Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Performance comparison of query-based techniques for anti-pattern detection ☆

Zoltán Ujhelyi [a,*], Gábor Szőke [b,c], Ákos Horváth [a], Norbert István Csiszár [b], László Vidács [d,*], Dániel Varró [a], Rudolf Ferenc [b]

[a] Department of Measurement and Information Systems, Budapest University of Technology and Economics, H-1117 Magyar tudósok krt. 2., Budapest, Hungary
[b] Department of Software Engineering, University of Szeged, H-6720 Dugonics tér 13., Szeged, Hungary
[c] Refactoring 2011 Kft., H-6722 Gutenberg u. 14., Szeged, Hungary
[d] MTA-SZTE Research Group on Artificial Intelligence, University of Szeged, H-6720 Tisza Lajos krt. 103., Szeged, Hungary

ARTICLE INFO

Article history:
Received 23 June 2014
Received in revised form 5 December 2014
Accepted 5 January 2015
Available online 22 January 2015

Keywords:
Anti-patterns
Refactoring
Performance measurements
Columbus
EMF-IncQuery
OCL

ABSTRACT

Context: Program queries play an important role in several software evolution tasks like program comprehension, impact analysis, or the automated identification of anti-patterns for complex refactoring operations. A central artifact of these tasks is the reverse engineered program model built up from the source code (usually an Abstract Semantic Graph, ASG), which is traditionally post-processed by dedicated, hand-coded queries.
Objective: Our paper investigates the costs and benefits of using the popular industrial Eclipse Modeling Framework (EMF) as an underlying representation of program models processed by four different general-purpose model query techniques based on native Java code, OCL evaluation and (incremental) graph pattern matching.
Method: We provide in-depth comparison of these techniques on the source code of 28 Java projects using anti-pattern queries taken from refactoring operations in different usage profiles.
Results: Our results show that general purpose model queries can outperform hand-coded queries by 2–3 orders of magnitude, with the trade-off of an increased in memory consumption and model load time of up to an order of magnitude.
Conclusion: The measurement results of usage profiles can be used as guidelines for selecting the appropriate query technologies in concrete scenarios.

1. Introduction

Program queries play a central role in various software maintenance and evolution tasks. Refactoring, an example of such tasks, aims at changing the source code of a program without altering its behavior in order to increase its readability, maintainability, or to detect and eliminate coding anti-patterns. After identifying the location of the problem in the source code the refactoring process applies predefined operations to fix the issue. In practice, the identification step is frequently defined by program queries, while the manipulation step is captured by program transformations.

Advanced refactoring and reverse engineering tools (like the Columbus framework [1]) first build up an Abstract Semantic Graph (ASG) as a model from the source code of the program, which enhances a traditional Abstract Syntax Tree with semantic edges for method calls, inheritance, type resolution, etc. In order to handle large programs, the ASG is typically stored in a highly optimized in-memory representation. Moreover, program queries are captured as hand-coded programs traversing the ASG driven by a visitor pattern, which can be a significant development and maintenance effort.

Models used in model-driven engineering (MDE) are uniformly stored and manipulated in accordance with a metamodeling framework, such as the Eclipse Modeling Framework (EMF), which offers advanced tooling features. Essentially, EMF automatically generates a Java API, model manipulation code, notifications for model changes, persistence layer in XMI, and simple editors and

☆ A previous version of this paper has been presented as the best paper at the IEEE CSMR-WCRE 2014 Software Evolution Week, Antwerp, Belgium, February 3–6, 2014.
* Corresponding authors. Tel.: +36 1 463 3579 (Z. Ujhelyi). Tel.: +36 62 544 143 (L. Vidács).
E-mail addresses: ujhelyiz@mit.bme.hu (Z. Ujhelyi), kancsuki@inf.u-szeged.hu (G. Szőke), ahorvath@mit.bme.hu (Á. Horváth), csiszar.norbert.istvan@stud.u-szeged.hu (N.I. Csiszár), lac@inf.u-szeged.hu (L. Vidács), varro@mit.bme.hu (D. Varró), ferenc@inf.u-szeged.hu (R. Ferenc).

http://dx.doi.org/10.1016/j.infsof.2015.01.003
0950-5849/© 2015 Elsevier B.V. All rights reserved.

viewers (and many more) from a domain metamodel, which significantly speeds up the development of EMF-compliant domain-specific tools.

EMF models are frequently post-processed by advanced model query techniques based on graph pattern matching exploiting different strategies such as local search [2] or incremental evaluation [3]. Some of these approaches have demonstrated to scale up for large models with millions of elements in forward engineering scenarios, but up to now, no systematic investigation has been carried out to show if they are efficiently applicable as a program query technology. If this is the case, then advanced tooling offered by the EMF could be directly used by refactoring and program comprehension tools without compromise.

The paper contributes a detailed comparison of (1) memory usage in different ASG representations (dedicated vs. EMF) and (2) run time performance of different program query techniques. For the latter, we evaluate five essentially different solutions: (i) hand-coded visitor queries implemented in native Java code (as used in Columbus), (ii) the same queries over EMF models, (iii) the standard OCL language, and generic model queries following (iv) a local search strategy and (v) incremental model queries, both using caching techniques from the EMF-IncQuery.

We compare the performance characteristics of these query technologies by using the source code of 28 open-source Java projects (with a detailed comparison of the largest 14 projects in the paper) using queries for 8 anti-patterns. Considering typical usage scenarios, we evaluate different usage profiles for queries (one-time vs. on-commit vs. on-save query evaluation). As a consequence, execution time in our measurements includes the one-time penalty of loading the model itself, and various number of query executions depending on the actual scenario.

This article is based on a conference paper [4] with extensions along four directions: two new types of anti-pattern queries were implemented, which are different from previous ones in their complexity and nature; OCL queries were included in the study as a fifth approach; the size of subject programs were increased from 1.9 M to 10 M lines of code, including three large programs (over 1 M lines of code each) to experiment with the limitations of the approaches; and the evaluation was extended, among others, with model and query metrics and with a lessons learned section.

Our main finding is that advanced generic model queries over EMF models can execute several orders of magnitude faster than dedicated, hand coded techniques. However, this performance gain is balanced by an up to 10–15-fold increase in memory usage (in case of full incremental query evaluation) and an up to 3–4-fold increase in model load time for EMF based tools and queries, compared to native Columbus results. Therefore, the best strategy can be planned in advance, depending on how many times the queries are planned to be evaluated after loading the model from scratch.

The rest of the paper is structured as follows. Section 2 introduces the queries to be investigated in the paper. Section 3 provides a technological overview including how to represent models of Java programs, while Section 4 describes how to capture queries as visitors, graph patterns and OCL queries. Section 5 presents the measurement environment including the measured applications and the measurement process. Our experimental results and their analysis are detailed in Sections 6 and 7. Section 8 discusses related work to ours, while Section 9 concludes the paper.

2. Motivation

The results presented in this paper are motivated by an ongoing three-year refactoring research project involving five industrial partners, which aims to find an efficient solution for the problem of software erosion. The starting point of the refactoring process is the detection of coding anti-patterns to provide developers with problematic points in the source code. Developers then decide how to handle the revealed issues. During the project, the first phase was a manual refactoring phase [5], where developers investigated the list of reported anti-patterns and manually solved the problems. Based on these experiences, the real needs of partners were evaluated, and a refactoring framework was implemented with support for anti-pattern detection and guided automated refactoring with IDE integration.

In this paper we focus on the detection of coding anti-patterns, the starting point of the refactoring process. At this step one has to find patterns of problems, like when two Java strings are compared using the `==` operator instead of the `equals()` method. After identifying an occurrence of such an anti-pattern, the problematic code is replaced with a new condition containing a call to the `equals()` method with an appropriate argument.

In the refactoring project, the original plan was to use the Columbus ASG as the program representation together with its API to implement queries, since the API provides a program modification functionality to implement refactorings as well. However, queries for finding anti-patterns and the actual modifications can be separated. The presented research builds on this separation to investigate the performance of various query solutions. Our aim was to involve generic, model based solutions in the comparison. Generic solutions offer flexibility and additional features like change notification support in the EMF and reusable tools and algorithms, such as supporting for high-level declarative query definitions [6,7]. Such features could reduce the effort needed to define refactorings as well.

In this paper, we investigate two viable options for developing queries for refactorings: (1) execute queries and transformations by developing Java code working directly on the ASG; and (2) create the EMF representation of the ASG and use EMF models with generic model based tools. Years ago, we experienced that typical modeling tools were able to handle only mid-size program graphs [8]. We now revisit this question and evaluate whether model-based generic solutions have evolved to compete with hand-coded Java based solutions. We seek for answers to questions like: *What are the main factors that affect the performance of anti-pattern detection (like the representation of program models, their handling and traversing)? What size of programs can be handled (with respect to memory and runtime) with various solutions? Does incremental query execution result in better performance?*

We note that while we present our study on program queries in a refactoring context, our results can be used more generally. For instance, program queries are applied in several scenarios in maintenance and evolution from design pattern detection to impact analysis; furthermore, we think that real-life case studies are first-class drivers of improvements of model driven tools and approaches.

In the first round of experiments we selected six types of anti-patterns based on the feedback of project partners and formalized them as model queries. The diversity of the problems was among the most important selection criteria, resulting in queries that varied both in complexity and programming language context ranging from simple traverse-and-check queries to complex navigation queries potentially with negative conditions. Here, we briefly and informally describe the selected refactoring problems and the related queries used in our case study.

Switch without default. Missing `default` case has to be added to the `switch`. *Related query:* We traverse the whole graph to find Switch nodes without a default case.

Catch problem. In a catch block there is an `instanceof` check for the type of the catch parameter. Instead of the `instanceof`