Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Incremental Static Analysis of Large Source Code Repositories

BACHELOR'S THESIS

Author
Dániel Stein

Advisors
Gábor Szárnyas, *PhD Student*
Dr. István Ráth, *Research Fellow*

2014

Contents

Kivonat i

Abstract ii

1 Introduction 1

1.1 Context . . . 1

1.2 Problem Statement and Requirements . . . 2

1.3 Objectives and Contributions . . . 2

1.4 Structure of the Thesis . . . 3

2 Background and Related Work 4

2.1 Big Data and the NoSQL Movement . . . 4

2.1.1 Sharding . . . 5

2.1.2 High Availability . . . 5

2.1.3 4store . . . 6

2.1.4 Query Languages and Evaluation Strategies . . . 6

2.2 Modeling . . . 7

2.2.1 Metamodels and Instance Models . . . 8

2.2.2 The Eclipse Modeling Framework . . . 8

2.2.3 JaMoPP . . . 9

2.2.4 Graph Data Models . . . 10

2.2.5 Model Queries over EMF and RDF . . . 13

2.3 Static Analysis in Practice . . . 14

2.3.1 Checkstyle . . . 14

2.3.2 FindBugs . . . 15

2.3.3 PMD . . . 15

2.4 Well-formedness Checking over Code Models 16

3 Overview of the Approach **18**

3.1 Architecture 18

3.1.1 Main Components 18

3.1.2 Steps of Processing 21

3.1.3 Serializing EMF Models into RDF 23

3.2 Incremental Processing 23

3.2.1 Processing Changes in the Source Code 24

3.2.2 Incremental Query Evaluation 25

3.3 Distributed Processing 25

4 Elaboration of the Workflow **26**

4.1 Case Study 26

4.1.1 Workspace 26

4.1.2 Source Code Snippet 27

4.1.3 Well-formedness Rules 27

4.2 Code Model Representation 28

4.2.1 JaMoPP Metamodel 28

4.2.2 JaMoPP Instance Model 28

4.2.3 Database Contents 29

4.3 Well-formedness Rules in Query Form 31

4.4 Evaluating a Query 34

4.5 Processing Changes 34

5 Evaluation **36**

5.1 Benchmarking Scenarios 36

5.1.1 Validating the Initial Codebase 36

5.1.2 Simulating a Change in the Source Code 37

5.1.3 Projects Selected for Verification 37

5.2 Benchmarking Environment 38

5.2.1 Hardware 39

5.2.2 Software 39

5.2.3 Configuration . . . 39
5.3 Measurement Results . . . 40
5.4 Result Analysis . . . 44
5.4.1 Threats to Validity . . . 45

6 Comparison of Currently Used Static Analyzers 46
6.1 Configurations . . . 46
6.1.1 FindBugs . . . 46
6.1.2 PMD . . . 47
6.1.3 Checkstyle . . . 47
6.1.4 Columbus . . . 47
6.2 Measurement Results . . . 49
6.3 Result Analysis . . . 49
6.3.1 Threats to Validity . . . 50

7 Conclusions 51
7.1 Summary of Contributions . . . 51
7.1.1 Scientific Contributions . . . 51
7.1.2 Practical Accomplishments . . . 51
7.2 Limitation and Future Work . . . 52

Acknowledgments 53

List of Figures iv

Bibliography ix

Appendix x
H.1 JaMoPP metamodel . . . x

HALLGATÓI NYILATKOZAT

Alulírott *Stein Dániel*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2014. december 12.

Stein Dániel
hallgató

Kivonat

Nagyméretű, komplex szoftverek fejlesztésekor a közreműködők számának növekedésével a kódban is gyakrabban fordulhatnak elő hibák. Ezek kiküszöbölésére jelenleg is rendelkezésre állnak megoldások, mint például a statikus analízis, amely többek között automatikusan ellenőrzi, hogy a kód megfelel-e a kódolási szabályoknak.

A statikus analízis a gyakorlatban sokszor lassú és drága művelet. Különösen folytonos integrációs alkalmazása során jelent skálázhatósági kihívást, mert minden módosítás után újra kell futtatni az ellenőrzést a teljes kódbázison. Ugyanakkor nagy igény jelentkezik a statikus analízis technológiák gyorsítására, mivel a gyakorlatban a tesztelést és így a fejlesztés egészét jelentősen gyorsíthatja, hiszen segíti a hibák korai szűrését.

Az egyik lehetséges megoldás az inkrementális feldolgozás. A szakdolgozat keretében olyan rendszert készítettem, mely segítségével a felhasználók által definiált problémákra magas szinten, gráfminták alapján lehet keresni. A rendszer inkrementalitásának lényege, hogy a lekérdezések kiértékelése és riport első generálása után a rendszer hatékonyan fel tudja dolgozni a kód változásait is, így a későbbi futások jóval hatékonyabbak.

A rendszer működőképességének igazolására olyan méréseket terveztem, melyek nyílt forrású programkódok analízisfolyamatainak végrehajtásával betekintést nyújtanak a rendszer skálázhatóságára az elemzett kódbázis méretének tükrében. Ezeket a méréseket elosztott környezetben végeztem el.

Abstract

In large-scale complex software development, the number of coding errors are noticeably increasing with the number of contributors. Mitigatory solutions for this problem exist, but they have their limitations. For example, static analysis methods that verify that the code is compliant with coding conventions are frequently very resource intensive in practice. Especially for continuous integration purposes, the size of the codebase would require a scalable solution because for every changeset in the source, the entire verification process needs to be reapplied on the whole codebase.

Incremental processing is one of the possible solutions for speeding up static analysis. In my thesis I present a system that is able to search for coding problems based on high level specifications by graph patterns. The system supports incremental evaluation by processing source code changes and mapping them efficiently to stages of the processing workflow. Hence, after the initial query evaluation and report generation, consecutive runs are significantly more efficient.

To investigate the scalability of the described system, I present benchmark results that are based on measurements carried out with open source code bases and standard coding validation rules. These benchmarks were executed in a distributed environment.

Chapter 1

Introduction

1.1 Context

Quality control plays an important role in large-scale software development. The more developers work together on developing a software, the more versatile their coding style and conventions are. To ensure the quality of the software source and help developers with their tasks, a solution is needed to continuously review the mistakes in the code and enforce the conventions.

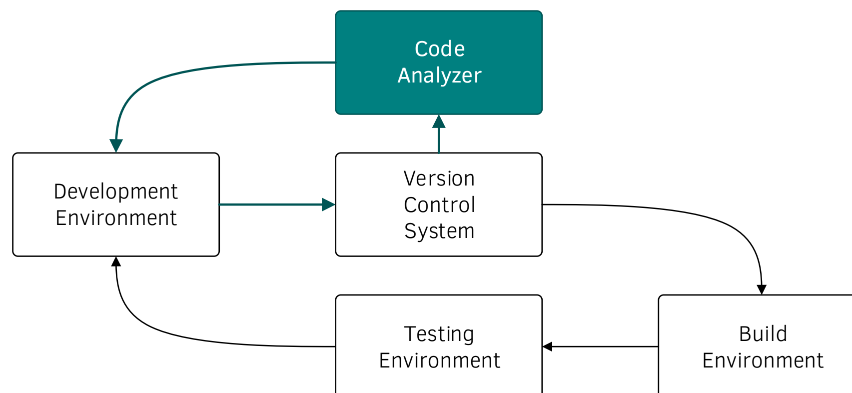

Figure 1.1: *Continuous Integration workflow extended with Static Analysis.*

Version control systems and continuous integration (CI) [1] are widely used tools of the modern software developers. Figure 1.1 shows an extended version of the generally used continuous integration workflow.

The basic workflow consists of the following steps. The developer makes modifications to the codebase using their *Development Environment*. The modifications are committed into a *Version Control System*, and this commit triggers the *Build Environment* to build the project. The *Testing Environment* can then perform runtime tests on latest build of

the project. After this process, the results — build and test logs — are presented to the developer.

This information helps the developers with discovering bugs and failures before the software is released. Producing this information often and early in the development workflow is vital for agile development.

In order to supplement the basic CI workflow, a proven method of enhancing software quality is using static program analysis. During this method, the code is analyzed without executing the application. In practice, this method is able to reveal problems that are undetectable with testing and thus is able to aid the developer in creating high quality software.

1.2 Problem Statement and Requirements

Static analysis techniques are often resource- and time-consuming. As the examined project grows in complexity and in the number of contributors, the number and frequency of source changes is increasing. Hence, static analysis requires even more resources.

For large code bases, static analysis may become so time-consuming that analyzing the software for each change is not practical any more. A temporary solution to tackle this problem is to process the changes in batches. To save resources, static analysis runs are carried out for a joined group of changes, rather than for every individual commit.

In an ideal situation, even before committing the changes, the developers receive feedback about the problems their modifications would imply.

1.3 Objectives and Contributions

Our main objective is to provide a solution for reducing the time required by software source analysis after a new modification is committed.

We aim to create a framework, ISAAC (Incremental Static Analyzer after Commit), that transforms a source code repository into a code model representation graph and performs code convention compliance checks.

Also, it should be able to process a subset of the repository, e.g. only the modifications introduced by the latest commit. This way, the system will be capable of incrementally processing the modifications in a commit. In this context, incremental processing means that only the modifications and their effects are merged into a continuously stored and updated report of the source project.

This framework requires a backend, e.g. a version control system, that is capable of sending notifications of the changes in the source code repository. Version control systems are not only able to provide the latest or the earlier revisions of the code, but also the changes that happened between revisions.

Our framework uses a dedicated component to determine the effects of the changes in addition to the changes provided by the version control system. This component also calculates the source code artifacts that have to be reprocessed with static analysis. This way, we only have to work on a subset of the source code, instead of reprocessing the whole project after a modification occurs.

In order to evaluate our framework, we created a benchmarking environment and conducted measurements on open-source software repositories.

1.4 Structure of the Thesis

This thesis is structured as follows. Chapter 2 introduces the background technologies required to build an incremental static analyzer. Chapter 3 shows the overview of our approach and gives detailed view of the main components of its architecture. Chapter 4 presents our implementation, ISAAC, and discusses the steps of the analysis. Chapter 5 demonstrates and evaluates the performance of ISAAC. Chapter 6 compares the currently used static analyzers. Chapter 7 concludes the thesis and presents our future plans.

Chapter 2

Background and Related Work

In this chapter we introduce the conceptual foundations and technologies which form the basis of our work. We discuss the building blocks required to create and incremental static analyzer. We also investigate similar systems and discuss related work.

2.1 Big Data and the NoSQL Movement

One of the building blocks we are looking for is a database backend that is capable of storing high volume of semi-structured data and can also evaluate queries which return a large amount of data. This section loosely follows [2] and explains why we first turned to NoSQL databases to store graph data.

Since the 1980s, database management systems based on the relational data model [3] dominated the database market. Relational databases have a number of important advantages: precise mathematical background, understandability, mature tooling and so on. However, due to their rich feature set and the strongly connected nature of their data model, relational databases often have scalability issues [4, 5]. They are typically optimized for transaction processing, instead of data analysis (see *data warehouses* for an exception).

In the last decade, large organizations struggled to store and process the huge amounts of data they produced. This problem introduces a diverse palette of scientific and engineering challenges, called *Big Data* challenges.

Big Data challenges spawned dozens of new database management systems. Typically, these systems broke with the strictness of the relational data model and utilized simpler, more scalable data models. These systems dropped support for the Structured Query Language (SQL) used in relational databases and hence were called *NoSQL databases*[1] [6].

As NoSQL databases look more promising than relational databases to store high volume of rich structured (graph) data, we experimented with NoSQL databases.

[1] The community now mostly interprets NoSQL as “not only SQL”.

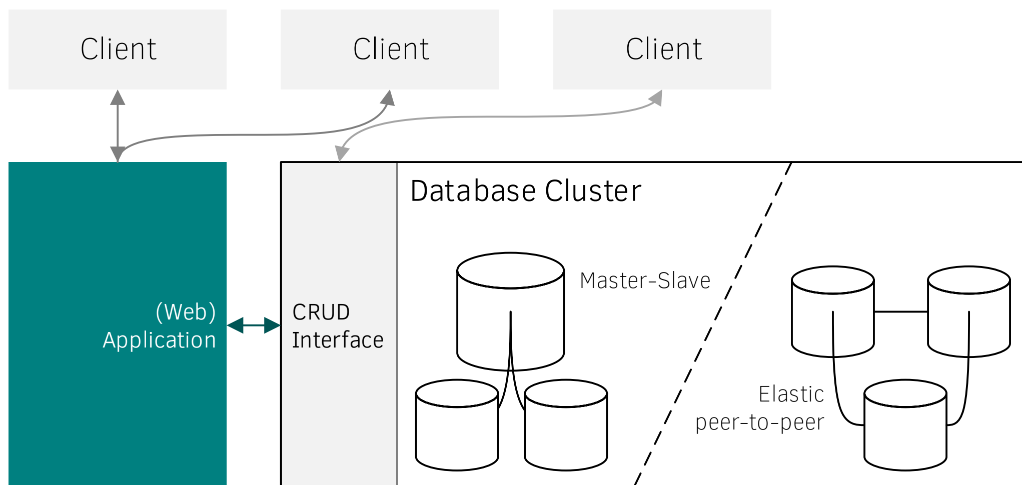

Figure 2.1: *Big Data database architecture.*

Figure 2.1 illustrates a "typical" Big Data database architecture. As we mentioned, many NoSQL databases dropped the support for complex query languages like SQL. They have an interface that provides the CRUD operations: Create, Read, Update and Delete. To perform more complex actions, usually another component, e.g. a web application has to translate them to a sequence of the CRUD operations.

Big Data databases are generally deployed in clusters in practice, as additional services, like sharding and high availability are also needed while storing a high volume of data to be available at any time.

2.1.1 Sharding

To provide scalable persistence and processing for large amounts of data, the data has to be split between multiple computers. This process is known as *data sharding*.

2.1.2 High Availability

High-availability is crucial for near real time data consumer systems, but it is also practical for less time-sensitive use cases. A group of computers arranged into a database cluster are able to provide continuous service even if parts of the system fail.

Without clustering, if a machine fails to provide the service to the clients (e.g. application crashes, network communication issues), the whole system may become unavailable until the problem is eliminated.

High availability clusters mitigate this problem, as a redundant and clustered, fault-tolerant system can provide its services, while it is partially disabled. Until the failed

components are fixed (automatically or with manual help), other components take over the tasks of the failed component.

Figure 2.1 shows two main approaches of cluster architecture, but they can vary for each database type.

2.1.3 4store

Although triplestores have been around for more than a decade, they share many similarities with NoSQL databases, including non-relational data model and support for large amounts of data.

4store is an open-source, distributed triplestore created by Garlik [7] and written in C. It is primarily applied for semantic web projects.

Architecture 4store was designed to work in a cluster with high-speed networks. 4store server instances are capable of discovering each other using the Avahi configuration protocol [8]. 4store offers a command-line and an HTTP server interface.

Data Model 4store's data model is an RDF graph (discussed in Section 2.2.4). It supports various RDF serialization formats, including RDF/XML, Turtle, etc. (see Section 2.2.4), which is processed using the Raptor RDF Syntax Library [9].

Sharding The *segmenting* mechanism in 4store distributes the RDF resources evenly across the cluster. 4store also supports replication by *mirroring* tuples across the cluster.

Query Language and Evaluation 4store uses the Rasqal RDF Query Library [10] to support SPARQL queries (discussed in Section 2.1.4).

2.1.4 Query Languages and Evaluation Strategies

There are numerous strategies to define and execute a query. Queries can be expressed in imperative programming languages over a data access interface such as the Eclipse Modeling Framework (EMF, see Section 2.2.2), or with declarative languages, processed by a query framework, such as OCL [11] or EMF-IncQuery [12].

Pattern matching, one of the various methods to retrieve data from a model is what we base our approach on. Following [13], we define graph patterns and discuss how they are used for querying.

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

Graph patterns 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 does not hold.

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,
3. and 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.

SPARQL SPARQL (recursive acronym for SPARQL Protocol and RDF Query Language) is an RDF query language. (RDF is discussed in detain in Section 2.2.4.) SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. SPARQL also supports aggregation, subqueries, negation, creating values by expressions, extensible value testing, and constraining queries by source RDF graph. The results of SPARQL queries can be result sets or RDF graphs. [14, 15]

For an example SPARQL query, see Section 2.2.5.

2.2 Modeling

Modeling is a very versatile concept, the word itself may refer to various topics. In the context of this thesis, by models we primarily mean data models. A data model, or sometimes called domain model organizes the data elements, how relate to each one another and what actions can be performed with them.

2.2.1 Metamodels and Instance Models

Metamodeling is a methodology for the definition of modeling languages. A metamodel specifies the abstract syntax (structure) of a modeling language. [2]

The metamodel contains the main concepts and relations of the domain specific language (DSL) and defines the graph structure of the instance models. To enrich the expressive power of the language, attributes are added to the concepts. By doing this, the language can be extended with predefined domains of data types (like integers, strings) that are supported by the metamodeling language. Additionally, some structural constraints might be specified with the elements like multiplicity.

Models describing a particular problem in the domain, called instance models, are defined using the elements of the metamodel.

2.2.2 The Eclipse Modeling Framework

Eclipse is a free, open-source software development environment and a platform with extensible plug-in system for customization. Eclipse comes with its own modeling tools, with the core framework called Eclipse Modeling Framework (EMF).

The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. EMF (core) is a common standard for data models, many technologies and frameworks are based on. [16]

Ecore

Ecore is the metamodeling language used by EMF. It has been developed in order to provide an approach for metamodel definition that supports the direct implementation of models using a programming language. Ecore is the de facto standard metamodeling environment of the industry, and several domain-specific languages are defined using this formalism. [2]

Figure 2.2 shows only a small fraction of the metamodel, as there is many more classes in the Ecore metamodel. The main classes are the following:

- `EAttribute` represents a named attribute literal, which also has a type.
- `EClass` represents a class, with optional attributes and optional references. To support inheritance, a class can refer to a number of supertype classes.
- `EDataType` is used to represent simple data types that are treated as atomic (their internal structure is not modeled). Data types are identified by their name.

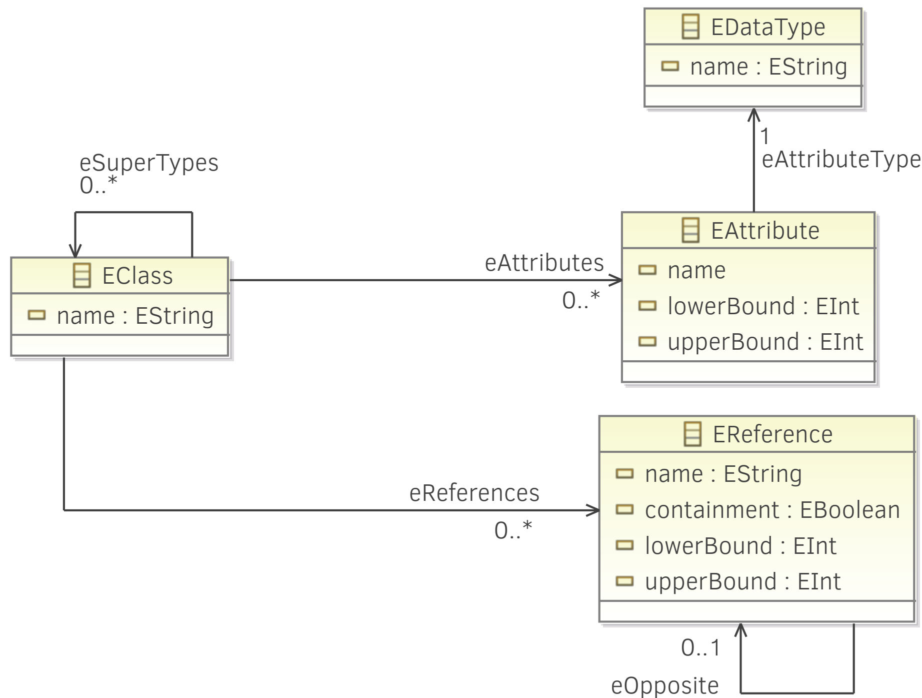

Figure 2.2: *An illustrative core part of Ecore, the metamodeling language of EMF.*

– `EReference` represents a unidirectional association between EClasses and is identified by a name. It is also possible to mark a reference as a containment that represents composition relation between elements. A bidirectional association should be modeled as two EReference instances mutually connected via their opposite references.

An Ecore model has a root object, representing the whole model. The children of this root object are packages, and the children of those are classes.

More detailed illustrations of the metamodel can be found in the EMF Documentation [17].

2.2.3 JaMoPP

JaMoPP [18] is a set of Eclipse plug-ins that can be used to parse Java source code into EMF-based models and vice versa. JaMoPP consists of:

– a complete Java 5 Ecore Metamodel,

– a complete Java 5 EMFText Syntax, and

– an implementation of Java 5's static semantics analysis.

Through JaMoPP, every Java program can be processed as any other EMF model. JaMoPP therefore bridges the gap between modeling and Java programming. It enables the application of arbitrary EMF-based tools on full Java programs. Since JaMoPP is developed through metamodeling and code generation, extending Java and embedding Java into other modeling languages, using standard metamodeling techniques and tools, is now possible. [18]

JaMoPP acts as a serialized file parser, when one wants to open a Java file programmatically. It parses the file and returns an EMF model representing the contents of the file. After this, it can be handled as an ordinary EMF model.

JaMoPP Metamodel Packages

The JaMoPP metamodel completely covers the Java 5 syntax with 18 packages, 80 abstract and 153 concrete classes. The metamodel contains all elements of the Java language, including the newly introduced elements in Java 5 [19]. Unfortunately Java 6 and later versions are not supported yet.

The JaMoPP metamodel packages and their main classes are listed in the Appendix Section H.1.

2.2.4 Graph Data Models

In this section, we enumerate the different graph model types and demonstrate how graph-oriented interpretation of software code model and NoSQL data representations is alike. This section loosely follows [2].

Along the well-known and widely used relational data model, there are many other data models. NoSQL databases are often categorized based on their data model (e.g. key–value stores, document stores, column families). In this thesis, we focus on *graph data models*.

The graph is a well-known mathematical concept widely used in computer science. To understand how the models are transformed in our approach, it is important to understand the concept of different graph data models.

The most basic graph model is the *simple graph*, formally defined as $G = (V; E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. Simple graphs are sometimes referred as textbook-style graphs because they are an integral part of academic literature. Simple graphs are useful for modeling homogeneous systems and have plenty of algorithms for processing.

Simple graphs can be extended in several different ways (Figure 2.3). To describe the connections in more detail, we may add directionality to edges (*directed graph*). To allow different connections, we may label the edges (*labeled graph*).

Typed graphs introduce types for vertices. *Property graphs* (sometimes called *attributed graphs*) add even more possibilities by introducing properties. Each graph element, both

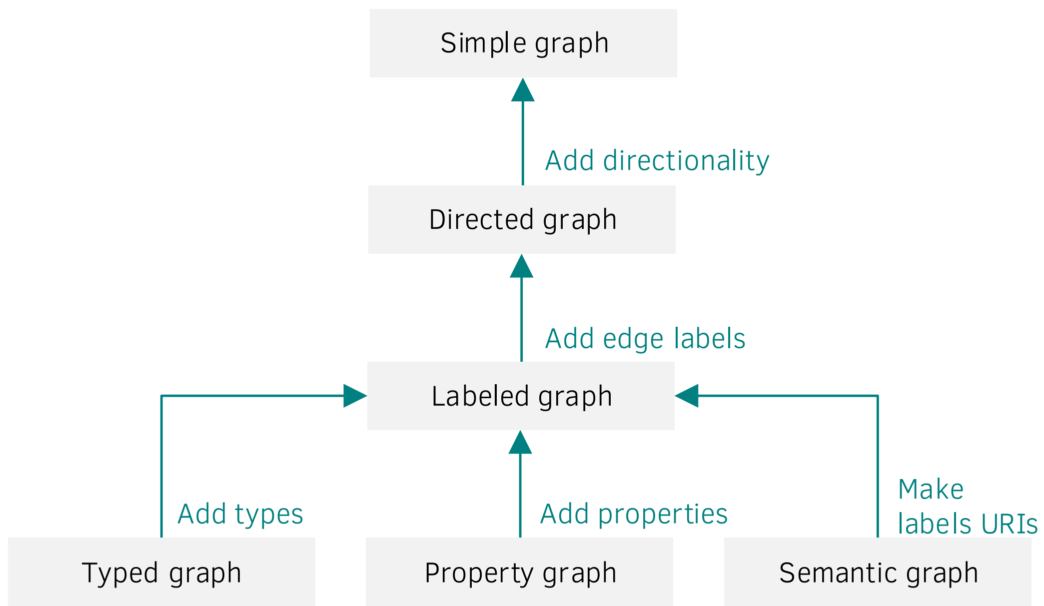

Figure 2.3: *Different graph data models (based on [20]).*

vertices and edges can be described with a collection of properties. The properties are key–value pairs, e.g. `type = 'Person'`, `name = 'John'`, `age = 34`. *Semantic graphs* use Uniform Resource Identifiers (URIs) instead of labels, otherwise they have similar expressive power as labeled graphs.

Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a family of World Wide Web Consortium (W3C) specifications originally designed as a metadata data model. The RDF data model is based on the idea of making statements about resources in the form of triples. A triple is a data entity composed of a subject, a predicate and an object, e.g. "John instanceof Person", "John is 34".

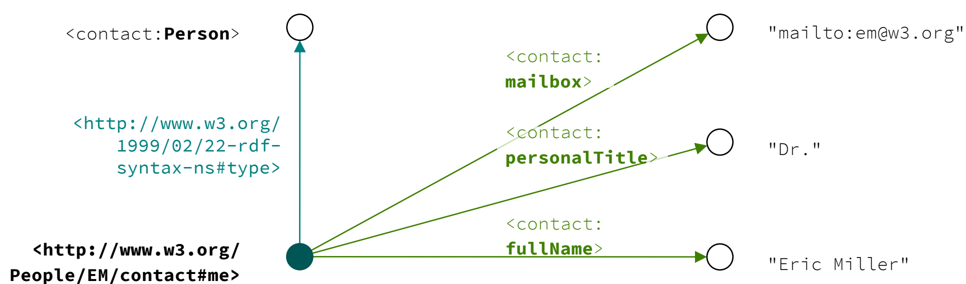

Figure 2.4: *W3C RDF example depicted as a graph.*

From the example in [21], the graph database content is visualized on Figure 2.4. The serialized statements in RDF/XML format of the database are found in Source 2.1.

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#">

  <contact:Person rdf:about="http://www.w3.org/People/EM/contact#me">
    <contact:fullName>Eric Miller</contact:fullName>
    <contact:mailbox rdf:resource="mailto:em@w3.org"/>
    <contact:personalTitle>Dr.</contact:personalTitle>
  </contact:Person>

</rdf:RDF>
```

Source 2.1: *W3C RDF example in RDF/XML format.*

In this example the following statements are expressed:

- `People/EM` identifies a `Person`
- his name is `Eric Miller`
- his title is `Dr.`
- his email address is `em@w3.org`

The RDF data model is capable of expressing semantic graphs. Although the semantic graph data model has less expressive power than the property graph data model, by introducing additional resources for each property, a property graph can be easily mapped to RDF.

There are various serialization formats for RDF, including:

- Turtle [22] — an easy to read, compact format.
- N-Triples [23] — a simple, easy-to-parse, line-based format, where every line contains a triple.
- N-Quads [24] — a superset of N-Triples, when multiple graphs are serialized.
- JSON-LD [25] — a JSON-based RDF format.
- RDF/XML [26] — an XML-based format, it was the first standard format of RDF.

The example in Source 2.1 was serialized in RDF/XML. The following RDF sources in this report are mostly in Turtle format.

There are several graph databases that are able to import graphs from RDF or triplestore formats (e.g. Neo4j [27], Titan [28, 29]/Cassandra [30, 29], Cayley [31], 4store [7]).

2.2.5 Model Queries over EMF and RDF

In this section, we demonstrate how queries can be performed on EMF and RDF data models. First, we show how one can iterate over the elements stored in an EMF model using the Java API provided by EMF (Section 2.2.2). Second, we present a graph pattern (Section 2.1.4) formulated in SPARQL (Section 2.1.4).

Also in Section 3.1.3, we describe in detail how an EMF model can be transformed or serialized into an RDF format making these two query methods interchangeable.

Java Queries over EMF Models

Using the Java API generated with EMF, it is possible to store and search information in the created model. To do so, we have to iterate over the object stored in the model and make sure that the result object and its relations fit to the pattern.

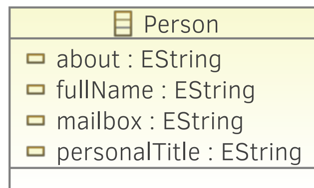

Figure 2.5: *W3C RDF example as an EMF class.*

Figure 2.5 shows the metamodel of the example in Figure 2.4 and Source 2.1. The code in Source 2.2 iterates over the whole model to find every name in the graph. The matches are collected as follows: every object is examined, whether its type is `Person`. If it is, we get its attribute with the identifier `fullName`. The return values are these attributes.

```
URI contactsUri = URI.createFileURI("contacts.xmi");
ResourceSet resourceSet = new ResourceSetImpl();
Resource resource = resourceSet.getResource(contactsUri, true);


List<String> names = new ArrayList<>();

TreeIterator<EObject> allContents = resource.getAllContents();

while (allContents.hasNext()) {
	EObject next = allContents.next();

	if (next.getClass() instanceof Person) {
		names.add(((Person) next).getFullName());
	}
}

return names;
```

Source 2.2: *Java source executing a query over an EMF instance model.*

SPARQL Query

The same query collecting the `fullName` attributes of each `Person` is expressed as a SPARQL query in Source 2.3. The query declares that the results are connected to a node of type `contact:Person` with an edge labeled as `contact:fullName`. The former restriction can be removed, if the metamodels specifies that `contact:fullName` label can only connect values to `contact:Person` nodes.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX contact: <http://www.w3.org/2000/10/swap/pim/contact#>

SELECT ?Name
WHERE
{
      ?Person rdf:type contact:Person .
      ?Person contact:fullName ?Name
}
```

Source 2.3: *Example SPARQL query over the shown RDF graph.*

The results (shown in Table 2.1) of this SPARQL query and the previous Java iteration is the same, and can be easily calculated manually with such a small instance model.

?Name
Eric Miller

Table 2.1: *The results of the query executed on the graph in Figure 2.4.*

2.3 Static Analysis in Practice

In this section we discuss four of the most widely-used static analysis solutions for Java.

2.3.1 Checkstyle

Checkstyle [32] is a development tool to help programmers write Java code that adheres to a coding standard. By default it supports the Sun Code Conventions, but it is highly configurable. It can be invoked with an ANT task and a command line program. [33]

The following paragraphs from [34] show the basic idea that Checkstyle follows for validating the source files:

A Checkstyle configuration specifies which modules to plug in and apply to Java source files. Modules are structured in a tree whose root is the Checker module. The next level of modules contains:

- FileSetChecks — modules that take a set of input files and fire error messages.
- Filters — modules that filter audit events, including error messages, for acceptance.

– AuditListeners — modules that report accepted events.

Many checks are submodules of the TreeWalker FileSetCheck module. The TreeWalker operates by separately transforming each of the Java source files into an abstract syntax tree and then handing the result over to each of its submodules which in turn have a look at certain aspects of the tree.

Checkstyle obtains a configuration from an XML document whose elements specify the configuration's hierarchy of modules and their properties.

2.3.2 FindBugs

FindBugs [35] is a free, open-source static analyzer, searching for problems and bugs in Java code. It can identify hundreds of potential faults in the code, which are given a rank 1–20, and grouped into the categories: scariest (rank 1–4), scary (rank 5–9), troubling (rank 10–14), and of concern (rank 15–20) [36].

According to FindBugs, the following can lead to bugs in the code:

– difficult language features

– misunderstood API methods

– misunderstood invariants when code is modified during maintenance

– garden variety mistakes: typos, use of the wrong boolean operator

FindBugs works by analyzing Java bytecode (compiled class files), which means that the source code of the examined project is not required. It also implies that the source code has to be compiled prior to checking.

Because its analysis is sometimes imprecise, FindBugs can report false warnings, which are warnings that do not indicate real errors. In practice, the rate of false warnings reported by FindBugs is less than 50%. [35]

FindBugs is distributed as a stand-alone GUI application, but there are also plug-ins for popular developer tools, such as Eclipse [37], NetBeans [38], IntelliJ IDEA [39], Maven [40] and Jenkins [41].

2.3.3 PMD

PMD is a free, open-source source code analyzer. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth. It supports Java, JavaScript, XML, XSL. Additionally it includes CPD, the copy-paste-detector. CPD finds duplicated code in Java, C, C++, C#, PHP, Ruby, Fortran, JavaScript. [42]

According to [43] PMD scans Java source code and looks for potential problems like:

- Possible bugs — empty try/catch/finally/switch statements
- Dead code — unused local variables, parameters and private methods
- Suboptimal code — wasteful String/StringBuffer usage
- Overcomplicated expressions — unnecessary if statements, for loops that could be while loops
- Duplicate code — copied/pasted code means copied/pasted bugs

SonarQube

SonarQube [44] (formerly called Sonar) in a free, open-source quality management platform for continuous inspection of source code quality. It is dedicated to continuously analyze and measure technical quality.

Although Sonar provides code analyzers, reporting tools as core functionality, it is also extensible with plug-ins, making it one of the most versatile source quality inspector.

Sonar supports more than 25 languages, including Java, C, C++, Objective-C, C#, JavaScript and Python. It provides reports on coding standards, duplicated code, unit tests and code coverage, potential bugs and others. Storing these reports makes it also able to report trends in these reports over time.

SonarQube cooperates with continuous integration tools, like Maven [40], Jenkins [41], and also has plug-in support for developer environments, such as Eclipse [37].

2.4 Well-formedness Checking over Code Models

This section follows [13].

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 [45] and the program understanding case of Transformation Tool Contest 2011 [46] both rely on program query evaluation followed by some transformation rules. The refactoring case was reused in [47] to select a query engine for a model repository, however, its performance evaluations do not consider incremental cases.

A series of refactoring operations were defined as graph transformation rules by Mens et al. [48], and they were also implemented for both the Fujaba Tool Suite and the AGG graph transformation tools. The Fujaba Tool Suite was also used to find design pattern applications [49]. As a Java model representation, the abstract syntax tree of the used parser generator was used, and the performance of the program queries were also evaluated. However, because of the age of the measurements, they are hard to compare with current technologies.

JaMoPP was used in [50] relying on the Eclipse OCL tool together with display of the found issues in the Eclipse IDE. However, the search time of the tool were not measured.

The EMF Smell and EMF Refactor projects [51] 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.

Several tools exist for a related purpose, finding coding problems in Java programs, such as the PMD checker [52], or the FrontEndART CodeAnalyzer [53], 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 [54] using Prolog clauses, the SOUL approach [55] relying on logic metaprogramming, while CodeQuest [56] is based on Datalog. However, none of these include a comparison with hand-coded query approaches. The DECOR methodology [57] provides a high-level domain-specific language to evaluate program queries.

Chapter 3

Overview of the Approach

In this chapter, we will overview the architecture of the proposed approach, look at the components one-by-one and discuss how they work together. We also discuss how the approach may increase the efficiency of certain static analysis techniques by adapting incremental query evaluation.

3.1 Architecture

In this section, we will discuss the components of the system, and how they cooperate, in detail.

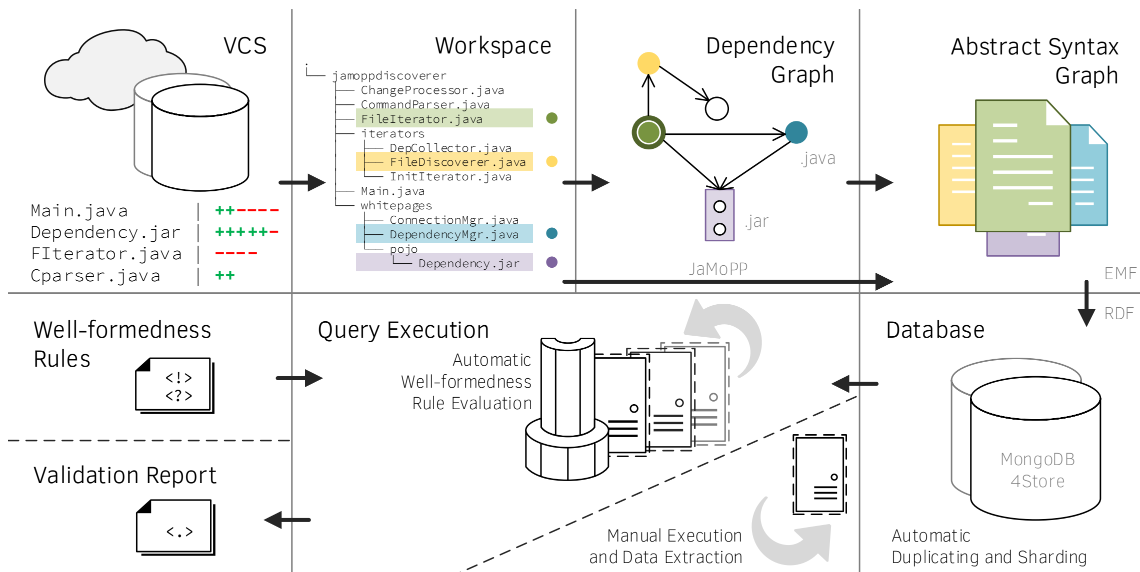

Figure 3.1: *Overview of the system.*

3.1.1 Main Components

VCS (Version Control System) Version control systems are the management systems of changes to files (e.g. documents, software code, etc.). They store revisions of the current

set of files managed by the system. Each revision is differentiated with a timestamp, and a person performing the changes are also associated with a revision.

Version control is one of the most essential collaboration tool. When developers work on the same codebase, especially when the codebase is large, they need to share the code and work on it at the same time. Using a VCS, one can investigate the version of the codebase at a selected revision. Also, it is possible to determine the changes made between two revisions, manage multiple development branches with the same root and merge the changes made in these.

The role of this component is to determine the changes made to the codebase and to provide the most current revision of files. Any VCS can be used with our approach, as long as they are able to provide the list of added, modified and removed files. The most known implementations are Git [58], Subversion (SVN) [59], Mercurial (Hg) [60], Concurrent Versioning System (CVS) [61].

Workspace The workspace is the part of the (local or remote) file system, where the sources of the projects are stored. All of the analyzed sources and the sources of their dependencies are aggregated here. Only these sources are reachable for the analyzer.

Dependency Graph The dependency graph describes the dependencies between the files of the source code. This graph contains a node for every classifier, such as classes and interfaces. The connections between two nodes represent dependency relationships.

In this context, the definition of dependency is derived from the definition of Java dependency. As the smallest unit of the processing in our approach is a source code file — and in Java, the compilation units are also a files —, the simplified definition of dependency is the following:

Compilation unit *A* depends on compilation unit *B*, if *A* uses a classifier (class, interface, enum) inside *B*. *A* can not be compiled without previously compiling *B*.

For every file in the workspace the following information is stored providing additional data about the source code:

- the relative path to the file,
- the package of the compilation unit,
- the fully qualified name of the outmost classifier in the compilation unit,
- the explicit dependencies of the file (the outgoing edges of the node),
- the list of files depending on this file (the incoming edges of the node),
- the list of interfaces that the main class implements,
- and the base class the main class in the file extends.

The Dependency Graph is stored in the Database and updated with every change, so in every step of the process it is accessible for every component. From this graph it is quick and easy to query the files required to process a file, and which files may be affected by the change.

Abstract Syntax Graph Abstract Syntax Tree (AST) is a tree representation of the structure of the source code written in a programming language. It is abstract, because it does not contain every detail in its representation. Each node of this tree identify a construct in the source code.

Abstract Syntax Graph (ASG) is a graph built from the AST. The nodes in AST, where identifiers were unresolved, referencing to elements not in the tree are resolved cross-references in the ASG. (This makes the set of trees into a graph.)

Transforming AST into an ASG is done by traversing the AST, while looking up matching references in every other accessible AST.

Every Java source file is parsed into an EMF-based model using JaMoPP. This model is also considered as an AST. Without their dependencies also processed, these models would have unresolved cross-references. To connect these unresolved references and achieve a globally connected graph, we have to process the dependencies of the given file (based on the Dependency Graph) and resolve the references.

Database The Database stores the content of the ASG between changes. Depending on the size of the data to be stored and other minor restrictions, this can be a cluster or a single database solution. The backend can be of any kind, as long as there is a graph database interface available (with some restrictions, e.g. the database should be able to evaluate elementary queries efficiently). For example 4store [7] and MongoDB [62] are tested to work well as a backend, but using relational databases is also possible.

Well-formedness Rules With large code bases, there is a need to ensure the compliance of the code to coding conventions, and static analysis is a way to enforce them. The set of rules described by the users specify the conventions that should be verified in the projects of the Workspace. Examples for these rules and conventions can be found in Section 4.1.3.

In practice, many of these conventions are defined in language specifications (e.g. Oracle Code Conventions for the Java Programming Language [63] or Google Java Style [64]). Static analysis software, such as FindBugs (see Section 2.3.2), collect these conventions into libraries. Also, generally users can specify their own conventions.

In our approach the we introduce a formalism to verify rules on code bases. This approach based on standard formats, integrating them into an unified solution:

- we specify queries on the JaMoPP metamodel,

- represent these models as RDF graphs,
- and formulate the queries as SPARQL graph patterns.

Query Execution This component validates the model in the database with graph queries based on the rules given by the user. A query can be evaluated either manually, executing the queries on demand, or automatically, processing the changes incrementally. In the latter the results of the queries are stored, and only the difference between changes is propagated through the evaluating system.

In this thesis we mainly discuss how the incremental approach can be adapted for static analysis.

Validation Report The result of the evaluation of the rules is the Validation Report. This is presented to the user every time the evaluation is done.

Although in this thesis we do not elaborate on generating reports incrementally, our approach makes it possible to produce validation reports efficiently.

3.1.2 Steps of Processing

The following enumeration presents the basic concept of the process parsing, storing and querying a code base.

0. **Set Up the Workspace**

   Every file in the project is checked out from the VCS and every dependency is placed into the workspace.

1. **Build the Dependency Graph**

   To collect every information mentioned in Section 3.1.1, every file is parsed and the information is accessed with regular expressions. This is performed without compiling the Java sources.

   The relative path to the file is calculated and stored. The source code is investigated line by line. If it contains the package identifier, an import or the name of the classifier, it is stored in the appropriate collection.

   The Dependency Graph itself is similar to a property graph. We parse every file and collect the needed information. We create a node for each fully qualified name (FQN) and connect it to its dependencies. If a node for the dependency FQN does not exist yet, we create one for it. If a compilation unit with this FQN is later processed, its properties will be connected to this node.

   Note that the FQN may not be globally unique and using the FQN as unique value is a great simplification of the problem of building a Dependency Graph. Handling FQN values properly is out of the scope of this thesis and subject to future work.

Proper handling of subclasses, transient class hierarchy (connecting every superclass of the classifier) is subject to future work as well.

2. **Assemble Parts of the Abstract Syntax Graph**

   We process only one file and its dependencies at a time, making the approach more memory-efficient and the processing granular. Serializing the ASG allows us to assemble only parts of the ASG at a time. The transformation process, and how it makes it possible to assemble the whole graph from part is described in Section 3.1.3.

   In this step, the following procedure is done for every file:

   a) parse the file with JaMoPP (from now on referred as main file),
   b) the model that was already parsed or parse all of its dependencies identified using the Dependency Graph,
   c) and parse every file in the same package as the main file, loading implicit dependencies.

3. **Transform and Store the Model into an RDF Graph**

   Using the reflective API of EMF, we iterate the objects stored in the model. For every object, we assemble a property graph from the following information about each object:

   a) classes and superclasses of the object,
   b) attributes of the object — the attributes are serialized with type information,
   c) and references of the object.

   This processing phase is detailed in Section 3.1.3.

4. **Store the Full Graph in the Database**

   Right after a part of the ASG is assembled, it can be immediately stored in the Database. This phase is implementation-dependent.

5. **Execute the Queries**

   After the whole ASG is stored, the queries can be executed. This phase is also implementation-dependent.

6. **Generate Report**

   Based on the results of the query execution, a report can be generated. This may require the system to query additional information from the database. (For example the location of the problem, or the content of surrounding lines.)

3.1.3 Serializing EMF Models into RDF

To serialize an EMF model, we have save every information stored inside it. Using the Ecore utilities, it is rather easy. There are many solutions available, but it can be performed manually as well. We save every information into an N-Triples document:

1. Iterate the list of contents inside the EMF model, for every object (`EObject`):
   a) Get the identifier of the object (Unique Resource Identifier, URI).
   b) Get the class and superclass of the object as an URI, and serialize these as triples `<object URI> rdf:type <class URI> .`, for each class, meaning the type of this object is `<class>`).
   c) Get every attribute of the object. Get the URI of the attribute name and the value literal. Serialize each of these attributes, one-by-one: `<object URI> <attribute name URI> <attribute literal> .`, where the attribute value is formatted as either untyped value or typed literal [65].
   d) Get every reference of the object. Get the URI of the reference name and the referenced object. Serialize each of these references, one-by-one: `<object URI> <reference name URI> <referenced object URI> .`.
2. After saving this document, it can be parsed and saved again in any other RDF format, for example the more compressed Turtle format.

The manually produced RDF document, where every triple is appended after each other is then parsed again and saved in a compressed format to save space.

These triples (similarly to the edges in a property graph) refer objects by their resource identifier (assembled using the EMF `EcoreUtil` helper class). The structure of these identifiers is the following: first an absolute path declares a resource containing a graph, and a relative path to the referenced node.

This structure allows us to store the graph of only one compilation unit at a time. If the referenced resource (in this case, compilation unit) had been saved, the referred node can be found. If the referred resource is not stored yet, the reference endpoint can be still stored. After the referred resource is stored, the two graphs are automatically connected.

3.2 Incremental Processing

In this section, we will discuss the defining characteristic of the incremental processing approach. Processing data incrementally in this scenario means that while the sources are changing, the results are stored and later updated rather than fully regenerated. After successfully parsing, processing the files and evaluating the queries, the results are stored.

This method typically consumes more memory and disk storage, but in distributed processing environments this can speed up the whole process. This approach, called space-

time trade-off, trading higher memory consumption for faster execution speeds, is well-known and widely used in computer science.

Applying incremental processing is only efficient, when the time required for changeset processing is proportional to the size of the changeset rather than the whole code base [66, 67].

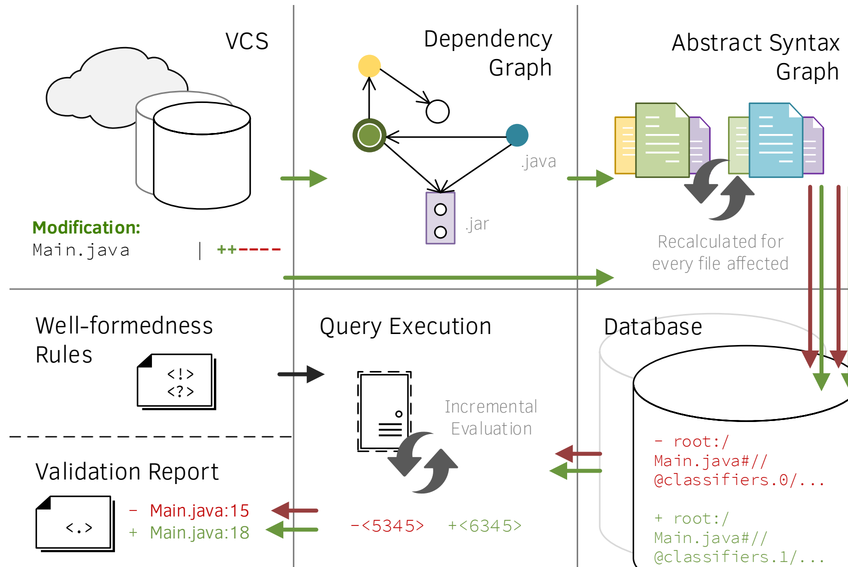

Figure 3.2: *Source code change propagating through the system.*

3.2.1 Processing Changes in the Source Code

As shown in Figure 3.2, a change in the VCS initiates a sequence of processes. There are two main steps for processing changesets: at first the dependency graph needs to be updated. This update could result in additional files to be processed, as the changes could affect other files.

Affected files can be the ones importing or being in the same package with a changed file. Also every extending or implementing classifier that is a descendant of a changed file in the class hierarchy is required to be reprocessed.

After the list of files to be reprocessed is calculated based on the Dependency Graph, the second step begins. Every affected file is reprocessed in this step, one by one.

This fine-grained processing may take more time, but makes the changesets propagating through the system smaller and easier to process. More importantly takes the consistent graph into another consistent state.

There are three main types of changes:

1. **New files are added**

   When a new file is added, all of its dependencies are either already in the dependency graph, or as they are also in new files, will be added in this changeset

processing session. There may be nodes in the graph already referencing to this file, so every one of them needs to be reprocessed in the second step.

2. **Existing files are removed**

   When a file is removed, every other one depending on it needs to be reprocessed in the second step. Also every dependency information of the file needs to be deleted.

3. **Existing files are modified**

   When a file is modified, it needs to be treated as a file that was removed, then added back. (As the granularity of the approach is file-level.)

3.2.2 Incremental Query Evaluation

Once the whole graph has been evaluated, it is feasible to monitor the changes and only process the changeset with the Query Evaluating component. Incremental query evaluation means having the results of the first full evaluation stored in the memory. This result is updated only with the changes coming through the Database.

After the initial query evaluation has been performed, the incremental query evaluator waits for notification only about additions and removals of the graph. (See EMF-IncQuery [12], INCQUERY-D [68].)

Applying incremental query evaluation can result in a fraction of the response time compared to the conventional evaluation method.

3.3 Distributed Processing

As discussed above in Section 3.1.2, the structure of RDF graph references makes it possible to process files one-by-one and store only the appropriate part of the graph under construction in the Database. This also means that there is no need to process every file of a big project or a big changeset sequentially. After the system collected the files for reprocessing, they could be parsed separately, even distributed.

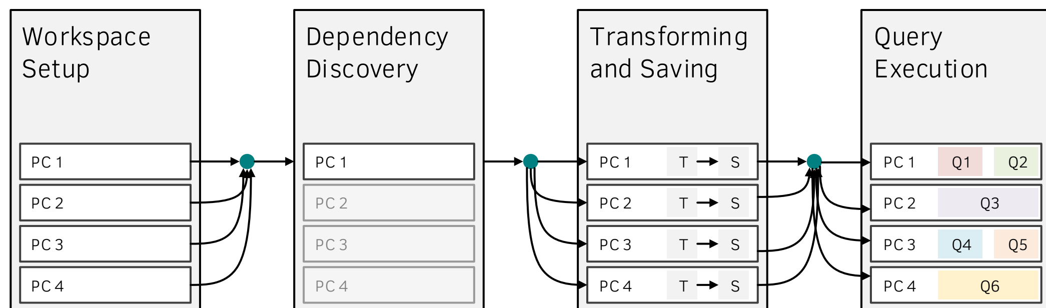

Figure 3.3: *Workflow of the distributed processing.*

Chapter 4

Elaboration of the Workflow

In this chapter we demonstrate the workflow of our approach, how it processes and validates a sample source code. We also look at the input and output for each component participating in the process.

4.1 Case Study

In this section, we present a smaller source code and show how the validation process is carried out. We also take two well-formedness rules commonly used in practice.

4.1.1 Workspace

For this case study, we investigate a small calendar application that has a main class and some utility classes. A snippet from the source code of a utility class is shown in Source 4.1.

To present, what the Dependency Graph looks like in this scenario, Figure 4.1 contains a part of the explicit and implicit dependencies inside of the workspace. The `Utils` package contains two utility classes, and the main application class is connected to the snippet, as the `Calendar.java` file (the main class) calls the `toMonth` method in Source 4.1.

The Snippet also depends on some JRE system library classes, such as `java.lang.System` and `java.lang.Object`. For the sake of clarity, the remaining dependencies were omitted from this visualization.

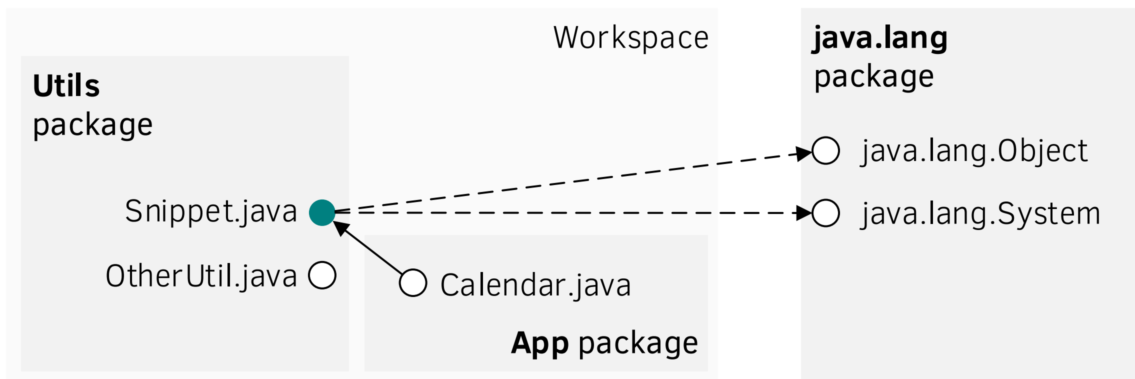

Figure 4.1: *Dependency Graph of the Workspace.*

4.1.2 Source Code Snippet

```
public static String toMonth(int month) {
   switch (month) {
      case 1: return "January";
      case 2: return "February";
      case 3: return "March";
      case 4: return "April";
      case 5: return "May";
      case 6: return "June";
      case 7: return "July";
      case 8: return "August";
      case 9: return "September";
      case 10: return "October";
      case 11: return "November";
      case 12: return "December";
      default: return "Invalid month";
   }
}
```

Source 4.1: *Example Java source, showing a Switch with SwitchCases and a DefaultSwitchCase.*

The code snippet found in Source 4.1 is presented in the Oracle Java Documentation [69] explaining the `switch` language element. The method in the snippet contains a switch, which has 12 "normal" cases and one default case. In Section 4.5 we discuss the differences between using a default case and omitting it.

4.1.3 Well-formedness Rules

The following rules are presented in [13] and are typical well-formedness rules for code models.

The rule definitions and illustrations follow the original article. We added SPARQL queries to demonstrate how these patterns could be translated to queries and search for matches in the instance model persisted in the Database.

Switch without Default

Switches left without default cases can lead to serious problems as the code evolves. It does not necessarily imply that a problem will occur, but the user needs to be informed about the possibility.

Catch Problem

In a `try-catch` block, a `catch()` statement expects an `Exception` of a given type. If the type of the `Exception` is the expected, the inner statements of the `catch` block are executed. These statements should not check the type of the `Exception` again with an `instanceof` expression, as its type has been checked in the `catch()` statement. Instead of the `instanceof` condition a new catch block should be added for the checked type, and the statements should be moved there.

Again, it is not a necessity that a problem will arise, and there are situations, where the problematic approach is correct. However, as a general rule of thumb, the user should get a warning about such threats.

4.2 Code Model Representation

In this section, we show how the source code is represented after various steps of the transformation process. While the previous section presented the original source snippet, here we show the process of the transformation: how it is transformed into a model and what format is the model serialized into. We also discuss the data pieces stored in the database and the query methods used for retrieving them.

4.2.1 JaMoPP Metamodel

The JaMoPP metamodel has been already introduced in Section 2.2.3. To illustrate its structure and make it easier to understand the queries in Section 4.1.3, the following Ecore diagrams show the relevant subsets of the whole metamodel for the queries "Switch without Default" (on Figure 4.2) and "Catch Problem" (on Figure 4.3).

4.2.2 JaMoPP Instance Model

As the source code snippet is parsed by JaMoPP, it is transformed into a JaMoPP Java instance model. This instance model contains every information that JaMoPP was able to extract. After parsing its dependencies into other instance models, the unresolved references in these instance models are resolved, if they can be resolved at all (if the code relies on 3rd party dependencies that are not in the Workspace, the references for these

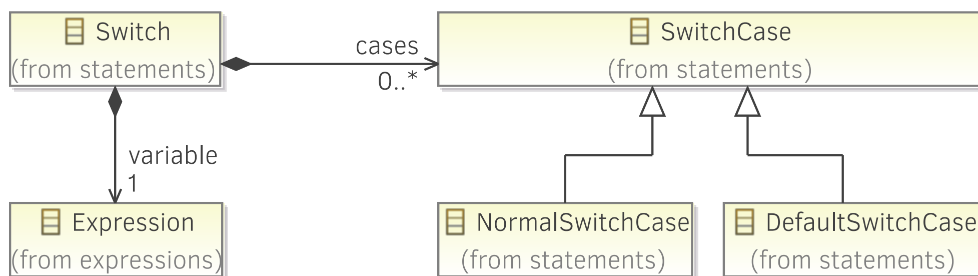

Figure 4.2: *Part of the JaMoPP Java metamodel showing the class hierarchy of the Switch and the various SwitchCase classes.*

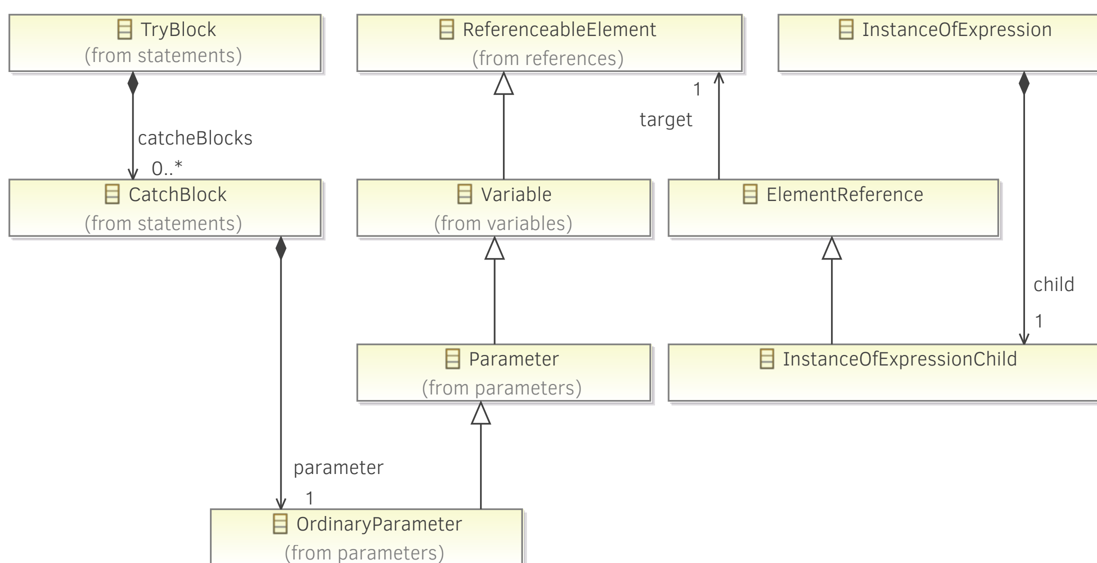

Figure 4.3: *Part of the JaMoPP Java metamodel showing the class hierarchy of the TryBlock, CatchBlock and InstanceOf-Expression classes (with related classes).*

dependencies cannot be resolved). For this sample code, there are no other compilation units, just dependencies to the JRE system library (e.g. `java.lang.Object`).

Figure 4.4 is a visualization of the instance model. The layout information and several `SwitchCases` are excluded from this illustration.

4.2.3 Database Contents

The process of transforming from JaMoPP instance models into RDF graphs is specified in Section 3.1.2. This specific instance model transformation is shown in Source 4.2.

The Switch block is assigned an XPath URI, as it shows the location of the block in the tree hierarchy of the file. This sample Switch can be found in the first Classifier (`@classifiers.0`), in its first Member (`@members.0`), and the Switch itself is that Member's second Statement (`@statements.1`).

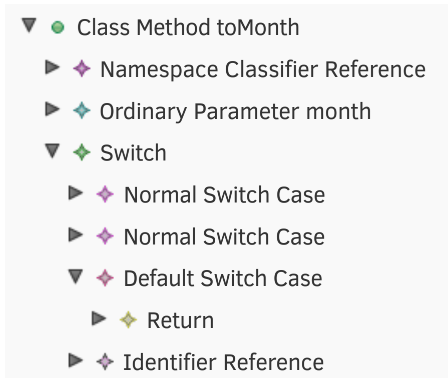

Figure 4.4: *Part of the instance model visualized with the Sample Reflective Ecore Model Editor.*

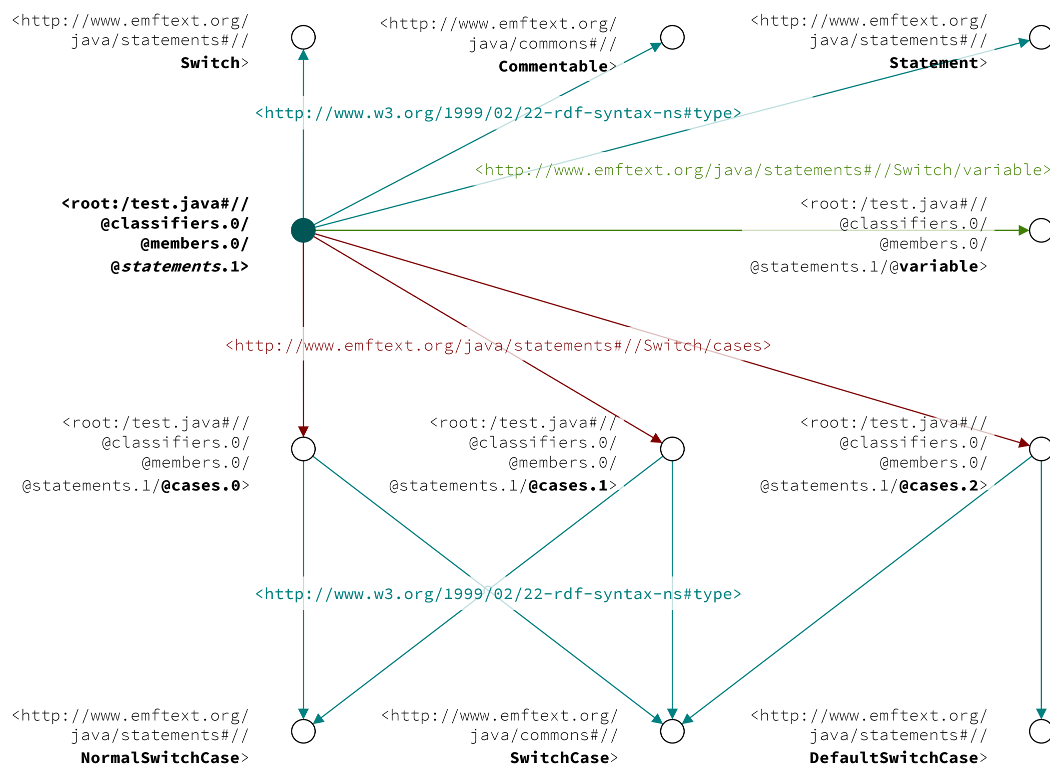

Figure 4.5: *Part of a JaMoPP instance model in RDF format showing a Switch with three SwitchCases, where one of them is a DefaultSwitchCase.*

The first three triples describe the switch itself: it is a `Switch`, a `Commentable` and a `Statement`. This representation also allows us to search for abstract classes or super-classes and get a heterogeneous collection as a result of a query.

The Switch has references to its SwitchCases (`#//Switch/cases`) and finally a Variable reference (`@variable`). In Figure 4.5 and Source 4.2 we only show three SwitchCases, for the sake of clarity. Although the difference between these SwitchCases is clearly visible: all three of them are `SwitchCases`, but only the first two are `NormalSwitchCases`. The third

one is not a `NormalSwitchCase`, rather a `DefaultSwitchCase`. This is why a query pattern could filter the default cases by their type. Additional triples about the SwitchCases and the Statements inside them are excluded from this example.

```
<root:/test.java#//@classifiers.0/@members.0/@statements.1>
      a     <http://www.emftext.org/java/statements#//Switch> ;

<root:/test.java#//@classifiers.0/@members.0/@statements.1>
      a     <http://www.emftext.org/java/commons#//Commentable> ;

<root:/test.java#//@classifiers.0/@members.0/@statements.1>
      a     <http://www.emftext.org/java/statements#//Statement> ;


<root:/test.java#//@classifiers.0/@members.0/@statements.1>
      <http://www.emftext.org/java/statements#//Switch/cases>
            <root:/test.java#//@classifiers.0/@members.0/@statements.1/@cases.0> ;

<root:/test.java#//@classifiers.0/@members.0/@statements.1>
      <http://www.emftext.org/java/statements#//Switch/cases>
            <root:/test.java#//@classifiers.0/@members.0/@statements.1/@cases.1> ;

<root:/test.java#//@classifiers.0/@members.0/@statements.1>
      <http://www.emftext.org/java/statements#//Switch/cases>
            <root:/test.java#//@classifiers.0/@members.0/@statements.1/@cases.2> ;


<root:/test.java#//@classifiers.0/@members.0/@statements.1>
      <http://www.emftext.org/java/statements#//Switch/variable>
            <root:/test.java#//@classifiers.0/@members.0/@statements.1/@variable> ;


<root:/test.java#//@classifiers.0/@members.0/@statements.1/@cases.0>
      a     <http://www.emftext.org/java/commons#//SwitchCase> ,
            <http://www.emftext.org/java/commons#//NormalSwitchCase> ;

<root:/test.java#//@classifiers.0/@members.0/@statements.1/@cases.1>
      a     <http://www.emftext.org/java/commons#//SwitchCase> ,
            <http://www.emftext.org/java/commons#//NormalSwitchCase> ;

<root:/test.java#//@classifiers.0/@members.0/@statements.1/@cases.2>
      a     <http://www.emftext.org/java/commons#//SwitchCase> ,
            <http://www.emftext.org/java/commons#//DefaultSwitchCase> ;
```

Source 4.2: *Example triples stored in the Database.*

Triples like these are stored in the Database. Every value is serialized (URIs, types, literals), and saved using the RDF graph interface supplied by the Database or a database driver implemented by a third-party. The serialization is done by our software, but storing, querying and retrieving the triples is done by the Database.

4.3 Well-formedness Rules in Query Form

To validate the well-formedness rules specified in Section 4.1.3, we need to translate the rules into database queries. The following figures and SPARQL queries show how these rules can be formulated as queries over the JaMoPP metamodel and the 4store database.

Switch without Default

The representation of the well-formedness rule described in Section 4.1.3 interpreted over the JaMoPP metamodel looks like the pattern on Figure 4.6. We search for `Switch` instances that either have no cases at all, or none of their cases are `DefaultSwitchCase`. (The red rectangle around `sc: DefaultSwitchCase` means that there should be no patterns, where a `Switch` and a `DefaultSwitchCase` is connected with an edge labeled as `cases`.)

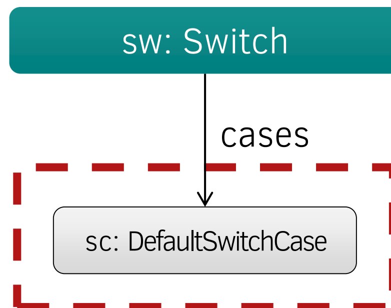

Figure 4.6: *Graph pattern representation of the "Switch without Default" problem.*

The same pattern is formulated as the SPARQL query in Source 4.3. We are searching for nodes that are of type `http://www.emftext.org/java/statements#//Switch`. We also try to find other nodes nodes connected with an edge labeled as `http://www.emftext.org/java/statements#//Switch/cases`. Also, the connected nodes have to be of the type `http://www.emftext.org/java/statements#//DefaultSwitchCase`. There is no negation as a language element in SPARQL 1.0, so we have to use another solution. We make these are optional, and invalidate the found Switch node as a result, if `sc` is bound.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?sw
WHERE
{
      ?sw rdf:type <http://www.emftext.org/java/statements#//Switch> .

      OPTIONAL {
        ?sw <http://www.emftext.org/java/statements#//Switch/cases> ?sc .
        ?sc rdf:type <http://www.emftext.org/java/statements#//DefaultSwitchCase>
      }
      FILTER (!BOUND(?sc))
}
```

Source 4.3: *SPARQL query searching for switches, where the "Switch without Default" problem pattern matches.*

Catch Problem

The representation of the well-formedness rule described in Section 4.1.3 interpreted over the JaMoPP metamodel looks like the pattern on Figure 4.7. We search for `CatchBlock` instances that have a `parameter` of the type `OrdinaryParameter`. We also search for an `InstanceOfExpression` that has a `InstanceOfExpressionChild` (connected with an edge labeled as `/child`), and the child is connected (labeled as `/target`) to the same parameter that the `CatchBlock` is connected to.

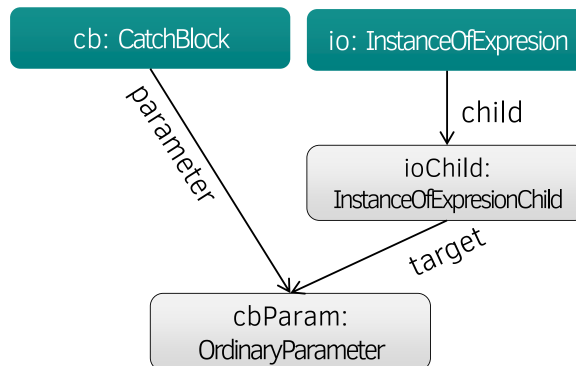

Figure 4.7: *Graph pattern representation of the "Catch Problem" problem.*

The same pattern is formulated as a SPARQL query in Source 4.4. Similarly to the previous SPARQL query, we are also translating the pattern into a SPARQL query. In this pattern there is no negation, meaning every triple shown on Figure 4.7 is expressed naming the nodes and binding the connections between them. If there is a match for the pattern, it is a result of the query.

```
SELECT ?cb
WHERE
{
      ?cb <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            <http://www.emftext.org/java/statements#//CatchBlock> .
      ?cb <http://www.emftext.org/java/statements#//CatchBlock/parameter>
            ?cbParam .

      ?io <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            <http://www.emftext.org/java/expressions#//InstanceOfExpression> .
      ?io <http://www.emftext.org/java/expressions#//InstanceOfExpression/child>
             ?ioChild .
      ?ioChild <http://www.emftext.org/java/references#//ElementReference/target>
            ?cbParam
}
```

Source 4.4: *SPARQL query searching for catch blocks, where the "Catch Problem" problem pattern matches.*

4.4 Evaluating a Query

In this section, we discuss the evaluation of the query specified in Source 4.3. This SPARQL query searches for `Switches`, which either do not have any cases or if they have, none of them are `DefaultSwitchCase`. This query is formalized with the following relational calculus expression:

$$\left\{ sw \mid \text{Switch}\,(sw) \land \neg\exists sc \left(\text{Switch:cases}\,(sw; sc) \land \text{DefaultSwitchCase}\,(sc) \right) \right\}$$

The result set of this expression will be empty, if run on the instance model (shown on Figure 4.5) of the sample snippet described in Section 4.1. This source only has one `Switch`, and one of the cases is a `DefaultSwitchCase`.

4.5 Processing Changes

In this section, we investigate the consequences of the default case being removed from Source 4.1. As the system is notified that a file (the one containing the source snippet) was modified, it needs to be reprocessed.

```
public static String toMonth(int month) {
   switch (month) {
      case 1: return "January";
      case 2: return "February";
      case 3: return "March";
      case 4: return "April";
      case 5: return "May";
      case 6: return "June";
      case 7: return "July";
      case 8: return "August";
      case 9: return "September";
      case 10: return "October";
      case 11: return "November";
      case 12: return "December";
      default: return "Invalid month";
   }
}
```

Source 4.5: *Modified source code snippet.*

1. The path of other compilation units depending on this file are collected for reprocessing. (As Figure 4.1 shows, the files depending on the snippet in Source 4.1 are the ones in the same package and the Calendar.java. These files are the candidates for reprocessing.)
2. The dependency information for this file is removed, recalculated and saved again. As there is no change in the imports, there is no difference between the two versions.
3. The RDF serialization of the previous code version is removed from the Database.

4. The second part of the process starts, every file affected is reprocessed, transformed into a JaMoPP model. The model is serialized into an RDF file and it is stored in the Database.

Our solution updates a file by removing and adding it again, without optimization. Using a difference generating algorithm can reduce the amount of data sent to the Database.

The difference between the two serialized models is only the deletion of three triples. The triples highlighted in Source 4.6 will be removed from the database, so the query will return a match.

```
<root:/test.java#//@classifiers.0/@members.0/@statements.1>
      a     <http://www.emftext.org/java/statements#//Switch> ;

<root:/test.java#//@classifiers.0/@members.0/@statements.1>
      a     <http://www.emftext.org/java/commons#//Commentable> ;

<root:/test.java#//@classifiers.0/@members.0/@statements.1>
      a     <http://www.emftext.org/java/statements#//Statement> ;


<root:/test.java#//@classifiers.0/@members.0/@statements.1>
      <http://www.emftext.org/java/statements#//Switch/cases>
            <root:/test.java#//@classifiers.0/@members.0/@statements.1/@cases.0> ;

<root:/test.java#//@classifiers.0/@members.0/@statements.1>
      <http://www.emftext.org/java/statements#//Switch/cases>
            <root:/test.java#//@classifiers.0/@members.0/@statements.1/@cases.1> ;

<root:/test.java#//classifiers.0/members.0/statements.1>
      <http://www.emftext.org/java/statements#//Switch/cases>
            <root:/test.java#//classifiers.0/members.0/statements.1/cases.2> ;


<root:/test.java#//@classifiers.0/@members.0/@statements.1>
      <http://www.emftext.org/java/statements#//Switch/variable>
            <root:/test.java#//@classifiers.0/@members.0/@statements.1/@variable> ;


<root:/test.java#//@classifiers.0/@members.0/@statements.1/@cases.0>
      a     <http://www.emftext.org/java/commons#//SwitchCase> ,
            <http://www.emftext.org/java/commons#//NormalSwitchCase> ;

<root:/test.java#//@classifiers.0/@members.0/@statements.1/@cases.1>
      a     <http://www.emftext.org/java/commons#//SwitchCase> ,
            <http://www.emftext.org/java/commons#//NormalSwitchCase> ;

<root:/test.java#//classifiers.0/members.0/statements.1/cases.2>
      a     <http://www.emftext.org/java/commons#//SwitchCase> ,
            <http://www.emftext.org/java/commons#//DefaultSwitchCase> ;
```

Source 4.6: *Modifications in the Database.*

sw
`<root:/test.java#//@classifiers.0/members.0/@statements.1>`

Table 4.1: *The results of the query after the source modification.*

Chapter 5

Evaluation

In order to evaluate the performance of the current implementation and compare it to existing static analyzers, we conducted a series of benchmarks. After discussing the setup, we present the benchmark results run on various projects and well-formedness rules.

As the main novelty of our approach is its capability for incremental processing, we simulate a change in the code and measure the time for our implementation to process the changes. This measurement is also presented and interpreted. We expect the incremental processing time to be a fraction of the time required to fully process the whole original project.

5.1 Benchmarking Scenarios

In this section, we discuss the measured parameters of each benchmarking scenario. We also discuss how each parameter is measured and what source projects they are measured on.

5.1.1 Validating the Initial Codebase

First, we validate the codebase as a whole. The Workspace only contains the sources for the one project to be processed. No libraries or additional files are present in the Workspace.

Initially we measure four main phases of processing:

1. building the Dependency Graph,
2. transforming the source code into Abstract Syntax Graph,
3. persisting the ASG to the Database,
4. evaluating the well-formedness queries.

5.1.2 Simulating a Change in the Source Code

For a certain project, we manually modify the source code to eliminate one violation of the well-formedness constraints. In this benchmark, we measure the time required to propagate the changes from source code to the execution of the query:

1. updating the Dependency Graph,
2. transforming the modified and affected sources into Abstract Syntax Graph,
3. updating the persisted ASG in the Database,
4. and re-evaluating the well-formedness queries.

5.1.3 Projects Selected for Verification

From several projects verified in [13], we selected the ones in Table 5.1. Their number of code lines are of different magnitudes, which allowed us to assess the scalability of our approach. Also we tried to collect projects covering most of the well-formedness constraint validations published in the article.

We benchmarked our approach on the following sources:

Project Name	Version	SLOC	Size
Physhun	0.5.1	4 227	0.202 MB
Java DjVu	0.8.06	41 497	1.2 MB
Xalan	2.7	338 316	12.8 MB

Table 5.1: *Selected source code repositories.*

All of these source projects are open-source software, with several contributors. The large amount of contributors, with different skills and background, imply that there are more coding styles and conventions applied in the codebase. Hence, these project are ideal candidates for evaluating our approach.

Physhun

Physhun is a framework for modeling, building and executing processes as Finite State Models in J2SE and J2EE environments. It allow complex processes to be laid out and managed graphically which promotes rapid development and ease of maintenance. Physhun is lightweight enough to run in simple standalone applications, but flexible and powerful enough to be used for orchestration in SOA. Physhun allows processes to be long lived, persistent and transactional. Processes can be purely synchronous, or can be interacted with asynchronously as is common in Workflow processes. [70]

```
     53 text files.
     53 unique files.
      0 files ignored.

http://cloc.sourceforge.net v 1.60 T=0.23 s (234.9 files/s, 18733.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Java                            53            598           1457           2172
-------------------------------------------------------------------------------
SUM:                            53            598           1457           2172
-------------------------------------------------------------------------------
```

Source 5.1: *Source analysis of Physhun.*

Java DjVu Viewer

Java DjVu provides an open source applet, and desktop viewer Java virtual machines. Versions of Java supported include Microsoft Java, Sun Java 1.1 and later, and J2ME. [71]

```
     70 text files.
     70 unique files.
      0 files ignored.

http://cloc.sourceforge.net v 1.60 T=0.69 s (101.2 files/s, 60013.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Java                            69           4677          13243          23570
Bourne Shell                     1              1              0              6
-------------------------------------------------------------------------------
SUM:                            70           4678          13243          23576
-------------------------------------------------------------------------------
```

Source 5.2: *Source analysis of Java DjVu Viewer.*

Xalan-Java

Xalan-Java is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. It implements XSL Transformations (XSLT) Version 1.0 and XML Path Language (XPath) Version 1.0 and can be used from the command line, in an applet or a servlet, or as a module in other program. [72]

5.2 Benchmarking Environment

In this section, we describe the hardware and software components used in the benchmarks. We also discuss the configuration of the components and present the steps for each benchmark.

```
    962 text files.
    953 unique files.
     24 files ignored.

http://cloc.sourceforge.net v 1.60 T=4.48 s (207.5 files/s, 75577.6 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Java                           907          45546         126812         165248
HTML                            22             48            394            268
-------------------------------------------------------------------------------
SUM:                           929          45594         127206         165516
-------------------------------------------------------------------------------
```

Source 5.3: *Source analysis of Xalan-Java.*

5.2.1 Hardware

All of the benchmarks were carried out on one of the DigitalOcean [73] Virtual Private Servers (VPS). Two virtual machines were booked for the benchmarks in the Amsterdam 3 region, both of them had the following specification:

- 4 CPU cores (Intel® Xeon® CPU E5-2630L v2 @ 2.40 GHz)
- 8 GB memory
- 80 GB SSD storage

5.2.2 Software

The following software components were installed on the machines (only the components relevant to the benchmarks are listed):

- Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-32-generic x86_64)
- Java 7 (1.7.0_67)
- 4store (v1.1.5)

5.2.3 Configuration

The first machine was used to process and transform the sources and execute the queries. Both of these machines were used as database backend servers in a two-node 4store cluster as well. These machines, although probably being in close physical proximity, communicated through the public network.

5.3 Measurement Results

In this section, we present the benchmark results of our approach in different phases of processing.

Every benchmark was run at least ten times, and every result presented here is the median of the measurements, filtering out some of the effects of running the measurements in the cloud.

Processing the Entire Codebase

The following benchmarks were run as the sources were added for the first time. The list of freshly added source files — given to our implementation for processing — contained every file in the source directory.

Building the Dependency Graph Figure 5.1 shows the time required for building the Dependency Graph for the fist time. The name of each project and the number of source lines of code (SLOC) is displayed on the horizontal axis, while the measured time in seconds is on the vertical axis. Note that the vertical axis is logarithmic.

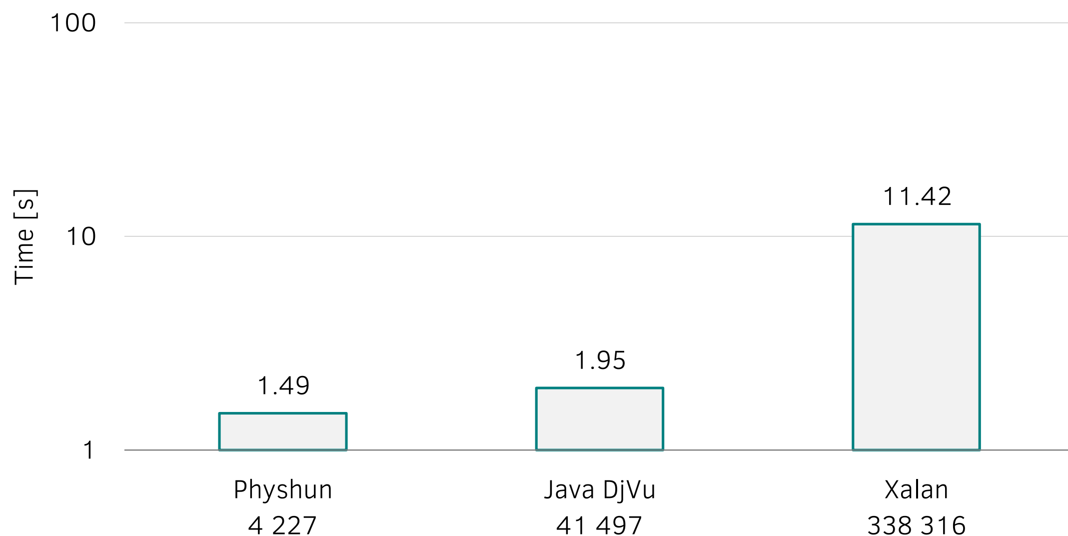

Figure 5.1: *Time required to build the Dependency Graph.*

It is also notable that the SLOC does not correlate with the time required to create the Dependency Graph, it is rather affected by the number of files processed. These projects have 53, 70 and 962 files iterated in the process, respectively.

Transforming the Code into an ASG Figure 5.2 shows the time required to assemble the ASG from parts for the fist time. As on the previous figure, the name of each project and the appropriate number of source lines of code (SLOC) is displayed on the horizontal

axis, while the measured time in seconds is on the vertical axis of this figure. Note that the vertical axis is also logarithmic.

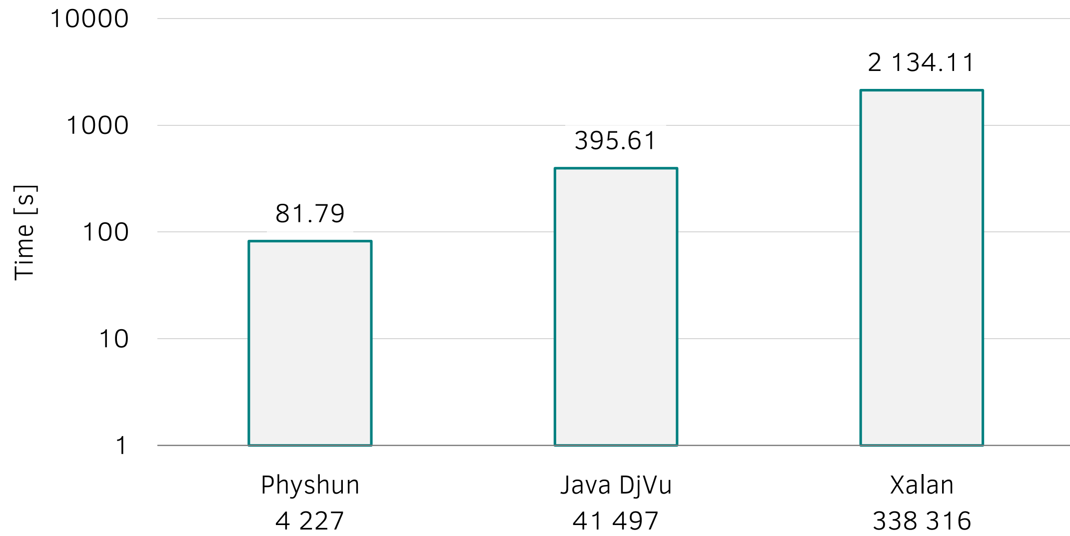

Figure 5.2: *Time required to transform the code into an ASG.*

To create the ASG, JaMoPP needs to parse every line of information, rendering the time required to create the ASG proportional to the number of lines or the byte size of the project. This correlation is visible on Figure 5.2.

Persisting the ASG Figure 5.3 shows the time required to load the ASG into the Database for the first time. As on the previous figure, the name of each project and the appropriate SLOC is displayed on the horizontal axis, while the measured time in seconds is on the vertical axis on this figure. Note that the vertical axis is also logarithmic.

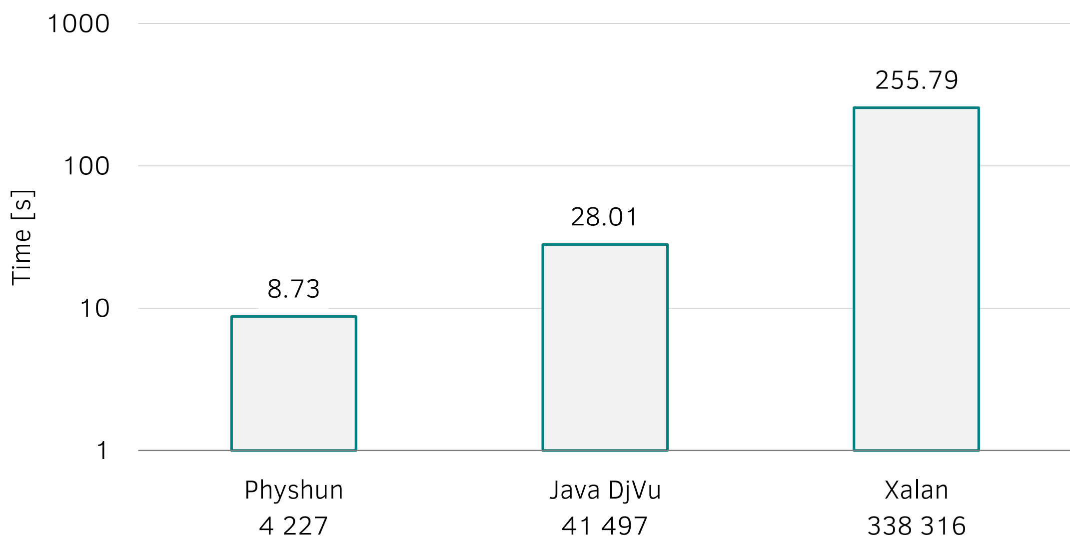

Figure 5.3: *Time required to persist the ASG in the Database.*

Executing Queries Figure 5.4 shows the time required to execute the queries on the Database for the fist time. Note that the unit of the time on the vertical axis of this diagram, contrary to the other diagrams, is millisecond, and the vertical axis is scaled linearly.

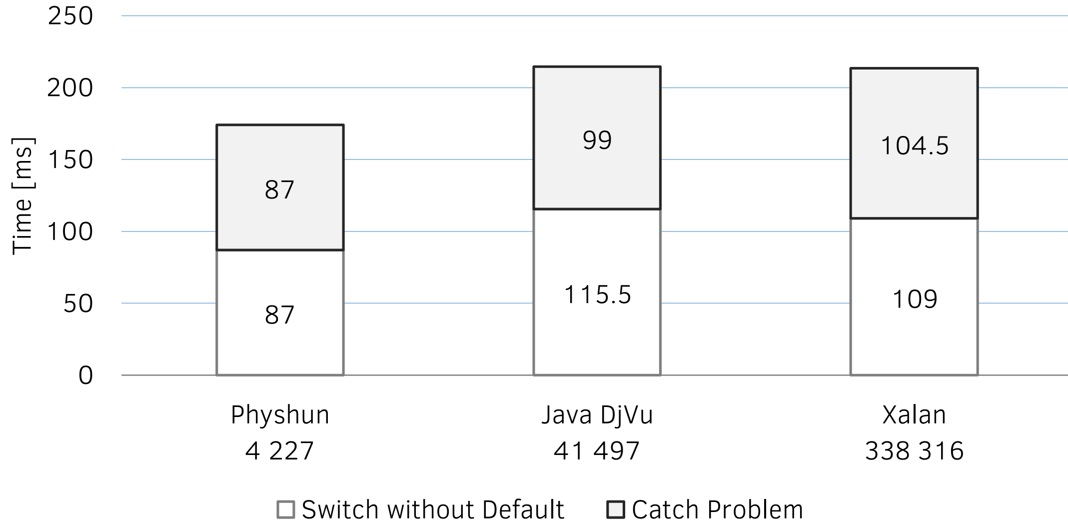

Figure 5.4: *Time required to execute the queries.*

The time required to execute these queries is not only proportional to the size of the projects, but it is also proportional to the number of constraint violations of the projects (detailed in Table 5.2).

Query Results The number of results of the query executions are collected in Table 5.2. To make sure that the results of our approach were correct, we manually confirmed the existence of found violations in the DjVu sources, and to ensure the validity, we also compared our result to the ones published in [13].

	Our Approach			Expert Audit		
	Physhun	**DjVu**	**Xalan**	**Physhun**	**DjVu**	**Xalan**
Switch without Default	0	15	32	0	14	26
Catch Problem	0	0	17	0	0	17

Table 5.2: *Constraint violations found in the source projects.*

The difference in some of these numbers may be attributed to the different version of the source projects (the article did not share the precise version numbers, but for our request, the authors provided approximations of the version numbers used in their benchmarks).

Processing an Incremental Update

In this section, we describe how we make a change in the sources, eliminate the problems violating the well-formedness rules, and notify the tool about the change. We measure the time required to process these changes to assess the viability of our approach.

Changes Made in the Source Code We make a change to a source file of Java DjVu, violating the "Switch without Default" well-formedness rule. The results of the query for this rule (presented in Source 4.3) show violations in the files listed in Source 5.4.

```
djvu/com/lizardtech/djvu/anno/Mapper.java
djvu/com/lizardtech/djvu/anno/Poly.java
djvu/com/lizardtech/djvu/anno/Rect.java
djvu/com/lizardtech/djvu/JB2Codec.java
djvu/com/lizardtech/djvu/text/DjVuText.java
djvu/com/lizardtech/djvubean/anno/AnnoManager.java
djvu/com/lizardtech/djvubean/keys/DjVuKeys.java
djvu/com/lizardtech/djvubean/menu/DjVuMenu.java
djvu/com/lizardtech/djvubean/toolbar/PopupChoice.java
```

Source 5.4: *Compilation units violating the "Switch without Default" well-formedness rule.*

In one of these files (`DjVuText.java`), we add a new default case (see Source 5.5) to the problematic switch block eliminating the violations.

```
      default:    System.err.println("ERR");
```

Source 5.5: *Added default case.*

Incremental Processing After the whole original sources were processed, and the modification has been carried out, we repeat the processing, although with an important difference. This time, our implementation is notified about the modified file and it is only processing the sources affected by the change (as it is specified in Section 4.5).

Figure 5.5 shows the time required to carry out each phase for the first time and doing the same the incremental manner. The name of each phase is displayed on the horizontal axis, while the measured time in seconds is on the vertical axis. Note that the vertical axis is logarithmically scaled.

It is important to note that the incremental approach is creating the ASG one order of magnitude faster than processing the whole project. Figure 5.3 and Figure 5.5 shows different values, because the comparing benchmarks were using an updated storing algorithm, which stored the models in a parallel fashion. Although the model modification is smaller than the model for the whole project, removing parts from the database was

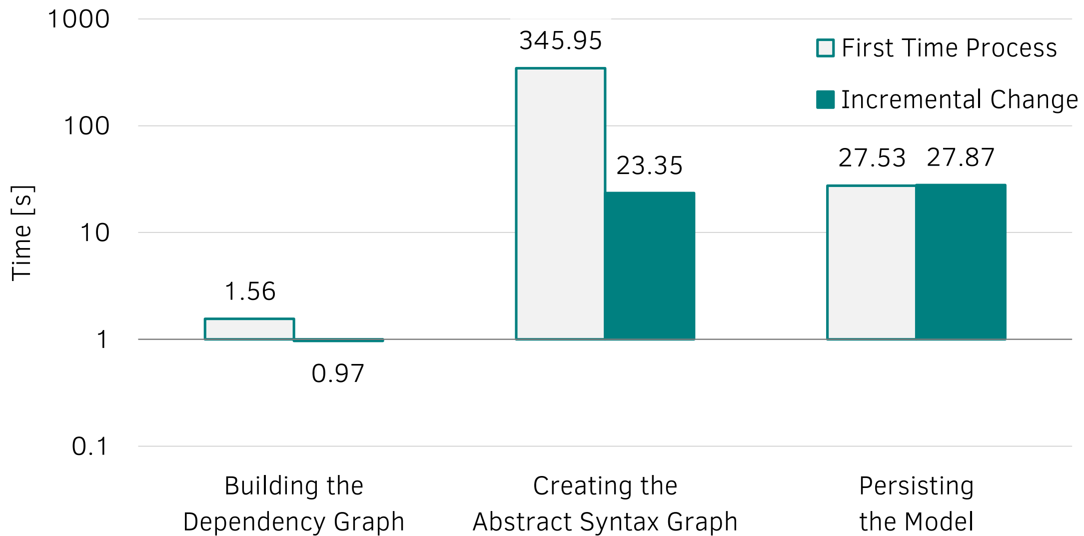

Figure 5.5: *Comparison of times needed to process the whole project or only incremental modification.*

slower than adding new ones. This caused the incremental algorithm to store the model in the nearly same amount of time.

The analysis of the results is presented in Section 5.4.

5.4 Result Analysis

In this section, we discuss the benchmark results and try to predict how this system would cope with inputs of different size.

As expected, the time required to process is proportional to the size of the input sources.

Figure 5.2 and Figure 5.3 shows that the processing and loading times are almost linearly proportional to the number of source lines of code, more specifically to the byte size of the sources. JaMoPP needs to parse every byte of the source and create the model. Next, we load these models together and try to resolve the references between them. Also, we iterate through the contents of the model and serialize it (see Section 3.1.3). All of these are proportional to the size of the source.

It is more difficult to derive conclusions from Figure 5.1. In a naive implementation, the time required to create the Dependency Graph is proportional to the byte size of the sources. However, if the algorithm is optimized correctly, it can be proportional to the number of source files, as only a part of the sources is needed.

Figure 5.5 confirms the validity of our approach, as the incremental iteration only takes a fraction of the time required to process the whole project.

The results show that the time required to create the ASG is two orders of magnitude greater than building the Dependency Graph and one order of magnitude greater than

the time of persisting the model. Figure 5.5 also shows that we successfully reached one of our objectives and decreased the time required to create the ASG to the magnitude of the persisting time.

5.4.1 Threats to Validity

Although we made everything to make our measurements valid, there may be factors beyond our control that affect the measurement results. In this section, we try to list the possible mistakes and also discuss the steps taken to mitigate their effects.

Benchmarking in the Cloud As a multiple access system, our virtual servers in the cloud can be easily affected by virtual neighbor machines using the same resources. The virtual machine manager can also limit the usage of these resources, if our machines disturb the other ones. We can neither control the resources assigned to our machines, nor influence their precise location and connections.

Our mitigation strategy is to run each benchmarks multiple times. To achieve this, we run each benchmark at least ten times and treat their median as the correct value.

Methodological Mistakes It is possible that we made mistakes while implementing our approach. It may not adhere the specification correctly, not perform the transformations correctly or measures correctly. For example, our implementation does not save the attributes of the model, as they are not used in the queries.

To check the validity of the results, we either checked the sources manually, or — as a form of expert audit — compared our results of others.

Measurement Mistakes We may have committed mistakes in the measurements or was led to the wrong conclusions evaluating these mistakes. In this thesis we performed a smaller set of measurements. Measuring with bigger source repositories and more well-formedness rules is subject to future work.

Chapter 6

Comparison of Currently Used Static Analyzers

In this chapter we compare different static analyzer tools introduced in Section 2.3.

6.1 Configurations

This section presents how we configured each of these tools to analyze the Java DjVu project and search for switches without a default case.

6.1.1 FindBugs

In order to analyze a project with FindBugs [35], it has to be compiled first. As a bytecode analyzer, the scope of the bugs FindBugs can find is limited to what it can recover from the bytecode. Having the code compiled, the command in Source 6.1 searches for problems labeled as the "SwitchFallthrough" category.

For example the following patterns are filled under this category according to [74]:

- SF: Switch statement found where one case falls through to the next case (`SF_SWITCH_FALLTHROUGH`)

  This method contains a switch statement where one case branch will fall through to the next case. Usually you need to end this case with a break or return.

- SF: Switch statement found where default case is missing (`SF_SWITCH_NO_DEFAULT`)

  This method contains a switch statement where default case is missing. Usually you need to provide a default case.

  Because the analysis only looks at the generated bytecode, this warning can be incorrect triggered if the default case is at the end of the switch statement and doesn't end with a break statement.

```
$ findbugs -textui -visitors SwitchFallthrough javadjvu/classes/
```

Source 6.1: *Command running FindBugs.*

6.1.2 PMD

As a source code analyzer, PMD [42] does not require the source code to be compiled. PMD executes static analysis based on rules that the user selects during configuration. For example, these could contain an XPath [75] expression (similar to a graph pattern we used in our approach) that specifies a certain problem.

There are many rulesets distributed with PMD, and one of these, the *Design* ruleset contains the `SwitchStmtsShouldHaveDefault` rule [76] that marks the `//SwitchStatement[not(SwitchLabel[@Default='true'])]` XPath expression as problematic. Source 6.3 is an extract from the ruleset only focusing on switches without a default case. The command in Source 6.2 executes PMD with the configuration.

```
$ run.sh pmd -d javadjvu/src/ -f text -R ~/PMD-noswitchdefault.xml
```

Source 6.2: *Command running PMD.*

6.1.3 Checkstyle

Checkstyle [32] is also a source code analyzer, hence it does not require the source code to be compiled. Checkstyle is distributed with many standard checks, containing the `MissingSwitchDefault` check [77] as well. It is implemented using the visitor pattern over the AST of the source code, and using our own visitor classes, Checkstyle can be also extended with additional checks. To use the standard check, the configuration in Source 6.5 has to be provided to the script.

```
$ java -jar checkstyle-6.1-all.jar -c ~/CS-MissingSwitchDefault.xml -r ~/javadjvu/src/
```

Source 6.4: *Command running Checkstyle.*

6.1.4 Columbus

We also benchmark Columbus, the tool measured in the foundation of this thesis, [13]. Columbus itself does not validate the code, the well-formedness validation is carried out by the framework using Columbus.

The following definition is closely based on [78].

```xml
<?xml version="1.0"?>

<ruleset name="Design"
  xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0
                 http://pmd.sourceforge.net/ruleset_2_0_0.xsd">

  <description>
    The Design ruleset contains rules that flag suboptimal code implementations.
    Alternate approachesare suggested.
  </description>

  <rule name="SwitchStmtsShouldHaveDefault"
    language="java"
    since="1.0"
    message="Switch statements should have a default label"
    class="net.sourceforge.pmd.lang.rule.XPathRule"
    externalInfoUrl="http://pmd.sourceforge.net/pmd-5.2.1/
              pmd-java/rules/java/design.html#SwitchStmtsShouldHaveDefault">
    <description>
      All switch statements should include a default option to catch any
      unspecified values.
    </description>
    <priority>3</priority>
      <properties>
         <property name="xpath">
            <value>
               <![CDATA[
                 //SwitchStatement[not(SwitchLabel[@Default='true'])]
               ]]>
            </value>
         </property>
      </properties>
    </rule>
</ruleset>
```

Source 6.3: *PMD configuration.*

```xml
<?xml version="1.0"?>
<!DOCTYPE module PUBLIC
        "-//Puppy Crawl//DTD Check Configuration 1.3//EN"
        "http://www.puppycrawl.com/dtds/configuration_1_3.dtd">

<module name="Checker">
   <module name="TreeWalker">
      <module name="MissingSwitchDefault"/>
   </module>
</module>
```

Source 6.5: *Checkstyle configuration.*

Columbus [78] is a reverse engineering framework, which has been developed in cooperation between the Research Group on Artificial Intelligence in Szeged, the Software Technology Laboratory of the Nokia Research Center and FrontEndART Ltd. Columbus is able to analyze projects and to extract data according to the Columbus Schema.

The main motivation behind developing the Columbus system was to create a tool which implements a general framework for combining a number of reverse engineering tasks,

and to provide a common interface for them. Thus Columbus is a framework which supports project handling, data extraction, data representation, data storage and filtering.

First, the Java source code is parsed into a model with the free version of SourceMeter [79]. SourceMeter is a source code analyzer tool, which can perform deep static analysis of the source code of Java systems. This free version of SourceMeter for Java contains many useful source code metrics, powerful clone detection capabilities, and professionally prioritized and selected PMD coding rule violations. [80]

Second, the framework presented in [13] analyzes the model for problems. We benchmarked different search methods (ASG, EMF, IQP, IQU, described in [13]). The ASG method was the fastest (min. 2 385 ms), and the IQU method was the slowest (max. 38 220 ms).

```
$ ./SourceMeter-Java-5.1.1-x86-linux-free/SourceMeter.sh -DprojectName=djvu
  -DprojectBaseDir=~/djvu-findbugs -DresultsDir=Results -DrunPMD false
$ java -Xms100M -Xmx5000M -XX:MaxPermSize=256m -XX:+UseCompressedOops
  -XX:-UseParallelGC -jar ./framework.jar sourcemeter/asg/djvu.ljsi
  -case ASG nodefaultswitch
```

Source 6.6: *Command running Columbus.*

The following software were used:

- Java SourceMeter version: 5.1
- Ant version: 1.9.3
- Java version: 1.7.0_72

6.2 Measurement Results

Figure 6.1 shows how much time is required to process the source project (Java DjVu). As FindBugs requires the project to be compiled in advance, the compile time is also presented in Figure 6.1. Note that the horizontal axis showing the required time to analyze the project is logarithmic.

6.3 Result Analysis

Note that these benchmarks were carried out for information purposes only, since the tools are not completely comparable.

These results show that, for now, the currently used static analyzers are faster than our tool. The main difference between the approaches is the method of processing the software being verified. FindBugs works on bytecode, Checkstyle and PMD parses the source into several ASTs, while ISAaC builds an ASG.

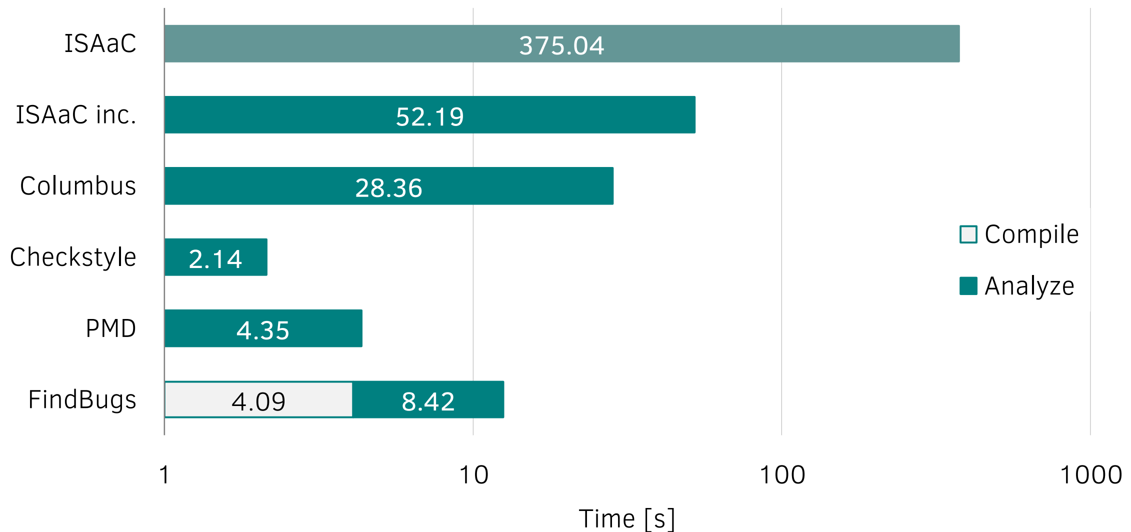

Figure 6.1: *Time required to analyze the original and the modified project with different tools.*

Checkstyle, FindBugs and PMD use in-memory data structures, ISAaC persists every model in a (distributed) database. This may slow down ISAaC on smaller projects, but it is designed to be able to scale with larger, even multiple projects. At some point, these scenarios may reach the limit of the other tools.

With proper optimization, using the incremental approach in every phase and processing in a distributed manner (discussed in Section 3.3), the speed of our tool is in the same order of magnitude as the others.

Examining these tools with a workload of validating more than one well-formedness rule showed that additional rules can slow them down. The runtime characteristics of ISAaC differ from the other tools. First, the slower transforming phase has to be carried out, making the subsequent processes (such as querying) faster. Running the queries in parallel makes ISAaC less sensitive to the number of well-formedness rules.

Also, our approach can find problems across multiple files or even multiple projects. These projects are not required to be processed at the same time. Processing connected projects subsequently can reveal additional related problems throughout the projects.

6.3.1 Threats to Validity

Although every tool was configured to perform only the necessary functions, without in-depth knowledge of their inner workflow and undocumented options, they may not operate in their most effective and efficient manner.

Since the benchmarking environment and methods were the same as in Section 5.2, the threats to validity mentioned in Section 5.4.1 remain as well in this comparison.

Chapter 7

Conclusions

This chapter summarizes the contributions presented in the thesis.

7.1 Summary of Contributions

We presented ISAAC, a modular, extensible, and scalable proof of our novel concept to perform incremental static analysis on source code repositories. Our proposal is based on software code modeling, model transformation and graph pattern matching. The feasibility of the approach was evaluated using manual validation and benchmarking.

7.1.1 Scientific Contributions

We achieved the following scientific contributions:

- We proposed a novel architecture for building an incremental static analyzer using standard formats and interchangeable components.
- We proposed an approach to transform Java source code into RDF graph data model in a distributed manner.
- We provided an algorithm to update the graph data model incrementally.

7.1.2 Practical Accomplishments

We achieved the following practical accomplishments:

- We created a modular, extensible and scalable incremental static analysis framework.
- We created a tool for transforming Java source code into RDF graph data model.
- We designed a benchmark to evaluate the approach.

7.2 Limitation and Future Work

The approach currently has the following limitations:

- The limitations of JaMoPP are projected to the whole approach. Projects using the new language elements of Java 6, 7 or 8 are not yet supported by the current implementation.
- The fully qualified names of the classifiers in the Workspace have to be unique. This is a significant simplification compared to the Java Classpath, but our concept can be extended in a straightforward way to remove this limitation.

The following tasks are subjects to future work:

- The benchmarks were only run on three source projects. More benchmarks are useful to improve the precision of the determination of the complexity of the phases of the analysis.
- Additional, more complex well-formedness rules could be formalized as graph patterns and used in our tests and benchmarks.
- Validation report generation was out of the scope of this thesis. However, in real use cases the developers should be informed about the current results of the analysis.
- Integration with version control and CI systems. After each commit a commit, the incremental analysis should be automatically invoked.
- Incremental query evaluation using INCQUERY-D [2]. Instead of the on-demand query execution, INCQUERY-D can be used to apply automatic and incremental query execution triggered by changes that can be calculated easily using information available within ISAAC. INCQUERY-D works in a distributed manner, which allows it to scale for models with more than 100 million elements.
- Integration to the Eclipse IDE. If the static analysis is integrated to the development environment, the developer receives instant and context-aware results, which greatly increases their productivity.

Acknowledgments

First, I would like to thank Zoltán Ujhelyi for providing the scientific foundations for my research and insight into software model verification and also for helping benchmarking Columbus.

I would like to thank my supervisors Gábor Szárnyas and Dr. István Ráth for their friendly advice and enthusiasm. Also, I wish to express my gratitude to Benedek Izsó for his help with RDF and SPARQL, Áron Tóth for helping my work in the cloud and all other colleagues in the Fault Tolerant Systems Research Group who provided numerous valuable observations and suggestions.

Last but not least, I am deeply grateful to my family and friends for their continuous support.

List of Figures

1.1 Continuous Integration workflow extended with Static Analysis. 1

2.1 Big Data database architecture. 5
2.2 An illustrative core part of Ecore, the metamodeling language of EMF. . . 9
2.3 Different graph data models (based on [20]). 11
2.4 W3C RDF example depicted as a graph. 11
2.5 W3C RDF example as an EMF class. 13

3.1 Overview of the system. 18
3.2 Source code change propagating through the system. 24
3.3 Workflow of the distributed processing. 25

4.1 Dependency Graph of the Workspace. 27
4.2 Part of the JaMoPP Java metamodel showing the class hierarchy of the Switch and the various SwitchCase classes. 29
4.3 Part of the JaMoPP Java metamodel showing the class hierarchy of the Try-Block, CatchBlock and InstanceOfExpression classes (with related classes). 29
4.4 Part of the instance model visualized with the Sample Reflective Ecore Model Editor. 30
4.5 Part of a JaMoPP instance model in RDF format showing a Switch with three SwitchCases, where one of them is a DefaultSwitchCase. 30
4.6 Graph pattern representation of the "Switch without Default" problem. . 32
4.7 Graph pattern representation of the "Catch Problem" problem. 33

5.1 Time required to build the Dependency Graph. 40
5.2 Time required to transform the code into an ASG. 41
5.3 Time required to persist the ASG in the Database. 41
5.4 Time required to execute the queries. 42

5.5 Comparison of times needed to process the whole project or only incremental modification. 44

6.1 Time required to analyze the original and the modified project with different tools. 50

Bibliography

[1] Continuous Integration. http://www.martinfowler.com/articles/continuousIntegration.html. Accessed: 2014-10-26.

[2] Gábor Szárnyas. Superscalable modeling. Master's thesis, Budapest University of Technology and Economics, Budapest, 12/2013 2013.

[3] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[4] Adam Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):36–44, August 2009.

[5] Sherif Sakr. Supply cloud-level data scalability with NoSQL databases. http://www.ibm.com/developerworks/cloud/library/cl-nosqldatabase/index.html, March 2013.

[6] NoSQL Databases. http://nosql-database.org/, May 2013.

[7] 4store. http://4store.org/, October 2013.

[8] Avahi. http://www.avahi.org/, October 2013.

[9] Redland RDF Libraries. Raptor RDF Syntax Library. http://librdf.org/raptor/, October 2013.

[10] Redland RDF Libraries. Rasqal RDF Query Library. http://librdf.org/rasqal/, October 2013.

[11] Object Management Group. *Object Constraint Language Specification (Version 2.3.1)*, 2012. http://www.omg.org/spec/OCL/2.3.1/.

[12] EMF-IncQuery. http://incquery.net/incquery, May 2013.

[13] Zoltán Ujhelyi, Ákos Horváth, Dániel Varró, Norbert István Csiszár, Gábor Szőke, László Vidács, and Rudolf Ferenc. Anti-pattern detection with model queries: A comparison of approaches. In *IEEE CSMR-WCRE 2014 Software Evolution Week*. IEEE, IEEE, 02/2014 2014. IEEE Best Paper Award, Acceptance rate: 31%.

[14] SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query/. Accessed: 2014-10-24.

[15] Steven Harris and Andy Seaborne. SPARQL 1.1 query language. W3C recommendation, W3C, March 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[16] The Eclipse Project. Eclipse Modeling Framework. http://www.eclipse.org/emf, October 2012.

[17] EMF documentation. Package org.eclipse.emf.ecore. http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html, October 2013.

[18] JaMoPP. http://www.jamopp.org/index.php/JaMoPP. Accessed: 2014-10-18.

[19] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. *Jamopp: The java model parser and printer*. Techn. Univ., Fakultät Informatik, 2009.

[20] Marko A. Rodriguez and Peter Neubauer. Constructions from dots and lines. *CoRR*, abs/1006.2361, 2010.

[21] RDF Primer — Turtle version . http://www.w3.org/2007/02/turtle/primer/. Accessed: 2014-10-18.

[22] Gavin Carothers and Eric Prud'hommeaux. RDF 1.1 turtle. W3C recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-turtle-20140225/.

[23] Andy Seaborne and Gavin Carothers. RDF 1.1 n-triples. W3C recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-n-triples-20140225/.

[24] Gavin Carothers. RDF 1.1 n-quads. W3C recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-n-quads-20140225/.

[25] Markus Lanthaler, Manu Sporny, and Gregg Kellogg. JSON-ld 1.0. W3C recommendation, W3C, January 2014. http://www.w3.org/TR/2014/REC-json-ld-20140116/.

[26] Dave Beckett. RDF/xml syntax specification (revised). W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

[27] Neo Technology. Neo4j. http://neo4j.org/, 2013.

[28] Titan. https://github.com/thinkaurelius/titan, May 2013.

[29] Titan – Big Data with Cassandra. http://www.slideshare.net/knowfrominfo/titan-big-graph-data-with-cassandra, August 2012.

[30] Apache Cassandra. http://cassandra.apache.org/, May 2013.

[31] google/cayley. https://github.com/google/cayley. Accessed: 2014-10-25.

[32] checkstyle – Checkstyle 6.1.1. http://checkstyle.sourceforge.net/. Accessed: 2014-12-07.

[33] checkstyle/checkstyle. https://github.com/checkstyle/checkstyle. Accessed: 2014-12-07.

[34] checkstyle – Configuration. http://checkstyle.sourceforge.net/config.html. Accessed: 2014-12-07.

[35] FindBugs™ - Find Bugs in Java Programs. http://findbugs.sourceforge.net/. Accessed: 2014-10-25.

[36] FindBugs 2™ - Find Bugs in Java Programs. http://findbugs.sourceforge.net/findbugs2.html. Accessed: 2014-10-25.

[37] Eclipse Project. http://www.eclipse.org, October 2012.

[38] Welcome to NetBeans. https://netbeans.org/. Accessed: 2014-10-25.

[39] IntelliJ IDEA — The Best Java and Polyglot IDE. https://www.jetbrains.com/idea/. Accessed: 2014-10-25.

[40] Apache Maven. http://maven.apache.org, October 2012.

[41] Welcome to Jenkins CI! | Jenkins CI. http://jenkins-ci.org/. Accessed: 2014-10-25.

[42] PMD. http://pmd.sourceforge.net/. Accessed: 2014-12-07.

[43] PMD – Welcome to PMD. http://pmd.sourceforge.net/pmd-5.2.1/. Accessed: 2014-12-07.

[44] SonarQube™. http://www.sonarqube.org/. Accessed: 2014-10-25.

[45] Javier Pérez, Yania Crespo, Berthold Hoffmann, and Tom Mens. A case study to evaluate the suitability of graph transformation tools for program refactoring. *International Journal on Software Tools for Technology Transfer*, 12(3-4):183–199, 2010.

[46] Tassilo Horn. Program understanding: A reengineering case for the transformation tool contest. In Pieter Van Gorp, Steffen Mazanek, and Louis Rose, editors, Proceedings Fifth *Transformation Tool Contest,* Zürich, Switzerland, June 29-30 2011, volume 74 of *Electronic Proceedings in Theoretical Computer Science*, pages 17–21. Open Publishing Association, 2011.

[47] Javier Espinazo Pagán and Jesús García Molina. Querying large models efficiently. *Information and Software Technology*, 56(6):586 – 622, 2014.

[48] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.

[49] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 338–348, New York, NY, USA, 2002. ACM.

[50] Mirko Seifert and Roland Samlaus. Static source code analysis using OCL. *Electronic Communications of the EASST*, 15(0), January 2008.

[51] Thorsten Arendt and Gabriele Taentzer. Integration of smells and refactorings within the Eclipse Modeling Framework. In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pages 8–15. ACM, 2012.

[52] PMD. *PMD checker*, 2014. http://pmd.sourceforge.net/.

[53] FrontEndART Software Ltd. *QualityGate CodeAnalyzer,* 2014. http://www.frontendart.com/.

[54] Daniel Speicher, Malte Appeltauer, and Günter Kniesel. Code analyses for refactoring by source code patterns and logical queries. In *1st Workshop on Refactoring Tools, WRT 2007,* pages 17–20, 2007.

[55] Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane 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*, PPPJ '11, pages 71–80. ACM, 2011.

[56] Elnar Hajiyev, Mathieu Verbaere, and Oege Moor. codequest: Scalable source code queries with Datalog. In Dave Thomas, editor, *ECOOP 2006 – Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin Heidelberg, 2006.

[57] 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*, 36(1):20–36, 2010.

[58] Git. `http://git-scm.com/`. Accessed: 2014-10-09.

[59] Apache Subversion. `https://subversion.apache.org/`. Accessed: 2014-10-09.

[60] Mercurial SCM. `http://mercurial.selenic.com/`. Accessed: 2014-10-09.

[61] Concurrent Versions System - Summary [Savannah]. `http://savannah.nongnu.org/projects/cvs`. Accessed: 2014-10-09.

[62] MongoDB. `http://www.mongodb.org/`, October 2013.

[63] Code Conventions for the Java Programming Language: Contents. `http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html`. Accessed: 2014-10-26.

[64] Google Java Style. `http://google-styleguide.googlecode.com/svn/trunk/javaguide.html`. Accessed: 2014-10-26.

[65] Apache Jena - Typed literals how-to. `https://jena.apache.org/documentation/notes/typed-literals.html`. Accessed: 2014-10-25.

[66] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. Incremental evaluation of model queries over emf models. In *Model Driven Engineering Languages and Systems, 13th International Conference, MODELS'10*. Springer, Springer, 10/2010 2010. Acceptance rate: 21%.

[67] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. Incquery-d: A distributed incremental model query framework in the cloud. In *ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems, MODELS 2014*, Valencia, Spain, 2014. Springer, Springer. Acceptance rate: 26%.

[68] Benedek Izsó, Gábor Szárnyas, István Ráth, and Dániel Varró. IncQuery-D: incremental graph search in the cloud. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 4:1–4:4, New York, NY, USA, 2013. ACM.

[69] The switch Statement (The Java™ Tutorials > Learning the Java Language > Language Basics). `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html`. Accessed: 2014-10-13.

[70] Physhun - J2EE Finite State Machine Framework. `http://physhun.sourceforge.net/`. Accessed: 2014-10-16.

[71] Java DjVu Viewer | SourceForge.net. http://sourceforge.net/projects/javadjvu/. Accessed: 2014-10-16.

[72] Xalan-Java Version 2.7.1. http://xml.apache.org/xalan-j/. Accessed: 2014-10-16.

[73] SSD Cloud Server, VPS Server, Simple Cloud Hosting | DigitalOcean. https://www.digitalocean.com/. Accessed: 2014-10-22.

[74] FindBugs Bug Descriptions. http://findbugs.sourceforge.net/bugDescriptions.html#SF_SWITCH_FALLTHROUGH. Accessed: 2014-11-20.

[75] Michael Dyck and Jonathan Robie. XML path language (XPath) 3.1. W3C working draft, W3C, April 2014. http://www.w3.org/TR/2014/WD-xpath-31-20140424/.

[76] PMD Java – Ruleset: Design. http://pmd.sourceforge.net/pmd-5.2.1/pmd-java/rules/java/design.html#SwitchStmtsShouldHaveDefault. Accessed: 2014-11-20.

[77] checkstyle – Coding. http://checkstyle.sourceforge.net/config_coding.html#MissingSwitchDefault. Accessed: 2014-11-20.

[78] Rudolf Ferenc, Ferenc Magyar, Árpád Beszédes, Ákos Kiss, and Mikko Tarkiainen. Columbus–tool for reverse engineering large object oriented software systems. 2001.

[79] SourceMeter - Free-to-use, Advanced Source Code Analysis Suite. https://www.sourcemeter.com/. Accessed: 2014-12-07.

[80] SonarQube - Open Wonderland. https://sonarqube.sourcemeter.com/dashboard/index/701903. Accessed: 2014-12-07.

Appendix

H.1 JaMoPP metamodel

annotations

Concrete classes for the Java annotations and abstract classes for the items that could be annotated.

arrays

Classes describing the dimensions and types of arrays, or the instantiation of the arrays.

classifiers

`Class, AnonymousClass, Interface, Enumeration, Annotation`

commons

Core package of the metamodel, containing abstract classes.

`Commentable, NamedElement, NamespaceAwareElement`

containers

`CompilationUnit, Package`

expressions

The expressions package contains the classes describing the assignments, castings, and, or, equality and other expressions, and their abstract child nodes to be superclasses for other expressions.

`ExpressionList, EqualityExpression, EqualityExpressionChild, NestedExpression`

generics

The generics package contains classes describing the constraints used in generic classifiers, methods e.g. `ExtendsTypeArgument`.

imports

The imports package contains classes describing the various import cases e.g.

`ClassifierImport, PackageImport, StaticClassifierImport, StaticMemberImport` .

instantiations

The instantiations package contains describing different types of constructor calls.

literals

The literals package contains classes describing literal values for the primitive data types e.g. `BooleanLiteral`, `CharacterLiteral`. It also contains many representations for the

`IntegerLiteral, LongLiteral, DecimalLiteral, FloatLiteral,`

also

`Super, NullLiteral, This .`

members

`Method, Field, Constructor`

modifiers

`Abstract, Final, Native, Public, Protected, Private, Static, Synchronized, Transient, Volatile`

operators

The operators package contains classes for the Java operators.

`Equal, NotEqual, GreaterThan, RightShift, MinusMinus`

parameters

Classes representing the method parameters `OrdinaryParameter`, `VariableLengthParameter`, and the `Parametrizable` abstract class for `Methods` and alike.

references

The references package contains classes representing references to other items.

`IdentifierReference, MethodCall, StringReference, ReflectiveClassReference, PrimitiveTypeReference, SelfReference`

statements

`Switch, TryBlock, CatchBlock, WhileLoop, ForLoop, ExpressionStatement, Return`

types

The types package contains classes for classifier or type references, primitive types.

`Boolean, Byte, Char, Double, Float, Int, Long, Short, Void`

variables

`Variable, LocalVariable, AdditionalLocalVariable`

(Fields are found in the members package.)