, including both parameters and local variables. Finally, in the right side (c) of the view, a graph representation is provided for the currently evaluated (partial) match, showing the current substitutions for the pattern variables along with the relationships between them.

Finally, to control the execution, standard debugging operations are available [7]: breakpoints can be assigned to search operations, and both step-by-step and continuous execution modes are available.

This view complements the debugging capabilities of the Query Explorer, as the latter one is useful for identifying problematic cases by providing live feedback when the model changes, the debugger visualizes the detailed execution of the search. The local search algorithm, in our experience, works similarly as a query developer reasons about a graph pattern, thus it eases the understanding of complex graph patterns.

5 Related Work

Local search-based pattern matching is commonly used in graph transformation tools, such as FUJABA [3], GrGen.NET [4] or FunnyQT [8]. The main difference between the various approaches are the supported modeling backends, the search planner algorithm and the cost estimation used during planning. For example, in [5] an adaptive algorithm is proposed that uses dynamic programming to estimate plan costs.

The debugging of graph transformations is already well-researched [7]; GrGen.NET [4] already incorporates a visual debugger for its transformation, that can visualize the models being transformed, and can highlight elements matched by a graph pattern; however, it does not support stepping through the pattern matching process manually.

The Eclipse OCL tool [9] reuses the debugger interface of Eclipse for stepping through models, including following the search steps directly in the OCL editor. The direct reuse of this debugging approach is not optimal for graph patterns, where, as opposed to OCL, the order of execution does not follow the order of definitions, making it hard to understand which elements were hidden.

In the database community, several development environments were proposed for SQL queries [6,10], providing query editing and evaluation support. Furthermore, to give insight to the performance of queries, visualizations are available of the execution plans of the queries, such as Graphical Explain Plans in case of Oracle Enterprise Manager.

The features of EMF-IncQuery introduced in the paper are novel in the sense that query definitions can be evaluated using either incremental or local search based techniques, and the corresponding tools for debugging incremental and local search strategies nicely complement each other.

6 Conclusion and Future Work

In this paper, we described a novel feature of the EMF-IncQuery framework, the support of local search-based pattern matching in addition to the previously available incremental evaluation. By reusing the existing pattern language and query development environment, it is possible to select the most appropriate strategy without modifications to already developed patterns. Furthermore, we presented a prototype graphical debugger that helps understanding complex patterns by visualizing the execution of the search process. Both contributions are included in the EMF-IncQuery project.

In the future, we plan to improve the local search support by providing a model-sensitive planner for local search [5], that is expected to enhance the performance. Another promising idea is the support of hybrid pattern matching [11]: by mixing incrementally evaluated and local search-based pattern matching, it is possible to fine-tune the performance characteristics (memory footprint or execution time), extending the range of problems that can be addressed.

References

1. Ujhelyi, Z., Szõke, G., Ákos Horvth, Csiszár, N.I., Vidács, L., Varró, D., Ferenc, R.: Performance comparison of query-based techniques for anti-pattern detection. Information and Software Technology (0) (2015) - Accepted
2. Ujhelyi, Z., Bergmann, G., Hegeds, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-Incquery: an integrated development environment for live model queries. Sci. Comput. Program. **98**(1), 80–99 (2015)
3. Nickel, U., Niere, J., Zündorf, A.: Tool demonstration: The FUJABA environment. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), pp. 742–745. ACM Press, Limerick, Ireland (2000)
4. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: GrGen: a fast spo-based graph rewriting tool. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 383–397. Springer, Heidelberg (2006)
5. Varró, G., Deckwerth, F., Wieber, M., Schürr, A.: An algorithm for generating model-sensitive search plans for emf models. In: Hu, Z., de Lara, J. (eds.) ICMT 2012. LNCS, vol. 7307, pp. 224–239. Springer, Heidelberg (2012)
6. Oracle: Enterprise Manager (2015). http://www.oracle.com/technetwork/oem/enterprise-manager/overview/index.html
7. Seifert, M., Katscher, S.: Debugging triple graph grammar-based model transformations. In: Fujaba Days, pp. 19–25 (2008)
8. Horn, T.: Model querying with funnyQT. In: Duddy, K., Kappel, G. (eds.) ICMB 2013. LNCS, vol. 7909, pp. 56–57. Springer, Heidelberg (2013)
9. Eclipse OCL Project: MDT-OCL website (2015). https://projects.eclipse.org/projects/modeling.mdt.ocl
10. IBM Software: InfoSphere Data Architect (2015). http://www-01.ibm.com/software/data/optim/data-architect/
11. Horváth, Á., Bergmann, G., Ráth, I., Varró, D.: Experimental assessment of combining pattern matching strategies with VIATRA2. Int. J. Softw. Tools Technol. Transfer **12**(3–4), 211–230 (2010)