Anti-pattern Detection with Model Queries: A Comparison of Approaches

Zoltán Ujhelyi, Ákos Horváth, Dániel Varró
Department of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary
{ujhelyiz,ahorvath,varro}@mit.bme.hu
Norbert István Csiszár, Gábor Szőke*, László Vidács†, Rudolf Ferenc
University of Szeged, Hungary
MTA-SZTE Research Group on Artificial Intelligence,† Hungary
Refactoring 2011 Kft.,* Hungary
csiszar.norbert.istvan@stud.u-szeged.hu, {kancsuki,lac,ferenc}@inf.u-szeged.hu

***Abstract*—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. Our paper investigates the use of the popular industrial Eclipse Modeling Framework (EMF) as an underlying representation of program models processed by three general-purpose model query techniques based on native Java code, local-search and incremental evaluation. We provide in-depth comparison of these techniques on the source code of 17 Java projects using queries taken from refactoring operations in different usage profiles. Our results show that general purpose model queries outperform hand-coded queries by 2-3 orders of magnitude, while there is a 5-10 times increase in memory consumption and model load time. In addition, measurement results of usage profiles can be used as guidelines for selecting the appropriate query technologies in concrete scenarios.**

I. 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 eliminate coding anti-patterns. The refactoring process starts with identifying the location of the problem in the source code, and then applying 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 systematic API, model manipulation code, notifications for model changes, persistence layer in XMI, and simple editors and 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 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 four essentially different solutions: (i) hand-coded visitor queries (as used in Columbus), (ii) queries implemented in native Java code over EMF models, (iii) generic model queries following a local search strategy and (iv) incremental model queries using a caching technique.

We compare the performance characteristics of these query technologies by using the source code of 17 open-source Java projects (with a detailed comparison of 5 projects in the paper) using queries for 6 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.

Our main finding is that advanced generic model queries

978-1-4799-3752-3/14 © 2014 IEEE

CSMR-WCRE 2014, Antwerp, Belgium

over EMF models execute several orders of magnitude faster than dedicated, hand coded techniques. However, this performance gain is balanced by a factor of 5-10 increase in memory usage and 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 II introduces the six queries to be investigated in the paper. Section III provides a technological overview including how to represent models of Java programs and capture queries as graph patterns. Our experimental results and their analysis are detailed in Section IV. Section VI lists research related to ours, while Section VII concludes our paper.

II. Case Study Setup

The results presented in this paper are motivated by an ongoing 3-year refactoring research project involving 5 industrial partners, which aims to find an efficient solution to the problem of software erosion. We focus on the detection of coding anti-patterns, the starting point of the refactoring process. In this step we have 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 anti-pattern, the problematic code is replaced with a new condition containing a call to the `equals()` method with an appropriate argument. Such queries for finding patterns and related refactorings are usually implemented on the top of an ASG. We investigate two viable options for implementing queries and refactorings: (1) to implement queries and transformations by developing Java code working directly on the ASG; (2) to create the EMF implementation of the ASG and use model based tools to do the job. Years ago, we experienced that typical modeling tools were able to handle only mid-size program graphs [4]. We now revisit this question and evaluate whether model-based solutions evolved to compete with hand-coded Java based solutions.

In the following we present our study on program queries in a refactoring context, however 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, real-life case studies are first-class drivers of improvements of model driven tools and approaches.

We capture anti-patterns as model queries using a high-level, declarative graph pattern based query language [5]. Compared to standard languages like OCL, this language offers advanced reuse and navigation. What is more, different execution strategies are available to evaluate them.

We selected 6 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 highest priority selection criteria, therefore, these queries vary both in complexity and programming language context ranging from simple traverse-and-check queries to complex navigation queries potentially with multiple negations. Here we briefly and informally describe the refactoring problems and related queries used in our case study.

a) 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.

b) Catch Problem: In a catch block there is an `instanceOf` check for the type of the catch parameter. Instead of the `instanceOf` 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 `instanceOf` operator and check whether it points to the parameter of the containing catch block.

c) Concatenation to Empty String: When a new `String` is created starting with a number, usually an empty `String` is added from 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 containing expression. If the container expression is an infix expression, then we also make sure that the string is located at the expression's left hand side and the kind of the infix operator is the String concatenation ("+").

d) 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 its only parameter is a string literal.

e) 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.

f) 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 places in the method body where parameters are used. However, there are specific cases when an unused parameter cannot be removed 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.

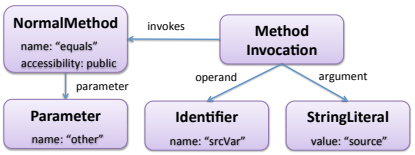
III. Technological Overview

In this section we first give a brief overview on how to represent Java programs as an ASG or EMF, then present the graph pattern formalism, and use it to capture various program queries. Finally, we detail the implementation of these queries using three methods: ASG visitors, local search techniques and Rete networks.

```
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

A. Managing Models of Java Programs

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], [6]). 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 [7], [8] and it conforms to our Java metamodel.

In order to keep the models of large programs in memory, the ASG implementation was 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 [9]. 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.

2) Java Application Models in EMF:

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

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 workflow 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; `EAttribute` 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 as in the Ecore language in Fig. 2 that focuses on method invocations as depicted in Fig. 1. The entire metamodel consists of 142 EClasses with 46 EAttributes and 102 EReferences.

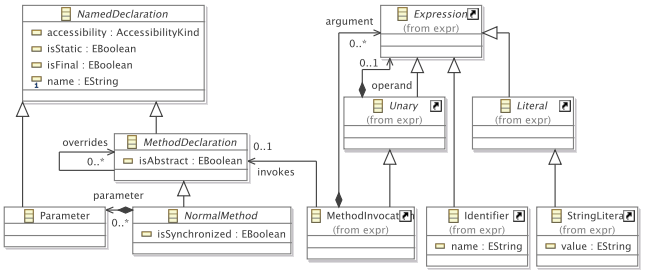
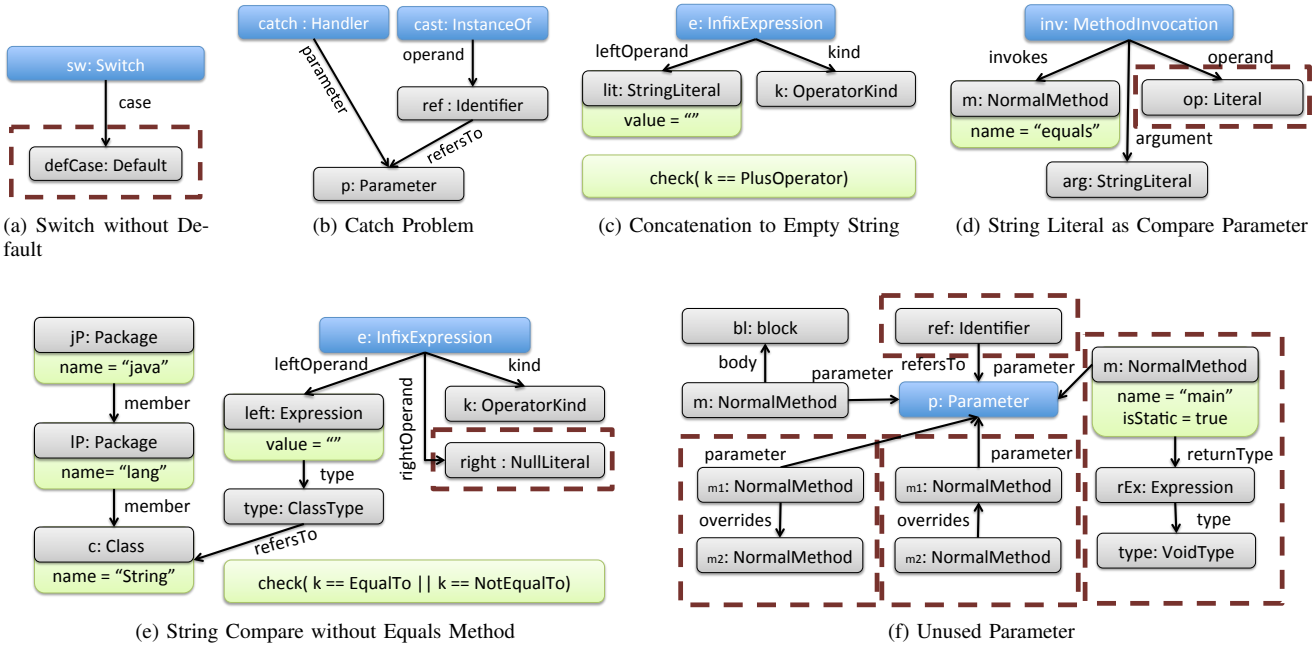


Fig. 2: A Subset of the Ecore Model of the Java ASG

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` selects a `MethodDeclaration` to execute; (2) the `argument` selects a list of expressions to use as the arguments of the called methods, and (3) the inherited `operand` EReference selects an expression representing the object the method is called on.

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



(a) Switch without Default (b) Catch Problem (c) Concatenation to Empty String (d) String Literal as Compare Parameter

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

Fig. 3: Graph Pattern Representation of the Search Queries

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 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 was 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 were handled by generating the standard EMF implementation together with Columbus compatibility methods.

B. Definition of Model Queries using Graph Patterns

Graph patterns [5] are a declarative, graph-like formalism representing a condition (or constraint) to be matched against instance model graphs. The 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.

A graph pattern consists of *structural constraints* prescribing the interconnection between nodes and edges of a given type. They are extended with *expressions* to define *attribute constraints* and *pattern composition* to reuse existing patterns. The called pattern is used as an additional set of constraints to meet, except if it is formed as *negative application condition* (NAC) describing cases when the original pattern is *not* valid.

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 fulfill all the following conditions: (1) have the same structure as the pattern; (2) satisfy all structural and attribute constraints; and (3) does not satisfy any NAC.

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 Sec. II 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 four nodes identified by the variables `inv`, `m`, `op` and `arg`, representing model elements of types `MethodInvocation`, `NormalMethod`, `Literal` and `StringLiteral`, respectively. The distinguishing (blue) formatting for the node `inv` describes its matches that are considered the result of each query.

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.

```
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
        if ("equals".equals(node.getInvokes().getName())) {
            //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 I: 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: Find `arg` that `inv.argument` → `arg`	Extend	Forward
5: Instance test: `arg` ⊂ `StringLiteral`	Check	
6: Find `op` that `inv.operand` → `op`	Extend	Forward
7: NAC analysis: `op` ⊄ `Literal`	Check	Called plan

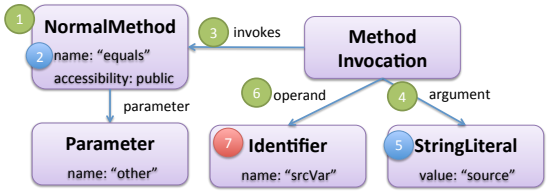


Fig. 5: Executing the Search Plan

C. Implementing Program Queries

1) Manual Search Code: The manually implemented ASG implementation allows to traverse the Java program models using the visitor [9] design pattern that can form the basis of the required search operations.

Visitor-based search implementations 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 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 hard to implement in a sound and efficient way.

2) Graph Pattern Matching with Local Search Algorithms: *Local search based pattern matching* (LS) are commonly used in graph transformation tools [10]–[12] 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 matching (e.g., by matching the target node along an edge), while *check* operations are used to validate constraints between 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 search, but this cost can be reduced by the use of a model indexer. Such an indexer can be set up while loading the model, and then updating it on model changes using the notification mechanism of EMF. If no such indexing mechanism is available (e.g., because of its memory overhead), the search plan generation 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 I 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 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 search states traversed, 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.

3) Incremental Graph Pattern Matching using the Rete algorithm: Incremental pattern matching [3], [13] 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 [14], which is well-known in rule-based systems, was efficiently adapted to several incremental pattern matchers [15]–[17]. The algorithm relies on a network of nodes to store *partial matches* of a graph pattern that enumerate the model elements that satisfy a subset of the graph pattern constraints. The partial matches of a node are readily available at any time as they are updated incrementally at model changes.

Input nodes represent underlying model elements (can be enumerated using an incrementally maintained index, such as EMF-IncQuery Base). *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 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 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 connected by join nodes that calculate the list of elements connected by `invokes`, `argument` and `operand` references, respectively. As both ends of the references 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. 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 same model element, such as the `MethodInvocation` in the example, can be used in multiple

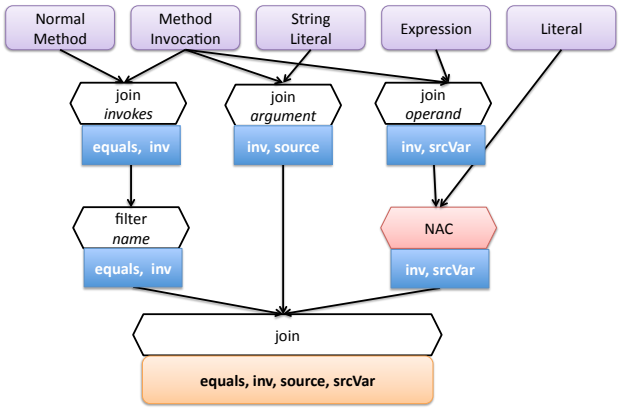


Fig. 6: Rete Network for the String Literal Compare Pattern

join operations; in such cases the final join is responsible for having only a single occurrence of that value (for a selected variable).

IV. Evaluation

To compare the performance characteristics of the different program query techniques, in this section we present detailed performance measurement experiments using program models obtained from Java projects.

A. Description of the Measurements

1) Java Projects: The approaches were evaluated on a test set of 17 open-source projects. The projects are sized between $1kLOC$ and $500kLOC$, and used in various scenarios. The list of projects include the ArgoUML[1] UML editor, JTransforms[2], a Fourier transformation library, the SVNKit[3] Subversion client, the online homework system WeBWorK[4], the Weka[5] data mining software, and many more.

Due to space constraints only the results related to these five projects are presented in full details, as they represent a wide range of application scenarios, and they are the largest ones tested. For a detailed test result with all models visit our website[6]. Table IIb describes the sizes of these projects in lines of code together with the size of the ASG and EMF models in terms of nodes and edges. The larger size of the EMF models are caused by the created proxy model elements.

2) Measurements: All measurements were executed on a dedicated Linux-based server with 6 GB RAM running Java 6. On the server the Java ASG of the Columbus Framework was installed together with EMF-IncQuery.

All program queries were implemented as both visitors for the ASG (by Columbus experts) and as graph patterns (by EMF-IncQuery experts). The visitor implementation was executed on both model representations, while the graph patterns were used to initialize local search and Rete network

[1] http://argouml.tigris.org/
[2] https://sites.google.com/site/piotrwendykier/software/jtransforms
[3] http://svnkit.com
[4] http://webwork.maa.org/
[5] http://www.cs.waikato.ac.nz/ml/weka/
[6] http://incquery.net/publications/program-query-comparison

TABLE II: Measurement Results

(a) Load Times (in seconds)

	ASG	EMF	LS	IQ
argouml	5.30±0.04	14.50±0.15	19.70±0.18	35.00±0.24
jtransforms	2.10±0.06	5.00±0.06	8.00±0.07	17.00±0.12
svnkit	3.90±0.03	10.00±0.16	15.10±0.14	30.00±0.31
webwork	2.30±0.01	5.20±0.06	8.00±0.08	15.00±0.12
weka	9.20±0.15	23.00±0.32	34.40±0.34	71.00±0.56

(b) Code (in LOC) and Model Sizes (# of elements)

		ASG		EMF	
	LOC	Nodes	Edges	Nodes	Edges
argouml	307069	700459	1468495	711121	1479044
jtransforms	43118	294067	651947	294910	652790
svnkit	118733	554165	1225837	566166	1237308
webwork	48472	217616	446727	224544	453641
weka	495412	1529023	3286541	1542205	3299530

(c) Memory Usage (in MB)

	ASG	EMF	LS	IQ
argouml	117.00±0.002	273.00±0.390	361.00±0.006	1571.00±0.572
jtransforms	50.00±0.066	132.00±0.002	174.00±0.0	668.00±0.642
svnkit	76.00±0.001	203.00±0.354	278.00±13.44	1206.00±1.303
webwork	55.00±0.004	84.00±0.002	121.00±0.458	404.00±0.002
weka	234.00±0.001	645.00±0.895	701.00±0.923	3308.00±28.149

(d) Program Query Execution Time (in seconds)

		catch	concate-nate	constant compare	no def. switch	string compare	unused parameter	Total
argouml	ASG	2.1	2.0	2.1	2.0	2.0	2.3	12.6
	EMF	1.3	1.3	1.3	1.3	1.2	1.5	7.9
	LS	0.02	0.67	0.12	0.01	0.14	0.20	1.15
	IQ	0.003	0.003	0.003	0.003	0.003	0.003	0.018
jtransforms	ASG	1.1	1.1	1.1	1.1	0.9	1.2	6.4
	EMF	0.6	0.6	0.6	0.6	0.5	0.6	3.3
	LS	0.03	0.51	0.03	0.02	0.07	0.09	0.75
	IQ	0.005	0.005	0.004	0.004	0.003	0.003	0.024
svnkit	ASG	1.9	2.3	1.9	1.9	1.8	2.2	12.0
	EMF	1.0	1.0	1.0	1.0	0.9	1.1	6.0
	LS	0.02	0.63	0.07	0.01	0.15	0.21	1.09
	IQ	0.003	0.003	0.003	0.003	0.003	0.003	0.019
webwork	ASG	0.9	0.8	0.9	0.8	0.8	1.0	5.2
	EMF	0.7	0.7	0.7	0.7	0.6	0.8	4.0
	LS	0.02	0.34	0.09	0.02	0.14	0.13	0.74
	IQ	0.004	0.005	0.004	0.004	0.005	0.005	0.027
weka	ASG	2.4	2.3	2.5	2.5	2.2	2.8	14.6
	EMF	1.8	1.8	1.8	1.8	1.6	2.0	10.7
	LS	0.03	1.43	0.12	0.01	0.26	0.32	2.17
	IQ	0.003	0.004	0.004	0.003	0.003	0.003	0.020

based pattern matchers for the EMF representation. In all cases, the time to load the model from its serialized form and the time to execute the program query was measured together with the maximum heap size usage.

Every program query was executed ten times, and the time and memory results were averaged. In order to minimize the interference between different runs, for each execution a new JVM was created and each query was run in isolation. The time to start up and shut down the JVM was not included in the measurement results.

B. Detailed Results

1) Load Time and Memory Usage: Table IIa presents the time to load the models in seconds. As our measurements showed that model load time is largely independent from the query selection, we only present an aggregated result table. It can be seen that the load time is 2-3 times longer when using an EMF-based implementation over the manual Java ASG, and further significant 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.

A similar increase can be seen on the memory usage in Table IIc: the EMF representation uses more than twice, while the incremental engine uses around 10 times more memory to store its partial result caches compared to the ASG.

The smaller memory footprint of the Java ASG representation is probably 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.

2) Search Time: Table IId presents the search time measurements. For each model and each program query the average search time is listed at first. The visitor implementations perform similarly, as they traverse the entire model to find the results. The time differences between the ASG and EMF visitor implementations are mainly the result of memory optimizations, as traversing the model requires more indirection than in the EMF case.

The local search and Rete based solutions provide a two or three orders of magnitude faster query execution, achieved by replacing model traversal by calls to a pre-populated (and incrementally updated) index.

Additionally, as Table IId shows, the runtime of visitor implementation increases linearly. This is in line with our expectation, as visitors have to traverse the entire model during search. On the other hand, the search time for incremental queries are roughly the same for all queries, as the search means only returning the results. In most of our patterns, local search is only an order of magnitude slower than incremental queries. However, the concatenation pattern (see Fig. 3c) executes as slow as the visitors in this regard. Based on our earlier experience [18] with different pattern matching strategies the execution performance for local search techniques depends on the query complexity and the model structure.

C. 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 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 re-executes all queries. In our measurements, 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, just it is executed more often. In our measurements, this mode is represented by a *load* operation followed by 100 *query* evaluations.

D. Evaluation of Results

We have calculated the execution times for the search profiles considering all projects by considering the time to load the models (Table IIa), and increasing it with 1, 10 and 100 times of the search time of all six queries one after another (the *Total* column in Table IId), respectively.

Fig. 7 shows our measurement results of total execution times on the various usage profiles from two points of view. First, we have included detailed graphs for the five selected models where load times and query times can be observed (Fig. 7a, Fig. 7b and Fig. 7c; note the differences in the time axis). Second, considering all Java program models, we have added a box plot showing average time values on a logarithmic scale in Fig. 7d. Overall, we found only a few outlier results, meaning our results are largely model-independent.

The results show that albeit the *visitor* implementations execute queries slow, 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 implementing all queries in a single visitor thus increasing implementation complexity. On the other hand, visitors behave worse regarding running time in case of repeated analysis: the mean time for executing 100 searches has increased from 14.7 to 1020 seconds for the ASG-based implementation for the five selected models (for all models it is 14.7 and 574 seconds).

The *incremental*, Rete-based pattern matching approach provides very fast model query times, as the results are always available in a cache. This makes such an algorithm ideal for repeatedly executed analysis scenarios, such as the Save-time analysis profile (mean time: 35.8/15.3 seconds for the five/all models). However, to initialize the caches, a lengthy preparation phase is needed, making the technique the slowest for one-time analysis scenarios (mean time: 33.7/12.6 seconds).

The *local search* based approach is noticeably faster than visitor-based solutions (with memory usage penalty), but consume much less memory to operate than Rete networks. The mean execution times range from 18.2 to 135 seconds (7.9-65.5 for all models). 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 model, such as in case of the Concatenation to Empty String pattern, its execution time may become similar to the visitor-based implementations.

When considering the size of *required memory* when analyzing large program models in the save-time analysis profile, it is possible that albeit incremental matchers should be the fastest, its large memory consumption could be a problem. In such cases, local search based approaches could still provide benefits over visitor-based solutions for execution time.

To sum up, we believe we have demonstrated three outcomes of this evaluation:

1) Generic model implementations, such as EMF, may substitute manually optimized, specific model implementations, as their $2-3$ times increased memory consumption might be acceptable.
2) Generic solutions require additional memory to execute up to a factor of 5 and an additional increased initialization time. However, this additional resource consumption seems acceptable for a large set of problems based on the evaluation.
3) The run time of the analysis itself using generic model query implementations is orders of magnitude faster than manually coded visitor queries.

V. Threats to Validity

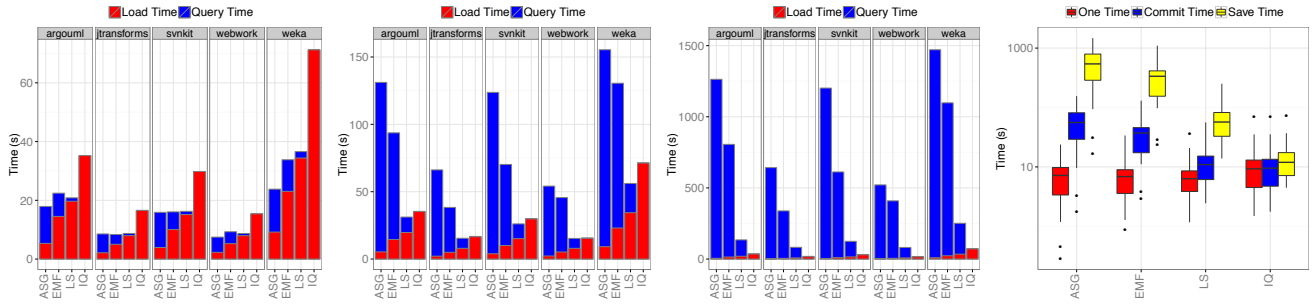
We identified several validity threats that can affect the construct, internal and external validity of our results.

Low *construct validity* may threaten our results of various usage profiles, as results do not include the time required to update the indexes and Rete networks on model changes. However, based on previous measurement results related to EMF-IncQuery [17] we believe that such slowdowns are negligible in most cases.

We tried to mitigate *internal validity* threats by comparing 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 implementations using the same search algorithm first, then comparing the EMF-based visitor to generic pattern matching solutions.

Considering *external validity*, the generalizability of our results largely depend 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, 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



(a) One-time (5 models) (b) Commit time (5 models) (c) Save time (5 models) (d) Usage Profiles (all models)

Fig. 7: Execution Time over Models

language constructs. However, considering projects from different programming languages additional evaluation may be needed to validate whether the results still hold.

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

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.

VI. Related Work

Program queries are a common use case for modeling and model transformation technologies, including transformation tool contests. The program refactoring case of GraBaTs Tool Contest 2009 [19] and the program understanding case of Transformation Tool Contest 2011 [20] both rely on program query implementation followed by some transformation rules. However, as the different submitted solutions store their models differently and use different algorithms, comparing the results from different approaches is problematic.

A series of refactoring operations were defined as graph transformation rules by Mens et al. [21], 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 [22]. As a Java model representation, the abstract syntax tree of the used parser generator was used, and the performance of the implemented program queries were also evaluated. However, because of the age of the measurements, they are hard to compare with current technologies. The Java Model Parser and Printer (JaMoPP) project [23] 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 transformation is to be reimplemented to execute it over the JaMoPP models. The EMF Smell and EMF Refactor projects [24] 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. However, there are no performance evaluations available for this tool.

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

Several tools exist for a related purpose, finding coding problems in Java programs, such as the PMD checker [25], or the FrontEndART CodeAnalyzer [26], which is built on the top of the Columbus ASG. These applications can be integrated into IDEs as plug-ins, and can be extended with searches implemented in Java code or in a higher level language, such as XPath queries in PMD. Furthermore, several approaches allow defining program queries using logical programming, such as JTransformer [27] using Prolog clauses, the SOUL approach [28] relying on logic metaprogramming, while CodeQuest [29] is based on Datalog. However, none of these include a comparison with hand-coded query approaches. The DECOR methodology [30] provides a high-level domain-specific language to evaluate program queries. It was evaluated on 11 open-source projects, however, performance results are hard to compare with our approach as the only listed execution time was of the much larger Eclipse project.

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 implementation for new queries. As a perspective, the model-based infrastructure allows defining and evaluating model queries on-the-fly.

VII. Conclusion

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 evaluated with an incremental and a local-search based strategy.

Our experiments carried out on 17 open source Java projects of different size and complexity demonstrate that encoding ASG as an EMF model results in a factor of 2-3 increase in memory usage and model load, while advanced generic model queries provided better run time compared to hand-coded visitors with 2-3 orders of magnitude, with an increased memory consumption of an order of magnitude.

To sum up, we emphasize the expressiveness and concise formalism of pattern matching solutions (like EMF-INCQUERY) over hand-coded approaches. They offer quick implementation and an easier way to experiment with queries; on the other hand, depending on the usage profile, their performance is comparable even on 500 000 lines of code.

ACKNOWLEDGMENT

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, and 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, Oct. 2002, pp. 172–181.

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

[3] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, and G. Varró, "Incremental pattern matching in the VIATRA transformation system," in *GRaMoT'08, 3rd International Workshop on Graph and Model Transformation*. 30th International Conference on Software Engineering, 2008.

[4] L. Vidács, "Refactoring of C/C++ Preprocessor constructs at the model level," in *Proceedings of ICSOFT 2009, 4th International Conference on Software and Data Technologies*, July 2009, pp. 232–237.

[5] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró, "A graph query language for EMF models," in *Theory and Practice of Model Transformations*, ser. Lecture Notes in Computer Science, J. Cabot and E. Visser, Eds. Springer Berlin / Heidelberg, 2011, vol. 6707, pp. 167–182.

[6] L. Vidács, A. Beszédes, and R. Ferenc, "Columbus Schema for C/C++ Preprocessing," in *Proceedings of CSMR 2004 (8th European Conference on Software Maintenance and Reengineering)*. IEEE Computer Society, Mar. 2004, pp. 75–84.

[7] L. Hamann, L. Vidács, M. Gogolla, and M. Kuhlmann, "Abstract runtime monitoring with USE," in *Proceedings of CSMR 2012 (16th European Conference on Software Maintenance and Reengineering)*. IEEE Computer Society, Mar 2012, pp. 549–552.

[8] L. Schrettner, L. Fülöp, R. Ferenc, and 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*, ser. PPPJ '10, ACM. New York, NY, USA: ACM, 2010, p. 148–157.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[10] U. Nickel, J. Niere, and A. Zündorf, "Tool demonstration: The FUJABA environment," in *The 22nd International Conference on Software Engineering (ICSE)*. Limerick, Ireland: ACM Press, 2000.

[11] *The ATLAS Transformation Language*, ATLAS Group, http://www.eclipse.org/atl.

[12] R. Geiss, G. V. Batz, D. Grund, S. Hack, and A. M. Szalkowski, "GrGen: A Fast SPO-Based Graph Rewriting Tool," in *Graph Transformations - ICGT 2006*, ser. Lecture Notes in Computer Science. Springer, 2006, pp. 383 – 397.

[13] D. Hearnden, M. Lawley, and K. Raymond, "Incremental Model Transformation for the Evolution of Model-Driven Systems," in *Proc. of 9th International Conference on Model Driven Engineering Languages and Systems (MODELS 2006)*, ser. LNCS, vol. 4199. Heidelberg, Germany: Springer Berlin, 2006, pp. 321–335.

[14] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, vol. 19, no. 1, pp. 17–37, Sep. 1982.

[15] "Drools - The Business Logic integration Platform," http://www.jboss.org/drools.

[16] A. Ghamarian, A. Jalali, and A. Rensink, "Incremental pattern matching in graph-based state space exploration," *Electronic Communications of the EASST*, vol. 32, 2011.

[17] G. Bergmann, A. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös, "Incremental evaluation of model queries over EMF models," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, D. Petriu, N. Rouquette, and O. Haugen, Eds. Springer Berlin Heidelberg, 2010, vol. 6394, pp. 76–90.

[18] Á. Horváth, G. Bergmann, I. Ráth, and D. Varró, "Experimental assessment of combining pattern matching strategies with VIATRA2," *International Journal on Software Tools for Technology Transfer*, vol. 12, pp. 211–230, 2010.

[19] J. Pérez, Y. Crespo, B. Hoffmann, and T. Mens, "A case study to evaluate the suitability of graph transformation tools for program refactoring," *International Journal on Software Tools for Technology Transfer*, vol. 12, no. 3-4, pp. 183–199, 2010.

[20] T. Horn, "Program understanding: A reengineering case for the transformation tool contest," in Proceedings Fifth *Transformation Tool Contest,* Zürich, Switzerland, June 29-30 2011, ser. Electronic Proceedings in Theoretical Computer Science, P. Van Gorp, S. Mazanek, and L. Rose, Eds., vol. 74. Open Publishing Association, 2011, pp. 17–21.

[21] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens, "Formalizing refactorings with graph transformations," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 4, pp. 247–276, 2005.

[22] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 338–348.

[23] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, "Closing the gap between modelling and Java," in *Software Language Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2010, no. 5969, pp. 374–383.

[24] T. Arendt and G. Taentzer, "Integration of smells and refactorings within the eclipse modeling framework," in *Proceedings of the Fifth Workshop on Refactoring Tools*, ser. WRT '12. ACM, 2012, pp. 8–15.

[25] PMD, *PMD checker*, http://pmd.sourceforge.net/.

[26] FrontEndART Software Ltd., *QualityGate CodeAnalyzer*, http://www.frontendart.com/.

[27] D. Speicher, M. Appeltauer, and G. Kniesel, "Code analyses for refactoring by source code patterns and logical queries," in *1st Workshop on Refactoring Tools, WRT 2007*, 2007, pp. 17–20.

[28] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers, "The SOUL tool suite for querying programs in symbiosis with Eclipse," in *Proc. of the 9th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '11. ACM, 2011, pp. 71–80.

[29] E. Hajiyev, M. Verbaere, and O. Moor, "codequest: Scalable source code queries with Datalog," in *ECOOP 2006 – Object-Oriented Programming*, ser. Lecture Notes in Computer Science, D. Thomas, Ed. Springer Berlin Heidelberg, 2006, vol. 4067, pp. 2–27.

[30] N. Moha, Y. Guéhéneuc, L. Duchien, and A. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20–36, 2010.