Information and Software Technology xxx (2015) xxx–xxx

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 xxxx

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/

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`

Please cite this article in press as: Z. Ujhelyi et al., Performance comparison of query-based techniques for anti-pattern detection, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.01.003

check a new catch block has to be added for the checked type and the body of the conditional has to be moved there. *Related query:* We search for identifiers on the left hand side of the `instanceOf` operator and check whether it points to the parameter of the containing catch block.

Concatenation to empty string. When a new `String` is created starting with a number, usually an empty `String` is added from the left to the number to force the `int` to `String` conversion, because there is no `int + String` operator in Java. A much better solution is to convert the number using the `String.valueOf()` method first. *Related query:* We search for empty string literals, and check the type of the containing expression. If the container expression is an infix expression, then we also make sure that the string is located at the left hand side of the expression and the kind of the infix operator is the String concatenation ("+").

String literal as compare parameter. When a `String` variable is compared to a `String` literal using the `equals()` method, it is unsafe to have the variable on the left hand side. Changing the order makes the code safe (by avoiding null pointer exception) even if the String variable to compare is `null`. *Related query:* We search for all method invocations with the name "equals". After that, we check that their only parameter is a string literal.

String compare without equals method. This refactoring is already mentioned above. *Related query:* We search for the `==` operator and check whether the left hand side operand is of type `java.lang.String`. We have to check for the right hand side operand as well: in case of `null` we cannot use the method call. In fact, it is not necessary because in this case the comparison operator is the right choice.

Unused parameter. When unused parameters remain in the parameter list they can be removed from the source code in most cases. *Related query:* We search for the places in the method body where parameters are used. However, there are specific cases when removing a parameter that is not used in the method body results in errors, such as (1) when the method has no body (interface or abstract method); (2) when the method is overridden by or overrides other methods; and (3) `public static void main` methods.

After the first round of our experiments described in [4], it turned out that all antipatterns can be evaluated by our selection of tools effectively. In order to find the limits of the approaches, we selected two additional, more complex antipatterns requiring additional capabilities.

Avoid rethrowing exception. The catch block is unnecessary if the exception handling code only re-throws the caught exception without further actions. We seek for a thrown exception in the catch block and check whether the thrown exception is the same (or descendant) as the caught one. However, simply rethrowing the exception is valid, if a specific exception is to be handled externally, while a more generic exception handler block is responsible for managing a superclass of the caught exception. This antipattern requires transitive closure calculation for the inheritance hierarchy as a new feature.

Cyclomatic complexity. Cyclomatic complexity measures the number of linearly independent paths through a program's source code, usually calculated for a function as the number of decision points +1. A highly complex code (e.g. by means of cyclomatic complexity) tends to be difficult to test and maintain and tend to have more defects. The pattern requires counting various types of program elements within a method body. This calculation relies on counting model elements together with simple arithmetic operations and extensive traversal around the containment hierarchy. To have the same validation format, we list the methods with cyclomatic complexity higher than 10.

3. Technological overview

In this section, we first give a brief overview on how to represent Java programs as an ASG or an EMF model, then present the graph pattern formalism and use it to capture various antipatterns.

3.1. Managing models of Java programs

3.1.1. Abstract semantic graph for Java

The Java analyzer of the Columbus reverse engineering framework is used to obtain program models from the source code (similarly as for the C++ language [1,9]). The ASG contains all information that is in a usual AST extended with semantic edges (e.g., call edges, type resolution, overrides). It is designed primarily for reverse engineering purposes [10,11] and it conforms to our Java metamodel.

In order to keep the models of large programs in memory, the ASG implementation is heavily optimized for low memory consumption, e.g., handling all model elements and String values centrally avoids storing duplicate values. However, these optimizations are hidden behind an API interface.

In order to support processing the model, e.g., executing a program query, the ASG API supports visitor-based traversal [12]. These visitors can be used to process each element on-the-fly during traversal, without manually coding the (usually preorder) traversal algorithm.

Example 1. To illustrate the use of the ASG, we present a short Java code snippet and its model representation in Fig. 1. The code consists of a public method called `equals` with a single parameter, together with a call of this method using a Java variable `srcVar`. The corresponding ASG representation is depicted in Fig. 1b, omitting type information and boolean attribute values such as the final flags for readability.

The method is represented by a `NormalMethod` node that has the name `equals` and `public` accessibility attribute. The method parameter is represented by a `Parameter` node with the name attribute `other`, and is connected to the method using a `parameter` reference.

The call of this method is depicted by a `MethodInvocation` node that is connected to the method node by an `invokes` reference. The variable the method is executed on is represented by an `Identifier` node via an `operand` reference. Finally, an `argument` reference connects a `StringLiteral` node describing the `"source"` value.

3.1.2. Java Application models in EMF

3.1.2.1. Metamodeling in the EMF. Metamodeling is a fundamental part of modeling language design as it allows the structural definition (e.g., abstract syntax) of modeling languages.

The EMF provides a Java-based representation of models with various features, e.g., notification, persistence, or generic, reflective model handling. These common persistence and reflective model handling capabilities enable the development of generic (search) algorithms that can be executed on any given EMF-based instance model, regardless of its metamodel.

The model handling code is generated from a metamodel defined in the *Ecore* metamodeling language together with higher level features such as editors. The generator work-flow is highly customizable, e.g., allowing the definition of additional methods.

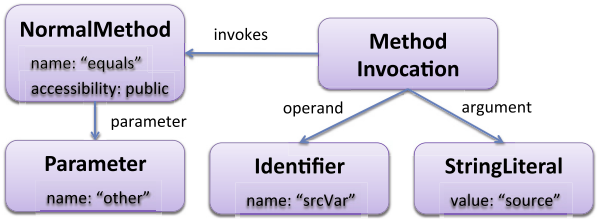
The main elements of the Ecore metamodeling language are the following: `EClass` elements define the types of objects; `EAttrib-`

Please cite this article in press as: Z. Ujhelyi et al., Performance comparison of query-based techniques for anti-pattern detection, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.01.003

```
public boolean equals(Object other) {...}

...
// Code inside another method
// The variable 'srcVar' is defined locally
srcVar.equals("source");
...
```

(a) Java Code Snippet



(b) ASG Representation

Fig. 1. ASG representation of Java code.

ute extend EClasses with attribute values while EReference objects present directed relations between EClasses.

Example 2. As an illustration, we present a small subset of the Java ASG metamodel realized in the Ecore language in Fig. 2 that focuses on method invocations as depicted in Fig. 1. The metamodel was designed to provide an equivalent representation of the ASG of the Columbus framework in the EMF, both on the model level and the generated Java API. The entire metamodel consists of 142 EClasses with 46 EAttributes and 102 EReferences.

The NormalMethod and Parameter EClasses are both elements of the metamodel that can be referenced from Java code by name. This is represented by generalization relations (either direct or indirect) between them and the NamedDeclaration EClass. This way, both inherit all the EAttributes of the NamedDeclaration, such as the name or the accessibility controlling the visibility of the declaration.

Similarly, the EClasses MethodInvocation, Identifier and StringLiteral are part of the Expression elements of Java. Instead of attribute definitions, the MethodInvocation is connected to other EClasses using three EReferences: (1) the EReference invokes points to the referred MethodDeclaration; (2) the argument selects a list of expressions to be used as the arguments of the called methods, and (3) the inherited operand EReference selects an expression representing the object the method is called on.

3.1.2.2. Notes on Columbus compatibility. The Java implementation of the Java ASG of the Columbus Framework and the generated code from the EMF metamodel use similar interfaces. This makes possible to create a combined implementation that supports the advanced features of the EMF, such as the change notification support or reflective model access, while remains compatible with the existing analysis algorithms of the Columbus Framework by generating an EMF implementation from the Java interface specification.

However, there are also some differences between the two interfaces that should be dealt with. The most important difference lies in multi-valued reference semantics, where the EMF disallows having two model elements connected multiple times using the same reference type, while the Columbus ASG occasionally relies on such features. To maintain compatibility, the EMF implementation is extended with proxy objects, which ensure the uniqueness of references. The implementation hides the presence of these proxies from the ASG interface while the EMF-based tools can navigate through them.

Other minor changes range from different method naming conventions for boolean attributes to defining additional methods to traverse multi-valued references. All of them are handled by generating the standard EMF implementation together with the Columbus compatibility methods.

3.2. Definition of model queries using graph patterns

Graph patterns [6] are a declarative, graph-like formalism representing a condition (or constraint) to be matched against instance model graphs. This formalism is usable for various purposes in model-driven development, such as defining model transformation rules or defining general purpose model queries including model validation constraints. In this paper, we give only a brief overview of the concepts, for more detailed, formal definitions see [13].

A graph pattern consists of *structural constraints* prescribing the interconnection between the nodes and edges of a given type and *expressions* to define *attribute constraints*. These constraints can be illustrated as a graph where the nodes are classes from the metamodel, while the edges prescribe the required connections of the 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 is a tuple of pattern parameters that fulfills all the following conditions: (1) has the same structure as the pattern; (2) satisfies all structural and attribute constraints; and (3) does not satisfy any NAC.

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 used as an additional set of

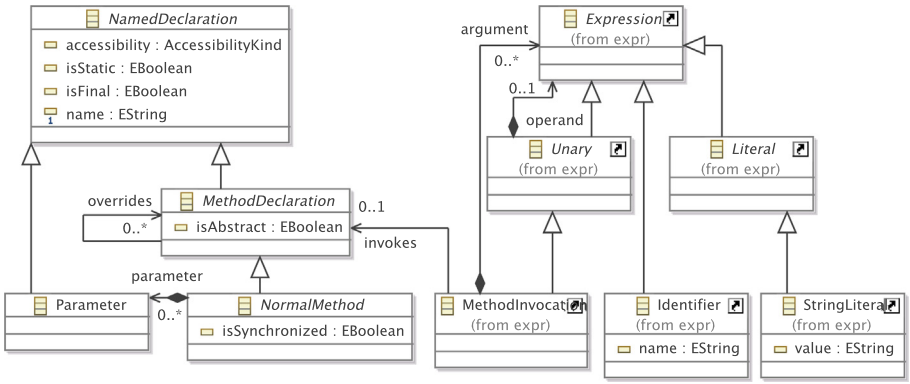


Fig. 2. A subset of the Ecore model of the Java ASG.

constraints to meet, while *negative application conditions* (NAC) describes the cases when the original pattern is *not* valid. Finally, *match set counting* constraints are used to calculate the number of matches a called pattern has, and use them as a variable in attribute constraints. Pattern composition constraints can be illustrated as a subgraph of the graph pattern.

When evaluating the results of a graph pattern, any subset of the parameters can be bound to model elements or attribute values that the pattern matcher will handle as additional constraints. This allows re-using the same pattern in different scenarios, such as checking whether a set of model elements fulfill a pattern, or list all matches of the model.

Example 3. Fig. 3 captures all the search problems from Section 2 as graph patterns. Here, we only discuss the String Literal as Compare Parameter problem (Fig. 3d) in detail, all other patterns can be interpreted similarly.

The pattern consists of five nodes named `inv`, `m`, `op` and `arg`, representing the model elements of the types `MethodInvocation`, `NormalMethod`, `Literal`, `Expression` and `StringLiteral`, respectively. The distinguishing (blue) formatting for the node `inv` describes that it is the parameter of the pattern.

In addition to the type constraints, node `m` shall also fulfill an attribute constraint ("equals") on its name attribute. The edges between the nodes `inv` and `m` (and similarly `arg`) represent a typed reference between the corresponding model elements. However, as the node `op` is included in a NAC block (depicted by the dotted red box), the edge `operand` means that either no operand should be given or the operand must not point to a `Literal` typed node.

Finally, to ensure that the invoked method has only a single parameter, the number of arguments are counted. The highlighted part of the pattern formulates a subpattern consisting of the arguments of the `MethodInvocation`, and the number of these subpattern matches is checked to be 1. This kind of checking could also be expressed using a NAC block describing a different parameter, but the use of match counting is easier to read.

After matching this pattern to the model from Fig. 1, the result will be a set containing a single element: the `MethodInvocation` instance.

4. Program queries approaches

In this section we give a brief overview of the possible approaches for implementing anti-pattern detection as program queries. At first, a visitor-based search approach is described, followed by two different graph-pattern based approaches (both supported by the EMF-IncQuery), and finally we use the OCL language to describe the query problems.

4.1. Manual search code

The ASG representation allows traversing the Java program models using the visitor [12] design pattern that can form the basis of the search operations.

Visitor-based searches are easy to implement and maintain if the traversed relations are based on containment references, and require no custom setup before execution. On the other hand, as the order of the traversal is determined outside the visitor, non-containment references are required to be traversed manually, typically with nested loops. Alternatively, traversed model elements and references can be indexed, and in a post-processing step these indexes can be evaluated for efficient query execution. In both cases, significant programming effort is needed for achieving efficient execution.

Example 4. The results of the String Literal as Compare Parameter (Fig. 3d) pattern can be calculated by collecting all MethodInvocation instances from the model, and then executing three local checks whether the invoked method is named `equals`, if it has an argument with a type of `StringLiteral`, and if it is not invoked on a `Literal` operand.

Fig. 4 presents (a simplified) Java implementation of the visitor. A single `visit` method is used as a start for traversing all `MethodInvocation` instances from the model, and checking the attributes and references of the invocation. It is possible to delegate the checks to different `visit` methods, but in that case the visitor has to track and combine the status of the distributed checks to prepare the results that is difficult to implement in a sound and efficient way.

The ASG does not initially contain reverse edges in the model. It provides an API to generate these extra edges in a second pass after loading the model, but this requires extra time and memory. As the subject queries in this study could be implemented without these extra resources, to keep the memory footprint low, we prefer not generating them.

4.2. Graph pattern matching with local search algorithms

Local search based pattern matching (LS) are commonly used in graph transformation tools [14–16] starting the match process from a single node and extending it step-by-step with the neighboring nodes and edges following a *search plan*. From a single pattern specification multiple search plans can be calculated [2], thus the pattern matching process starts with a plan selection based on the input parameter binding and model-specific metrics.

A search plan consists of a totally ordered list of *extend* and *check* operations. An *extend* operation binds a new element in the calculated match (e.g., by matching the target node along an edge), while *check* operations are used to validate the constraints between the already bounded pattern elements (e.g., attribute constraints or whether an edge runs between two matched nodes). If an operation fails, the algorithm backtracks; if all operations are executed successfully, a match is found.

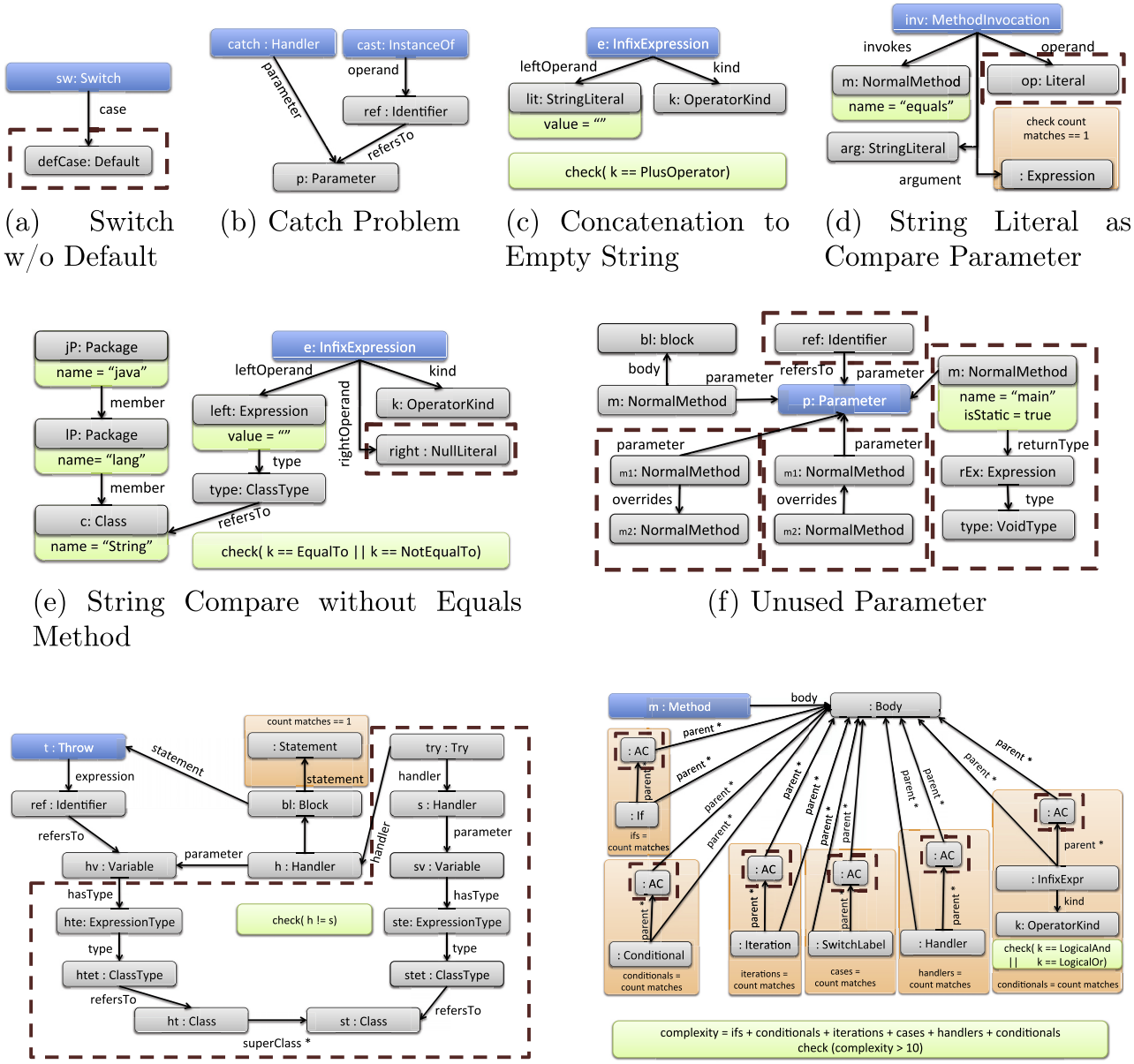
Some extend operations, such as finding the possible source nodes of an edge or iterating over all elements of a certain type might be very expensive to execute during a search, but this cost can be reduced by the use of an incremental model indexer, such as the EMF-IncQuery Base.[1] Such an indexer can be set up while loading the model, and then updating it on model changes using the notification mechanism of the EMF. If no such indexing mechanism is available (e.g., because of its memory overhead), the search planner algorithm should consider these operations with higher costs, and thus provide alternative plans.

Example 5. To find all String Literals appearing as parameters of `equals` methods, a 7-step search plan presented in Table 1 was used. First, all `NormalMethod` instances are iterated over to check for their name. Then a backward navigation operation is executed to find all corresponding method invocations to check its argument and operand references. At the last step, a NAC check is executed by starting a new plan execution for the negative subplan, but only looking for a single solution.

Fig. 5 illustrates the execution of the search plan on the simple instance model introduced previously. In the first step, 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

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

Please cite this article in press as: Z. Ujhelyi et al., Performance comparison of query-based techniques for anti-pattern detection, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.01.003



(a) Switch w/o Default (b) Catch Problem (c) Concatenation to Empty String (d) String Literal as Compare Parameter

(e) String Compare without Equals Method (f) Unused Parameter

(g) Avoid Rethrow (h) Cyclomatic Complexity

Fig. 3. Graph pattern representation of the search queries.

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.

It is important to note that the search begins with listing all `NormalMethod` elements as opposed to the visitor-based implementation, which starts with the `MethodInvocations`. This was motivated by the observation that in a typical Java program there are more method invocations than method definitions, thus starting this way would likely result in less traversed search states, while still finding the same results in the end. However, this optimization relies on having an index which allows cheap backward navigation during pattern matching for step 3 (on the contrary to the ASG based solution where this information is not available without extra traversal).

4.3. Incremental graph pattern matching using the Rete algorithm

Incremental pattern matching [3,17] is an alternative pattern matching approach that explicitly caches matches. This makes the results available at any time without further searching, however, the cache needs to be incrementally updated whenever changes are made to the model.

The Rete algorithm [18], which is well-known in rule-based systems, was efficiently adapted to several incremental pattern matchers [19–21]. The algorithm uses an extended incremental caching approach that not only indexes the basic model elements but also indexes *partial matches* of a graph pattern that enumerates the model element tuples that satisfy a subset of the graph pattern constraints.

Please cite this article in press as: Z. Ujhelyi et al., Performance comparison of query-based techniques for anti-pattern detection, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.01.003

```
public class CompareParameterVisitor extends Visitor {

    //A set to store results
    private Set<MethodInvocation> invocations
        = new HashSet<MethodInvocation>();

    @Override
    public void visit(MethodInvocation node) {
        super.visit(node);
        //Checking invoked method name and number of parameters
        if ("equals".equals(node.getInvokes().getName())
                        && node.getArgument().size() == 1) {
            //Node argument
            Expression argument = node.getArgument(0);
            //Node operand
            Expression operand = node.getOperand();
            //Type checking for argument
            if (argument instanceof StringLiteral
                    //NAC checking for operand
                    && !(operand instanceof Literal)) {
                //Result found
                invocations.add(node);
            }
        }
    }
}
```

Fig. 4. Visitor for the string literal as compare parameter problem.

Table 1
Search plan for the string literal compare pattern.

Operation	Type	Notes
1: Find all `m` that `m` ⊂ `NormalMethod`	Extend	Iterate
2: Attribute test: `m.name==`*"equals"*	Check	
3: Find`inv` that `inv.invokes` →`m`	Extend	Backward
4: Count of `inv.argument` →`arg` is 1	Check	Called plan
5: Find `arg` that `inv.argument` →`arg`	Extend	Forward
6: Instance test: `arg` ⊂ `StringLiteral`	Check	
7: Find `op` that `inv.operand` →`op`	Extend	Forward
8: NAC analysis: `op` ⊄ `Literal`	Check	Called plan

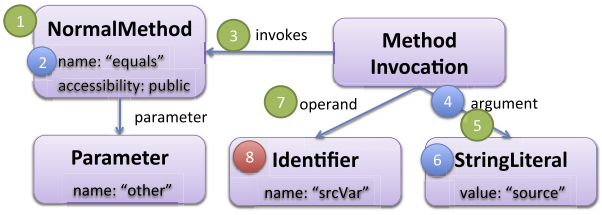


Fig. 5. Executing the search plan.

These caches are organized in the graph structure called Rete network that can be incrementally updated at model changes.

The *input nodes* of Rete networks represent the index of the underlying model elements. The *intermediate nodes* execute basic operations, such as filtering, projection, or join, on other Rete nodes (either input or intermediate) they are connected to, and store the results. Finally, the match set of the entire pattern is available as an *output (or production) node*.

When the network is initialized, the initial match set is calculated and the input nodes are set up to react on the model changes. When receiving a *change notification*, an *update token* is released on each of their outgoing edges. Upon receiving such a token, a Rete node determines how (or whether) the set of stored tuples will change, and releases update tokens on its outgoing edges. This way, the effects of an update will propagate through the network, eventually influencing the result set stored in the production nodes.

Example 6. To illustrate a Rete-based incremental pattern matching, we first depict the Rete network of the String Literal as Compare Parameter pattern in Fig. 6.

The network consists of five input nodes that store the instances of the types `NormalMethod`, `MethodInvocation`, `StringLiteral`, `Expression` and `Literal`, respectively. The input nodes are coupled by join nodes that calculate the list of elements connected by `invokes`, `argument` and `operand` references, respectively. As both ends are already enumerated in the parent nodes, both forward and backward references can be calculated efficiently. The invoked method list (output of the `invokes` join node) is filtered by the `name` attribute of Methods, while the argument lists are `filtered` for one per call. The NAC checking is executed by removing the elements with `Literal` types from the result of the `operand` join. Finally, all partial matches are joined together to form the resulting matches.

It is important to note that the Rete node, such as the `MethodInvocation` in the example, can be used in multiple join operations; in such cases the final join is responsible for filtering out the unwanted duplicates (for a selected variable).

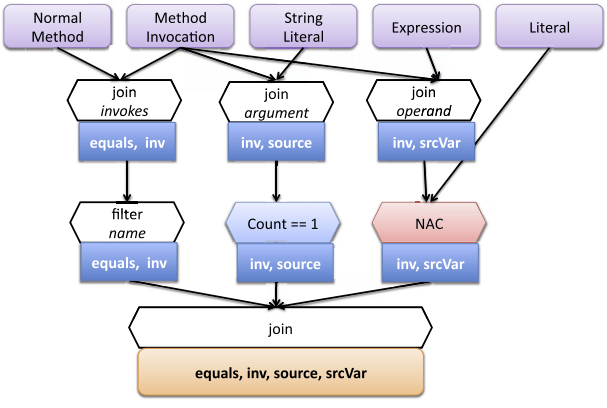


Fig. 6. Rete network for the string literal compare pattern.

Please cite this article in press as: Z. Ujhelyi et al., Performance comparison of query-based techniques for anti-pattern detection, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.01.003

```
context MethodInvocation:
def: stringLiteralAsCompareParameter : Boolean =
self.invokes.name = 'equals'
   and self.arguments -> exists(oclIsKindOf(StringLiteral))
   and self.arguments -> size() = 1
   and not self.operand.oclIsKindOf(Literal)
```

Fig. 7. The OCL expression of the string literal as compare parameter problem.

4.4. Model queries with OCL

OCL [7] is a standardized, pure functional model validation and query language for defining expressions in the context of a metamodel. The language itself is very expressive, exceeding the expressive power of first order logic by offering constructs such as collection aggregation operations (`sum()`, etc.). The rest of the section gives a basic overview of OCL expressions, for a more detailed description of the possible elements consult the specification [7].

Variables of an OCL expression refer to instance model elements and a set of basic types including strings, various number formats and different kinds of collections. For these types, built-in operations are defined such as comparison operators or membership testing.

Furthermore, OCL expressions are compositional, allowing the definition of sub-expressions in more complex expressions, including the `let` expression for defining additional variables, the `if` expression for implementing conditions or *iterator* expressions that evaluate subexpressions on all members of a collection.

Each OCL expression is valid in a *context*, described as a metamodel type. The OCL standard allows the definition of multiple context variables, however OCL implementations often support only a single one.

Example 7. To illustrate the capabilities of OCL, Fig. 7 formulates the String Literal as Compare Parameter problem as an OCL query. The query can be evaluated starting from a `MethodInvocation` context variable, that is referred to throughout the query as `self`.

The query is described as the conjunction of 4 different subexpressions:

1. It is checked whether the target of the invocation has a `name` attribute with the value of `'equals'`. The type of the invoked call is not checked, as based on the metamodel it is known to be correct.
2. It is checked whether the list of `arguments` contain an element that has the type of (StringLiteral). The `exists` operation is one of the iterator operations, that detects whether any member of the collection fulfills the condition.
3. It is checked whether the size of the arguments collection is exactly 1.
4. Finally, the operand type is checked *not* to be `Literal`.

OCL expressions can be evaluated as a search of the model, where the corresponding search plan is encoded in the expression itself. This makes the manual optimizations of the queries possible, however it needs a detailed understanding of both the instance and metamodels and the underlying OCL engine as well.

5. Measurement context

To provide a context for our performance evaluation, in this section we describe the executed measurements of this paper. This includes a detailed evaluation of all our instance models and queries using different complexity metrics and the description of our measurement process. The selection of metrics was motivated by earlier results of [22] where the values of different metrics are compared to the execution time of different queries.

The use of metrics helps to identify which queries/models are more difficult for the selected tools. Furthermore, it would allow to compare both the models and the queries to other available performance benchmarks.

5.1. Java projects

The approaches were evaluated on a test set of 28 open-source projects. The projects are sized between 1*kLOC* and 2*MLOC*, and used in various scenarios. The list of projects include the ArgoUML editor, the Apache CloudStack infrastructure manager tool, the Eclipse Platform, the Google Web Toolkit (GWT) library, the Tomcat Java application server, the SVNKit Subversion client, the online homework system WeBWorK, the Weka data mining software, and many more. Table 2 contains the full list of projects and their analyzed versions (projects where snapshots were used are marked in the table).

To compare these models, Table 2 shows different metrics about them, including their size in lines of code and in number of nodes, edges and attributes of the graph representation, the number of metamodel types used and the indegree and outdegree of the graph nodes. The graph structure of all models are similar: they use about 90–100 of the types specified in the metamodel, and the average indegree and outdegree is 3. The large numbers in the maximum indegree column are related to the representation of the Java type system: a few types, such as `String` or `int` are referred to many times throughout the code.

In the remainder of the section, only the results related to the programs larger than 100*kLOC* are presented, as they still represent a wide range of Java applications, and in the case of smaller models the differences between the tools are much smaller (but similar to those presented here).[2]

5.2. Query complexity

The antipatterns used different approaches in the various tools, resulting in different query complexity in each case. To compare them, Table 3 describes the complexity of queries implemented in the various tools. We have selected different complexity measures for the different formalisms to understand how query complexity changes with the different approaches.

In the case of visitors we are calculating the lines of Java code required together with its cyclomatic complexity. The six original queries were written in less than 100 lines of code and had a cyclomatic complexity of 10–20. The two new queries were more complex both in lines of code and cyclomatic complexity.

For graph patterns, we rely on metrics defined in [22]: the number of query *variables* and *parameters*, the number of *edge* and *attribute* constraints, the number of subpattern *calls* and the combined number of negative pattern calls and match counters *NEG*. It is important to note that the metrics were not calculated from the

[2] For a detailed test result containing all models and raw measurement data visit our website: http://incquery.net/publications/extended-program-query-comparison.

Please cite this article in press as: Z. Ujhelyi et al., Performance comparison of query-based techniques for anti-pattern detection, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.01.003

Table 2
Model metrics.

	Version	LOC	Node count	Edge count	Attribute count	Type count	Avg/max InDegree		Avg/max OutDegree	
ArgoUML	0.35.1 (*)	174,516	1,002,129	2,973,258	6,895,018	100	3	72,230	3	445
CloudStack	4.1.0	1,369,952	5,390,662	16,478,218	36,650,136	100	3.1	631,140	3.1	1198
Eclipse	3.0.0	2,294,146	8,403,914	26,254,507	58,219,100	97	3.1	1,245,390	3.1	1958
Frinika	0.5.1	64,828	429,407	1,292,961	3,065,383	99	3	54,286	3	844
GWT	2.3.0	1,078,630	3,219,239	9,986,705	22,364,819	101	3.1	392,098	3.1	1206
Hibernate	3.5.0	773,166	2,444,419	7,563,207	16,789,330	102	3.1	193,769	3.1	522
Jackrabbit	2.8	590,420	1,765,882	5,341,431	12,145,662	100	3	271,217	3	708
Java DjVu	0.8.06	23,570	129,068	372,444	926,653	92	2.9	26,918	2.9	1026
javax.usb	1.0.1	1161	12,231	32,388	89,399	83	2.6	969	2.6	148
JFreechart	1.2.0	327,865	865,148	2,663,967	6,022,410	93	3.1	50,658	3.1	445
JML	1.0b3	10,159	72,598	212,544	520,599	94	2.9	4908	2.9	221
JTransforms	2.4	38,400	295,009	945,643	2,053,900	80	3.2	117,775	3.2	217
Makumba	0.8.1.9	65,065	378,204	1,127,797	2,637,424	98	3	62,717	3	445
OpenEJB	4.5.2	575,363	1,785,660	5,428,385	12,377,185	101	3	152,624	3	540
Physhun	0.5.1	4935	36,962	108,888	263,091	86	2.9	2944	2.9	148
ProteinShader	0.9.0	22,651	137,416	391,322	997,679	88	2.8	9654	2.8	445
Qwicap guess	1.4b24	443	7903	21,222	59,069	85	2.7	918	2.7	107
Robocode	1.5.4	28,245	204,362	599,556	1,500,298	97	2.9	17,323	2.9	445
sdedit	3.0.5	14,717	145,453	413,998	1,075,471	97	2.8	12,643	2.8	445
Stendhal	0.75.1	105,411	667,142	2,037,645	4,688,300	98	3.1	49,556	3.1	445
Struts2	1.4.0	274,092	927,163	2,849,021	6,452,090	100	3.1	95,272	3.1	620
Superversion	2.0b8	29,282	238,842	705,875	1,731,692	94	3.0	2041	3.0	445
SVNKit	1.3.0.5847	114,189	698,753	2,203,436	4,843,209	93	3.2	57,987	3.2	272
Tomcat	8.0.0 (*)	459,579	1,338,601	4,084,668	9,302,681	102	3.1	116,637	3.1	620
WebWork	2.2.7	46,208	285,372	853,724	2,018,672	95	3	36,439	3	445
Weka	3.7.10 (*)	205,537	1,615,637	4,989,653	11,259,543	99	3.1	216,651	3.1	550
Xalan	2.7	349,681	708,445	2,093,338	4,937,831	93	3	87,447	3	445
Xins	2.2a2	21,698	164,989	472,003	1,193,822	89	2.9	15,169	2.9	445

Table 3
Query complexity metrics.

	Visitor		Query						OCL
	LOC	CC	Param.	Variables	Edges	Attr.	Calls	NEG	MC
Catch	78	14	4	6	3	0	1	0	9
Concatenate	32	8	6	8	3	1	3	0	4
Constant compare	39	10	6	11	5	0	2	2	7
No default switch	53	11	2	3	1	0	0	1	2
String compare	56	15	10	17	10	1	7	2	15
Unused parameter	88	21	11	19	8	0	6	1	21
Avoid rethrow	210	54	11	24	12	0	2	1	23
Cyclomatic complexity	114	22	23	40	5	2	9	7	34

graphical notation of Fig. 3, but their implementation in the EMF-IncQuery, where different subpatterns were created to facilitate reuse both in the design level and during runtime. A subpattern call introduces new variables for the parameters of the subpattern that are equal to some parameters at their call site; this might cause an increased number of variables compared to the number of edge and attribute constraints.

To measure the complexity of OCL queries, we used a minimum complexity (MC) metric presented in [23] that is based on either calculating or estimating the number of model elements visited during the execution of its search, where multiple visits of the same element accounts as different ones. However, the metric definition relies on the model structures; in order to have a model-independent metric, estimates need to be provided for the models.

In the current paper, we calculate a lower bound of this metric by underestimating the number of visited model elements with stating that each OCL expression or operation will be evaluated with at most one model element that relates to the number of conditions to evaluate. This way, it is possible to get a lower bound of the complexity for instance models that have at least one single result for the query.

The complexity of the queries over the different approaches behave similarly for almost all cases except for the following three: (i) the *no default switch* case uses the most simple pattern and OCL query, while in the case of visitors, (ii) the *concatenation* case uses the simplest visitor. (iii) Conversely, the calculation of *cyclomatic complexity* is clearly the most complex query in the graph patterns formalism and OCL, while its visitor is considerably simpler than the *avoid rethrow*. We believe that this difference is based on the fact that the calculation of cyclomatic complexity needs only the traversal of the containment hierarchy that visitors excel in.

5.3. Measurement process

All measurements were executed on a dedicated Linux-based server with 32 GB RAM running Java 7. On the server the Java ASG of the Columbus Framework was installed together with the EMF-IncQuery (supporting graph pattern matching using both the local search and the Rete-based incremental approaches) and the Eclipse OCL [24] tool.

All program queries were implemented as both visitors for the ASG (by a Columbus expert from the University of Szeged) and as graph patterns (by a model query expert from the Budapest University of Technology and Economics) – a different reviewer than the original implementer of the query. In the case of OCL expressions, we relied on our previous experience in comparing model query tools for [22], where OCL experts were asked to verify the developed queries. Visitors were executed on both model repre-

Please cite this article in press as: Z. Ujhelyi et al., Performance comparison of query-based techniques for anti-pattern detection, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.01.003

sentations, while the graph patterns (both for local search-based and incremental queries) and the OCL queries were evaluated on the EMF representation. In order to also be able to reason about use cases where multiple queries are executed together, indexes were built for all queries. In all cases, the time to load the model from its serialized form and the time to execute the program query were measured together with the maximum heap size usage.

The query implementations were manually verified to return the same values for all tools in three ways. At first, (1) the created specifications were reviewed to fulfill the original, textual specifications. Then, (2) in a selection of smaller programs all instances were manually compared to return exactly the same issues. Finally, (3) in case of all models, the number of found issues was reported and compared.

Every program query was executed ten times, and the standard deviation of the results was verified. After that, we averaged time and memory results without the smallest and the largest values. In order to minimize the interference between the different runs, for the execution of model, tool and query a new JVM was created and ran in isolation. Additionally, all measurements were executed with a 10 min timeout: if loading the model, initializing and executing the query took more than the timeout, the measurement was considered a failed one. The time to start up and shut down the JVM was not included in the measurement results.

6. Measurement results

To compare the performance characteristics of the different program query techniques, in this section we present the detailed performance measurement results.

6.1. Load time and memory usage

Table 4a presents the time required to load the models in seconds. As our measurements showed that the model load time is largely independent of the query selection, we only present an aggregated result table. The only exception to this rule is the *cyclomatic complexity* pattern with incremental pattern matching: in that case we found that indexing the transitive closure of the containment hierarchy was prohibitively expensive both in terms of load time and memory usage. For this reason, we executed two sets of measurements: (1) one without initializing the *cyclomatic complexity* pattern (INC), and (2) another that also includes this pattern (INC-CC).

Fig. 8 depicts the detailed load time and memory usage measurements for the Jackrabbit tool in box plots; the diagrams for the other cases were similar. In general, the diagrams show that the repeated measurements of the test cases show generally very little differences, except a few cases, while there are large differences when comparing the results of different techniques.

It can be seen that the load time is 3–4 times longer when using an EMF-based implementation over the manual Java ASG, and further increases can be seen when initializing the pattern matchers for local search and incremental queries. The two-phase load algorithm for the EMF model (EMF case), and the time to set up the indexes (local search) and partial matches (Rete) can account for these increases. As OCL does not use any specific index, no additional load overhead over the EMF visitor implementation is measured.

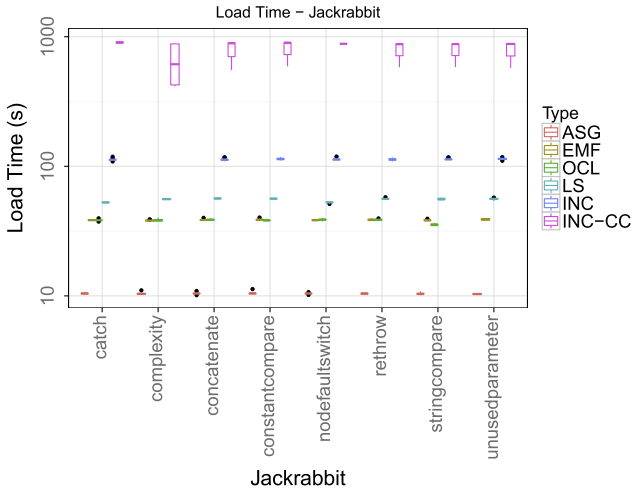
A similar increase can be seen for the memory usage in Table 4b: the EMF representation uses around twice as much memory, while the incremental engine may require an additional 10–15 times more memory to store its partial result caches compared to the ASG. When adding the *cyclomatic complexity* pattern as well, an additional increase in memory usage is observed, resulting in a memory exhaustion for the largest models (over 500kLOC, or 1.7 M graph nodes).

The smaller memory footprint of the Java ASG representation is the result of model-specific optimizations not applicable in generic EMF models. The additional increase for local search and Rete-based pattern matchers mainly represent the index and partial match set sizes, respectively. Similarly to load times, the use of

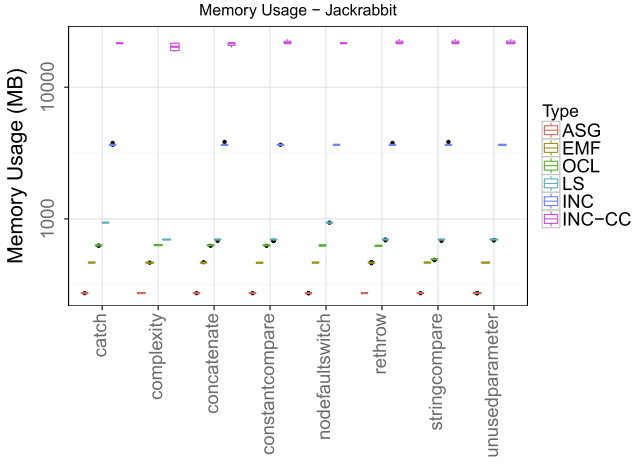
Table 4
Measurement results.

	ASG	EMF	OCL	LS	INC	INC-CC
(a) Load Time (in seconds)						
CloudStack	27.5 ± 0.6	115 ± 3.2	115 ± 1.8	156 ± 3.0	343 ± 5.9	NA
ArgoUML	6.7 ± 0.1	25 ± 0.6	25 ± 0.5	35 ± 0.6	52 ± 1.3	312 ± 53.3
Eclipse	41.7 ± 0.7	169 ± 2.3	171 ± 2.8	238 ± 3.2	470 ± 4.1	NA
GWT	16.1 ± 0.1	80 ± 2.1	80 ± 0.5	102 ± 2.7	199 ± 2.3	NA
Hibernate	13 ± 0.2	58 ± 1.7	57 ± 1.8	83 ± 1.9	146 ± 2	NA
Jackrabbit	10.4 ± 0.2	39 ± 0.5	38 ± 0.6	55 ± 0.7	113 ± 2.3	796 ± 152
JFreeChart	5.6 ± 0.2	21 ± 0.4	21 ± 0.4	30 ± 0.5	44 ± 1.2	277 ± 7.0
OpenEJB	10.6 ± 0.2	44 ± 0.8	43 ± 0.7	60 ± 0.8	117 ± 3.1	NA
Stendhal	4.4 ± 0.1	17 ± 0.5	17 ± 0.4	23 ± 0.4	36 ± 1.2	239 ± 11.7
SVNKit	4.4 ± 0.1	18 ± 0.3	18 ± 0.4	25 ± 0.5	39 ± 14	268 ± 7.7
Struts2	5.7 ± 0.1	23 ± 0.4	23 ± 0.4	32 ± 0.6	49 ± 1.1	292 ± 8.7
Tomcat	8.3 ± 0.2	33 ± 0.6	33 ± 0.6	43 ± 0.6	69 ± 1.7	484 ± 15.8
Weka	9.4 ± 0.2	38 ± 0.7	37 ± 0.3	52 ± 0.4	111 ± 2.4	526 ± 29.5
Xalan	4.8 ± 0.1	19 ± 0.3	19 ± 0.2	25 ± 0.3	38 ± 1.1	254 ± 9.4
(b) Memory usage (in MB)						
CloudStack	2189 ± 0.47	3503 ± 1.39	3925 ± 38	4017 ± 2.7	10,414 ± 58.88	NA
ArgoUML	198 ± 0.81	404 ± 0.9	461 ± 2.3	549 ± 9.1	5068 ± 42.09	11,974 ± 841
Eclipse	2453 ± 0.66	4054 ± 1.87	4641 ± 3.9	4745 ± 1848	17,754 ± 753.93	NA
GWT	2579 ± 0.12	1967 ± 2.49	2178 ± 2.9	3566 ± 1.3	5973 ± 32.93	NA
Hibernate	2086 ± 0.14	2524 ± 1.73	2788 ± 2.4	2995 ± 37.5	4507 ± 2.54	NA
Jackrabbit	309 ± 0.04	583 ± 4.62	651 ± 63	955 ± 9.8	3652 ± 59.45	22,123 ± 1593
JFreeChart	160 ± 0.06	360 ± 2.18	429 ± 67	530 ± 82.6	4400 ± 0.34	10,560 ± 273
OpenEJB	344 ± 0.26	656 ± 2.89	662 ± 82	946 ± 6.5	3889 ± 23	NA
Stendhal	109 ± 0.06	229 ± 0.51	431 ± 36	460 ± 124.2	3383 ± 68.85	7783 ± 629
SVNKit	129 ± 0.48	252 ± 3.12	401 ± 2.6	409 ± 2.8	3717 ± 4819	9835 ± 556
Struts2	159 ± 0.03	359 ± 2.71	479 ± 2.6	521 ± 2.9	4893 ± 70.27	11,636 ± 180
Tomcat	246 ± 0.04	547 ± 6.05	601 ± 7.6	788 ± 66.7	6637 ± 64.05	16,929 ± 2169
Weka	290 ± 0.07	616 ± 6.08	615 ± 151	695 ± 10.6	3427 ± 1	20,357 ± 1377
Xalan	146 ± 0.59	260 ± 2.85	441 ± 1.7	445 ± 9	3600 ± 0.52	8259 ± 535

Please cite this article in press as: Z. Ujhelyi et al., Performance comparison of query-based techniques for anti-pattern detection, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.01.003



(a) Load Time (in seconds)



(b) Memory Usage (in MB)

Fig. 8. Distribution of load time and memory usage of the Jackrabbit project.

OCL does not result in a change in memory usage compared to the EMF model.

The memory footprint increase of the *cyclomatic complexity* pattern is caused by the indexing of the transitive closure of the parent relation. As every model element has a parent and the containment hierarchy is usually deep, this transitive closure can alone become several times of the size of the entire model making it very expensive to index. On the other hand, the containment hierarchy can be effectively traversed using search operations, thus the other approaches can handle this query much better.

Generally, neither for load times nor memory usage were the standard deviation of the results significant compared to the other values, with the notable exceptions of the load time of the Jackrabbit tool with INC-CC, and the SVNKit applications memory usage with INC. The first one can be explained with garbage collection, as the memory usage was close to the 25 GB limit. For the latter one we have no clear explanation; however as we have witnessed no other fluctuations of this size, we believe that it was caused by a temporary issue during our measurements.

6.2. Search time

Table 5 presents the search time measurements (and uses *NA* if the measurement timed out). For each model and each program query the average search time is listed at first. Furthermore, in Fig. 9, we have highlighted the results of the Jackrabbit project in a box plot, where there are only minimal differences between the different executions of the same case, similar to load and search times.

Both visitor implementations perform similarly, producing similar execution times for queries, but increasing with model size as they traverse the entire model to find the results. The time differences between the ASG and EMF visitors are mainly the results of the memory optimizations of the original ASG implementation that avoided storing the same values multiple times, but required additional indirections during the model traversal. The reverse navigation option is not used in our measurements.

The local search and Rete based solutions provide a two or three orders of magnitude faster query execution, achieved by replacing the model traversal by calls to a pre-populated (and incrementally updated) index. Additionally, the search time of incremental queries is largely independent of model size, while in the case of local search it increases much slower than in the case of the visitor executions. As the search times for INC queries were exactly the same regardless whether the *cyclomatic complexity* query was loaded or not, their rows merged in the table.

The execution of OCL queries include a traversal of the model together with additional search operations, making its search slower than the visitor implementations. An exception from this to note is the unused parameter query: in that case the search operation timed out every time. This is most likely caused by the usage of the *allInstances* function that is used to find the source of an edge without reverse navigation options.

Additionally, as Table 5 shows, the execution time of visitor implementations increases linearly. This is in line with our expectation, as visitors have to traverse the entire model during the search. On the other hand, the search time for incremental queries are roughly the same for all queries, as the search simply means returning the results. In most of our patterns, the local search is an order of magnitude slower than incremental queries. However, the concatenation pattern (see Fig. 3c) executes as slow as the visitors in this regard. This is in line with our earlier experience [25] with different pattern matching strategies that the execution performance for local search techniques depends on the query complexity and the model structure.

To validate the results, for each program and tool combination we have the maximum standard deviation in percentage of their corresponding search time. In most cases, the standard deviation is low; only 9 rows contain deviations over 20%. As our measurements have shown time differences of orders of magnitude, these differences do not invalidate our conclusions during the analysis.

7. Evaluation of usage profiles

In addition to the raw evaluation of the measurement results, in this section, we discuss how the different approaches are compared in various usage profiles, and we summarize our findings. Furthermore, we discuss the different threats to validity, and the ways they were managed.

7.1. Usage profiles

In order to compare the approaches, we calculated the total time required to execute program queries for three different *usage profiles*: one-time, commit-time, and save-time analysis. The profiles were selected by estimating the daily number of commits and file changes for a small development team.

One-time analysis consists of loading the model and executing each program query in a batch mode. In case the analysis needs

Table 5
Query execution time (in seconds).

	Catch	Cyclomatic complexity	Concatenate	Constant compare	No default switch	Avoid rethrow	String compare	Unused parameter	Maximum deviation (%)
CloudStack									
ASG	5.3	7.6	6.0	5.5	5.3	5.9	5.4	6.0	16
EMF	3.9	5.0	3.7	3.7	4.1	4.0	3.6	4.4	15
OCL	6.2	90.7	6.8	9.0	6.6	7.1	7.4	NA	6
LS	0.13	81.50	0.55	0.28	0.02	0.26	1.09	0.76	24
INC	0.012	NA	0.010	0.024	0.012	0.013	0.013	0.020	18
ArgoUML									
ASG	1.8	2.4	1.7	1.9	1.7	1.8	1.6	1.9	5
EMF	1.3	1.5	1.3	1.3	1.2	1.3	1.1	1.3	9
OCL	2.4	14.3	2.0	2.9	1.6	2.1	2.2	NA	11
LS	0.05	6.94	0.16	0.09	0.01	0.06	0.17	0.26	13
INC	0.012	0.011	0.010	0.013	0.012	0.012	0.012	0.012	13
Eclipse									
ASG	8.0	11.3	8.3	9.2	7.8	7.7	7.0	9.3	11
EMF	5.6	7.4	5.5	5.6	5.9	5.7	5.3	6.1	10
OCL	10.3	122.2	10.0	12.4	9.4	9.9	12.1	NA	3
LS	0.20	99.82	0.85	0.25	0.08	0.21	1.03	1.45	8
INC	0.010	NA	0.009	0.013	0.014	0.010	0.011	0.022	11
GWT									
ASG	5.2	11.2	5.4	5.1	6.9	5.9	5.4	6.4	24
EMF	3.0	3.9	2.8	2.8	2.9	2.9	2.8	3.2	8
OCL	4.7	37.5	4.6	5.8	4.0	4.6	4.6	NA	9
LS	0.05	29.15	0.47	0.15	0.03	0.10	0.53	0.39	4
INC	0.010	NA	0.009	0.012	0.012	0.011	0.011	0.013	7
Hibernate									
ASG	4.2	5.3	4.6	3.9	4.5	3.9	5.1	4.5	19
EMF	2.8	3.2	2.7	2.6	2.4	2.7	2.3	2.8	9
OCL	3.8	34.4	3.7	6.0	3.3	4.3	3.7	NA	10
LS	0.05	14.58	0.23	0.13	0.02	0.10	0.30	0.37	5
INC	0.011	NA	0.009	0.011	0.011	0.010	0.010	0.011	14
Jackrabbit									
ASG	2.8	3.6	2.8	2.8	2.7	2.8	2.6	3.0	4
EMF	1.8	2.3	1.8	1.8	1.8	1.8	1.6	1.9	6
OCL	2.9	24.1	2.9	4.1	2.6	3.3	3.2	NA	8
LS	0.10	50.93	0.26	0.10	0.04	0.13	0.32	0.36	36
INC	0.012	0.013	0.010	0.011	0.011	0.011	0.011	0.012	13
JFreeChart									
ASG	2.3	2.9	2.2	2.3	2.2	2.3	2.1	2.4	8
EMF	1.2	1.4	1.1	1.2	1.1	1.1	1.0	1.2	6
OCL	1.9	12.1	1.9	2.7	1.4	1.8	2.3	NA	6
LS	0.05	6.94	0.16	0.10	0.01	0.06	0.10	0.21	28
INC	0.009	0.010	0.010	0.012	0.012	0.010	0.012	0.012	23
OpenEJB									
ASG	2.6	3.5	2.6	2.8	2.5	2.6	2.4	2.9	3
EMF	1.8	2.3	1.8	1.9	1.9	1.9	1.7	2.0	9
OCL	2.9	20.4	2.8	3.9	2.3	3.4	3.1	NA	7
LS	0.12	12.40	0.25	0.11	0.02	0.17	0.36	0.36	32
INC	0.012	NA	0.009	0.012	0.011	0.011	0.011	0.013	16
Stendhal									
ASG	1.6	2.1	1.7	1.8	1.6	1.7	1.5	1.9	9
EMF	0.9	1.2	1.0	1.0	1.0	1.0	0.9	1.0	2
OCL	1.5	10.1	1.7	2.2	1.2	1.5	1.8	NA	3
LS	0.03	4.64	0.16	0.09	0.01	0.05	0.19	0.23	20
INC	0.010	0.011	0.010	0.013	0.012	0.010	0.012	0.012	14
SVNKit									
ASG	1.6	1.9	1.6	1.7	1.6	1.6	1.5	1.8	7
EMF	1.0	1.2	1.0	1.0	1.0	1.0	0.9	1.0	9
OCL	1.5	13.3	1.8	2.2	1.2	1.6	2.3	NA	6
LS	0.06	9.49	0.16	0.06	0.01	0.09	0.19	0.25	18
INC	0.012	0.012	0.010	0.012	0.011	0.011	0.010	0.012	16
Struts2									
ASG	2.2	2.9	2.2	2.3	2.3	2.2	2.0	2.4	7
EMF	1.2	1.5	1.2	1.2	1.2	1.2	1.1	1.3	4
OCL	2.0	12.7	2.0	2.8	1.5	1.9	2.1	NA	7
LS	0.05	7.09	0.15	0.08	0.02	0.07	0.23	0.26	16
INC	0.012	0.011	0.010	0.011	0.012	0.011	0.011	0.013	14

Please cite this article in press as: Z. Ujhelyi et al., Performance comparison of query-based techniques for anti-pattern detection, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.01.003

Table 5 (*continued*)

	Catch	Cyclomatic complexity	Concatenate	Constant compare	No default switch	Avoid rethrow	String compare	Unused parameter	Maximum deviation (%)
Tomcat									
ASG	2.3	3.0	2.4	2.4	2.4	2.4	2.2	2.6	7
EMF	1.5	2.0	1.5	1.5	1.5	1.5	1.3	1.6	15
OCL	2.6	21.3	2.5	3.3	2.1	2.7	3.1	NA	8
LS	0.08	13.48	0.23	0.10	0.02	0.13	0.33	0.31	16
INC	0.013	0.012	0.011	0.013	0.012	0.012	0.013	0.012	24
Weka									
ASG	3.2	4.3	3.2	3.3	3.1	3.2	3.0	3.4	5
EMF	1.7	2.2	1.7	1.7	1.7	1.7	1.5	1.8	4
OCL	2.8	26.2	3.1	3.6	2.3	2.8	3.2	NA	7
LS	0.06	21.46	0.27	0.09	0.02	0.09	0.39	0.33	23
INC	0.010	0.013	0.009	0.011	0.010	0.010	0.011	0.011	11
Xalan									
ASG	1.7	2.0	1.7	1.7	1.6	1.7	1.6	1.8	9
EMF	1.1	1.3	1.1	1.2	1.1	1.1	1.0	1.2	8
OCL	1.9	12.5	1.8	2.2	1.3	1.8	2.2	NA	3
LS	0.05	11.32	0.16	0.07	0.02	0.08	0.21	0.24	3
INC	0.011	0.011	0.010	0.013	0.012	0.012	0.011	0.013	11

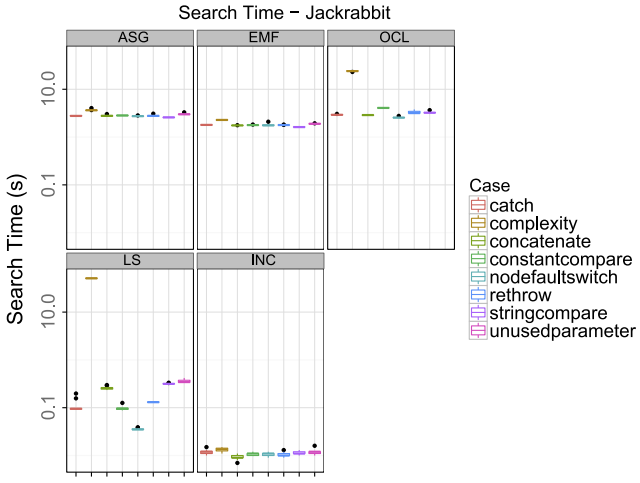


Fig. 9. Distribution of search times for the Jackrabbit project.

to be repeated, the model is reloaded. In our measurements, this mode is represented by a *load* operation followed by a single *query* evaluation.

Commit-time analysis can be used in a program analysis server that keeps the model in-memory, and on each commit, it is updated as opposed to be reloaded, and then it re-executes all queries. In our case, this mode is represented by a *load* operation followed by 10 *query* evaluations.

Save-time analysis is executed whenever the programmer saves a file in the IDE, and then the IDE either executes the analysis itself, or notifies the analysis server. It is similar to commit-time analysis, but it is executed more often. In our measurements, this mode is represented by a *load* operation followed by 100 *query* evaluations.

7.2. Usage profile analysis

We calculated the execution times for the search profiles considering all projects by considering the time to load the models (Table 4a), and increasing it with 1, 10 and 100 times of the search time of six queries one after another, respectively. As the *unused parameter* and *cyclomatic complexity* query could not always be executed in OCL and the incremental matcher, respectively, to keep the results comparable, they were excluded from this calculation.

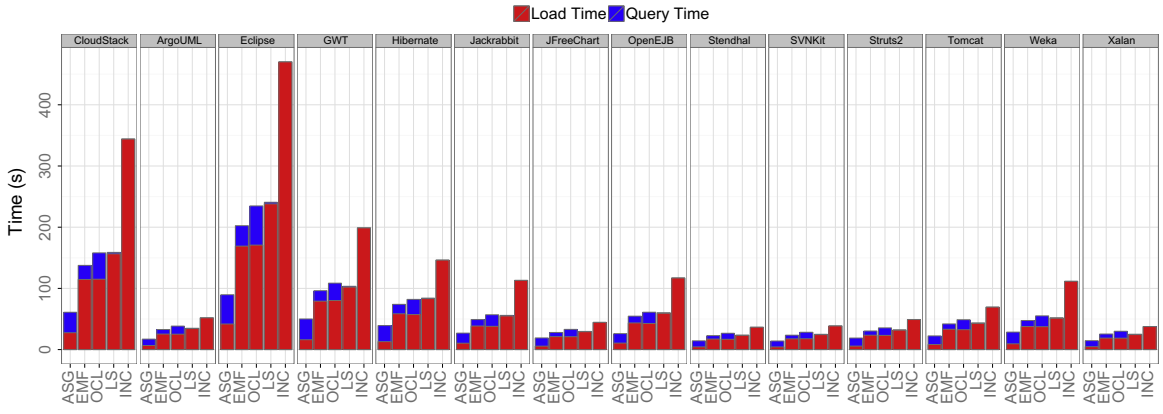
Fig. 10 shows our measurement results of total execution times on the various usage profiles from two points of view. We have included detailed graphs for the selected models where load times and query times can be observed (note the differences in the time axis).

The results show that albeit the *visitor* approaches execute queries slowly, as there are no additional data structures initialized, the lower load time makes this approach very effective for one-time, batch analysis. However, as all visitors are implemented separately, to execute all of them would require six model traversals; reducing this would get further time advantage of this solution over the local search based ones. This issue could be managed by combining all queries in a single visitor, thus increasing its complexity. On the other hand, visitors behave worse regarding run time in the case of repeated analysis: the mean time for executing 100 searches has increased from 32 to 1967 s for the ASG-based implementation (and from 62 to 1257 when executed over EMF).

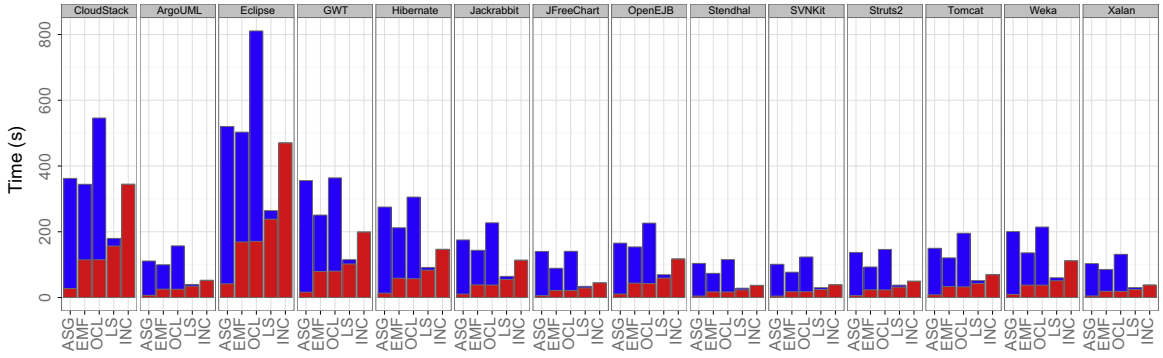
OCL queries behave similarly to visitor-based searches: no indexing is used, but the model is traversed during search. Executing a single query is more expensive than executing a single visitor, and during the measurements nothing is shared between the different executions making the mean one-time execution time of the six queries 71 s (almost the same as the result of the local search based pattern matcher), repeating it a hundred times is done in 2204 s (slower than the ASG version). However, selecting an OCL execution mode that evaluates multiple OCL queries during a single traversal if possible could considerably reduce the total search time, making this approach also a viable alternative to the hand-coded visitors.

The *local search* based approach is noticeably faster than visitor-based solutions with memory usage and initialization time penalties introduced by the use of caching. The mean execution times range from 69 to 171 s. These properties make the approach work very well in the Commit-time analysis profile, and other profiles with a moderate amount of queries. However, if a bad search plan is selected for a query, such as in the case of the Concatenation to Empty String pattern, its execution time may become similar to the visitor-based implementations.

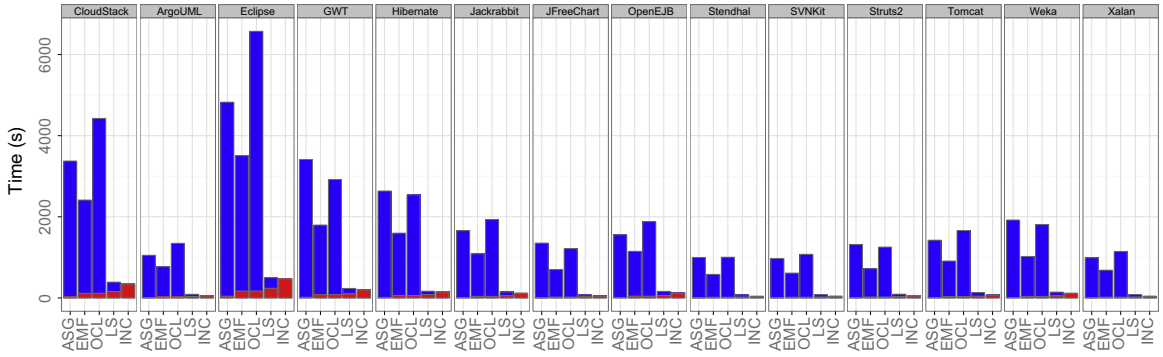
The *incremental*, Rete-based pattern matching approach provides instantaneous model query times, as the results are always available in a cache. This makes such an algorithm powerful for repeatedly executed analysis scenarios, such as the Save-time



(a) One-time



(b) Commit time



(c) Save time

Fig. 10. Execution time over models.

analysis profile (mean time: 131 s, the lowest from all approaches). However, to initialize the caches, a lengthy preparation phase is needed making the technique the slowest for one-time analysis scenarios (mean time: 394 s).

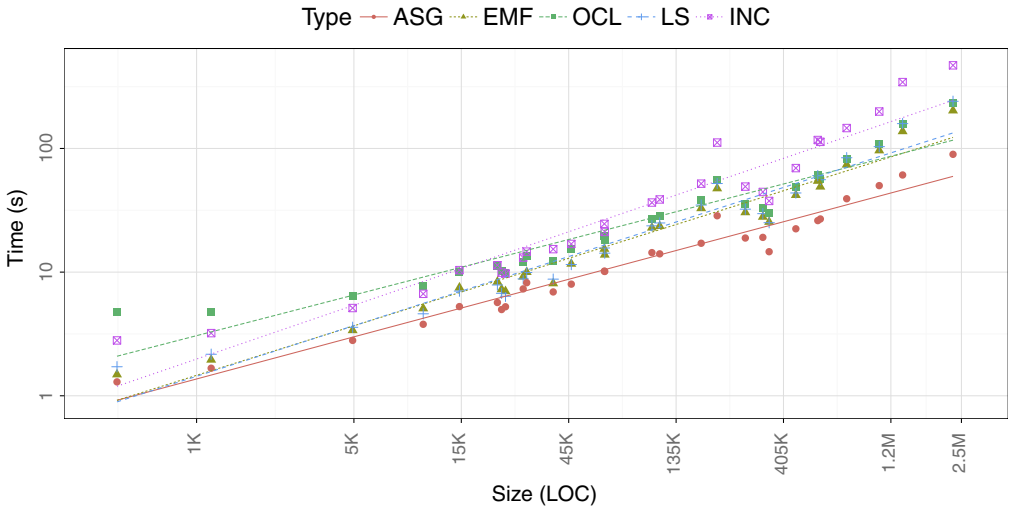
If the save-time analysis profile is used and the *required memory* of the incremental approach cannot be met, it is possible to use the complementing local search matcher that still has a performance benefit over the visitor-based solutions. Additionally, by moving the analysis to a distributed, cloud-based system, it is possible to manage even larger models using the incremental approach [26].

Additionally, we evaluated how execution times changed when increasing the model size. Fig. 11 depicts the analysis time using different tools over the model size in each usage profile and adds linear trend lines to compare the levels of increase. We have found observed our findings are consistent over different models: regardless of the model size, the same relative ordering can be observed in the case of each profile.

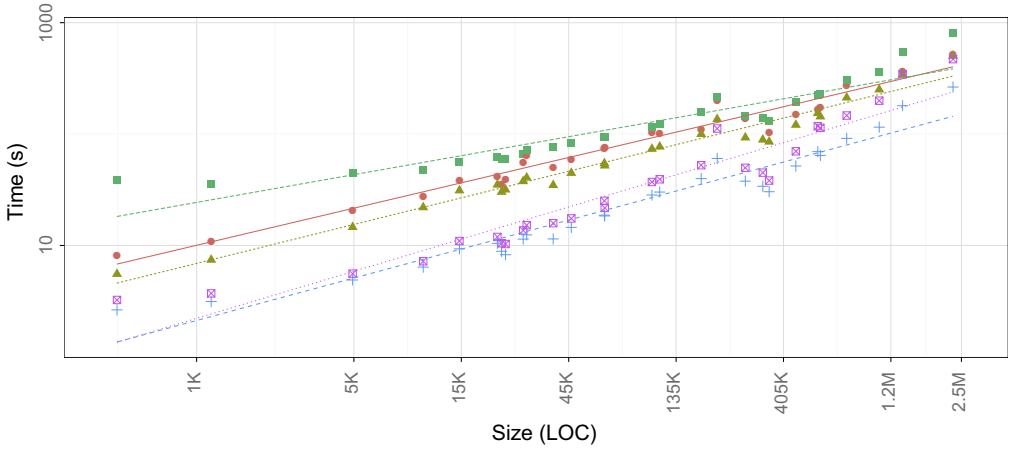
7.3. Lessons learned

From a memory consumption perspective, the manually optimized ASG excels while providing fast query execution for the one-time usage profile. However, generic model implementations, such as EMF, may be viable alternatives when additional features of these frameworks are used and the doubled memory usage is acceptable. Furthermore, the use of generic model implementations allows generic query approaches to become an alternative for manually coded searches based on usage profiles:

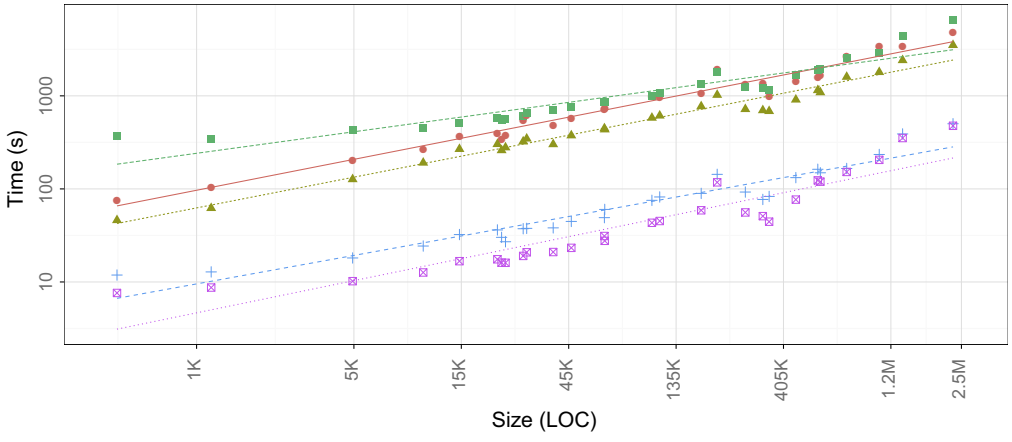
Please cite this article in press as: Z. Ujhelyi et al., Performance comparison of query-based techniques for anti-pattern detection, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.01.003



(a) One-time Usage Profile



(b) Commit time Usage Profile



(c) Save time Usage Profile

Fig. 11. Execution time with regards to model sizes.

- Batch solutions, such as the Eclipse OCL implementation have minimal additional memory requirements while their performance is similar to manually written visitors.
- Full incremental solutions, such as the Rete-based pattern matcher of the EMF-IncQuery, provide results instantaneously even after model changes, making it very useful for recurring queries and evolving source code, if its memory requirements are met.

Please cite this article in press as: Z. Ujhelyi et al., Performance comparison of query-based techniques for anti-pattern detection, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.01.003

- The local search implementation of the EMF-IncQuery uses an incremental indexer to speed up search implementations, achieving query evaluation times that are still orders of magnitude faster than non-indexed solutions, but with a lesser memory consumption. This result is in line with the idea of hybrid pattern matching [25], where incremental and search-based approaches are complementing each other for better performance characteristics.

Both the OCL and the graph pattern formalism provides a higher-level specification of program queries resulting in a compact query description compared to manually coding visitors, and in our subjective experience, they are easier to understand and reduce query development time. Advanced features, such as the computation of transitive closures, are also supported, further reducing the length of query descriptions.

Regardleess of the modeling technology, optimizing the queries, either for performance or memory consumption, may require a deep understanding of the underlying algorithms. In some cases, this means a complete reformulation of the query, e.g. in the case of the *catch problem*, the pattern description requires an inverse navigation between the catch parameters and its references, while the visitor implementation traverses the containment subtree instead.

We have also identified cases where one of the selected tools works noticeably better or worse than the other candidates:

- If inverse relations are not modeled, some queries in OCL cannot be implemented efficiently (e.g. without iterating all instances of a type). Unsurprisingly, adding inverse relations increases the memory usage of the model.
- Navigating the containment hierarchy (especially transitively) requires huge amount of memory with the Rete-based incremental approach, as it requires storing many model element-ancestor pairs in the memory.
- Visitor-based solutions can very effectively traverse the containment hierarchy. In the case of the *cyclomatic complexity* calculation, this is the main reason why the visitor implementations outperform all the others.

Additionally, as a rule of thumb, we have created a simplified representation (see Fig. 12) of the lessons we learned from the results in a form of a decision model to choose the best suited tools for the different usage scenarios. The figure stands as a supplementary guide to help the understanding of our observations above, but it is not a standalone presentation of our results.

In the refactoring project, as mentioned in the motivation section, a refactoring framework is implemented. In this framework, the one time scenario is applied, as the usage scenario was planned for the ASG which does not support incremental model updates. In addition, a large, 4 M LOC proprietary program is refactored, so the decision during this project was to keep the ASG and the one time approach. From this research, we have concluded that generic solutions are viable alternatives and using by an incremental tool setup a huge performance gain can be achieved when enough memory is available.

7.4. Threats to validity

We have identified several validity threats that can affect the construct, the internal and external validity of our results.

Low *construct validity* may threaten our results of various usage profiles, as the results do not include the time required to update the indexes and Rete networks on model changes. However, based on the previous measurement results the related to EMF-IncQuery [21], we believe that such slowdowns are negligible in the cases where the change size is small compared to the model.

Furthermore, in the case of very large heap sizes (over 10 GB) the garbage collection of JVM instances may block the program execution for minutes, in a non-deterministic way. To make the measurements reproducible, the JVM instances were allocating their maximum heap size during startup instead of gradually extending it as needed.

We tried to mitigate *internal validity* threats by comparing the measurements changing only one measurement parameter at a time. For example, the EMF implementation of the Java ASG allows to differentiate between the changes caused by different internal model representations by comparing the different model representations using the same search algorithm first, then com-

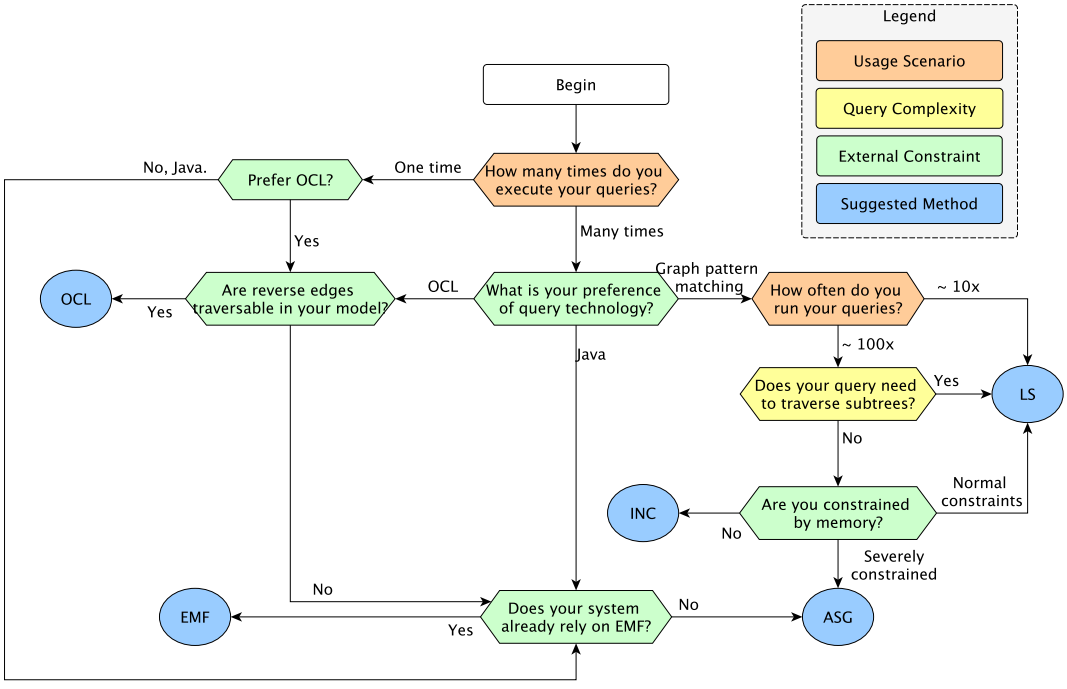


Fig. 12. Decision model (simplified representation).

Please cite this article in press as: Z. Ujhelyi et al., Performance comparison of query-based techniques for anti-pattern detection, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.01.003

paring the EMF-based visitor to generic pattern matching solutions.

An important threat in a study to compare various methods is that the evaluation is done through actual implementations. The decisions in the implementation may affect the overall outcome and the judgement of the methods. To weaken this threat, the implementation is performed by experts of the given technologies. Hence, the same query is implemented in a slightly different way in each method depending on the features of the methods like the availability of reverse edges.

Note that the authors are not experts of the OCL tools, furthermore, the metamodel itself does not favor the structure expected by OCL. However, as we have found that OCL performs comparably to the visitor-based implementations, it is clearly a viable alternative to manually coded searches.

Considering *external validity*, the generalizability of our results largely depends on whether the selected program queries and models are representative for general applications. The queries were selected prior to the projects and scenarios. These refactorings were marked important by project partners and were selected to cover several aspects of transformations.

The selected open-source projects differ in size and characteristics – including computational intensive programs, applications with heavy network and file access and with graphical user interface. Furthermore, the projects were selected from the testbed of the Columbus Java static analyzer and ASG builder program where the aim was to cover a wide range of Java language constructs.

Considering projects from different programming languages requires a corresponding metamodel and instance models. The Columbus framework itself provides metamodels and code analyzers to create these models for various languages, such as C/C++, C# or RPG, and these metamodels can be ported similarly to the EMF. However, an additional evaluation may be needed to validate whether the results still hold, as the properties of those program models may differ significantly.

Another issue is the selection of model query tools. Although several other tools are available, based on the results of more than 10 years of research in efficient graph pattern matching techniques we believe that other pattern matcher tools would provide similar results to either our local search or incremental measurements.

In our work, we used Java-based tools and the EMF framework so that the results of the tools could be comparable. On the other hand, the investigated tools support additional languages. For example, the Columbus API is available in C++ as well, OCL tools are available for different modeling formalisms and languages. The EMF-IncQuery framework is implemented in Java and focuses on EMF models; however the language and runtime are being adapted to different formalisms such as RDF or the metamodeling core of MPS.

OCL queries expect that a context object is selected from the environment and expressions can be evaluated from this point. However, the standard does not specify how to select this context object, and different OCL tools support varying query execution modes. Such modes include the Impact Analyzer of the Eclipse OCL tool [24] that tracks model changes and recomputes only those results that rely on the changed model elements; or the model invariant formulation that can evaluate multiple boolean queries parallel. In order to be able to measure the execution times of single queries, we selected all possible context objects by traversing the entire source model. To evaluate the effects of choosing a different context selection strategy or execution mode, additional measurements are needed.

Altogether, our results were similar for all the models and queries, so we believe our results will generalize well to other program queries and models, until the memory requirements of indexing or Rete building are met.

8. Related work

In our comparison, we evaluated solutions that are specific to program models and generic methods not restricted to the domain of program models. We present related research in two groups starting from generic to program model-specific solutions.

8.1. Software analysis using generic modeling techniques

Program queries are a common use case for modeling and model transformation technologies including transformation tool contests. The program refactoring case of the GraBaTs Tool Contest 2009 [27] and the program understanding case of the Transformation Tool Contest 2011 [28] rely on a program query evaluation followed by some transformation rules, focusing on the applicability of modeling tools for refactoring and reverse engineering. In 2011, six tools entered the contest (GreTL, VIATRA2, Edapt, MOLA, GrGen.NET and Henshin), some of them were EMF-based, others relied on a different metamodeling approach, and in the case of all tools the tasks were executed in a few seconds (albeit sometimes after costly model import operations). This paper extends these results by comparing the costs of using generic modeling environments to manually optimized refactoring models; and extends the performance comparisons with a larger pool of real-world software models and the use of different model queries.

The refactoring case was reused in [29] to select a query engine for a model repository, however, its performance evaluations did not consider incremental cases.

A series of refactoring operations were defined as graph transformation rules by Mens et al. [30], and they were also implemented for both the Fujaba Tool Suite and the AGG graph transformation tools. Although the paper presents that graph transformations are useful as an efficient description of refactoring operations, no performance measurements were included. The Fujaba Tool Suite was also used to find design pattern applications [31]. As a Java model representation, the abstract syntax tree of the used parser generator was used, and the performance of the queries were also evaluated.

The Java Model Parser and Printer (JaMoPP) project [32] provides a different EMF metamodel for Java programs. It was created to directly open and edit Java source files using EMF based techniques, and the changes were written back to the original source code. On the other hand, the EMF model of the JaMoPP project does not support any existing model query or refactoring approaches, every program query or refactoring is to be reimplemented to execute it over the JaMoPP models. This approach was used in [33] relying on the Eclipse OCL tool together with a display of the found issues in the Eclipse IDE.

The EMF Smell and EMF Refactor projects [34] offer to find design smells and execute refactorings over EMF models based on the graph pattern formalism. As Java programs can be translated into EMF models, this also allows the definition and execution of program queries.

As a distinguishing feature from the above mentioned related works, we have compared the performance characteristics of hand-coded and model-based query approaches.

When comparing the performance of the different approaches, an additional factor needs to be considered: as there are multiple different (sometimes not even EMF-based) metamodels used to describe Java applications, additional measurements are required to evaluate the effects of metamodel selection. However, we believe that our test setup is general enough to handle the large set of tools, approaches and queries proposed by these papers.

The train benchmark described in [21] focuses on measuring the performance of incremental model query approaches. It relies on synthetic models scalable to any model size, and defines both

Please cite this article in press as: Z. Ujhelyi et al., Performance comparison of query-based techniques for anti-pattern detection, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.01.003

query and model manipulation steps to measure the real impact of query re-evaluation. [22] aimed to predict the query evaluation performance based on metrics of models and queries both. In the current paper, we reused these metrics on real-world models to evaluate the query engine instead of synthetic models, and while our results were largely similar, further a detailed comparison is required to analyze their usefulness.

8.2. Software analysis designed for program models

For detecting coding issues in Java programs several tools exist. The closest solutions to our ASG+Visitor method are for example the PMD checker [35] and FrontEndART's FaultHunter [36], which is, in fact, built on the top of the Columbus ASG. These applications can be integrated into IDEs as plug-ins, and can be extended with the searches implemented in Java code or in a higher level language, such as XPath queries in PMD. PMD provides rules for a great variety of coding problems, but the provided model and query API is not as flexible as the solutions used in this research. The main usage scenario of these tools is to run the checkers once on (any version of) the source code and find coding issues, they do not support incremental model updates.

On the contrary to generic solutions, there are several systems that support (meta) modeling and querying especially program models. FAMIX [37] is a language-independent meta-model for representing procedural and object-oriented code, used in the Moose reverse engineering environment [38]. The MOOSE environment provides query possibilities in Smalltalk. The authors state that their approach is not Smalltalk specific and can be applied Java as well. The Rascal [39] metaprogramming language is designed for source code analysis and manipulation; its analysis features are based on relational calculus, relation algebra and logic programming systems. Its tool support includes an Eclipse based IDE, and the language provides Java integration: for any task not (easily) expressible in RASCAL, one may use Java method bodies inside Rascal functions. These solutions use their own meta model to represent Java programs, on the contrary to solutions in our research, where the Columbus meta model is used through the EMF. However, these tools are candidates for comparative research in the future.

Furthermore, several approaches allow defining program queries using logical programming, such as the JTransformer [40] using Prolog clauses, the SOUL approach [41] relying on logic metaprogramming, while CodeQuest [42] is based on Datalog. However, none of these include a comparison with hand-coded query approaches. The DECOR methodology [43] provides a high-level domain-specific language to evaluate program queries. It was evaluated on 11 open-source projects, including the Eclipse project for performance; it took around one hour to find its defined smells. These results are difficult to compare to ours, as the evaluated queries are different (and some of them more complex than the ones defined in our paper), but they are described in enough detail to extend our environment. However, evaluating the effects of representation and tool selection is problematic, as neither the model representation, implementation structure nor the used programming language is shared between the different approaches.

An important benefit of our approach is the ability to select the query evaluation strategy based on the required usage profile. Additionally, it is possible to re-use the existing program query implementations while using a high-level, graph pattern-based definition for the new queries.

9. Conclusions and future perspectives

We evaluated different query approaches to locate anti-patterns for refactoring Java programs. In a traditional setup, an optimized Abstract Semantic Graph was built by the state-of-the-art static code analysis tool called Columbus, and processed by hand-coded visitor queries. In contrast, an EMF representation was built for the same program model which offers various advantages from a tooling perspective. Furthermore, anti-patterns were identified by generic, declarative queries in different formalisms evaluated with an incremental and a local-search based strategy.

Our experiments that were carried out on 28 open source Java projects of different size and complexity demonstrated that encoding ASG as an EMF model results in an up to 2–3-fold increase in memory usage and an up to 3–4-fold increase in model load time, while incremental model queries provided a better run time compared to hand-coded visitors with 2–3 orders of magnitude faster execution, at the cost of an additional increase in memory consumption by a factor of up to 10–15. Additionally, we provided a detailed comparison between the different approaches making it possible to select one over the other based on the required usage profile and the expressive capabilities of the queries.

To sum up, we emphasize the expressiveness and concise formalism of pattern matching solutions over hand-coded approaches. They offer a quick implementation and an easier way to experiment with queries together with different available execution strategies; on the other hand, depending on the usage profile, their performance is comparable even on 2,000,000 lines of code.

This work provides basis for the improvement of the ASG and its API towards incremental model handling. In addition, several research aims are identified during this work. We plan to involve additional solutions that are designed for program analysis like Rascal and FAMIX. We also plan to extend the empirical analysis from anti-pattern detection to the whole refactoring process including model transformations. The actual transformations can be programmed using Java or the Xtend language, and can be defined as graph transformations as well, where an empirical comparison is of our interest. Another promising idea is provide automatic refactoring fixes to anti-patterns based on primitive transformations and design space exploration techniques.

Acknowledgments

The authors would like to thank István Ráth of Budapest University of Technology and Economics for his help in validating our measurement environment and evaluation.

This paper was partially supported by the Hungarian National Grant GOP-1.2.1-11-2011-0002, the CERTIMOT Project (ERC_HU-09-1-2010-0003) and the EU FP7 STREP Projects MONDO (ICT-611125) and REPARA (ICT-609666).

References

[1] R. Ferenc, Á. Beszédes, M. Tarkiainen, T. Gyimóthy, Columbus – reverse engineering tool and schema for C++, in: Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002), IEEE Computer Society, 2002, pp. 172–181.

[2] G. Varró, F. Deckwerth, M. Wieber, A. Schürr, An algorithm for generating model-sensitive search plans for EMF models, in: Theory and Practice of Model Transformations, Lecture Notes in Computer Science, vol. 7307, Springer, Berlin, Heidelberg, 2012, pp. 224–239.

[3] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, G. Varró, Incremental pattern matching in the VIATRA transformation system, in: Proceedings of 3rd International Workshop on Graph and Model Transformation (GRaMoT 2008), 30th International Conference on Software Engineering, ACM, 2008, pp. 25–32.

[4] Z. Ujhelyi, A. Horváth, D. Varró, N.I. Csiszár, G. Szőke, L. Vidács, R. Ferenc, Anti-pattern detection with model queries: a comparison of approaches, in: Proceedings of IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE 2014), 2014, pp. 293–302 (Software Evolution Week).

[5] G. Szőke, G. Antal, C. Nagy, R. Ferenc, T. Gyimóthy, Bulk fixing coding issues and its effects on software quality: is it worth refactoring? in: Proceedings of

the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014), 2014, pp. 95–104.
[6] G. Bergmann, Z. Ujhelyi, I. Ráth, D. Varró, A graph query language for EMF models, in: Theory and Practice of Model Transformations, Lecture Notes in Computer Science, vol. 6707, Springer, Berlin, Heidelberg, 2011, pp. 167–182.
[7] OMG, Object Constraint Language Specification (Version 2.3.1), Object Management Group, 2012. <http://www.omg.org/spec/OCL/2.3.1/>.
[8] L. Vidács, Refactoring of C/C++ preprocessor constructs at the model level, in: Proceedings of the 4th International Conference on Software and Data Technologies (ICSOFT 2009), 2009, pp. 232–237.
[9] L. Vidács, A. Beszédes, R. Ferenc, Columbus schema for C/C++ preprocessing, in: Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004), IEEE Computer Society, 2004, pp. 75–84.
[10] L. Hamann, L. Vidács, M. Gogolla, M. Kuhlmann, Abstract runtime monitoring with USE, in: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR 2012), IEEE Computer Society, 2012, pp. 549–552.
[11] L. Schrettner, L. Fülöp, R. Ferenc, T. Gyimóthy, Visualization of software architecture graphs of Java systems: managing propagated low level dependencies, in: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ 2010), ACM, New York, NY, USA, 2010, pp. 148–157.
[12] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
[13] G. Bergmann, Incremental Model Queries in Model-Driven Design, Ph.D. dissertation, Budapest University of Technology and Economics, Budapest, 2013.
[14] U. Nickel, J. Niere, A. Zündorf, Tool demonstration: the FUJABA environment, in: Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), ACM Press, Limerick, Ireland, 2000, pp. 742–745.
[15] ATL, The ATLAS Transformation Language, ATLAS Group, 2014. <http://www.eclipse.org/atl>.
[16] R. Geiß, G. Batz, D. Grund, S. Hack, A. Szalkowski, GrGen: a fast SPO-based graph rewriting tool, in: Graph Transformations, Lecture Notes in Computer Science, vol. 4178, Springer, Berlin, Heidelberg, 2006, pp. 383–397.
[17] D. Hearnden, M. Lawley, K. Raymond, Incremental model transformation for the evolution of model-driven systems, in: Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 4199, Springer, Berlin, Heidelberg, 2006, pp. 321–335.
[18] C.L. Forgy, Rete: a fast algorithm for the many pattern/many object pattern match problem, Artif. Intell. 19 (1982) 17–37.
[19] Drools, Drools – The Business Logic integration Platform, 2014. <http://www.jboss.org/drools>.
[20] A. Ghamarian, A. Jalali, A. Rensink, Incremental pattern matching in graph-based state space exploration, Electron. Commun. EASST 32 (2011).
[21] Z. Ujhelyi, G. Bergmann, Ábel Hegedü, Ákos Horváth, B. Izsó, I. Ráth, Z. Szatmári, D. Varró, EMF-IncQuery: an integrated development environment for live model queries, Sci. Comput. Programm. 98, Part 1 (2015) 80–99. Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012).
[22] 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 2013), IEEE, Silicon Valley, CA, USA, 2013, pp. 412–431.
[23] J. Cabot, E. Teniente, A metric for measuring the complexity of OCL expressions, in: Proceedings of the Model Size Metrics Workshop @ MoDELS 2006, 2006.
[24] Eclipse OCL Project, MDT-OCL website, 2014. <https://projects.eclipse.org/projects/modeling.mdt.ocl>.
[25] Á. Horváth, G. Bergmann, I. Ráth, D. Varró, Experimental assessment of combining pattern matching strategies with VIATRA2, Int. J. Softw. Tools Technol. Transfer 12 (2010) 211–230.
[26] G. Szárnyas, B. Izsó, I. Ráth, D. Harmath, G. Bergmann, D. Varró, IncQuery-D: a distributed incremental model query framework in the cloud, in: Model-Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 8767, Springer, International Publishing, 2014, pp. 653–669.
[27] J. Pérez, Y. Crespo, B. Hoffmann, T. Mens, A case study to evaluate the suitability of graph transformation tools for program refactoring, Int. J. Softw. Tools Technol. Transfer 12 (2010) 183–199.
[28] T. Horn, Program understanding: a reengineering case for the transformation tool contest, in: Proceedings Fifth Transformation Tool Contest, Zürich, Switzerland, June 29–30, 2011, Electronic Proceedings in Theoretical Computer Science, vol. 74, Open Publishing Association, 2011, pp. 17–21.
[29] J.E. Pagán, J.G. Molina, Querying large models efficiently, Inform. Softw. Technol. 56 (2014) 586–622.
[30] T. Mens, N. Van Eetvelde, S. Demeyer, D. Janssens, Formalizing refactorings with graph transformations, J. Softw. Maintenance Evol.: Res. Pract. 17 (2005) 247–276.
[31] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, J. Welsh, Towards pattern-based design recovery, in: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), ACM, New York, NY, USA, 2002, pp. 338–348.
[32] F. Heidenreich, J. Johannes, M. Seifert, C. Wende, Closing the gap between modelling and Java, in: Software Language Engineering, Lecture Notes in Computer Science, vol. 5969, Springer, Berlin, Heidelberg, 2010, pp. 374–383.
[33] M. Seifert, R. Samlaus, Static source code analysis using OCL, Electron. Commun. EASST 15 (2008).
[34] T. Arendt, G. Taentzer, Integration of smells and refactorings within the eclipse modeling framework, in: Proceedings of the Fifth Workshop on Refactoring Tools, (WRT 2012), ACM, New York, NY, USA, 2012, pp. 8–15.
[35] PMD, PMD checker, 2014. <http://pmd.sourceforge.net/>.
[36] FrontEndART Software Ltd., SourceMeter module: FaultHunter, 2014. <http://www.frontendart.com/>.
[37] S. Demeyer, S. Ducasse, S. Tichelaar, Why unified is not universal, in: «UML»'99 The Unified Modeling Language, Lecture Notes in Computer Science, vol. 1723, Springer, Berlin, Heidelberg, 1999, pp. 630–644.
[38] S. Ducasse, T. Girba, A. Kuhn, L. Renggli, Meta-environment and executable meta-language using Smalltalk: an experience report, Softw. Syst. Model. 8 (2009) 5–19.
[39] P. Klint, T. van der Storm, J.J. Vinju, Rascal: a domain specific language for source code analysis and manipulation, in: Proceedings of IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009), IEEE, 2009, pp. 168–177.
[40] D. Speicher, M. Appeltauer, G. Kniesel, Code analyses for refactoring by source code patterns and logical queries, in: Proceedings of the 1st Workshop on Refactoring Tools (WRT 2007), 2007, pp. 17–20.
[41] C. De Roover, C. Noguera, A. Kellens, V. Jonckers, The SOUL tool suite for querying programs in symbiosis with eclipse, in: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ 2011), ACM, 2011, pp. 71–80.
[42] E. Hajiyev, M. Verbaere, O. Moor, Codequest: scalable source code queries with datalog, in: ECOOP 2006 -Object-Oriented Programming, Lecture Notes in Computer Science, vol. 4067, Springer, Berlin, Heidelberg, 2006, pp. 2–27.
[43] N. Moha, Y. Guéhéneuc, L. Duchien, A. Le Meur, DECOR: a method for the specification and detection of code and design smells, IEEE Trans. Softw. Eng. 36 (2010) 20–36.

Please cite this article in press as: Z. Ujhelyi et al., Performance comparison of query-based techniques for anti-pattern detection, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.01.003