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

Soma Lucz

Static analysis algorithms for JavaScript

Bachelor's Thesis

Supervisors
Dávid Honfi
Gábor Szárnyas

Budapest, 2017

Contents

Contents iii

Kivonat vii

Abstract viii

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Problem Statement and Requirements . . . 2
- 1.3 Objectives and Contributions . . . 3
- 1.4 Structure of the Thesis . . . 3

2 Preliminaries 5
- 2.1 Static Analysis . . . 5
  - 2.1.1 Introduction . . . 5
  - 2.1.2 Source Code Transformation . . . 6
  - 2.1.3 Use Cases and Limitations . . . 7
- 2.2 JavaScript . . . 8
  - 2.2.1 Brief History of JavaScript . . . 8
  - 2.2.2 The ECMAScript as a Standard and as a Language . . . 8
  - 2.2.3 The Process of Transpiling . . . 9
  - 2.2.4 Looking into the Goals of JavaScript Static Analysis . . . 10
- 2.3 Graph Databases . . . 11
  - 2.3.1 The Property Graph Data Model . . . 11
  - 2.3.2 Neo4j . . . 13
  - 2.3.3 Cypher . . . 13
- 2.4 Running Example . . . 14

3 Related Work 15
- 3.1 Static Analysis Tools for JavaScript . . . 15
  - 3.1.1 TAJS (Type Analysis for JavaScript) . . . 15

3.1.2 Flow . . . 16
3.1.3 Tern . . . 16
3.1.4 SonarQube . . . 17
3.1.5 Shift . . . 17
3.1.6 Esprima . . . 18
3.1.7 Comparison of the Featured Tools . . . 18
3.2 Static Analysis Tools for Java . . . 19
3.2.1 FindBugs . . . 19
3.2.2 PMD . . . 19
3.2.3 jQAssistant . . . 19
3.3 Static Analysis Tools for C and C++ . . . 20
3.3.1 Clang . . . 20
3.3.2 PolySpace . . . 21
3.3.3 Coverity . . . 21
3.4 Most Used Error-Checking Constraints . . . 22

4 Overview of the Approach 23
4.1 Rearchitecturing the Codemodel-Rifle Framework . . . 23
4.1.1 Open-Sourcing and Licensing Issues . . . 25
4.1.2 Decomposing the Architecture . . . 25
4.1.3 Optimising for Testing Purposes . . . 27
4.1.4 Solutions to Speed-Related Issues . . . 27
4.1.5 Other Performances . . . 30
4.1.6 Summary of Refactoring . . . 31
4.2 In Development: Steps of Building New Analyses . . . 32
4.2.1 Visualising the Defect with Codemodel-Visualisation . . . 32
4.2.2 Describing the Defect Pattern . . . 32
4.2.3 Implementing the Analysis . . . 32
4.3 In Production: Steps of Operating Live . . . 33
4.3.1 Import: Synchronising the Repository into the Framework . . . 33
4.3.2 Interconnect: Connecting the Related ECMAScript Modules . . . 34
4.3.3 Analyse: Performing Analyses . . . 34

5 Elaboration of the Workflow 35
5.1 Interconnecting Related ECMAScript Modules . . . 35
5.1.1 The ECMAScript Module System . . . 36
5.1.2 Export Syntaxes and Cases . . . 36
5.1.3 Import Syntaxes and Cases . . . 38
5.1.4 Number of Export-Import Combinations . . . 38
5.1.5 Compatibility of the Export-Import Cases . . . 39
5.1.6 Unsupported Cases . . . 39

5.1.7 Pattern Generalisation Techniques . . . 41
5.1.8 Implementing the Interconnection Algorithms . . . 44
5.2 Simple Analyses by Pattern Matching . . . 48
5.2.1 Uninitialised Variables . . . 48
5.2.2 Globally Unused Exports . . . 49
5.2.3 Division By Zero (restricted) . . . 50
5.2.4 Misuse of Negative Integers as Function Arguments (restricted) . . . 51
5.3 Complex Analyses with the Qualifier System . . . 52
5.3.1 Transitive Defects . . . 52
5.3.2 Introduction: The Qualifier System . . . 54
5.3.3 The Running Example's Division By Zero (transitive) . . . 55
5.3.4 Misuse of Negative Integers as Function Arguments (transitive) . . . 57
5.3.5 Unreachable Code Caused by Exception (transitive) . . . 57
5.4 Limitations of the Analyses . . . 58

6 Evaluation of Performance **59**
6.1 Evaluation Environment . . . 59
6.1.1 Computer Configuration . . . 59
6.1.2 Software Configuration . . . 59
6.2 Measurement Goals and Methods . . . 60
6.2.1 Selection Criteria of the Analysed Source Code Repositories . . . 60
6.2.2 Key Performance Indices . . . 60
6.2.3 Process of Measurement . . . 60
6.3 Measurement Results . . . 61
6.3.1 Synchronisation . . . 61
6.3.2 Interconnection . . . 63
6.3.3 The Qualifier System . . . 64
6.3.4 Analysis . . . 65
6.3.5 Total Duration of the Analysis Process . . . 66
6.4 Defects Found by the Framework . . . 68
6.5 Threats to Validity . . . 68

7 Conclusion and Future Work **69**
7.1 Summary of Contributions . . . 69
7.1.1 Scientific Contributions . . . 69
7.1.2 Engineering Contributions . . . 70
7.2 Future Work . . . 70

Acknowledgements **71**

References **73**

Appendix **79**
A Cypher Queries for Interconnecting the ASGs of Related Modules 79
A.1 exportAlias–importAlias 79
A.2 exportAlias–importDefault 80
A.3 exportAlias–importName 81
A.4 exportDeclaration–importAlias 82
A.5 exportDeclaration–importName 83
A.6 exportDefaultDeclaration–importAlias 84
A.7 exportDefaultDeclaration–importDefault 85
A.8 exportDefaultDeclaration–importName 86
A.9 exportDefaultName–importAlias 87
A.10 exportDefaultName–importDefault 88
A.11 exportDefaultName–importName 89
A.12 exportName–importAlias 90
A.13 exportName–importName 91
B Cypher Queries of the Analyses 92
B.1 nonInitialisedVariable 92
B.2 unusedExport — exportName-exportAlias 93
B.3 unusedExport — exportDefault-exportDefaultName 94
B.4 unusedExport — exportDeclaration 95
B.5 divisionByZero-literal — restricted 96
B.6 squareRootNegativeArgument-literal — restricted 97
B.7 divisionByZero-variable — transitive 98
B.8 squareRootNegativeArgument-variable — transitive 99
B.9 unreachableCode-exception — transitive 100
C Cypher Queries of the Qualifier System 101
C.1 Initialising the Qualifier System 101
C.2 Tagging literals with EqualsZero 101
C.3 Tagging throw statements with ExceptionThrown 101
D Cypher Queries for Qualifier Propagation 102
D.1 Propagation along function calls 102
D.2 Propagation along function declarations 102
D.3 Propagation along function return statements 102
D.4 Propagation along throw statements in functions 102
D.5 Propagation along variable declarations 103
D.6 Propagation along variable declaration statements 103
D.7 Propagation along variable initialisations 103
D.8 Propagation along variable references 103
E Selected Open-Source Repositories for the Evaluation 104
E.1 Repository and Graph Data 104
E.2 Measurement Results 106

Kivonat Összetett szoftverek fejlesztése során a kódbázis növekedésével általában a kódban megjelenő fejlesztői hibák száma is nő. Ezen hibák fokozott kockázatot jelenthetnek, hiszen az esetlegesen helytelen, nemkívánatos működés mellett jelentős biztonsági réseket eredményezhetnek. Kiaknázásuk által rosszindulatú támadók a szoftvert számukra kedvező, az eredetileg tervezettől eltérő módon futtathatják.

A statikus forráskódanalízis egy, az iparban gyakran használt, általánosan elfogadott szoftvertesztelési megközelítés. Célja, hogy minél több szoftverfejlesztői hibát minél előbb, még a program fejlesztési szakaszában – a kód lefordítása és lefuttatása nélkül – tárjon fel, csökkentve ezzel a működés közben felmerülő programhibák számát, és így a telepítés utáni hibajavítással járó pluszköltségeket. Felhasználási lehetőségei közé tartozik a csoportos, vállalati kódolási szabályoknak, stílusoknak való megfelelés ellenőrzése, illetve egyre több statikus analízis eszköz nyújt támogatást egyre komolyabb logikai hibák fordítási vagy akár kódírási idejű feltárásához is.

Napjaink folytonos integrációs infrastruktúrájába illesztve a statikus analízis hatékony eszköz lehet a fejlesztői hibák feltárásában, és ezáltal az állandó kódminőség biztosításában. Nagymértékű népszerűsége ellenére a JavaScript nyelvhez – annak dinamikus és gyenge típusosságából eredő sajátosságok következményeként – kevés statikus analízis-eszköztár létezik, és a rendelkezésre álló eszközök sem nyújtanak teljeskörű megoldást nagyméretű, vállalati szintű JavaScript forráskódtárak összefüggő elemzéséhez. Gyakran felmerülő probléma emellett az analitikus komplexitással általában fordítottan arányos sebesség: sem folytonos integrációs infrastruktúrába, sem fejlesztőkörnyezetbe nem illeszthető olyan eszköz, amely miatt a fordítási idő akár órákkal növekszik.

Dolgozatomban egy már létező, a fenti követelményeknek nagy részben eleget tevő statikus kódanalízis-keretrendszer bővítését tervezem meg, fejlesztem ki és értékelem. A bővítés során egyrészt új – logikai és formai – JavaScript-alapú statikus analízis-kikötéseket implementálok a rendszerhez. Másrészt lehetővé teszem, hogy a rendszer több összefüggő JavaScript-modulon (forrásfájlon) átívelő, globális analízis-kikötések kiértékelésére is képes legyen. Ezt kihasználva újabb kikötéseket implementálok, immáron több JavaScript-modult összefüggően elemző analízisekhez.

Abstract In complex software development, the number of developer errors usually increases with the growth of the code base. These errors can be sufficiently dangerous: besides causing improper or undesirable operation, they can lead to serious security vulnerabilities. By exploiting them, malicious attackers can take control over the software in some ways to run it according to their goals, or at least differently than originally intended.

Static source code analysis is a widely used, generally approved software testing approach. Its goal is to discover as many human errors as possible, as early as possible — meaning during development, without compiling and running the code —, in order to reduce the number of software failures in production, and to minimize the extra costs of fixing bugs after deployment. Possible applications of static analysis include verifying whether the code complies with enterprise coding standards and styles, but more and more analysis toolsets provide ways to detect more complex logical errors during compilation time, or even development time.

In our days, static analysis toolsets integrated into Continous Integration (CI) workflows can be an efficient way to detect developer errors at commit- and build-time, and thus to provide constant code quality. Despite its widespread popularity, the JavaScript language does not have extensive static analysis tooling — a possible cause can be the language's dynamic and weak typing —, and the available tools do not provide a full-scale solution to coherently analyse large, enterprise-grade code repositories either. Moreover, increased analysis complexity generally means significant reduction of speed, and of course a tool can not be integrated neither into a CI workflow, nor into a development environment, if it increases the build time with even several hours.

In this thesis I design, implement and evaluate the extension of an existing static code analysis framework complying with most of the above detailed requirements. With the extension on the one hand, I implement new JavaScript static analysis constraints — logical and formal — for the framework. On the other hand, I extend the system with the capability of analysing more than one JavaScript source code files coherently, thus I provide a way to evaluate global analysis queries over more than one JavaScript modules related to each other. Then I implement more analysis constraints, but now for coherently analysing more than one, related JavaScript modules.

Chapter 1

Introduction

1.1 Context

Software development is a highly complex process involving many people, tools and methods. As a source code repository grows, code quality becomes an important aspect of the development procedure: the software gets more and more complex, the number of human errors in the implementation gets higher and higher. It is important to find and fix these errors as soon as possible: software defects found after deployment are 15 times more costly than if they were found during implementation [1]. According to NIST, software bugs cost approximately $59.5 billion for the US government annually [2].

Today's developer tools in commercial and open-source projects generally include *version control systems (VCS)* and *continuous integration (CI)* toolsets [3, 4]. Integrating code quality assurance tools into the CI platform, or into the developers' *integrated development environment (IDE)*, seems to be the practical choice for enforcing project-/company-wide coding style compliance, and analysing the code deeper whether it contains defects.

A CI platform can be configured to scan and analyse the source code with external tools when the developer commits their code to the central code repository. A common workflow is the following:

1. the developer edits the code,
2. the developer commits and pushes the modified code into the central repository,
3. the VCS triggers a hook to inform the subscribers of the hook (including the CI platform) that new code has been committed,
4. the CI platform analyses the source code with the static analysis platforms integrated and configured by the user, and creates reports about the analyses,
5. the CI platform builds the code with its dependencies and passes on the built artifact for further testing, and finally for deployment.

The reports created by the integrated static analysis tools give the developers insights about code quality, and help them discover faults in the software before they reach the testing or production stage.

This thesis focuses on the static analysis of JavaScript projects. As JavaScript is an interpreted language, it is generally considered not to require to be built before executing in browsers and external runtimes. Nevertheless, it is sensible to involve CI into JavaScript-projects for code quality and testing purposes, for a so-called transpiling step[1], and for automated deployment.

1.2 Problem Statement and Requirements

Despite being one of the most commonly used programming language in the world [5], JavaScript does not have extensive static analysis tooling. There are static analysers for the language, but either their capabilities are very limited, or they require special preparations, like code annotations or special syntax flavours to work appropriately. There are only a couple of analysers, which analyse more than one JavaScript modules coherently.

One solution is to modify already existing JavaScript projects according to the needs of the analysis toolsets. If developers annotate their objects and/or use specially extended, non-standard flavours of the JavaScript language, they can get benefits like type inference. For already existing projects being developed for a longer amount of time, this solution is far from ideal. Since more complex projects can excede 1 million lines of code in size, utilising annotations or special, non-standard syntax flavours would involve huge refactoring costs.

Another possible solution would be a general JavaScript analysis framework with a static type system and other analytical benefits based on nothing else, but the current JavaScript standard[2] [6]. This solution would require:

- a JavaScript parser complying with the latest ECMAScript standards to parse the source files into a data structure that can be processed and manipulated effectively,
- a database technology for storing the data structure,
- an interface to manipulate the data structure for the purposes of the analyses,
- and — necessarily — the analyses' algorithmic descriptions themselves, which reveal the potential defects' location in the inspected software.

The solution can introduce other usability requirements as well, like incremental processing of source code repositories for speed, multi-version data model in accordance with VCSs so the analysis framework can be used by many developers simultaneously, or even a centralised interface for collecting, storing, and presenting previous analysis results for

[1] The procedure of transpiling will be detailed in Chapter 2.

[2] According to the standard [6], the official name of the JavaScript language is ECMAScript.

fine-grained, per-person or per-workgroup efficiency analytics. This thesis focuses on the source code analyses themselves.

1.3 Objectives and Contributions

Dániel Stein created a graph-based static analysis framework for JavaScript (ECMAScript), called *Codemodel-Rifle* [7]. The project's source code is available on GitHub [8]. The framework stores the analysed source code repository's each parsed file as a distinct property graph, called an *Abstract Semantic Graph (ASG)*, and gives us an interface to run analyses via graph queries.

My main goal is to extend *Codemodel-Rifle* with several static analysis constraints. This involves providing ways for evaluating analysis queries over more than one JavaScript modules related to each other.

The framework and the analyses are tested with open-source projects and a closed-source, security-oriented industrial product from Tresorit [9], a cloud security company located in Budapest, Hungary.

1.4 Structure of the Thesis

The thesis is structured as follows. *Chapter 2* presents the concept of static analysis, shortly summarises JavaScript and its static analysis approaches to be detailed in Chapter 3, and gives insights to the background technologies of the previously mentioned *Codemodel-Rifle* framework. It also provides an example which will accompany the reader throughout the thesis. *Chapter 3* specifies the currently known approaches and related work. *Chapter 4* gives an overview of my approach of JavaScript static analysis using the *Codemodel-Rifle* framework, and describes all performance- and modularity-related *architectural* changes of the framework. *Chapter 5* encompasses all *semantic* changes of the framework: it details the implementation of the analysis algorithms and the additional proceedings about coherently analysing more than one JavaScript modules related to each other. *Chapter 6* demonstrates and evaluates the implemented analyses. *Chapter 7* concludes the thesis and presents possible future research directions.

Chapter 2

Preliminaries

This chapter presents the concept of static analysis, shortly summarises the JavaScript language and its static analysis approaches, and gives insights to the background technologies of the previously mentioned *Codemodel-Rifle* framework.

2.1 Static Analysis

2.1.1 Introduction

Static source code analysis is a software testing approach performed without compiling and executing the program itself. Usually the source code of the analysed software first gets transformed to a mathematical data structure — which is mostly a tree or a different form of graph —, then the data structure is inspected by automated tools with the goal of finding software defects. As static analysis is performed without actually executing the program, software can be analysed in as early as its source code state, before getting to testing or deployment.

Techniques for static analysis exist for almost 50 years [10]. A 1995 research paper concludes, that “Static analysis is effective and complementary to dynamic testing. Hence its use is to be recommended in the context of the majority of critical software.” [11] In 2017, open- and closed-source static analysis tooling is quite extensive, and publicly available not only for the academia and the commercial industry, but for open-source projects as well [12].

The sophistication and the generated reports’ quality of static analysis tools vary: some report potential fault locations, others use mathematical tools to verify properties of a software and its specification. Besides general code quality-related applications, static analysis acquires a growing market share also in safety- and mission-critical systems for exploring defects [13].

2.1.2 Source Code Transformation

Software source code is a text, usually consisting of human-readable characters. Characters formulate sequences of instructions by the specified grammar of the programming language. To be executed on a computer, most programming languages need to be *compiled* by a *compiler* first, meaning the source code has to be transformed into *binary code* or *bytecode* to be executed. Other languages, called interpreted languages, do not need to be compiled, they are interpreted and executed at runtime.[1]

Compiled languages' are always analysed, at least at compilation time by the compiler. If the software contains severe errors (like type association errors in strongly typed languages), the compiler will abort its operation, thus the software can not be run, since it has not been compiled. Considering interpreted languages do not need to be compiled, they are not analysed by a compiler before running, and — generally — not analysed at all. Interpreted languages' static analysis is therefore beneficial to compensate the lack of a compiler-like component in the software processing chain.

More than one static analysis methods can be run simultaneously on a project. As static analysis inspects the source code without modifying the original, the operations of several such tools are independent from each other. Therefore at compiled languages, added static analyses can only compliment the compiler's necessary analysis.

Usually three abstract data structures are used to represent software source code in a mathematically defined form.

Abstract Syntax Tree (AST)

If the compiler or an analysis tool processes the source code and its parser transforms the source code into an abstract data structure, it usually creates an *Abstract Syntax Tree*. It is the tree-representation of the code, meaning every node in the tree is a semantic element of the source code. The *source code to AST transformation* is vica versa unambiguous, meaning the two structures are identical to each other regarding the program logic. It is abstract in the sense of syntax: not all elements of the syntax is preserved in its AST, meaning without the language grammar, transformation would not be possible.

Abstract Semantic Graph (ASG)

A more abstract representation of the source code (or an AST) can be an *Abstract Semantic Graph*. Derived semantic information added to the AST can result in a graph which provides more insights into the structure of the program: it can reveal data about variable and function scopes, and much more to be detailed later.

[1] JavaScript is an interpreted language.

Control-Flow Graph (CFG)

Control-Flow Graphs or Execution Graphs contain all possible execution paths of a program. They are essential to compiler optimisations and widely used in static analysis tools.

2.1.3 Use Cases and Limitations

Static analysis use cases are generally code quality-related: on the one hand, the program under development should comply to specified programming styles and rules, on the other hand, the number of defects in the software should be as low as possible, ideally zero. If the software under development is part of a mission-critical solution, finding and fixing defects is essential.

Code style analysers and code formatters are used to enforce team- or company-wide coding styles. Linters are rule-based tools: they reveal simple programming errors and poorly used programming constructs. Pattern-matching techniques supplemented with algorithms to manipulate the representing data structure can be efficient to obtain deeper insights of the source code: this approach is to be detailed later being one of the subjects of this thesis. Static analysis with methods of formal verification uses mathematical models and methods to prove well-defined statements about the inspected source code.

Static analysis is limited in many ways. It often provides false values: *false positives* are issues which do not have real significance or are not even true, *false negatives* are real issues not being reported by the analysis tool. A framework is considered to be *sound* if all defects checked for are reported by the tool: there are no false negatives but there can be false positives. A general approach of static analysis frameworks is to be sound, and simultaneously avoid extensive reporting of false positives [10].

Regarding limitations, time and resources are also important aspects. An analysis tool can not be utilised efficiently, if the amount of either time or resources consumed by an analysis is too high. Even if it was theoretically possible to create a tool which finds every possible defects in a piece of source code, this tool would presumably consume so much time and resources for an analysis that there would be no appropriate use case for operating it [14].

Exploring execution paths greatly benefits static analysis proceedings, as it provides extra information about program states. Nevertheless, exploring all possible execution paths of a program is very costly: if a procedure contains n branches without loops, the number of intraprocedural execution paths would be 2^n [14]. And even if a tool would encapsulate so much resources that it would be capable of exploring all possible executions paths, the set of possible inputs, whose cardinality is typically infinite, would still not be taken into account. Since Alan Turing proved the halting problem to be generally undecidable over Turing-machines [15], we can conclude that — generally — some questions about a software can not be answered only by inspecting its source code.

2.2 JavaScript

JavaScript is a high-level, run-time interpreted language, featuring object-oriented capabilities. Being part of the core of the World Wide Web [16], it is one of the most commonly used programming languages in the world [5].

2.2.1 Brief History of JavaScript

Like all new technologies, JavaScript evolved very fast in the beginnings. The basics of the language was developed in 10 days by Brendan Eich, then-employee of Netscape Communications [17]. The language had multiple names over the time: first it was Mocha, then LiveScript, then in December 1995, it was renamed to JavaScript as a sort of marketing movement [18], after seeing the then-popularity of the heavyweight Java language developed by Sun Microsystems.

Initially, non-professional programmers were aimed by the idea to provide a portable, embeddable programming language that can be executed in web browsers. Since the syntax was closely similar to the syntax of C / C++ / Java, JavaScript rapidly gained traction. In the time of writing this thesis, the language features browser-based client- and separate runtime-based server-side capabilities [19] as well, and extensive tooling, package management [20], testing and build systems are available for automated operations in even larger software development organisations.

2.2.2 The ECMAScript as a Standard and as a Language

There is a significant aspiration to standardise the JavaScript language with its core capabilities and data structures. The first intentions of standardisation began in 1997 by Ecma International [18], resuling in *Standard ECMA-262* [21]. The newest standard currently is the *ECMA-262, 7th Edition (ES7)* [6]. Apart from standardisation, there are several implementations of JavaScript in interpreters like Chakra[1], JScript[2] and Google's V8[3] [7].

Today's growing traction of standardised JavaScript, henceforth also referenced as ECMAScript, can be explained with several reasons. But — due to being untyped[4] and dynamic[5] —, static analysis of JavaScript is difficult. The ECMAScript standard enhances plain JavaScript with several new programming structures making the language more expressive and sometimes

[1] `https://github.com/Microsoft/ChakraCore`
[2] `https://msdn.microsoft.com/library/hbxc2t98.aspx`
[3] `https://github.com/v8/v8`
[4] In JavaScript, no static types are assigned to entities.
[5] Meaning of dynamic here: contrary to static languages where compilation time checks play an important role in verifying various properties of the program, dynamic languages' several common programming behaviours are executed only at run-time. Considering a common example: in JavaScript we have the `eval()` function to execute source code at run-time.

more simple [22]. Users of the language can write more coherent code by applying these new language constructs as well as best practices making it easier to interpret the program by a static analysis tool.

2.2.3 The Process of Transpiling

Transpiling is a word came into existence by mixing *transforming* and *compiling*. It is a generally used process in the ECMAScript developer community to ensure backwards compatibility of newer ECMAScript language standards, like ES6 and ES7.

Compiler A *compiler* is a software with the primary goal of transforming software source code written in a high-level programming language into machine language, usually into a form of binary code called object code [23]. Compiled languages like C, C++, Java, or C# need to be compiled to be executed on a specific processor architecture.

Transpiler A *source-to-source compiler* or *transpiler* is a software which transforms software source code written in a high-level programming language into another high-level programming language. Ideally the two source codes are logically equivalent, meaning that with given abstractions, the operation of the two different software is the same.[1]

Transpiling and compiling have a set of common processing steps [24]. First the source code is parsed into an abstract mathematical form for effective manipulation, then, after optimisations and transformations, both methods yield another kind of code. While compilers' output, being low-level machine code, can be generally executed on a computer architecture without further transformation steps, transpilers' output need further processing. Considering transpiling interpreted languages', the main use case is to provide compatibility with older or other versions of the language.

	Chrome 58	IE 11	iOS 9	Android 5.1
default function parameters	●	—	—	—
spread (...) operator	●	—	○	—
for..of loops	●	—	○	○
const	●	○	○	○
let	●	○	—	—
arrow functions	●	—	—	—

Table 2.1 Excerpt from an ECMAScript 6 compatibility table [25]

[1] The two executed programs need to be logically equivalent, but do not need to correspond in every technical aspects: there can be differences in machine-level operations and low-level proceedings.

In the world of JavaScript, compatibility is a ubiquitous problem, see Table 2.1. Considering all the different browsers and server runtimes, and the slow progression of adopting JavaScript standards, transpiling has an important role in ensuring that the software works on a broad scale of platforms: code written in a modern syntax like ES6 can easily be transpiled into an universally supported syntax like plain JavaScript.

Figure 2.1 shows two logically equivalent pieces of code: the second one (plain JavaScript) is created by transpiling the first one (ECMAScript 6) with a popular, automated transpilation tool, *babel*[1]. As the example shows, new language constructs can make the code much more concise, while the transpiled alternative provides compatibility with older desktop browsers and server runtimes.

The first piece of code uses ES6 constructs for simplicity:

```
[1, 2, 3].map(n => n ** 2);
```

The second piece of code uses widely-supported plain JavaScript constructs only, and is created by transpiling the first piece of code with *babel*:

```
[1, 2, 3].map(function (n) {
  return Math.pow(n, 2);
});
```

Figure 2.1 A transpilation example

2.2.4 Looking into the Goals of JavaScript Static Analysis

As JavaScript is an interpreted language not being checked by a compiler by default at compilation time [11], it is recommended to apply static analysis during the development of, or before deploying a JavaScript application. Due to its dynamic and untyped[2] nature [16], static analysis for the language is a challenging task. There are several existing approaches focusing mainly on defects detection [27, 28, 29], but few of them are ready for production usage, and most of them lack compatibility with recent ECMAScript versions.

Being untyped, an obvious analysis goal is type inference: checking type correctness can eliminate several defects from software. Security demands imply that deeper, logical analysis of JavaScript code is needed. Besides security, the development procedure itself can also benefit from static analysis: there are features like automatic stub generation or auto-complete [27] in several development tools [30, 31].

[1] `http://babeljs.io`

[2] TypeScript, a strict superset of JavaScript adds static typing to the language [26].

2.3 Graph Databases

Being graphs, developing new data structures for Abstract Syntax Trees, Abstract Semantic Graphs and Control-Flow Graphs would be superfluous: they can be practically stored in graph databases. There are established vendors on the open-source and also on the closed-source market [32, 33, 34, 35, 36] providing databases with either a native graph storage model, or with support for storing graphs over an underlying data model other than a graph. For manipulating data, they provide well-defined and well-documented interfaces instead of ad-hoc solutions.

Graphs are mathematically defined data structures being broadly used in several fields of computer science. Recent technologies and implementations made possible for developers to easily embed graph data models into their applications. There are numerous real-world scenarios which can be represented more efficiently as graphs (*nodes* connected to each other by *edges*), than with the traditional, relational approach.

2.3.1 The Property Graph Data Model

It is a common way to define graphs as a set of objects, in which some object pairs are connected to each other. In this model, an object is called *vertex* or *node* or *point*, and a connection between two *vertices* is called *edge* or *relation*. Connections can be detailed further by specifying their directionality, also they can be *labeled* to define them even more. Similarly labeling vertices leads to the model of *typed graphs*. If we assign properties to the nodes or relations, we get the model of *property graphs*. Properties, as shown in Figure 2.2, are usually key-value pairs in the format of `key = 'value'`. Generally, keys are strings, and values represent common data types like string, integer, float, etc.

Figure 2.2 Two people's relationship modeled with a property graph

The Codemodel-Rifle framework uses property graphs for its internal data storage. The parsed source code's AST is transformed into an ASG, and is stored as a property graph: nodes in the AST become property graph nodes, nested AST nodes are connected to each other via labeled graph relations. Figure 2.3 shows the ASG of the simple JavaScript program `const PI = 3.141593;` produced and visualised by Codemodel-Rifle.[1]

[1] Administrative properties and labels are omitted for the sake of simplicity, e.g. no identifiers are shown.

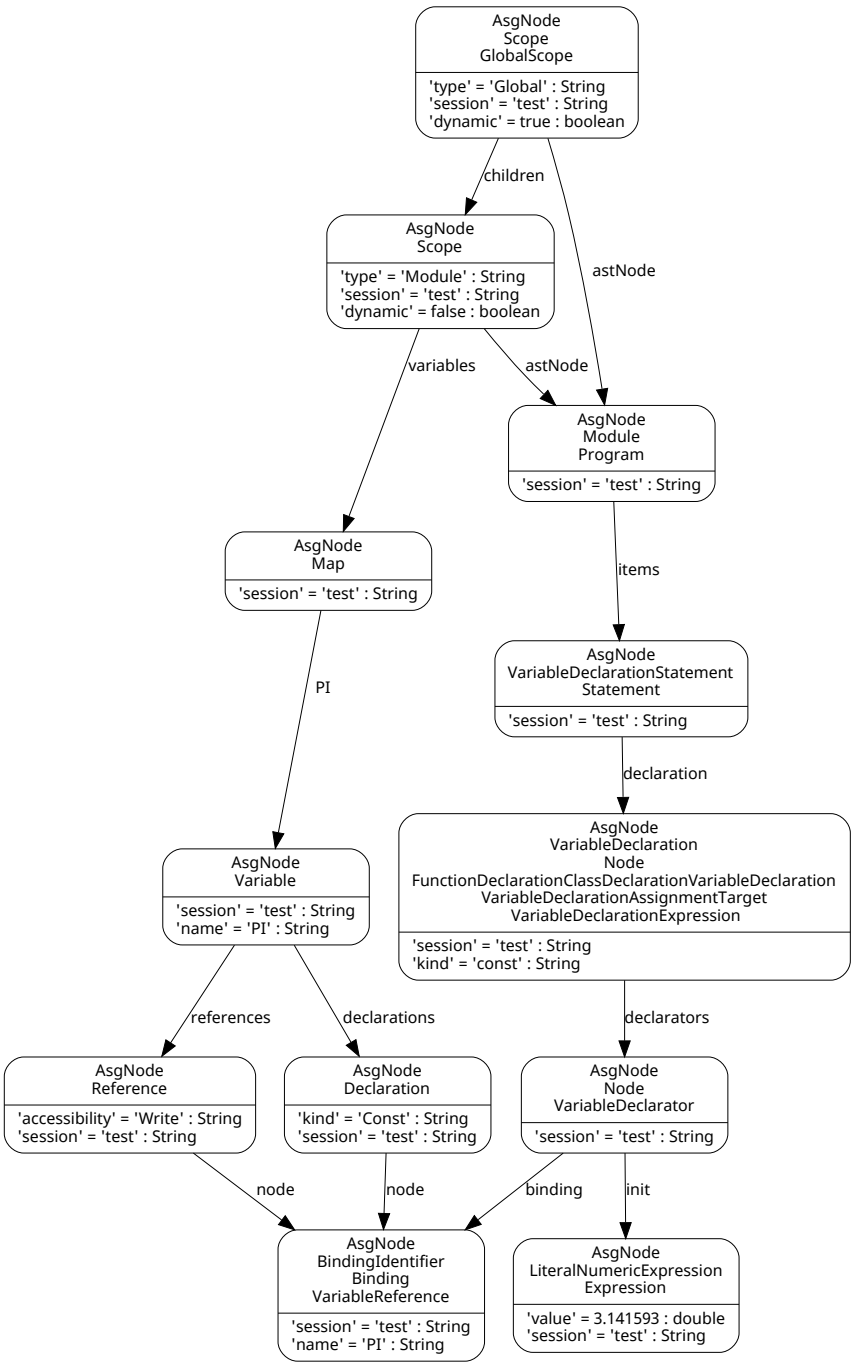

Figure 2.3 `const PI = 3.141593;` in Abstract Semantic Graph format

2.3.2 Neo4j

Amongst a handful of graph database vendors [37], Neo Technology's Neo4j is the most popular one [38]. It features a pure graph data model, contrary to other vendors' multi-model approaches. Besides Neo Technology, Neo4j is backed by the open-source community as well [39]. There are two variants: *Community Edition* and *Enterprise Edition* with an extended feature set. Interestingly, open-source licensing is available for the Enterprise Edition as well [40] (for closed-source software, commercial licensing is available [41]).

Neo4j provides two access models, described in the following paragraphs.

Embedded mode For JVM-based languages, a native API is exposed for data operations with a very low latency. This makes the database directly embeddable to any software written in a JVM-compatible language, but provides less scalability than the *server mode*.

Server/Remote mode The database can be operated as a separate server listening on its binary *Bolt* protocol as well as on its HTTP REST interface. From scalability aspects, the Enterprise Edition's master-slave database replication[1] is only available in server mode.

The Codemodel-Rifle framework uses Neo4j for its property graph storage. At first, the database was embedded into the software, but due to licensing issues, the framework had to be refactored to use Neo4j in server mode.[2]

2.3.3 Cypher

Cypher is a query language developed especially for graph databases by Neo Technology [43]. Contrary to the usage of the native API, it is mostly used when Neo4j is deployed in server mode. Figure 2.4 shows that the language uses a sort of ASCII-art to represent nodes and relationships: nodes are in parentheses, relationships are in brackets surrounded by relationship direction information.

```
(Bob)-[:LOVES]->(Alice)
```

Figure 2.4 A basic Cypher example

Cypher syntax is elegant and expressive, thus very readable. Besides using it to represent nodes and relationships, we can utilise it to access the database's indexing capabilities and stored procedures as well. Since complex pattern-matching conditions can be expressed easily and intuitively in Cypher, it should be the primary way of accessing Neo4j instead of the little bit faster but less readable API.

[1] At the time of this writing, multi-master replication is not offered by Neo4j [42].

[2] The licensing issues and the details and results of the refactoring are described in Section 3.1.

2.4 Running Example

In this section I provide a couple of ECMAScript codes as a software defect example, which accompanies the reader throughout the thesis. This example is to be used whenever a new static analysis concept is introduced. There are two JavaScript modules in the example: module `exporter` in the source file `exporter.js` and module `importer` in the source file `importer.js`.

The first one exports a function, which happens to return 0, as a variable. The second one imports the variable and tries to divide a number with the return value of the imported function variable. Through this example, I present that this and similar software defects can be revealed by graph-based static analysis, even if the defect spans more than one ECMAScript modules (source files), and includes patterns which can not be directly matched by one general graph pattern description. Figure 2.5 presents `exporter.js`. Figure 2.6 presents `importer.js`.

```
var a = 0;

export default function b() {
  let c = function d() {
    return a;
  };

  return c();
};
```

Figure 2.5 Source file `exporter.js`

```
import defaultName from "exporter";

let a = 5 / defaultName();
```

Figure 2.6 Source file `importer.js`

Chapter 3

Related Work

This chapter specifies the currently known approaches and related work of static analysis in general, and specifically for JavaScript.

3.1 Static Analysis Tools for JavaScript

This section introduces several static analysis tools for the main subject of this thesis, the JavaScript language.

3.1.1 TAJS (Type Analysis for JavaScript)

TAJS is a static data flow analysis tool for JavaScript with the capability of inferring detailed and sound type information using abstract interpretation [29]. In the time of this writing, it fully supports the 3rd version of ECMAScript, and partially supports the 5th version[2], including its standard library, the HTML DOM, and the browser API [45].

The abstract interpretation approach consists of the following main points [46]:

1. construct the *Control-Flow Graph* of the program,
2. define a data flow lattice [29], which abstracts program data flow into a mathematically interpreted format,
3. define transfer functions, which abstracts the operations on the data flow lattice.

There is an Eclipse plug-in for TAJS, but according to the creators of the framework, it is not ready for production usage [47].

[2] ECMAScript 5 is the most popular, and most broadly used version of ECMAScript, supported by most of the desktop and mobile browsers and external runtimes [44]. This is the ECMAScript version I referred to previously as *plain JavaScript*.

3.1.2 Flow

Flow is a static type checker for JavaScript developed and maintained by the Facebook Open Source community [48]. Flow checks the code for defects based on *static type annotations* [49]. Without explicit type annotations, Flow is still able to work by attempting to infer types implicitly. Thus, into larger codebases, Flow can be introduced incrementally.

Like many other static analysis tools, Flow also aims for soundness, while preventing extensive reporting of false positives. The developers of the tool identified two main goals: precision and speed. According to the very imprecise documentation [50], Flow is made to be practically precise by modeling the language's essential characteristics accurately enough to differentiate between intentional solutions and unintentional mistakes.

Flow's speediness means to be part of the editing process: the goal is to be fast enough for an IDE to show type information in real-time, during editing the code. To achieve this speed, Flow uses file-level incremental processing, meaning only those files need to be processed, which were changed since the last analysis.

3.1.3 Tern

From the Tern website: "Tern is a stand-alone code-analysis engine for JavaScript. It is intended to be used with a code editor plug-in to enhance the editor's support for intelligent JavaScript editing." [51] Tern provides features like editor auto-completion of variables and properties, function argument hints, automatic refactoring, and finding the definition of functions or variables. Being written in JavaScript, it is capable of running on external runtimes and in web browsers as well.

The software is maintained on GitHub [52] by Marijn Haverbeke, developer of the Acorn lightweight JavaScript parser. Acorn is used as the underlying parser for the Tern infrastructure, which consists of several components: the editor plug-ins communicate with the Tern server, which is implemented on top of the server module, which uses the inference engine to perform analyses [51].

Tern's editor plug-ins' list contains editors with significant or growing popularity:

- Emacs
- Vim
- Sublime Text
- Brackets
- Eclipse

At the time of this writing, the newest version of Tern is 0.21, implying that the tool is not yet aimed for heavyweight production usage, but rather for experimental purposes.

3.1.4 SonarQube

SonarQube (formerly Sonar) is an open-source platform providing "Continuous Code Quality as a Service" [53], backed by a Swiss software company called SonarSource. The platform offers two functionality model for source code analysis:

- Used **as a service**, SonarQube analyses GitHub repositories online: an analysis is triggered every time if new code is pushed to the repository. Analysis settings and results are available on a customisable, per-project interface within the SonarQube website after authentication.
- Used as an **offline tool**, SonarQube can be integrated into the build process with plug-ins available for popular build and continuous integration tools like Maven, Gradle, Jenkins and Apache Ant. It has a command-line interface as well, allowing build-independent analyses.

Following the documentation [53], the platform's *Code Quality Model* is based on three types of rule-based constraints:

- **bugs** track code that is highly likely to yield unexpected behavior of the software,
- **vulnerabilities** are raised on code that is potentially vulnerable to exploitation, and
- **code smells** are code snippets that confuse maintainers being measured primarily in terms of the time they will take to fix.

The platform supports a wide variety of programming languages: in the time of this writing, there are rules for Java (411), C++ (315), Python (238), C# (229), C (225) and JavaScript (186), besides others. As implementing constraints for new problems is highly encouraged in the community, the list of rules is continuously expanding.

Apart from the basically linting-based rules of *code smells* constraints, the software is capable of detecting common bugs, pitfalls and vulnerabilities over JavaScript source codes. Constraints in the *bug* category include inspecting whether non-empty statements alter the control-flow, if non-existent variables or properties are referenced, or if conditionally executed code blocks are not reachable, amongst others.

The inspections in the *vulnerability* category check for vulnerable functionality usage patterns including dynamically injected and executed code, debugger messages, and using the local storage of the browser, amongst others.

3.1.5 Shift

Shift is not a static analysis tool, but an AST toolset created and developed by Shape Security, consisting of several tools [54]. Besides others, Shift features a parser, a code generator, and a scope analyser. It supports the full *ECMAScript* 7^{th} *Edition* [54], and its parser and scope analyser are foundations of the Codemodel-Rifle framework.

It is to be mentioned here, that Shift uses its own AST format, first announced by Shape Security in late 2014, as their first open-source contribution. According to their reasoning, a new ECMAScript AST format was needed because its predecessor, Mozilla's SpiderMonkey AST was not specifically created for static analysis purposes, but rather for an internal representation only for interpretation.

Shift AST is said to comply with all aspects of a good AST-format, as

- "it minimizes the number of inhabitants that do not represent a program,
- it is at least partially homogenous to allow for a simple and efficient visitor,
- it does not impede moving, copying, or replacing subtrees,
- it discourages duplication in code that operates on it." [55]

3.1.6 Esprima

Esprima is an ECMAScript parser with extended capabilities, like syntax validation. It supports the full standard of *ECMAScript 7th Edition*. The open-source software is created by Ariya Hidayat, engineer of *Shape Security*, and is maintained on GitHub [56].

3.1.7 Comparison of the Featured Tools

Table 3.1 presents a functional comparison of the featured JavaScript static analysis tools. From the version number and open-source attributes like the number of contributors and the license, the tool's maturity and usability can be inferred.

	TAJS	**Flow**	**Tern**	**SonarQube**
ECMAScript support	ES3	ES5	ES6	ES7
open-source	●	●	●	●
number of contributors	1	335	87	59
license	Apache 2.0	BSD 3	MIT	LGPL 3.0
current version	v0.9-10	v0.45.0	0.21.0	6.3.2
infers types	●	●	●	—
needs non-standard syntax	—	●	—	—
checks code style	—	—	○	●
analyses vulnerabilities	—	—	○	●
functionally extensible	—	—	○	●
analyses related files	—	—	●	●

Table 3.1 Comparison of the featured JavaScript static analysis tools

3.2 Static Analysis Tools for Java

This section introduces static analysis tools for Java, mainly for earning new ideas regarding static analysis.

3.2.1 FindBugs

FindBugs is a static analysis tool for detecting bug patterns in Java code [57]. One of its main techniques is to syntactically match source code to programming constructs marked as suspicious programming practise. “For example, FindBugs checks that calls to `wait()`, used in multi-threaded Java programs, are always within a loop–which is the correct usage in most cases. In some cases, FindBugs also uses dataflow analysis to check for bugs. For example, FindBugs uses a simple, intraprocedural (within one method) dataflow analysis to check for null pointer dereferences. FindBugs can be expanded by writing custom bug detectors in Java. We set FindBugs to report ‘medium’ priority warnings, which is the recommended setting.” [58]

3.2.2 PMD

Similarly to FindBugs, PMD performs syntactic analysis on Java programs, but is does not have a data flow component. “In addition to some detection of clearly erroneous code, many of the ‘bugs’ PMD looks for are stylistic conventions whose violation might be suspicious under some circumstances. For example, having a try statement with an empty catch block might indicate that the caught error is incorrectly discarded. Because PMD includes many detectors for bugs that depend on programming style, PMD includes support for selecting which detectors or groups of detectors should be run.” [58]

3.2.3 jQAssistant

A German technology firm, Buschmais developed a component-based static analysis tool for Java, called jQAssistant [59]. Similarly to the Codemodel-Rifle framework, jQAssistant is built upon the Neo4j graph database. According to the documentation [60], the tool is to be integrated into the build process to detect constraint violations and generate reports about user defined concepts and metrics.

Analysis rules can be expressed in Neo4j’s graph query language, Cypher. However, instead of the semantics of the source code itself, jQAssistant focuses on the software components and their connections. Its features include validating dependencies between modules in a project, enforcing naming conventions e.g. for test classes, packages, JPA entities, and detecting common architectural problems like cyclic dependencies [60]. The products is licensed under GNU General Public License v3, allowing developers to use it in open-source projects [61].

3.3 Static Analysis Tools for C and C++

This section introduces static analysis tools for C and C++, mainly for earning new ideas regarding static analysis.

3.3.1 Clang

Besides serving as a compiler front-end for LLVM, Clang has a static analyser component for finding bugs in C, C++, and Objective-C programs [62]. The tool can be used either as a standalone command-line tool, or as an Xcode[1] plug-in.

Clang uses static analysis based on compiler techniques. It is designed to report much more information than GCC, using control-flow graph analysis. It features flow- and path-sensitive analyses while preserving the overall form of the original source code [63]. The tool can be integrated into IDEs, and supports automated refactoring.

Following [62], the checkers of Clang can be divided into six groups.

Core checkers Core checkers model core language features and analyse general software defects like division by zero or null pointer deference. It features checks for arrays initialised with zero size, uninitialised values used in assignments or branch conditions, or undefined return values of a function.

C++ checkers As the name implies, these checkers perform analyses specifically for defects related to the C++ language. Without counting checkers marked as experimental, the category has only one member; it analyses double-free, use-after-free and offset problems involving the `delete` keyword.

Dead code checkers This category also has only one member, which checks for values stored to variables that are never read afterwards.

OS X checkers These checkers perform Objective-C-specific checks and analyse if Apple's SDKs and APIs are used appropriately.

Security checkers Members of this category check for insecure API usage and perform analyses based on the CERT Secure Coding Standards. Checks include verifying if return values of insecure API calls are checked, or if a float value is used as a loop counter.

UNIX checkers These checkers analyse UNIX-specific defect possibilities, like mismatched memory deallocation, incompatible types used in `malloc` calls, or insecure API usage.

[1] Xcode is Apple's integrated development environment only available for Apple's macOS, containing a suite of development tools for Apple platforms: macOS, iOS, watchOS and tvOS.

3.3.2 PolySpace

PolySpace Technologies, which first developed the PolySpace Verifier static analysis tool, was later acquired by MathWorks. PolySpace Verifier has been reorganised into a suite, which now features static analysis for C and C++. Similar to TAJS's approach, PolySpace uses the classic lattice-theoretic abstract interpretation technique. The underlying analyser relies on a sound approximation of the set of all reachable states [10]. The tool features a mathematical data structure named *convex polyhedron*[1], several *convex polyhedra* encodes the sets of states [64].

The tool is sound in the meaning that, given the full code base of the project, it computes the superset of every reachable state. It is flow-sensitive, context-sensitive, features inter-procedural analyses, and supports aliasing. "The properties checked by PolySpace Verifier are in many cases similar as those checked e.g. by other commercial systems, but the analysis is more sophisticated taking account of non-trivial relationships between variables (taking advantage of convex polyhedra) while other static analysis tools seem to cater only for simple relationships (e.g. equalities between variables and variables being bound to constant values or intervals of values)." [10]

PolySpace Verifier features checks for array conversion range extensions, return value initializations, variable initializations, pointer initializations, scalar/float under- and over-flows and division by zero, non-termination of calls and loops, correctness of function arguments, unreachable code and many others [10].

In today's product portfolio [65], Polyspace Bug Finder™ features the goal of locating defects with static analysis, and Polyspace Code Prover™ is said to prove the absence of run-time errors in C and C++ source code.

3.3.3 Coverity

Coverity Prevent, now part of Synopsys [66], is a static analysis tool created as a spin-off from a research group at Stanford University. "In 2006 Coverity and Stanford were awarded a substantial grant from the U.S. Department of Homeland Security to improve Coverity tools to hunt for bugs and vulnerabilities in open-source software. During the first year 5,000 defects were fixed in some 50 open source projects. Updated results of the analyses can be found on the web.

The tool itself is a data-flow analysis tool featuring inter-procedural analyses. The analysis is neither sound nor complete, that is, there may be both defects which are not reported and there may be false alarms. A substantial effort has however been put on eliminating false positives, and the rate of these is clearly low (reportedly around 20 per cent)." [10]

[1] A convex polyhedron is an n-dimensional geometric shape where for any pair of points inside the shape the straight line connecting the points is also inside the shape [10].

Coverity features a different set of C and C++ checkers. For C, Coverity checks for resource leaks, dereferencing/deallocating already deallocated memory, uninitialised variables, unused pointer values, dead code, null pointer dereferences, misuse of negative integers and functions that may return negative integers, and null returns, amongst others. For C++, Coverity checks for errors in overriding virtual functions, resource leaks because of missing destructors, past-the-end STL iterators, and uncaught exceptions, amongst others.

Coverity has concurrency and security checkers as well, such as checks for double locks and missing releases, dangerous function calls like `gets` or `strcpy`, string overflows, and incorrect usage of the `chroot` system call [10].

3.4 Most Used Error-Checking Constraints

According to the above related work, the following error-checking constraints are the most widely used ones in static analysis tools:

- type correctness,
- uninitialised variables,
- unreachable code,
- division by zero,
- misuse of negative integers as function arguments.

Chapter 4

Overview of the Approach

This chapter gives an overview of my approach of JavaScript static analysis using *Codemodel-Rifle*, and describes all — modularity- and performance-related — *architectural* changes of the framework.

4.1 Rearchitecturing the Codemodel-Rifle Framework

Dániel Stein, creator of the Codemodel-Rifle framework, details the design of the framework in his Master's Thesis [7]. Following his thesis and my experiences with the framework, Figure 4.1 and the below specification summarises the software's original architecture:

- A source code file is delivered to Codemodel-Rifle via the HTTP REST API of the framework's embedded webserver as a text.
- The framework parses the incoming source file into an AST model with Shape Security's Shift parser.
- The framework performs scope analysis on the AST model with Shape Security's scope analyser, transforming the AST model into an ASG model.
- The ASG model is transformed to a property graph and is stored in the framework's embedded Neo4j graph database.
- Apart from importing a file, the framework is able to perform analyses on or visualisation of a graph stored in its database, if requested over its REST API.
- Analysing more than one ECMAScript modules coherently is only minimally supported: interconnecting the related modules' subgraphs along the *export* and *import* ECMAScript statements is implemented for one use case only, out of more than 80.
- The result of the analyses or the visualisation is returned via the REST API in JSON or in a visual file format.

Codemodel-Rifle was notably refactored since then. This section introduces why refactoring was necessary, and presents the details and the results of the process.

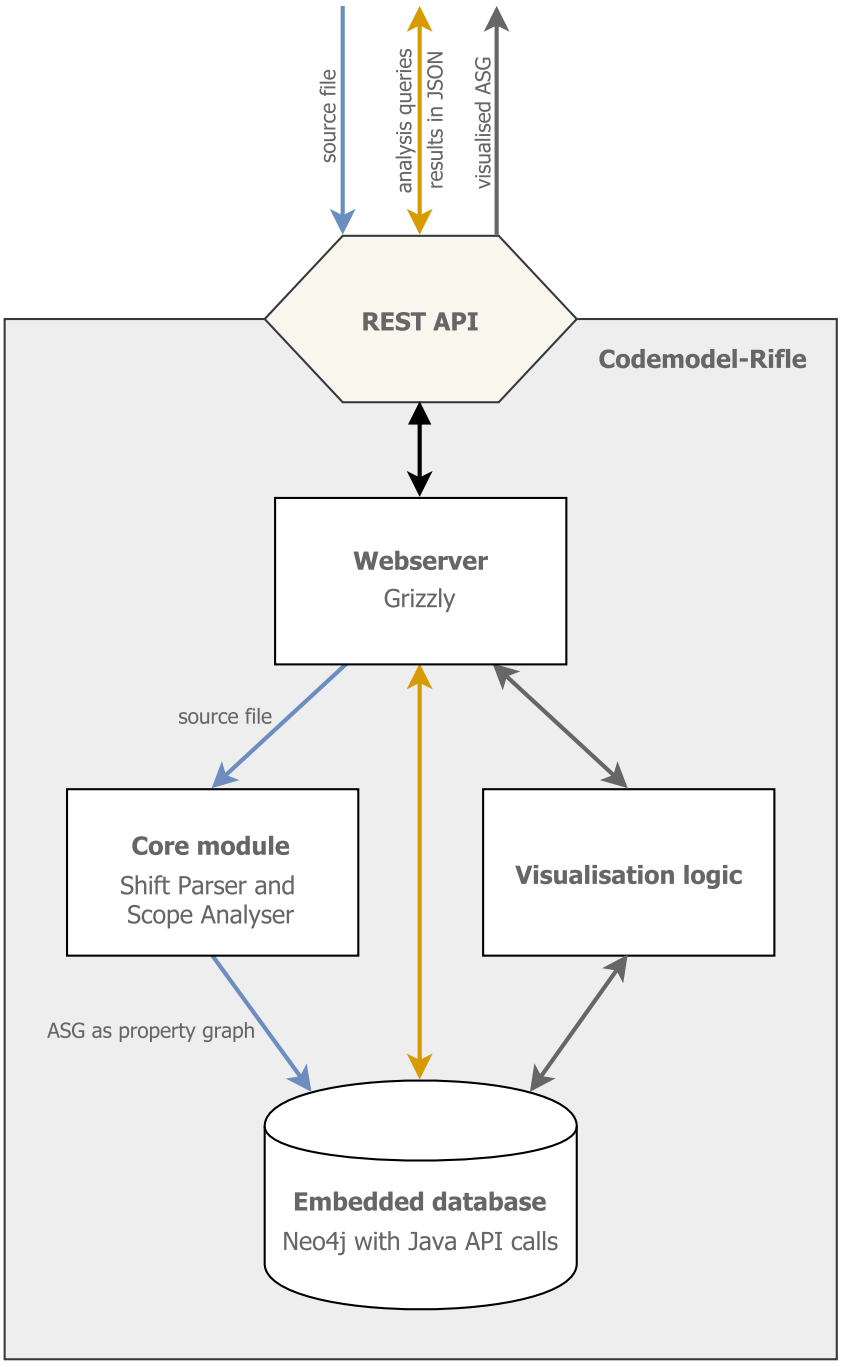

Figure 4.1 The original architecture of the Codemodel-Rifle framework

4.1.1 Open-Sourcing and Licensing Issues

The development of the Codemodel-Rifle framework was supported by the Fault-Tolerant Systems Research Group (FTSRG) of the Budapest University of Technology and Economics. FTSRG's decision — with the support of Dániel Stein — was to open-source the framework under the Eclipse Public License, version 1.0 (EPLv1) [67]. This introduced licensing problems as follows.

The framework uses Neo4j as its internal graph data storage, and Neo4j was embedded into Codemodel-Rifle [7]. From the point of licensing, there is an important difference between *using* the database *via a network connection* and *embedding* the database *into software*. Since Neo4j's Community Edition, used by Codemodel-Rifle, is licensed under GPLv3 [41], it can be used remotely via a network connection with practically any license because of the so-called *application service provider loophole* [68], but it can not be embedded into applications which do not comply with GPLv3. As EPLv1 and GPLv3 are incompatible, Neo4j can not be embedded into the open-sourced Codemodel-Rifle.

Consequently, a necessary step was to switch from embedded Neo4j to remote Neo4j accessed via a driver. But, as native API-calls, which were extensively used by Codemodel-Rifle, can not be used with driver-accessed remote Neo4j, this caused further problems; these are subjects of the next subsections.

4.1.2 Decomposing the Architecture

Codemodel-Rifle's first architecture was monolith. It *embedded* four key modules:

- a Neo4j graph **database**,
- a **webserver** exposing an HTTP REST API for interactions,
- the **core module** responsible for transforming source code into an ASG,
- and **other application logic**, e.g. for displaying and exporting AGSs into visual file formats like PDF or PNG.

Analysing the graph was possible either by running built-in Cypher queries via dedicated REST endpoints (e.g. `/unusedfunctions`), or by submitting custom Cypher queries to the embedded database via the `/run` endpoint.

Decoupling, or minimising direct interdependencies between components is an important aspect of software engineering. If a software is decomposed into smaller components along well-defined interfaces, it becomes modular: any module's inner functionality can be changed without affecting other modules, as long as the module implements the interface it was bound to. Motivations to alter a module include performance issues, scalability efforts, or changed domain logic. Codemodel-Rifle's first architecture was well-designed for easy manual testing and seemed to be an obvious solution for creating a small-scale analysis software. But several reasons required the framework to become modular to adapt.

Detaching the Database

Apart from the licensing issues detailed above, using a remote Neo4j server as a database instead of the embedded version comes with several benefits. The database can be outsourced onto a separate hardware or infrastructure: since analyses and graph maintenance can be demanding over large code repositories, providing dedicated resources for the database is an obvious solution for possible performance issues and scalability.

With a remote Neo4j database, a custom database driver can be utilised. This driver can be capable of incremental processing on the graph database level.[1] Or it can provide an impermanent, in-memory local database instance for testing and for developing new analyses — to eliminate the need of installing a complete database server when long-term persistency is not explicitly needed.[2]

As a result of the aforementioned benefits and licensing issues, the framework was refactored to use a remote Neo4j server via a driver. This meant native API-calls were no longer possible: interacting with the database has been restricted to Cypher queries provided via the database driver. The Codemodel-Rifle framework extensively used native API-calls, so all these function calls had to be rewritten into distinct Cypher queries. As Cypher queries turned out to be notably slower than the API, when executed many queries at once, this introduced *performance issues*. Solutions to these issues are described in the next sections.

Eliminating the Web Interface

The framework contained an embedded Grizzly [70] web server to expose an HTTP REST API for user interactions. This was a convenient way for manual testing and a sensible approach for operating the software in a prospective production environment as well. All communication with the Codemodel-Rifle framework (operating as a server) could be achieved via its HTTP REST API with tool like curl [71] or Postman [72] (in development), or with an IDE or CI plugin (in production).

For automated testing, however, an HTTP REST API is inconvenient: solving important testing issues like exception handling are not straightforward. Since the framework is not yet ready for production use at all, but is heavily under development, an architectural decision was to eliminate the web server, and focus on the core functionality: the analyses. After removing the webserver from the architecture, the in-development way to supply code repositories to the framework for analysis is via unit tests: each test has its resources shipped along with the framework's source code.

[1] Gábor Szárnyas et al. are developing a graph database driver named *ingraph* with the goal of evaluating openCypher queries incrementally [69].

[2] Currently, the default configuration of the framework is to use an impermanent, in-memory graph database accessed via a Neo4j-compatible database driver.

Separating the Visualisation Logic into an Isolated Project

Visualising the ASG of an imported JavaScript source code is key to get familiar with Codemodel-Rifle's ASG-semantics, as well as for developing new analyses. Figure 2.3 displays an example of an ASG created and visualised by Codemodel-Rifle. However, the framework does not explicitly need this feature to perform analyses. Therefore it was a rational step to separate the visualisation logic into an isolated project, which is called *Codemodel-Visualization*.

4.1.3 Optimising for Testing Purposes

The framework used embedded Neo4j as its storage: the project's folder contained a directory named `database`, in which the full Neo4j embedded graph database was stored. Being embedded, the database instance was managed entirely by Codemodel-Rifle. After refactoring the framework to use an external Neo4j database server accessed via a driver because of the aforementioned licensing issues, testing became difficult. The following database-related steps were needed to run unit tests:

- the Neo4j Community Edition server software needed to be downloaded,
- the designated directory to hold the database data needed to be selected,
- the Neo4j server software needed to be started,
- after the tests, the server needed to be stopped,
- the database needed to be flushed after each test to ensure the necessary level of independence amongst the test cases.

This process can be partially automated with scripts, but it is still not a clean way to perform automated unit tests of Codemodel-Rifle.

As a solution, Gábor Szárnyas advised to use his *neo4j-drivers* project [73]. The package contains wrappers for the Neo4j Java driver: the `EmbeddedTestkitDriver` makes possible to use a local, in-memory `ImpermanentGraphDatabase` accessed via a driver. Using an impermanent, local database is convenient for use cases where persistency is not explicitly needed — e.g. testing and developing new analyses —, since no external Neo4j database needs to be installed and run. At the same time, Codemodel-Rifle can be easily reconfigured for production environments, where the framework needs to persist its data in an actual remote database. This reconfiguration only involves changing the framework's database driver in the `DbServicesManager` class to another Neo4j-compatible one.

4.1.4 Solutions to Speed-Related Issues: Object-Graph Mapping and the Cypher Query Builder

Converting from embedded Neo4j to external, driver-accessed Neo4j, involving converting from *persistent driver-accessed Neo4j* to *in-memory driver-accessed Neo4j* introduced no-

table slowness, making testing and developing new analyses inconvenient again. Table 4.1 compares the duration of visualising a simple JavaScript program (the runnning example's `exporter` module seen on Figure 2.5) with the old embedded, and the new in-memory driver-accessed approach.

	embedded database	**in-memory driver-accessed database**
importing, transforming, storing	82 ms	14,816 ms
visualization	1,832 ms	2,456 ms
total	1,914 ms	17,272 ms

Table 4.1 Speed comparison between the two database approach[1]

Seeing measurement results in Table 4.1, it was necessary to optimise the framework's performance for the in-memory driver-accessed database scenario, because extensive testing would not have been possible with such slowness. Apart from testing, optimisations benefit the in-production performance as well, since the testing and the production environments share the same interface: in both scenario, the database is accessed via a Neo4j driver. Ideally, the optimisations should be configurable to adapt to both the testing and the production environment. In the following paragraphs, I will summarise the optimisations I performed on the Codemodel-Rifle framework.

In Dániel Stein's implementation [7], translating the ASG model to the property graph model happens simultaneously with actually storing the property graph model in the database. If an element of the ASG model has been successfully translated into the property graph model, it is stored in the database immediately. This can be optimised: by creating a property graph model stored in Java objects, and then implementing a storage logic to perform saving the objects into the database, the operative parameters of the storage logic can be optimised directly to the currently used database driver.

Creating a specialised Object-Graph Mapping (OGM) Layer

Importing a repository can be summarised by two types of database-level action.

1. *Creating nodes* — the property graph model's nodes get created in Neo4j.
2. *Setting relationships* — the property graph model's relationships get set in Neo4j.

Therefore, a mapping layer basically needs to translate two object types: *nodes* and *relationships*. I mapped these two object types with the `AsgNode` and `AsgRelation` Java classes. An

[1] These measurements are only for demonstrating that the framework was so slow after the necessary refactorings that it needed to be optimised even for testing. They are not aimed to be fully accurate and complete. Evaluating the framework's performance with accurate measurements is the subject of Chapter 6.

`AsgNode` stores its properties in a `HashMap`, and its labels and relations in two separate `List` members. An `AsgRelation` has a `fromNode`, a `toNode`, and a `relationshipLabel` member. Storing relationship properties was omitted, since the Codemodel-Rifle framework semantics does not contain relationship properties.

Identifying nodes is achieved with a universally unique identifier (UUID), instead of the earlier approach of using Neo4j's discouraged `id()` function to get the nodes' built-in identifier. Each `AsgNode` object has an `id` member, which contains a value generated using the `java.util.UUID` package. The `id` member gets automatically translated into the property graph as well as all other properties. With a mapping layer like the above, it is possible to customise the procedure of storing the model in the database e.g. by optimising query granularity.

The Cypher Query Builder

A main bottleneck identified with the `ImpermanentGraphDatabase` instance of the `EmbeddedTestkitDriver` interface was the speed of parsing queries. The example presented in Table 4.1 requires 201 property graph nodes and 340 relationships to be created, so it normally requires 541 distinct Cypher queries to be run. As per my experience of manually performed testing, if several distinct queries are merged into one, it increases speed significantly. Accordingly, it was a reasonable step to implement a configurable, specialised query builder, which manages storing the property graph model with a coarser query-granularity (by creating more than one nodes or setting more than one relationships within one executed database query).

The query builder I implemented is capable of creating Cypher queries specially for the aforementioned OGM layer, following its internal configuration of how many *node creator* queries and how many *relation setter* queries should be merged (compressed) into one. The builder assembles and prepares the queries, and then returns them in a list. Each database query in the list is ready to be executed without further modifications.

Refactoring the Core Logic to Utilise the OGM and the Query Builder

After implementing and testing the mapping layer and the query builder, I modified the core import logic of the framework in the `ASTScopeProcessor` class to utilise the new components. Instead of immediately storing the translated ASG model as a property graph model in the database, the processor first stores the property graph model in Java objects with my custom OGM layer. Then, benefiting from the query builder, the model is sent to the database in optimally sized chunks following the query builder's configuration.

Table 4.2 shows the optimal configuration values of the query builder in testing environment (with the `EmbeddedTestkitDriver`), and in a prospective production environment (with the official Neo4j driver) for test cases run on my computer.

	testing	production
nodes created in one query	16	20
relationships set in one query	1	2

Table 4.2 Optimal configuration of the query builder for my computer

Results of Speed-Related Refactorings

Table 4.3 shows a comparison between the speed of two versions of the framework when importing the `exporter` module of the running example, presented in Figure 2.5. Both versions presented here use the in-memory driver-accessed database, but the first does not use the optimisations implemented (the mapping layer and the query builder), while the second one does.

	without optimisations	with optimisations
importing, transforming, storing	14,816 ms	7,031 ms
visualization	2,456 ms	2,432 ms
total	17,272 ms	9,463 ms

Table 4.3 Speed comparison with and without optimisations[1]

4.1.5 Other Performances

After the refactoring, the framework's package structure got very complex. Several main features of the software — like source code parsing and other actions to be exposed onto the external interface for user interactions — were mixed with internal operations like database management and utilities. I separated the packages this way: `actions` contains features to be exposed to the user, `database` contains database-related operations, `tasks` contains internal features not to be exposed, and `utils` contains utilities.

The final version of Dániel Stein's framework used the `v2.2.0` version of Shape Security's Shift parser and scope analyser. This version only supports the 6th Version of ECMAScript. Since then, version `es2016-v1.1.1` supporting the full ES7 specification was released by Shape Security [54, 74]. I updated the framework's dependencies to use the new version of the parser and scope analyser.

[1] These measurements are only for demonstrating that the framework became notably faster after the speed-related refactorings. They are not aimed to be fully accurate and complete. Evaluating the framework's performance with accurate measurements is the subject of Chapter 6.

4.1.6 Summary of Refactoring

Figure 4.2 presents a high-level overview of the refactored architecture of the Codemodel-Rifle framework. Besides becoming modular, the framework has gone through a series of optimisations to simplify testing and developing new analyses.

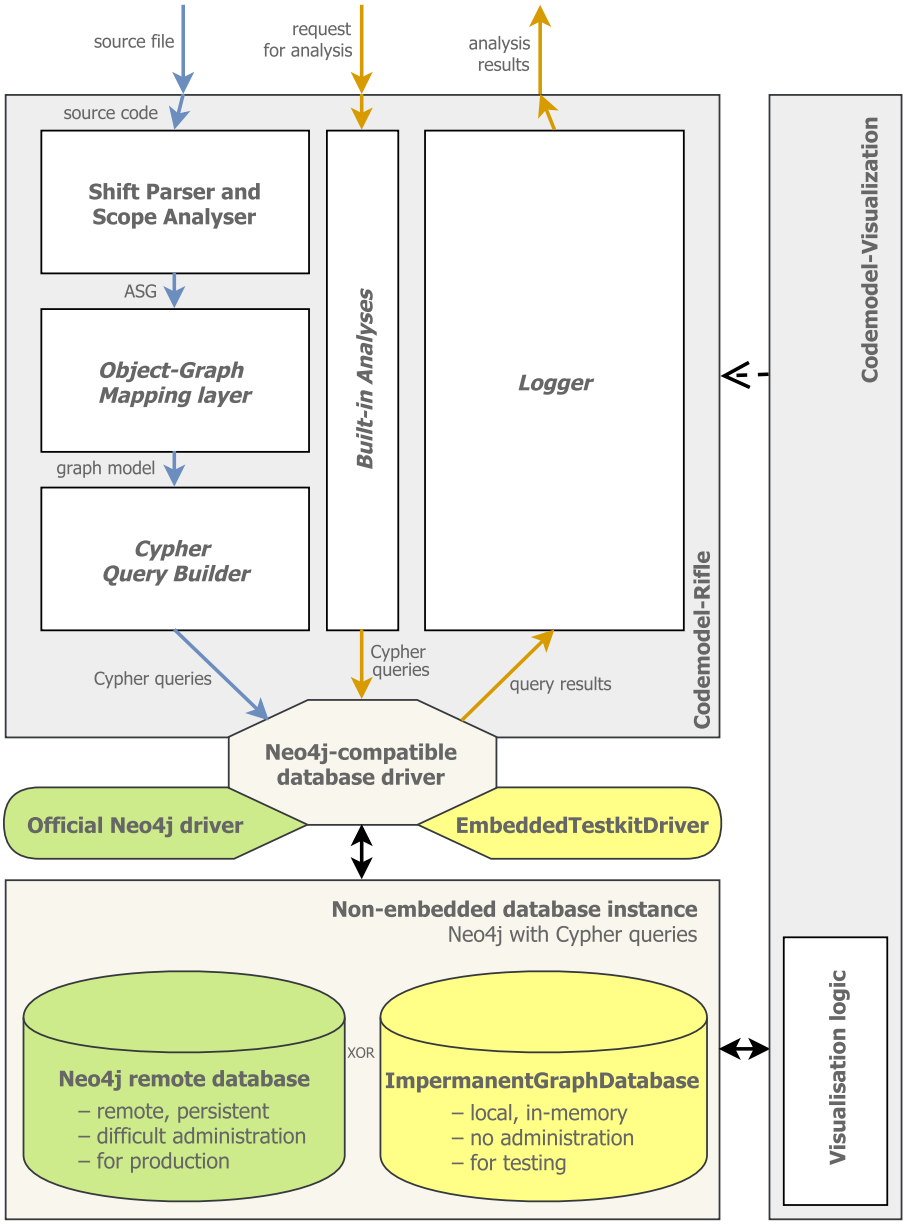

Figure 4.2 The new architecture of Codemodel-Rifle with my contributions *emphasised*

4.2 In Development: Steps of Building New Analyses

Building new analyses for software defects basically consists of three steps. The steps are detailed in the following subsections.

4.2.1 Visualising the Defect with Codemodel-Visualisation

Without seeing what to search for, new analyses can not be implemented. A defect's signature has to be inspected with Codemodel-Rifle's semantics first. For visualising a defect pattern, a new unit test has to be created in the Codemodel-Visualization project. The JavaScript modules containing the defect should be included as test resources.

Using Codemodel-Rifle as a dependency, Codemodel-Visualization first parses the JavaScript files given as test resources and translates them to separate property graphs. If more than one source files were imported, their graphs are interconnected along the export and import semantics of ECMAScript.[1] Finally, the full property graph model gets exported into a visual file format, like PDF or PNG. The export format is configurable in the unit test.

4.2.2 Describing the Defect Pattern

The file exported by Codemodel-Visualization precisely mirrors the property graph instance model translated by Codemodel-Rifle, but some nodes and edges are not displayed to preserve the transparency of the visualised graph.[2] Any pattern seen in the visualised graph can be directly matched by Codemodel-Rifle.

4.2.3 Implementing the Analysis

Analyses are basically Cypher queries. If a defect's pattern can be expressed with a Cypher query, it can be detected by the framework.

Some defects are more high-level or more general than to present their patterns in an intact graph directly. Detecting complex errors like these may require to extensively manipulate the graph to dredge defect patterns for matching. In cases involving *transitive* defects, like in the running example[3] presented in Chapter 2, a flag like `EqualsZero` has to be propagated through the graph along specified edges: variable assignments, variable references, function call and function return statements, etc.

[1] The process of interconnecting ECMAScript modules along export and import statements is one of the key subjects of this thesis. It will be detailed in Chapter 5.

[2] Ignored nodes and edges are listed in the `GraphWalker` class as filtered entities from the underlying visitor pattern implementation. As an earlier architectural decision, this is not configurable externally.

[3] The running example is to detect a division by zero scenario. But zero is not a numeric literal 0, but the indirectly referenced return value of a nested function stack with variable assignments and also a module boundary in between.

Transitive graph manipulations can be achieved by introducing *qualifiers* into the analysis. The concept of qualifiers will be described in detail in Chapter 5.

If an analysis matches the specified pattern, it returns the following:

- a **message** to explain the type of the defect for a human reader,
- an **entity name** (or an empty string) to identify defects bound to named entities like variables and functions,
- the **path** of the containing module,
- the **line** in which the defect was found,
- the **column** of the line at which the defect begins.

In my current implementation, the above items are uniformly[1] returned from the database as elements of a Neo4j `Record`, and they are handed over to a central logger to be immediately printed after minimal formatting. This is not a flexible solution; in the future, this basic defect processing logic should be refined. The found defects could be returned as JSON objects from the database to be easily parsed into a Java class named `Defect`. They could also be collected into a per-analysis data structure. This way, the framework could display defects found at an analysis according to various aspects and criteria, and it could also produce machine-readable output. With a clean API, this would allow the framework to be embedded into other software.

4.3 In Production: Steps of Operating Live

The prospective live operation of the framework basically consists of three steps, which are managed by the framework. Ideally, the operation should be automatic and transparent: if a change is done in the IDE, or a new commit is pushed to the central repository, the framework should perform analyses over the changed code repository. The steps of a full analysis procedure are detailed in the following subsections.

4.3.1 Import: Synchronising the Repository into the Framework

First, the code repository is imported into the framework. This involves listing and parsing all files with configured extensions (currently only `.js`), then saving the created property graph models into the database.

The word synchronising expresses that Codemodel-Rifle aims to be incremental; but while it does so, its capabilities are still very limited. According to plans, the framework will cooperate with VCSs to detect changes, thus it will be able to import only those files that changed since the last import process.

[1] Exactly these items are returned in all cases, regardless of the defect's type. This is not flexible, since an *unreachable code* defect may require other arguments to be logged, than a *non-initialised variable* defect.

4.3.2 Interconnect: Connecting the Related ECMAScript Modules

To evaluate analyses over more than one ECMAScript modules, the related modules' separate property graphs are interconnected along the export and import semantics of ECMAScript. This process is described in detail in Chapter 5.

4.3.3 Analyse: Performing Analyses

Performing analyses can be broken down into two substeps.

Manipulating the Graph

Complex analyses may require to extensively manipulate the graph. These manipulations involving qualifiers are processed first.

Querying the Graph

The graph is queried with Cypher, with matching predefined graph patterns developed with the aforementioned steps. If a defect pattern matches, it gets logged onto the console with the semantics described in Section 4.2.3, in the format seen in Figure 4.3.

```
message: entityname at line:column in path
```

Figure 4.3 The framework's console output if a defect was found

Chapter 5

Elaboration of the Workflow

This chapter details the implementation of the analyses and the additional proceedings about analysing more than one ECMAScript modules coherently. Thus, this chapter encompasses all *semantic* changes of the framework.

Following Dániel Stein [7] and Chapter 4 of this thesis, a full analysis procedure of the Codemodel-Rifle framework can be broken down to three distinct phases:

1. **Import:** Every ECMAScript source file (containing the source code of one ECMAScript module) of the analysed code repository is imported into Codemodel-Rifle. The modules are translated to Abstract Semantic Graph models. The ASGs are stored as distinct, per-module property graphs in the underlying Neo4j graph database.
2. **Interconnect:** The related modules' separate graphs are interconnected along the *export* and *import* semantics of ECMAScript. This makes possible to evaluate analyses over more than one modules coherently.
3. **Analyse:** The predefined analyses are executed.
   a) The graph manipulations of the *Qualifier System* are performed.
   b) The defect patterns are matched.

Since I have not made any semantic changes to the Import phase, this chapter focuses to the Interconnect and the Analyse phases.

5.1 Interconnecting Related ECMAScript Modules

This section describes the work I made to support analysing more than one ECMAScript modules coherently. The approach follows [7], and completes it by developing the semantics of missing use cases, and then implementing them. To shortly summarise: in order to coherently analyse several related ECMAScript modules with the Codemodel-Rifle framework, the related modules' separate property graphs are interconnected by well-defined

rules. As previously already mentioned, these rules are built upon the *export* and *import* semantics of ECMAScript [75]. Equivalently, ECMAScript modules are considered to be *related*, if they refer to each other by using *export* and *import* statements.

5.1.1 The ECMAScript Module System

As the language gained traction, JavaScript projects rapidly grown to a size where modularisation became critical in order to keep the code logically organised. Today's largest ECMAScript code bases include Google's Gmail[1] with approx. 400,000 lines of code [76], Ruben Daniels' Cloud9 IDE[2] with approx. 300,000 lines of code [77], and Lucidchart[3] with approx. 200,000 lines of code [78]. The product of Tresorit featured in this thesis consists of approx. 35,000 lines of ECMAScript code.

Plain JavaScript does not have built-in support for modules [75], there are only community-provided solutions like *RequireJS*[4]. In contrary, the 6th version of ECMAScript has language-level support for modules: each source file represents exactly one module. Entities like variables and functions defined in one module, or even complete modules themselves can be *exported* to be *imported* to a different module. By default, modules are referred by their relative pathname, without the containing file's extension. Entities that are not explicitly exported remain *private*, meaning they can not be imported to other modules.

In ECMAScript 6, there are several ways of exporting and importing entities [75], these are detailed in the next subsections. The Codemodel-Rifle framework had only minimal demonstrative support for interconnecting several ECMAScript modules; I extended Dániel Stein's work by covering the most used *export-import case combinations*.

5.1.2 Export Syntaxes and Cases

By default, each entity can only be accessed in the scope of the module it was declared in. To be accessed in other modules, the entity has to be explicitly exported first. Figure 5.1 presents export syntax examples of ECMAScript 6, based on [79]. Since these statements can be almost arbitrarily combined, and the number of exported variables is not limited in theory, the list of differing export syntaxes of ECMAScript 6 is practically endless.

Therefore, *export syntaxes* need to be distinguished from *export cases*. An *export case* is identified by the *basic form* of an *export syntax*. An *export syntax* written in *basic form* does not combine diverse syntaxes, and exports only one entity per export statement. Figure 5.1 displays all syntaxes in *basic form*, thus it lists all members of the *distinct export cases*' finite set. Each different *export case* has a unique graph pattern in the ASG.

[1] `https://www.gmail.com`
[2] `https://c9.io`
[3] `https://www.lucidchart.com`
[4] `http://requirejs.org`

```
// exportName
export { name1, ... };
// exportDefaultName
export default name1;
// exportAlias
export { name1 as exportedName1, ... };
// exportAsDefault
export { name1 as default, ... };
// exportEmptyLetDeclaration
export let name1, ... ;
// exportEmptyVarDeclaration
export var name1, ... ;
// exportLetDeclaration
export let name1 = ..., ... ;
// exportVarDeclaration
export var name1 = ..., ... ;
// exportConstDeclaration
export const name1 = ..., ... ;
// exportClass
export class name1 { ... }
// exportFunction
export function name1(...) { ... }
// exportGenerator
export function* name1(...) { ... }
// exportDefaultClass
export default class name1 { ... }
// exportDefaultFunction
export default function name1(...) { ... }
// exportDefaultGenerator
export default function* name1(...) { ... }
// exportDefaultExpression
export default expression;
// exportDefaultAnonymousClass
export default class { ... }
// exportDefaultAnonymousFunction
export default function (...) { ... }
// exportDefaultAnonymousGenerator
export default function* (...) { ... }
// exportExpression
export expression;
// reexportNamespace
export * from ...;
// reexportName
export { name1, ... } from ... ;
// reexportAlias
export { import1 as importedName1, ... } from ...;
```

Figure 5.1 Export syntax examples of ECMAScript 6

5.1.3 Import Syntaxes and Cases

An entity declared in module **A** can be accessed in module **B**, if **A** exports, and **B** imports the entity. All exported entities of a module can be imported as well: in this case an object is created with the name of the imported module's alias, and with members listing the exported entities of the imported module. Figure 5.2 present import syntax examples of ECMAScript 6, based on [80]. Like the exports, these statements can also be combined with each other, making the list of the possible import syntax combinations endless.

Thus, *import syntaxes* need to be distinguished from *import cases*, similarly to the exports. An *import case* is identified by the *basic form* of an *import syntax*. Figure 5.2 displays all syntaxes in *basic form*. Each different *import case* has a unique graph pattern in the ASG.

```
// importName
import { name1, ... } from "exporter";
// importAlias
import { name1 as importedName1, ... } from "exporter";
// importDefault
import defaultName from "exporter";
// importNamespace
import * as exportedModule from "exporter";
// importModule
import "exporter";
```

Figure 5.2 Import syntax examples of ECMAScript 6

5.1.4 Number of Export-Import Combinations

Let $\mathbb{E}$ be set of all the distinct export cases, and let $\mathbb{I}$ be the set of all the distinct import cases. As Figure 5.1 and Figure 5.2 show, $|\mathbb{E}| = 23$, and $|\mathbb{I}| = 5$. If all export cases would be compatible with all import cases according to the ECMAScript grammar, set $\mathbb{C}$ containing all combinations would be $\mathbb{C} = \mathbb{E} \times \mathbb{I}$ with the cardinality of $|\mathbb{C}| = |\mathbb{E}| * |\mathbb{I}| = 23 * 5 = 115$.

```
// exporter.js
export let name1 = ...;
// importer.js
import defaultName from "exporter";
```

Figure 5.3 An example of incompatible export-import cases

Let $\mathbb{S}$ be the set of the export-import combinations supported by Codemodel-Rifle, and let α be the number of distinct algorithms needed to be implemented for supporting every

element of $\mathbb{S}$. The following applies: $\alpha \leq |\mathbb{S}|$, since the framework needs one separate algorithm for each export-import case at most. As not all export cases are compatible with all import cases (a counterexample is displayed on Figure 5.3), the set of *semantically valid* export-import combinations is narrower than $\mathbb{C}$. Codemodel-Rifle should interconnect only semantically valid export-import cases, so $\mathbb{S} \subset \mathbb{C}$. Also, α can be reduced further by involving ASG-specific knowledge: with graph pattern generalisation techniques, several export cases can be handled as one at implementing the interconnections, while preserving semantics. Therefore several export cases can be covered by one algorithm, so $\alpha < |\mathbb{S}|$. In addition, by choosing particular export and import cases not to be supported by Codemodel-Rifle, α can be lowered even further. Case compatibility, unsupported cases and pattern generalisation techniques are detailed in the following subsections.

5.1.5 Compatibility of the Export-Import Cases

An export-import combination is considered to be *semantically valid*, if it complies with the ECMAScript grammar [81, 82]. Accordingly, semantically valid export-import combinations consist of *compatible* export-import cases: export case **E** and import case **I** are considered to be *compatible* with each other, if the entity exported by **E** can be imported by **I**, following the ECMAScript grammar. Figure 5.3 shows an example of incompatible export-import cases. Table 5.1 displays a compatibility matrix for ECMAScript export-import cases.

As only semantically valid export-import combinations are required to be supported by Codemodel-Rifle to evaluate analyses over several ECMAScript modules coherently, incompatible cases do not need to be covered. This reduces α from 115 to 84 (see Table 5.1).

5.1.6 Unsupported Cases

There are export and import cases which I chose not be supported by Codemodel-Rifle because of implementation difficulties, or the cases' irrelevant usage. This reduces α from 84 to 33. The unsupported export and import cases are the following:

- **`exportDefaultExpression`, `exportDefaultAnonymousClass`, `export- DefaultAnonymousFunction` `exportDefaultAnonymousGenerator`**: There is no clear way for interconnecting the exported entities with the importer module.
- **`exportExpression`**: Unnamed expressions (e.g. **`export`** `1 + 2;`) can not be imported, because they can not be referenced.
- **`reexportName`, `reexportAlias`, `reexportNamespace`**: According to my experiences, re-exporting is used very little.
- **`importNamespace`**: There is no clear solution for including all exported variable of the imported module as an object into the ASG.
- **`importModule`**: It only loads the module, does not import anything. The first such import in a program executes the body of the module [75].

Table 5.1 displays the unsupported export and import cases with grey background. With excluding the incompatible and the unsupported cases from the interconnection process, α is reduced by more than 71%, from 115 to 33. This saves a significant amount of work without notable loss of the analyses' credibility — unsupported cases were mostly chosen because of their unpopularity. Nevertheless, these cases need to be covered later as well.

	importName	importAlias	importDefault	importNamespace	importModule
exportName	●	●	○	●	●
exportDefaultName	●	●	●	●	●
exportAlias	●	●	○	●	●
exportAsDefault	○	○	●	○	●
exportEmptyLetDeclaration	●	●	○	●	●
exportEmptyVarDeclaration	●	●	○	●	●
exportLetDeclaration	●	●	○	●	●
exportVarDeclaration	●	●	○	●	●
exportConstDeclaration	●	●	○	●	●
exportClass	●	●	○	●	●
exportFunction	●	●	○	●	●
exportGenerator	●	●	○	●	●
exportDefaultClass	●	●	●	●	●
exportDefaultFunction	●	●	●	●	●
exportDefaultGenerator	●	●	●	●	●
exportDefaultExpression	○	○	●	○	●
exportDefaultAnonymousClass	○	○	●	○	●
exportDefaultAnonymousFunction	○	○	●	○	●
exportDefaultAnonymousGenerator	○	○	●	○	●
exportExpression	○	○	○	○	●
reexportName	●	●	○	●	●
reexportAlias	●	●	○	●	●
reexportNamespace	●	●	●	●	●

Table 5.1 Export-import compatibility matrix with unsupported cases in grey

5.1.7 Pattern Generalisation Techniques

After excluding the incompatible and the unsupported cases, 33 different import-export combinations still need to be covered by the interconnection process. This would imply that $\alpha = 33$ algorithms are needed for all combinations, but α can be reduced further by involving ASG-specific knowledge. At interconnecting modules, several export cases' graph patterns can be matched by one, generalised pattern description, and thus several export cases can be interconnected with the same algorithm. For import cases, generalisation is neither possible nor necessary, since only three, semantically different import cases are supported by Codemodel-Rifle. To proceed, two concepts are defined.

Semantically correct interconnection An export-import interconnection between two modules' property graphs is *semantically correct* to Codemodel-Rifle, if the interconnection is reversible, it correlates with the semantics of ECMAScript, and the interconnected property graphs contain the same information as the separate property graphs.

Isomorphic export case Two export cases are *isomorphic*, if they contain ASG patterns which can be interconnected to an import case along the same nodes and edges, applying the same algorithm, and the interconnection is semantically correct.

Applying the two definitions, my workflow was the following for finding the isomorphic export cases in order to reduce α:

1. I inspected the similar export cases' ASG patterns, whether they can be described by one, generalised graph pattern description.
2. If yes, I examined if the two export cases can be interconnected with import cases along the same nodes and edges, with the same algorithm.
3. If yes, I performed the interconnections, and inspected them whether they are semantically correct.
4. If yes, the two export cases are isomorphic.

Figure 5.4 presents two distinct ASGs of two isomorphic export cases as an example. These two cases are described below with also specifying their location on the figure:

- ON THE LEFT: `export let name1 = "name1Value"`
- ON THE RIGHT: `export function name1() { return "name1Value"; }`

The two export cases are isomorphic because of the following.

a) *The two graphs contain patterns which can be matched by one pattern description.* Even though these patterns (indicated with thicker outlines) contain nodes and edges with different labels and properties, in Neo4j it is possible to match both of them with only one Cypher expression.

b) *Both patterns can be interconnected to an import along the same nodes and edges applying the same algorithm.* Applying the semantics of Codemodel-Rifle developed along practical reasons, only the node labeled as `Declaration` (indicated with blue filling) needs to be connected to the import module's ASG in both cases.

c) *The interconnection is semantically correct.* In both cases, the interconnection is reversible, and no information is lost. The interconnection also correlates with the semantics of ECMAScript: in both cases, it expresses that a *named declaration* has been imported from another module. For the aim of the Codemodel-Rifle framework — which is revealing possible errors in software by static analysis — this is a satisfactory way of implementing the interconnections.

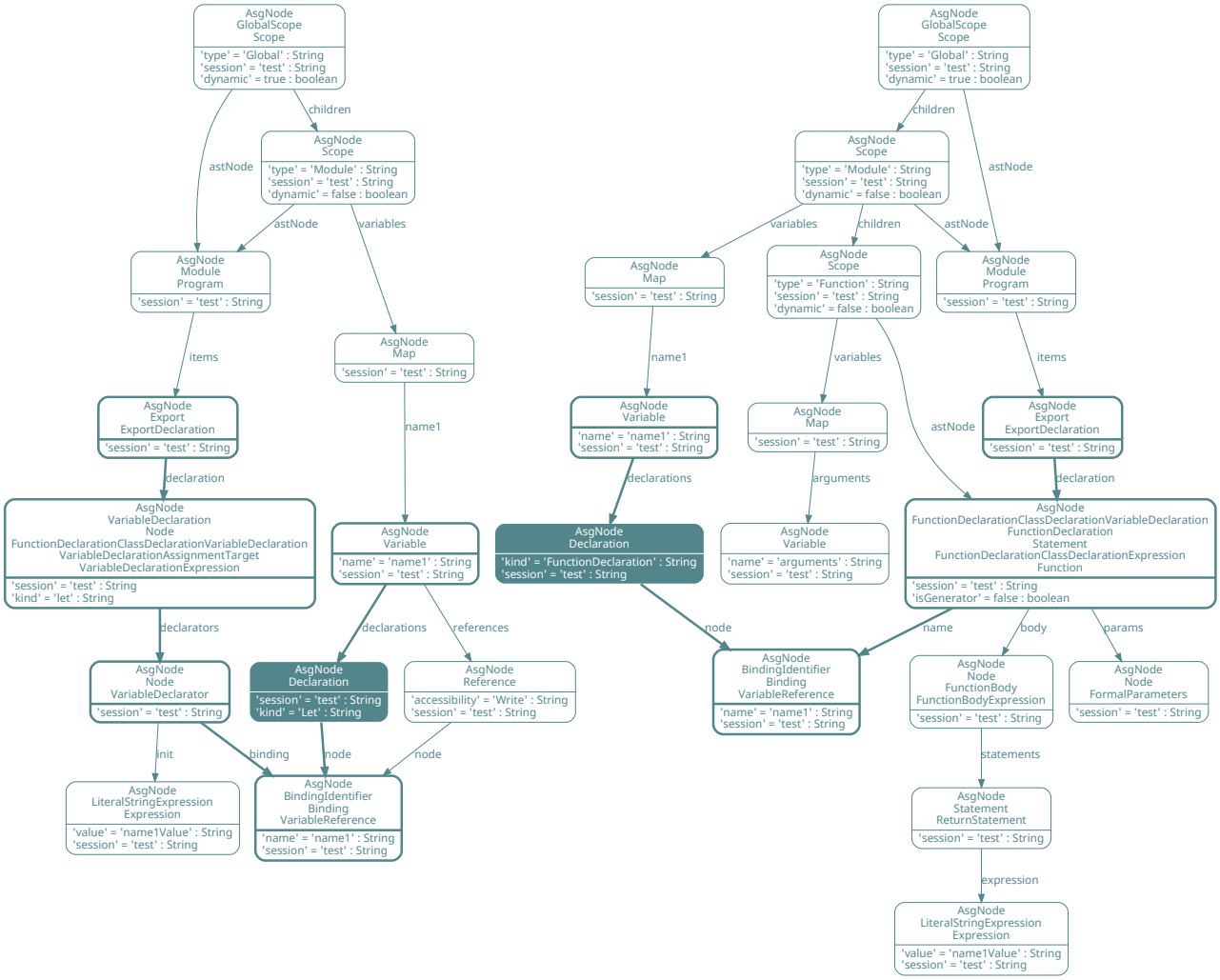

Figure 5.4 Two isomorphic export cases contain the same pattern

The process of pattern generalisation needs to be performed carefully. The generalised patterns must match only those export cases' patterns that can be interconnected with imports in a semantically correct way. If the patterns are too broadly generalised, they will match more export cases than intended, resulting semantically incorrect interconnections (between incompatible export-import cases). In contrary, if they are too narrowly specified, they will match only one export case, resulting no reduction of α.

In the following, I list all export cases I found to be isomorphic in groups. Each group's name implies why the elements are isomorphic in the group. Since every element can be interconnected with imports using the same algorithm per group, an isomorphic group with its elements can be considered as one generalised export case regarding the ASG interconnection process of the Codemodel-Rifle framework. The following 5 isomorphic export groups have been formed:

- **`exportName`**
  - `exportName`
- **`exportDefaultName`**
  - `exportDefaultName`
- **`exportAlias`**
  - `exportAlias`
  - `exportAsDefault`
- **`exportDeclaration`**
  - `exportEmptyLetDeclaration`
  - `exportEmptyVarDeclaration`
  - `exportLetDeclaration`
  - `exportVarDeclaration`
  - `exportConstDeclaration`
  - `exportClass`
  - `exportFunction`
  - `exportGenerator`
- **`exportDefaultDeclaration`**
  - `exportDefaultClass`
  - `exportDefaultFunction`
  - `exportDefaultGenerator`

Based on the above, having 5 isomorphic export groups means that the number of distinctly handled export cases has been reduced to 5. Table 5.2 shows the updated compatibility table with export cases grouped by their isomorphism, without listing the unsupported cases. By this time, with excluding incompatible and unsupported cases, and applying pattern generalisation techniques, α has been reduced to 13, meaning only 13 separate algorithms have to be implemented in order to cover most of the export-import cases.

	importName	importAlias	importDefault
exportName	●	●	○
exportDefaultName	●	●	●
exportAlias	●	●	●
exportDeclaration	●	●	○
exportDefaultDeclaration	●	●	●

Table 5.2 Export-import compatibility matrix with exports grouped by their isomorphism

5.1.8 Implementing the Interconnection Algorithms

After thoroughly inspecting the ASG signatures of the numerous export and import cases for minimising the number of algorithms to be implemented, actually implementing the algorithms was straightforward. In this section, I will not present all combinations in detail. Instead, I describe the general steps of the interconnection process, and I provide a complete example with one concrete combination. In the Appendix, all export-import case combinations are listed with their interconnection algorithms.

The steps of the interconnection process in general can be described as follows:

1. Match each to-be-exported entities of the exporter module with strictly unique patterns containing all necessary identifiers and information for the export.
2. Match each to-be-imported entities of the importer module with strictly unique patterns containing all necessary identifiers and information for the import.
3. Perform interconnections between the exporter module and the importer module by finding corresponding entities in the two modules based on identifiers like names and/or default export/import bindings.
4. Clean the graph, so it will not contain duplicate nodes or edges after the interconnection process.

```
// exporter.js
let name1 = "name1Value";
export { name1 };

// importer.js
import { name1 as importedName1 } from "exporter";
```

Figure 5.5 Modules for demonstrating the *exportName–importAlias* combination

I chose the fully detailed combination to be the *exportName–importAlias*. The *exportName* case is in the `exporter` module, the *importAlias* case is in the `importer` module. Figure 5.5 shows the source code of the two modules.

Figure 5.6 displays the process of interconnecting the `exporter` module's graph with the `importer` module's graph along the *exportName–importAlias* case combination. The following steps are performed on the ASGs of the modules:

1. Find the exported `Variable` with its `Declaration` in the `exporter` module marked with blue colour. The full matched pattern is indicated with thicker outlines.
2. Find the imported `Variable` with its `BindingIdentifier` and its `Declaration` in the `importer` module marked with crimson colour. The full matched pattern is indicated with thicker outlines.
3. Check if the `Import` node's `moduleSpecifier` attribute is equal to the exporter module's name, which is currently `exporter`.
4. Check if the `name` attribute of the `IdentifierExpression` node (connecting to the `ExportLocalSpecifier` node) is equal to the `ImportSpecifier` node's `name` attribute. In this particular *importAlias* case, checking `ImportSpecifier` node's `name` attribute instead of the imported `Variable` node's `name` attribute provides the support for the *aliased* import.
5. Create a `declarations` edge from the imported `Variable` node to the exported `Declaration`. This is indicated with a thick black outline.
6. Create a `node` edge from the exported `Declaration` node to the imported variable's `BindingIdentifier` node. This is indicated with a thick black outline.
7. Delete the original `Declaration` node of the imported variable with its edges.[1] These are indicated with dashed outlines.

These steps are translated to Cypher, and sent to the database. Each export-import combination featured in Table 5.2 has a separate Cypher query. As these *export-import interconnection queries* are *independent* from each other — they do not modify the others' results in any way — they can be executed in any order. The queries are also *idempotent*: they can be re-executed arbitrarily many times without different outcomes on the same dataset.

Figure 5.7 presents the full Cypher query of the *exportName–importAlias* combination. The query contains a node with a label that is not displayed in the visualised graph: `CompilationUnit`. At translating the modules into ASGs, Codemodel-Rifle creates a node with the label `CompilationUnit` for each distinct source file. Each module's all graph nodes are connected to the module's `CompilationUnit` node. The node also stores information about the parsed module's file path. As displaying the `CompilationUnit` nodes with all their connections would make the graph very dense, they are omitted.

[1] This step does not cause loss of information: the graph still contains the information that the variable was *imported*.

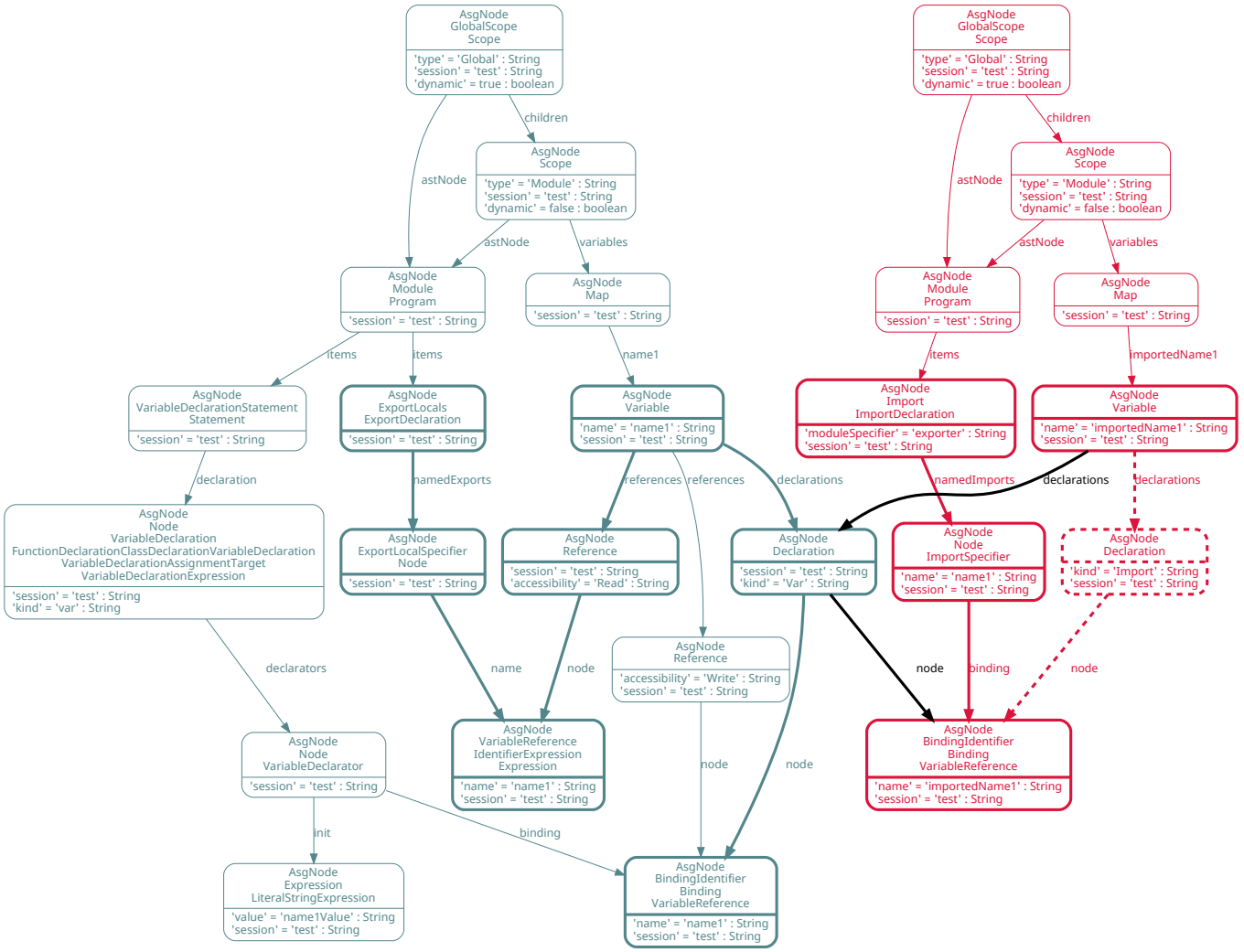

Figure 5.6 Interconnecting the `exporter` module with the `importer` module in the export-import combination *exportName–importAlias*

```
MATCH
// exporter.js: let name1 = "name1Value"; export { name1 };
   (exporter:CompilationUnit)-[:contains]->(:ExportLocals)
      -[:namedExports]->(:ExportLocalSpecifier)
      -[:name]->(exportBindingIdentifier:IdentifierExpression)
      <-[:node]-(:Reference)
      <-[:references]-(:Variable)
      -[:declarations]->(declarationToMerge:Declaration)
      -[:node]->(:BindingIdentifier),

// importer.js: import { name1 as importedName1 } from "exporter";
   (importer:CompilationUnit)-[:contains]->(import:Import)
      -[:namedImports]->(importSpecifier:ImportSpecifier)
      -[:binding]->(importBindingIdentifierToMerge:BindingIdentifier)
      <-[:node]-(declarationToDelete:Declaration)
      <-[:declarations]-(importedVariable:Variable)

   WHERE
   exporter.parsedFilePath CONTAINS import.moduleSpecifier
   AND exportBindingIdentifier.name = importSpecifier.name

MERGE
   (importedVariable)-[:declarations]->(declarationToMerge)
      -[:node]->(importBindingIdentifierToMerge)

DETACH DELETE
   declarationToDelete
```

Figure 5.7 The Cypher query interconnecting the *exportName–importAlias* combination

5.2 Simple Analyses by Pattern Matching

In the Codemodel-Rifle framework, analyses are basically Cypher queries. If a defect's pattern in the Abstract Semantic Graph can be expressed with a Cypher query, it can be detected by the framework. This section details those analyses I implemented for Codemodel-Rifle, which use only pattern matching and do not require to alter the graph.

I developed the analyses by the process I presented in Section 4.2. After visualising the defect's pattern with Codemodel-Visualization, I created the description of the defect by implementing a Cypher query for matching its pattern in the ASG model. The results of the analyses are returned as strings containing defect properties, as described in Section 4.2.3.

5.2.1 Uninitialised Variables

A variable is uninitialised if it was declared but had no value assigned. In most programming languages, uninitialised variables do have *some* value, but it is usually unpredictable *memory garbage* originating from prior values stored at the variable's memory location.

Contrarily in JavaScript, uninitialised variables do not contain random memory garbage. A method or statement evaluating a variable that has not been assigned a value returns `undefined`; a primitive value and also a primitive type of JavaScript. Uninitialised variables are of type `undefined` with the value `undefined`. Per se, uninitialised variables are not defects, but if an uninitialised variable is used without checking whether it is `undefined`, it can break code execution in several ways: e.g. making the result of the evaluating expression `undefined` too, or throwing a `ReferenceError`.

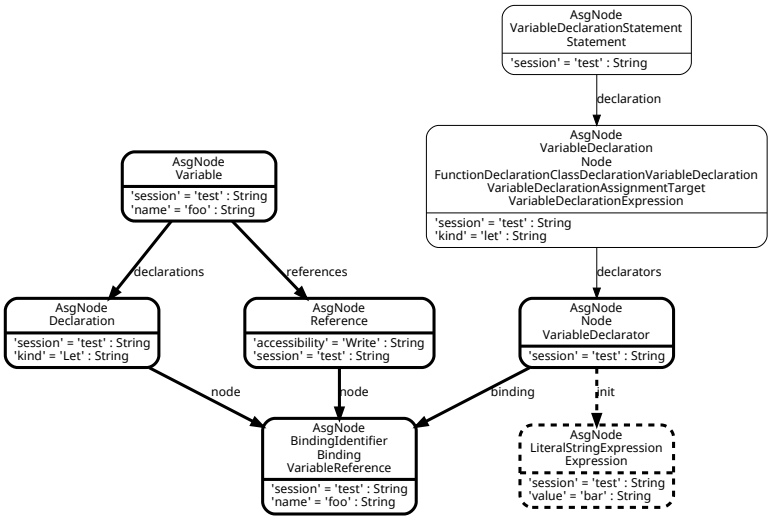

Figure 5.8 Matching the `nonInitialisedVariable` analysis pattern

Regarding uninitialised variables, my analysis in Codemodel-Rifle reports if a variable was not explicitly initialised with an assignment expression.[1] ASG-semantically, this means verifying that the variable's `VariableDeclarator` node has no `init` relationship. Figure 5.8 presents a partial ASG demonstrating how an uninitialised variable is revealed. The nodes and edges with thicker outlines are members of the pattern matching expression, the dashed outlines represent entities being checked for existence. The source code of the analysis is available in the Appendix.

5.2.2 Globally Unused Exports

The ECMAScript module system provides a practical solution for keeping code bases organised: logically separated, but practically cooperating software components can be implemented. Exporting only particular entities from a module allows to hide several sensitive information from the outside, such as internal functionality and implementation details, or even security-related specialities. Thus, a best practice is to only export what is explicitly intended to be public, and keep everything else private.

My analysis for detecting unused exports report if an entity is exported, but never imported to any other module. It is based on the semantics of the module interconnections described in Section 5.1.

Figure 5.9 presents a partial ASG demonstrating how an unused export is revealed. The exporter module's graph is indicated with blue colour, the importer module's graph is indicated with crimson colour. The nodes and edges with thicker outlines are members of the pattern matching expression, the dashed outlines represent entities being checked for existence. The source code of the analysis is available in the Appendix.

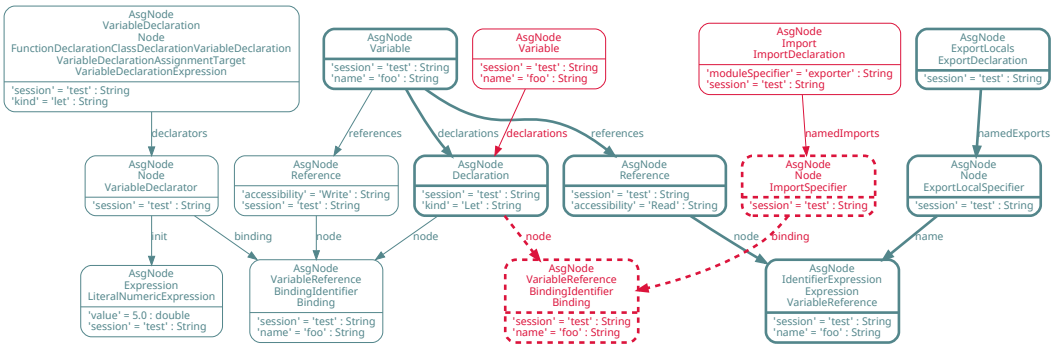

Figure 5.9 Matching the `unusedExport_exportName` analysis pattern

[1] The analysis only covers unconditional cases, reporting results of conditional value assignments is currently not supported.

5.2.3 Division By Zero (restricted)

Division by zero is one of the most basic software defects. JavaScript usually does not throw an error if it evaluates such expressions, but returns `undefined`, `NaN` or `Infinity` instead, depending on the environment and the runtime. As stated before, this can break program execution in several ways.

Detecting a division by zero scenario generally is rather challenging by using only static tools. As the right-hand operator of a division expression can be a variable, whose value can be anything — even originate from several other variables —, it needs much more effort than simple pattern matching. Detecting such *transitive* division by zero cases is the subject of the next section, involving the Qualifier System.

However, finding division expressions in the ASG, where the right-hand operator is a numeric literal with the value zero is not complicated. My analysis for this *restricted* case reports such division by zero defects by simple graph pattern matching.

Figure 5.10 presents a partial ASG demonstrating how a division by zero defect is revealed when zero is a numeric literal. The nodes and edges with thicker outlines are members of the pattern matching expression, and the `'value'` property of the `LiteralNumericExpression` is checked if it equals zero.

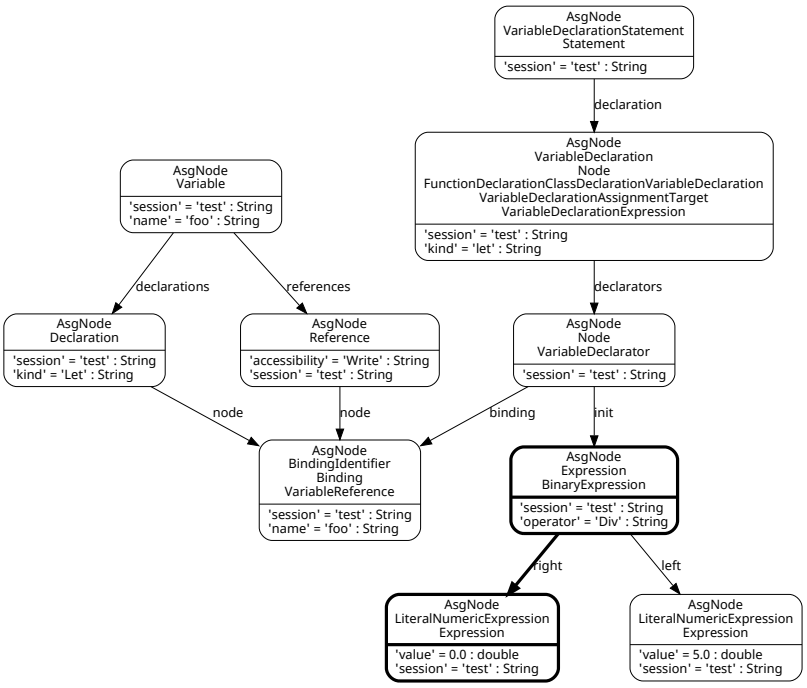

Figure 5.10 Matching the `divisionByZero-literal` analysis pattern

5.2.4 Misuse of Negative Integers as Function Arguments (restricted)

Generally used functions in JavaScript's `Math` library do not support complex numbers. Therefore, if a developer supplies a negative numeric value to a function like `Math.sqrt()` or `Math.log()`, the expression will return `NaN` or `undefined`, depending on the environment and the runtime.

My analysis for detecting the misuse of negative integers as function arguments reports if the argument of a `log()` or a `sqrt()` call is a negative numeric literal.[1]

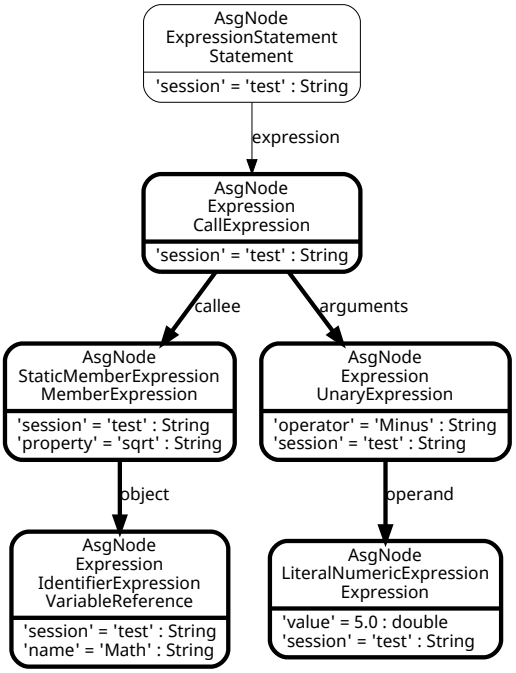

Figure 5.11 Matching the `squareRootNegativeArgument-literal` analysis pattern

Figure 5.11 presents a partial ASG demonstrating how a square root called with a negative argument defect is revealed when the argument is a numeric literal. The nodes and edges with thicker outlines are members of the pattern matching expression. The `'name'` property of the `VariableReference` connected to the `StaticMemberExpression` is checked whether it is `Math`, the `'property'` property of the `StaticMemberExpression` is checked whether it is `sqrt`, and the `'operator'` property of the `UnaryExpression` connected to the `LiteralNumericExpression` is checked whether it is `Minus`. The source code of the analysis is available in the Appendix.

[1] This analysis does not cover transitive cases, where the function argument is a variable. That is the subject of the next subsection.

5.3 Complex Analyses with the Qualifier System

Some defects are more general than to present their patterns in an intact graph directly. Detecting complex errors like these may involve to deduce variable and function return values, and it may require to manipulate the graph to dredge defect patterns for matching. Implementing complex analyses for these defects involved the creation of the Qualifier System, a generic graph constraint propagation strategy for revealing — otherwise generally unmatchable — *transitive* defect patterns. This section details the analyses I implemented for Codemodel-Rifle involving extensive graph manipulations, using the Qualifier System.

5.3.1 Transitive Defects

In this thesis, the term transitive defect is used as follows. A software defect is considered *transitive*, if its effect propagates through multiple variable value assignments and/or function calls. Patterns of transitive defects generally can not be directly matched in the ASG, because the graph pattern of such defects — spanning an indeterminate number of functions or variable assignments — can not be described by one general pattern description. But, patterns of transitive defects can be deduced in the ASG by following their propagation, and marking the intermediate nodes with constraints.

Demonstratively, the running example presented in Chapter 2 contains a transitive division by zero defect. In the example's `exporter` module, there is a variable given the value zero, then the variable is nested into several levels of variable assignments and function return expressions, finally into the `exporter` module's default function export. The exported function will return zero, too. After the example's `importer` module imports the default export of `exporter`, it divides numeric literal 5 with the return value of the imported function, practically by zero. The defect is transitive in the meaning that the zero is not a numeric literal 0, which could be revealed easily by simple pattern matching. Instead, that zero comes from nested variable assignments and functions — it *transits* along variable assignments and functions. This transitivity can be deduced by the Qualifier System.

This deduction of values is similar to the approach of data-flow analysis. By propagating *qualifiers* in the ASG node-by-node, until the system reaches a fixpoint (where no further propagation is possible), basically the nodes' local data-flow equations are solved.

Figure 5.12 presents the propagation of the running example's transitive division by zero defect in the `exporter` module's partial ASG. The graph pseudo-node marked with crimson filling is the `importer` module. The node with blue filling is the `LiteralNumericExpression`, which finally causes the `importer` module's `function defaultName()` to return 0. The propagation of the transitive defect starts at the assignment of the literal zero (the blue node), exits the `exporter` module, then — as the two related modules' graphs are interconnected with each other — enters and ends in the `importer` module.

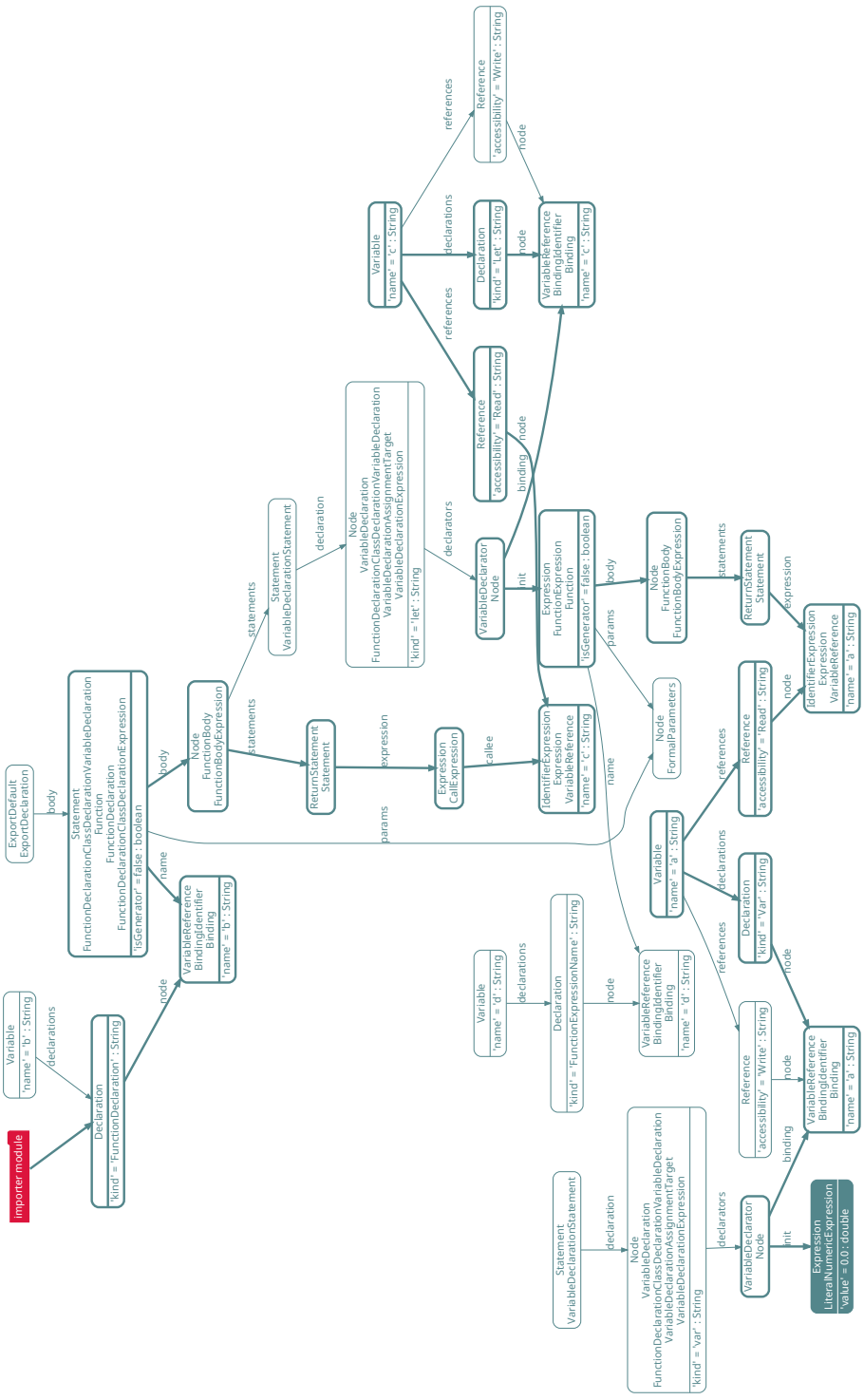

Figure 5.12 The transition path of the running example's division by zero defect

5.3.2 Introduction: The Qualifier System

The Qualifier System is the generalisation of Dániel Stein's Type System [7]. The system assigns well-defined constraints to ASG nodes satisfying certain criteria, then propagates these constraints through the graph by certain rules. These constraints — the *qualifiers* — are *instances* of the Qualifier System: they are represented by graph nodes, connected to a central `QualifierSystem` collector node with an `:_instance` relationship.

The graph manipulations of the Qualifier System are performed:

- **after** the analysed repository is imported/synchronised, the source files' ASGs are constructed, and the related modules' graphs are interconnected to each other,
- **before** the defect patterns of the analyses are matched.

This allows to first manipulate the graph in several ways by assigning and propagating the qualifiers, and then build pattern matching expressions specifically for the qualifier instances. This way, *transitive* defects — like the running example of Chapter 2, where the division by zero is passed along multiple functions and variable assignments — can be detected by deducing the transitions by the qualifiers.

The basic operation of the system is the following. In the enumeration below, the **phase description** is followed by a concrete demonstrative case based on the running example presented in Chapter 2.

1. **Initialise the Qualifier System. Create the `QualifierSystem` collector node and the qualifier instance nodes.** In the running example, the analysis is based on propagating the `EqualsZero` qualifier instance.
2. **Identify all literals which can be directly marked with a qualifier instance. Connect them to the right qualifier instance with the edge `:_qualifier`.** In the running example, the `LiteralNumericExpression` node of the `var a = 0;` variable declaration statement is connected to the `EqualsZero` instance.
3. **Connect adjacent nodes to the same qualifier if they satisfy propagation criteria.** In the running example, the `VariableDeclarator` node is also connected to the `EqualsZero` qualifier instance, because it satisfies the propagation criterion of being connected to a `LiteralNumericExpression` by an `init` relationship.
4. **Repeat the previous step until there is no modification in the graph.**[1] In the running example — after the propagation finished — the `EqualsZero` qualifier will be connected to every entity that is caused to be zero because of the `var a = 0;` assignment, including the exported `function b()`, and thus the imported `defaultName()` function — which is the right-hand side value of the division. Therefore, the transitive division by zero defect can be detected by simply checking whether the right-hand side of the expression has an `EqualsZero` qualifier.

[1] There has to be a stop condition for unintentional infinite loops.

If the propagation of the Qualifier System finishes, then all transitive defects are *closed* in the meaning that every spread of the defect is marked with a qualifier, so it can be easily detected by a simple pattern matching expression. The following subsections present examples for detecting transitive defects with the Qualifier System.

5.3.3 The Running Example's Division By Zero (transitive)

Detecting a transitive division by zero defect — when the zero expression is not a numeric literal 0, but a variable or a function providing the value zero — requires the right-hand value of the division expression to be deduced. If this value comes from several nested variable assignments and functions, like presented in the running example, the originating value has to be found: a variable assignment with a numeric literal.

Finding a variable assignment, where a numeric literal is the assigned value, can be carried out by simple pattern matching. If this value equals zero, its graph node, the `LiteralNumericExpression` is qualified by using an `EqualsZero` qualifier instance. After this assignment has been qualified, its adjacent nodes are inspected whether they can be also qualified, according to the propagation rules[1] of the Qualifier System. In the current case, after the `LiteralNumericExpression`, its only adjacent ASG node, the `VariableDeclaration` gets qualified too by `EqualsZero`. This is valid, because the `init` edge connecting the two nodes is allowed to propagate an `EqualsZero` qualifier instance. After the `VariableDeclaration` has been qualified, its adjacent nodes are inspected whether they can be qualified, and the propagation algorithm continues until there are no more paths the qualifier instance could be propagated further on.

In the running example, the propagation of the `EqualsZero` qualifier instance stops at the right-hand value of the importer module's division expression. Semantically, the meaning that the right-hand value of the division expression is qualified with an `EqualsZero` instance: the division's right-hand side value (the value returned by the `defaultName()` function) has been successfully deduced, and found to be equal to zero.

After the propagation of qualifiers, a simple pattern matching query checks if there are any right-hand values of division expressions qualified by `EqualsZero`. If yes, then a division by zero is performed, so it is reported to the developer.

Figure 5.13 presents the propagation of the `EqualsZero` qualifier, regarding the running example's transitive division by zero defect. Although the figure is analogous to Figure 5.12 by showing the same path, Figure 5.12 shows the propagation path of the *defect*, while Figure 5.13 shows the propagation of the `EqualsZero` *qualifier*, following the defect.

[1] The propagation rules or propagation criteria of Codemodel-Rifle's Qualifier System is implemented as pattern matching queries. Let N_1 be qualified by a qualifier instance, and let N_2 be an adjacent node of N_1 via the relationship R. If R is member of the set of qualifier propagator relationships, then N_2 gets qualified too.

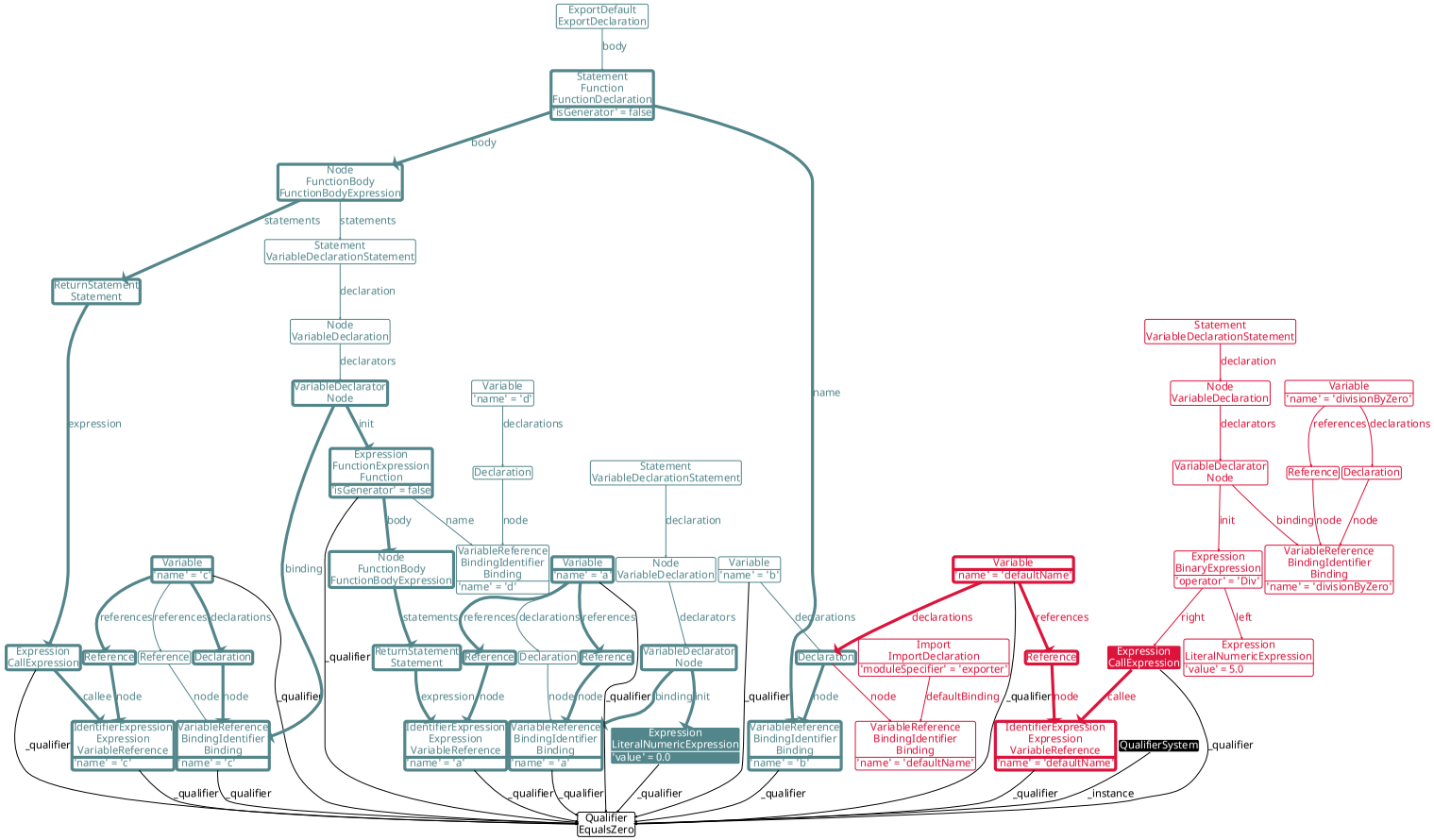

Figure 5.13 The propagation path of the `EqualsZero` qualifier instance at analysing the running example

5.3.4 Misuse of Negative Integers as Function Arguments (transitive)

Detecting the misuse of negative function arguments in trasitive cases — when the argument is not a numeric literal, but a variable, whose value can be anything — needs the same value deduction, as detecting a transitive division by zero defect. The difference is the usage of qualifiers: in this case, a `NegativeNumeric` qualifier[1] is utilised instead of an `EqualsZero`.

The `NegativeNumeric` qualifier is propagated through the graph along the variable assignments and function return expressions, similarly to the `EqualsZero`, with the same stop condition. After the propagation of the qualifier, a simple pattern matching query checks if there are any `Math.sqrt()` or `Math.log()` function calls with their arguments marked as `NegativeNumeric`. If yes, then it is a misuse of negative function argument defect, so it is reported to the developer.

5.3.5 Unreachable Code Caused by Exception (transitive)

The exception handling of the ECMAScript language has the same semantics as Java. If exceptions thrown with the **`throw`** keyword are surrounded by a **`try..catch`** block context, they get caught, and they can be processed or thrown further.

```
function throwsException() {
   return function () {
      throw new SQLException;
   };
}
let a = throwsException;
let b = function () {
   return function () {
      let c = throwsException();
      return 42;
   }
};
console.log(b());
console.log(42);
```

Figure 5.14 Deeply nested exception in ECMAScript

An exception halts the execution of the program, and yields it to the exception handler —

[1] The qualifiers' names can be arbitrary, the semantic design of a qualifier-based analysis requires no predefined naming system. The name of a qualifier instance matters only at implementing the pattern matching algorithms for the analyses: e.g. an `EqualsZero` qualifier is only an error if it is connected to the right-hand side of a division expression.

the catch block —, at least if there is such handler implemented by the developer. Source code following an exception throwing statement is not executed, therefore it is unreachable or dead code. Most static analysis tools detect dead code caused by exceptions, but only very shallowly.

Figure 5.14 presents a program with an exception nested into several levels of functions. The program — instead of logging 42 to the console — will halt, since calling `function b()` will eventually cause an `SQLException` to be thrown.

By introducing the `ExceptionThrown` qualifier instance into the Qualifier System, the propagation path of an exception can be tracked. At analysing the code of Figure 5.14, first the **`throw new`** `SQLException` statement gets marked by the `ExceptionThrown` qualifier. Then after several steps of propagation, `function b()` also gets marked, therefore it can be easily found by pattern matching.

My analysis for exception-caused unreachable code reports that:

- an exception is thrown at the execution of the statement `console.log(b())`, and
- because of the exception, the statement `console.log(42)` will never be executed.

5.4 Limitations of the Analyses

Though the graph-based static analysis approach is a promising novelty from several aspects, my analyses presented in this thesis are limited in many ways. Implementing ECMAScript module interconnections and introducing the Qualifier System were both relevant acts, but they are only supporting elements of the analyses themselves.

Codemodel-Rifle's variable and function value deductions are primitive: no arithmetic operations are supported, the framework tracks only raw, unmodified values. Conditional cases are not covered either: an exception gets detected only if it is unconditionally thrown.

Implementing sound and complete analyses with the Codemodel-Rifle framework is not the subject of this thesis. But — building upon the work of Dániel Stein — the first steps have been made to create a versatile graph-based static analysis tool capable of inspecting enterprise-grade source code repositories coherently.

Chapter 6

Evaluation of Performance

In this chapter, I evaluate the framework's performance by measuring the duration of analysing several source code repositories.

6.1 Evaluation Environment

6.1.1 Computer Configuration

The measurements were performed on my computer for the sake of simplicity. During a measurement session, my computer was plugged in, it was configured to utilise its full performance, and only those software were running, which were explicitly necessary for the measurements. Each measurement session was preceded by a full system restart.

The major points of the my computer's configuration are the following:

- **Brand and model:** Apple MacBook Pro, Mid-2014, 13 inches;
- **CPU:** Intel Core i5 (4278U), 2.6 GHz;
- **Memory:** 8 GB 1600 MHz DDR3 RAM;
- **Storage:** 250 GB SSD.

6.1.2 Software Configuration

As currently the Codemodel-Rifle framework does not have any interface to interact with, the measurements were performed as per-repository unit tests with logging test results onto the console.

- **Runtime:** JetBrains IntelliJ IDEA Ultimate 2016.3.4
- **Java Runtime Environment:** 1.8.0_112-release-408-b6 x86_64
- **Java Virtual Machine:** OpenJDK 64-Bit Server VM by JetBrains s.r.o (initial memory allocation pool: 4 GB, maximum memory allocation pool: 8 GB)

- **Database:** Neo4j Community Edition 3.1.3 Server (initial heap size: 4 GB, maximum heap size: 8 GB, page cache size: 8 GB, transaction log retention policy: 1 day)
- **Database driver:** Neo4j Bolt driver for Java 1.1.1

For better performance, a database index was defined in Neo4j for the `'id'` property of all nodes labeled with `'AsgNode'` (practically all nodes created by Codemodel-Rifle).

6.2 Measurement Goals and Methods

6.2.1 Selection Criteria of the Analysed Source Code Repositories

The evaluation was performed on popular open-source JavaScript code repositories randomly chosen and downloaded from GitHub, and on a closed-source, security-oriented product from Tresorit, called `webclient`. The altogether 40 repositories (listed in the Appendix) differ in size, in the number of lines of code, and in the number of source files.

6.2.2 Key Performance Indices

The goal of the performance evaluation was to determine the time characteristics of the extended Codemodel-Rifle framework, especially the implemented analyses. Based on the production operation of the framework detailed in the introduction of Chapter 5, the following Key Performance Indices have been determined to be measured.

- **The duration of synchronisation:** the time period between starting the importing process of a code repository (excl. finding all `.js` files, and reading the contents of the source files) and saving the last module's last `AsgNode` into the database.
- **The duration of interconnection:** this time period encompasses searching for semantically valid interconnections amongst related modules' property graphs, and actually performing the interconnections.
- **The duration of running the Qualifier System:** this time period encompasses initialising the Qualifier System, and propagating the qualifiers.
- **The duration of performing the analyses:** this time period encompasses trying to match all predefined analysis patterns and logging analysis results.
- **The total duration of the analysis process:** the calculated sum of the above four.

Besides the Key Performance Indices, the number of graph nodes and relationships created during the synchronisation of a repository is also recorded.

6.2.3 Process of Measurement

All analysed code repositories were measured four times in a session in order to avoid biases caused by the environment. Each session was preceded by a full system restart. The final measurement results of a repository are averaged from the four different values.

6.3 Measurement Results

In this section, I present and evaluate the measurement results of the aforementioned Key Performance Indices.

6.3.1 Synchronisation

At first, a repository needs to be synchronised into Codemodel-Rifle. In this phase, the source code files of the repository get translated to distinct, per-module property graphs.

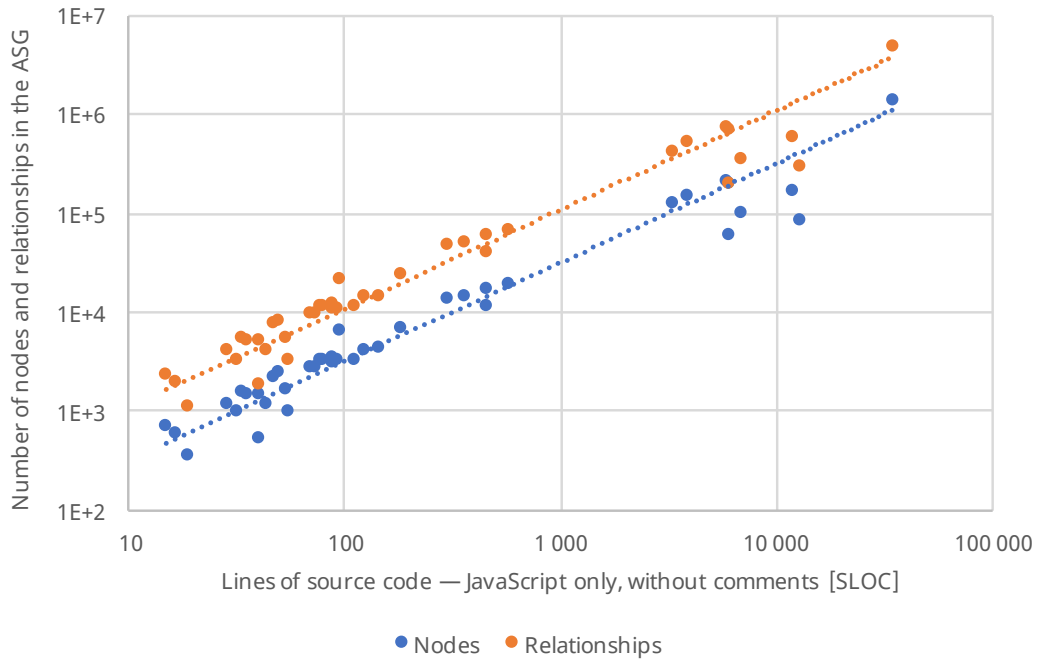

Figure 6.1 The characteristics of synchronising repositories into Codemodel-Rifle

There are many coding styles and conventions, and the contents of the source files can vary from per-line exported configuration constants to program codes without physical line breaks. Nevertheless, there is a linear relationship between the number of code lines and the number of created ASG nodes and relationships in the analysed repositories.

Figure 6.1 presents the correlation of the source lines of code (SLOC) and the number of ASG nodes and relationships created during synchronising the code bases into Codemodel-Rifle. In terms of SLOC, the smallest repository imported was `initialstate/silent-doorbell` with 15 lines of code (686 nodes and 2,306 relationships), while the largest was `tresorit/webclient` with 34,546 lines of code (1,346,776 nodes and 4,576,319 relationships).

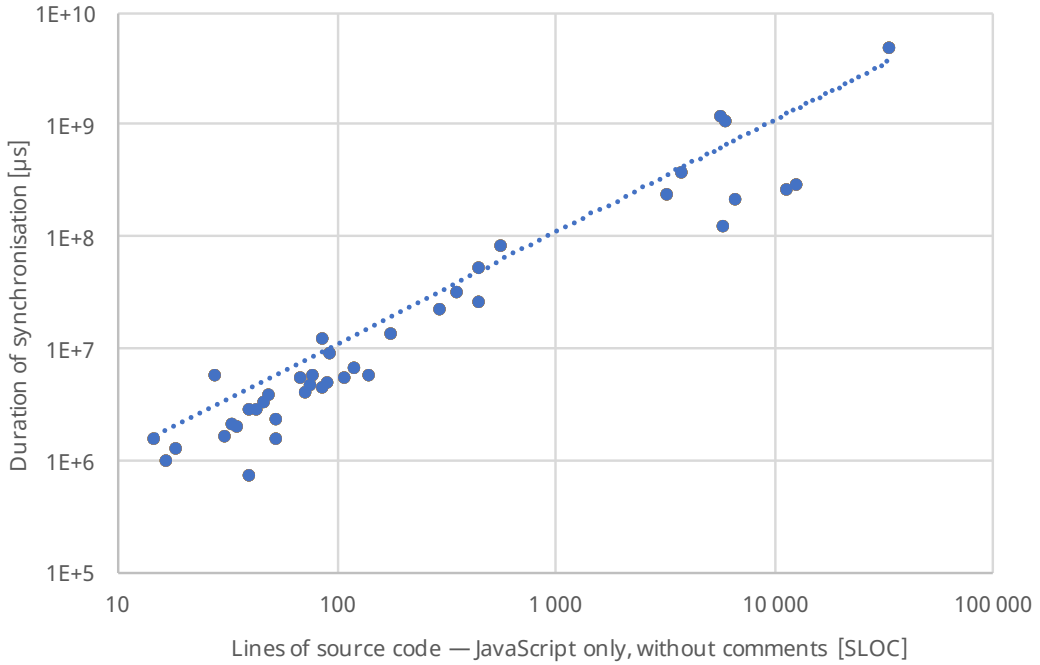

Figure 6.2 The characteristics of synchronising repositories into Codemodel-Rifle

As for the duration of the synchronisation phase, the correlation is linear in this case too. The bigger the repository is, the more time it consumes to synchronise the code base into Codemodel-Rifle (as more graph nodes and relationships need to be created). Figure 6.2 shows the correlation between the duration of synchronisation and the repository size. The shortest synchronisation duration belongs to the `facundoolano/promise-log` repository with altogether 41 SLOC in 1 module, it was imported into the framework in around 7 milliseconds. The import of the largest repository, `tresorit/webclient` with 34,546 SLOC in 609 modules, took about 78 minutes.

A more evident approach is to inspect the relationship between the duration of synchronisation and the repository size measured with the number of created graph nodes and relationships. As Figure 6.3 shows, the relationship between these two values is also linear. Practically, this means the underlying graph database, Neo4j is able to handle very large volumes of node and relationship creations linearly. By using the database index on the `'id'` attribute, the nodes are retrieved faster at creating the relationships.

An important thing to consider regarding Neo4j is transaction granularity. According to my experience with the production server run on my laptop with the configuration mentioned earlier, the graph database tends to freeze if a very large amount of queries are committed

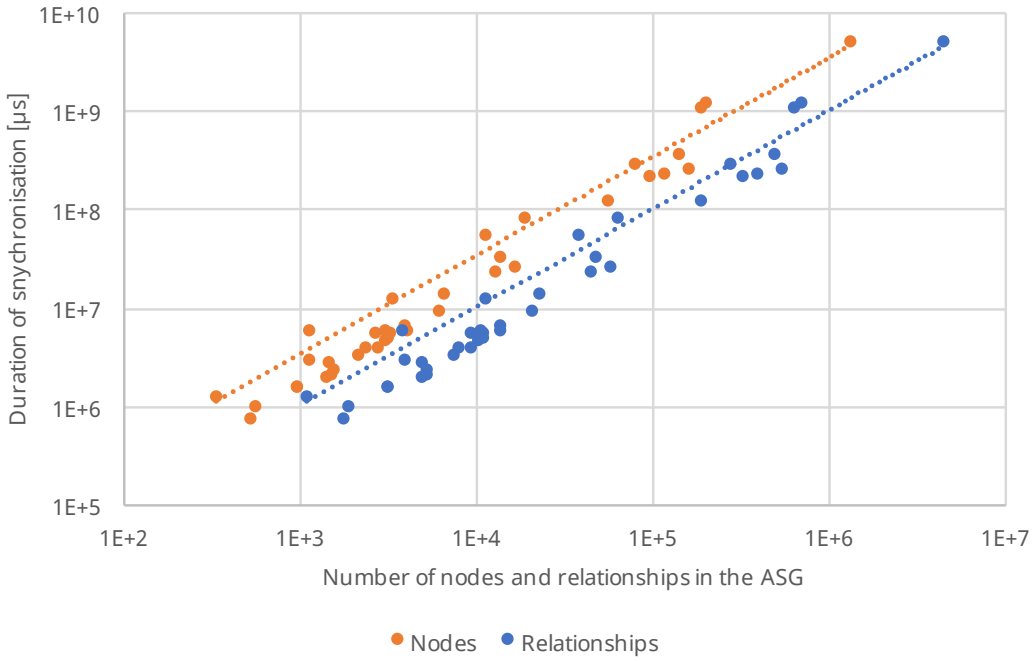

Figure 6.3 Synchronising repositories into Codemodel-Rifle

within one transaction. In the framework's current implementation, it is not possible to configure the maximum number of queries committed in one turn. It is hard-coded into the framework to handle each file in a separate transaction as a whole, to preserve at least file-level consistency. The transactions of synchronising larger files (several hundred kilobytes, several thousand SLOC) should be configurably split into multiple smaller ones in the future, in order to ensure solid operation.

6.3.2 Interconnection

In theory, interconnecting ECMAScript modules is a very slow operation. The 13 implemented interconnection algorithms are run one by one, each matching two complex graph patterns for finding the compatible export and import cases.

In practice, however, the interconnection phase was the fastest of all. Even at the largest analysed repository, `tresorit/webclient`, having 609 distinct ECMAScript modules, 1,346,776 graph nodes and 4,576,319 graph relationships, the interconnection phase took less than 30 seconds. At smaller repositories, or at repositories having only one module, the duration the interconnection phase was a small, sub-second value.

Regarding the characteristics of the export-import interconnections, no explicit relationship can be determined between the repository size or the number of modules and the duration of the interconnection phase. This is understandable: altering the number of distinct modules in an ECMAScript project does not necessarily cause the number of *related* modules to change.

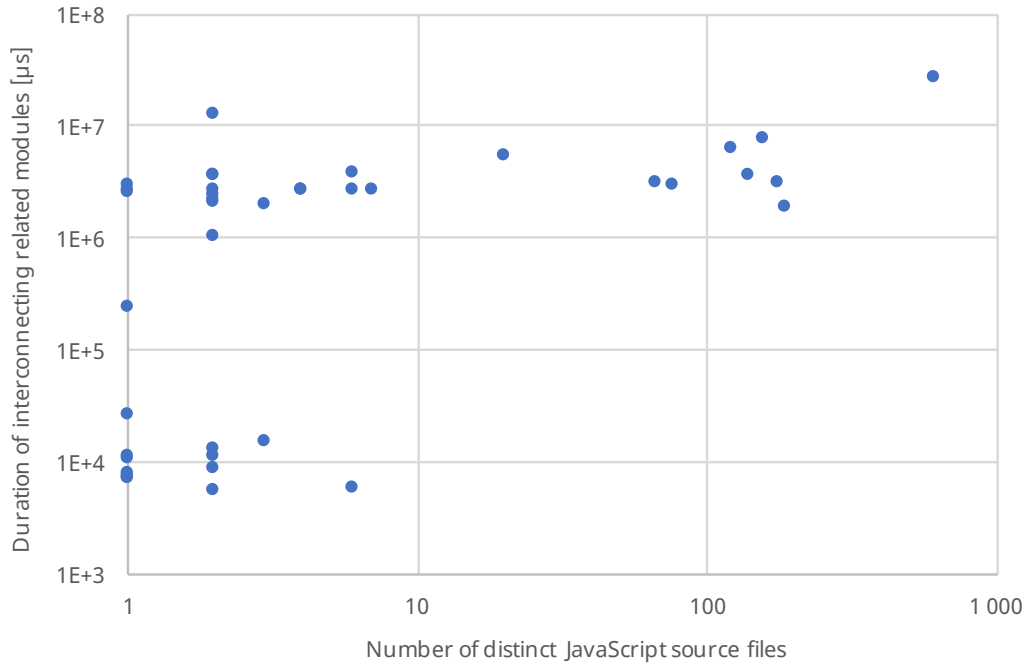

Figure 6.4 The characteristics of interconnecting related modules

Figure 6.4 shows the duration of interconnecting related modules in the light of the number of distinct modules in the repository. The figure shows that no relationship can be determined between the two values. Since the design of modularisation varies for every project, simply the number of distinct modules does not indicate how many of those modules are related to each other.

6.3.3 The Qualifier System

Spreading qualifiers along possible propagation paths in the Abstract Semantic Graph is a long process. In each step, a particular qualifier can traverse only one relationship at a time — similarly to solving data-flow equations locally, based on the solution of the preceding equation. In larger graphs containing long transitive paths, producing a full transitive closure can involve many steps.

The more “entry points” has a particular software for the Qualifier System (e.g. literals and **`throw`** statements can be entry points: they are the first to be marked with qualifiers at the initialisation of the Qualifier System), the more propagation paths need to be closed up. The bigger the repository is, the more likely it is to contain such entry points.

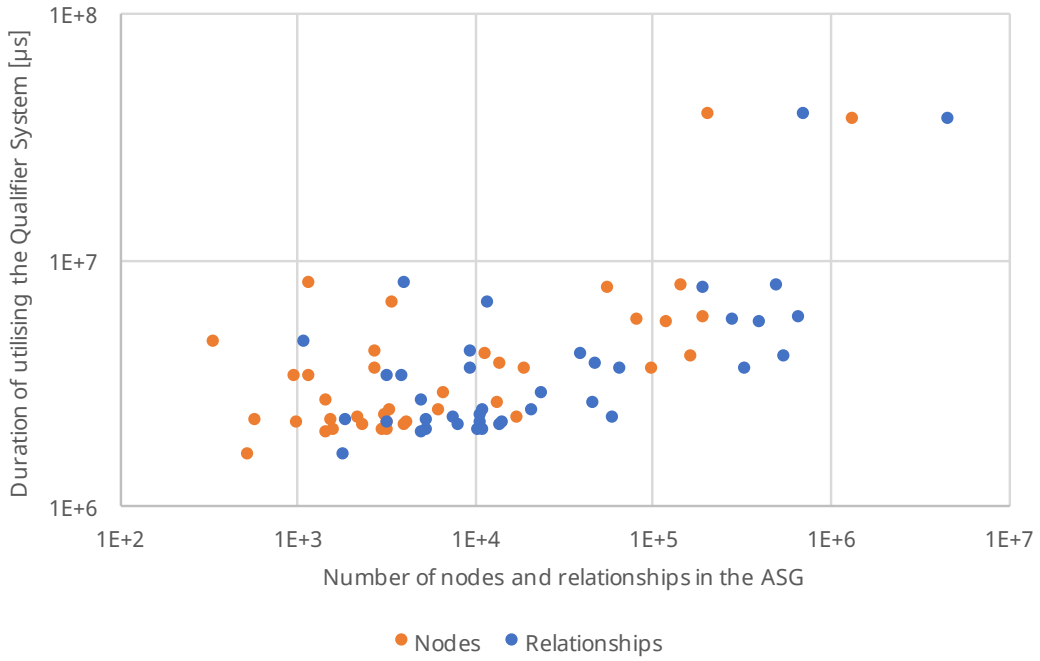

Figure 6.5 The characteristics of running the Qualifier System

Figure 6.5 presents the characteristics of the Qualifier System. The relationship between the duration of running the Qualifier System and the number of graph nodes and relationships is not unequivocal. The two outlier values — 36 seconds for the `tresorit/webclient` and 38 seconds for the `alvin198761/web-os` — can be explained by either the large number of transitive defects, or simply the size of the repository. It is worth to mention though, that while the `web-os` contains only 5,922 SLOC, the `webclient` has 34,546 SLOC. The similar duration of running the Qualifier System with this difference could imply that the `web-os` contains much more transitive defect paths to propagate qualifiers on.

6.3.4 Analysis

Importing, interconnecting, and applying the Qualifier System are only preparatory steps for running the actual analyses. All analyses involve matching complex patterns, even the ones using the results of the Qualifier System: besides qualifiers, several other attributes

need to be queried for returning a complete set of analysis results, like code location information, and the containing module's file path.

Figure 6.6 presents the characteristics of the analyses. The duration of the analysis phase seemingly does not have any relationship with the number of graph nodes and relationships of a code repository. This is plausible, since the number of defects in a repository does not necessarily depend on the code base's size.

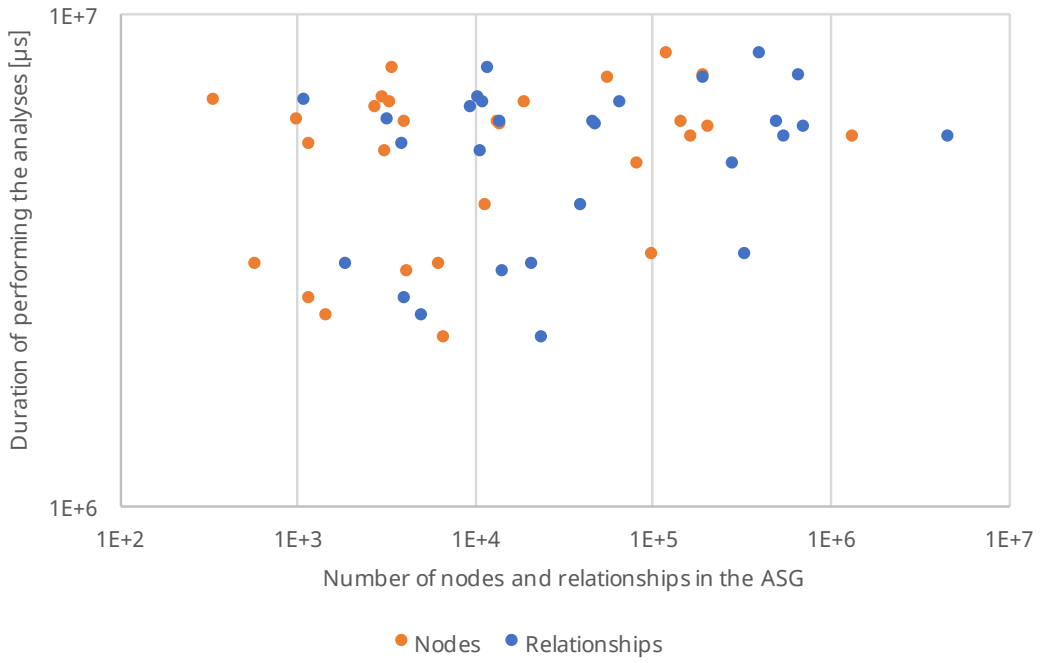

Figure 6.6 The characteristics of performing the analyses

6.3.5 Total Duration of the Analysis Process

The total duration of analysing a repository seems to be in linear relationship with the repository's size. Figure 6.7 and Figure 6.8 present that both measured in SLOC and in the number of graph nodes and relationships, the correlation is linear: the bigger the source code repository is, the more time it takes to perform a complete analysis process on it.

It is interesting to notice, how the proportion of the synchronisation phase's duration increases with the size of the repository (see the Appendix for details). While for the smaller repositories, the import makes up only 30–40% of the total duration, for the larger repositories, it increases to 80–90%. For the `tresorit/webclient` repository, the synchronisation phase alone makes up 98% of the duration of the total analysis process.

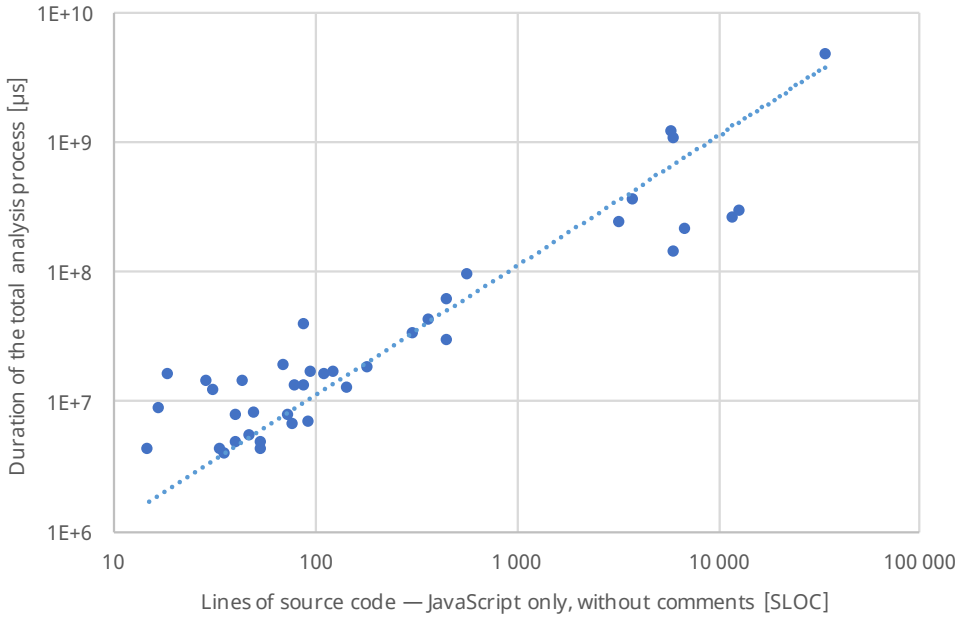

Figure 6.7 The characteristics of the full analysis process

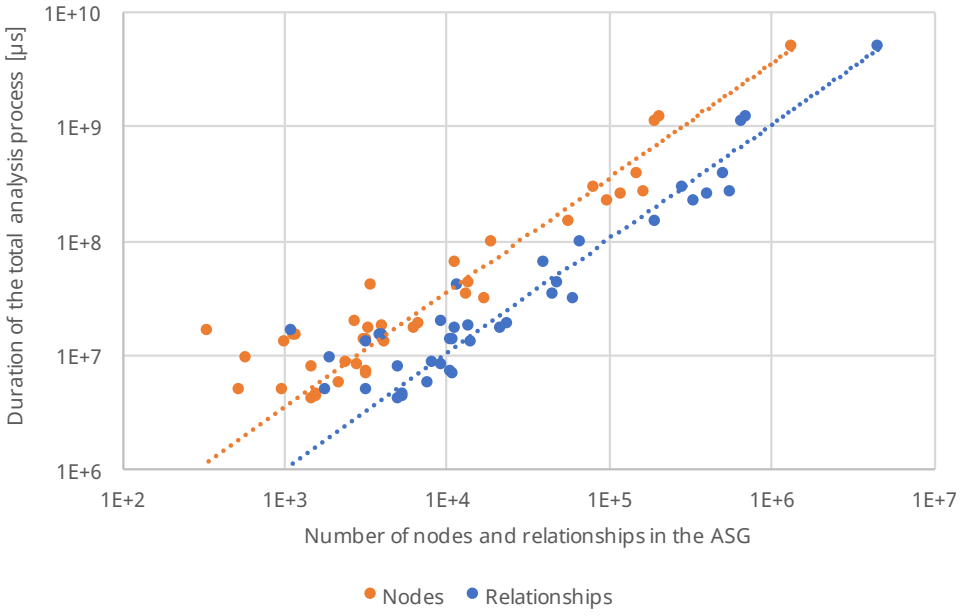

Figure 6.8 The characteristics of the full analysis process

6.4 Defects Found by the Framework

The framework detected only two types of defects in the 40 analysed repositories: 897 cases of uninitialised variables, and 134 cases of globally unused exports were found. As the analysis is neither sound, nor complete, these numbers can be inaccurate. However, I inspected a randomly chosen subset of the found defects manually, and — according to my experience — the defects were indeed present in all cases.

6.5 Threats to Validity

I designed the measurements to be as accurate and complete as possible. Nevertheless, there are factors which I could not fully control, and these may have influenced the results. In this section, I summarise the factors which could bias the measurements.

Measurements on a Consumer Laptop Since my computer runs an operating system targeted for consumer usage, it may contain software running in the background, which influence measurement factors like processor or memory usage. I tried to mitigate this by configuring the computer to utilise all resources for the measurement procedure, by running the measurements multiple times, and by analysing a larger number of code repositories independently.

Graph Query Optimisations I tried to optimise the graph queries of the interconnections and the analyses as much as I could. However, since I am not an expert in the internals of Cypher queries, it is possible that some queries can be optimised further. Therefore, the characteristics of the interconnections or the analyses may not be fully correct.

Methological Mistakes It is possible that I made other methodological mistakes at implementing the analyses or the measurements. Using a fluid, internal semantics for the interconnection of modules incorrectly can be an example of a such mistake.

Chapter 7

Conclusion and Future Work

My primary object was to extend the Codemodel-Rifle framework with analysis algorithms. To make the framework practically usable, this involved several other supporting features to be planned and implemented.

Codemodel-Rifle was rearchitectured to become modular. Therefore, by changing components if necessary, the framework can adapt to various requirements and use-cases. The software was also reworked semantically, by elaborating the capability of performing analyses on multiple modules coherently, the Qualifier System, and the analyses themselves.

Once the framework contains enough analyses, it can be a practical tool for helping developers in finding defects. By this time, utilising module interconnections and the Qualifier System, it is expressive enough to cover a large set of statically analysable use-cases.

7.1 Summary of Contributions

I contributed to the development of the framework in two ways. Scientific contributions encompass the performances regarding the analysis of ECMAScript, and the language itself. Engineering contributions cover designing the architecture of a large-scale, modular code analysis software, and implementing a proof-of-concept prototype.

7.1.1 Scientific Contributions

I have achieved the following scientific contributions:

- Defined the semantics of interconnecting multiple Abstract Semantic Graphs along the export-import statements of the ECMAScript language.
- Proposed an approach to evaluate graph-based static analyses over multiple ECMAScript modules coherently.

- Provided an extensible data model and an algorithm for analysing the data flow of ECMAScript software.

7.1.2 Engineering Contributions

I have also achieved the following engineering contributions:

- Designed a modular architecture for an analysis framework to be capable to scale and adapt to various requirements.
- Created a specialised Object-Graph Mapping layer for optimising the transformation of Abstract Syntax Trees into Abstract Semantic Graphs.
- Implemented a specialised Query Builder for the Cypher language.
- Elaborated several graph-based analyses for the ECMAScript language.

7.2 Future Work

The goal of the work described in this thesis was to extend the Codemodel-Rifle framework with analysis algorithms. By implementing several other supporting features, the scope broadened: it is now possible to analyse multiple modules coherently, and to inspect the data flow of ECMAScript software. Implementing more — and more precise — analyses, which utilise these new capabilities is a task for the future.

Further optimisations can be done at various points of the architecture. By collaborating with version-control systems like Git for file-level incremental processing, the speed of the analysis procedure can be increased significantly.

To involve Codemodel-Rifle into various software development methods, the framework should be able to communicate with other applications. Thus, the capability of producing machine-readable output is to be implemented. Also, creating plugins for continuous integration platforms would make possible to embed the framework into well-known software production architectures.

Acknowledgements

I would like to thank my supervisors Dávid Honfi and Gábor Szárnyas for their friendly advice and enthusiasm, and for being available and fully responsive at any time.

I would like to thank Dániel Stein for creating and documenting the foundations of my research, and for providing guidance and continuous support regarding the Codemodel-Rifle framework. I would also wish to express my gratitude to Ádám Lippai for his numerous valuable suggestions, and for making the `webclient` proprietary software of Tresorit available for analysis. I would also like to thank András Vörös, Kristóf Marussy, Oszkár Semeráth, and other members of the Fault Tolerant Systems Research Group for providing assistance during writing this thesis.

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

MTA–BME Lendület This work was partially supported by the MTA-BME Lendület Research Group on Cyber-Physical Systems.

References

[1] Maurice Dawson, Darrell N Burrell, Emad Rahim, and Stephen Brewster. “Integrating Software Assurance into the Software Development Life Cycle (SDLC)”. In: *Journal of Information Systems Technology and Planning* 3.6 (2010), p. 51.

[2] Gregory Tassey. “The Economic Impacts of Inadequate Infrastructure for Software Testing”. In: *National Institute of Standards and Technology, RTI Project* 7007.011 (2002).

[3] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. “Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects”. In: *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE. 2016, pp. 426–437.

[4] Martin Fowler. *Continuous Integration*. URL: `http://www.martinfowler.com/articles/continuousIntegration.html` (visited on 04/24/2017).

[5] StackOverflow. *Developer Survey Results 2016*. URL: `http://stackoverflow.com/insights/survey/2016` (visited on 04/25/2017).

[6] ECMA International. *Standard ECMA-262, 7th Edition*. URL: `https://www.ecma-international.org/publications/standards/Ecma-262.htm` (visited on 04/25/2017).

[7] Dániel Stein. “Graph-Based Source Code Analysis of JavaScript Repositories”. Master’s thesis. Budapest University of Technology and Economics, 2016.

[8] Fault Tolerant Systems Research Group (Budapest University of Technology and Economics). *Codemodel-Rifle – Graph-based incremental static analysis of ECMAScript 6 source code repositories*. URL: `https://github.com/ftsrg/codemodel-rifle` (visited on 05/01/2017).

[9] Tresorit. *End-to-End Encrypted Cloud Storage for Businesses*. URL: `https://tresorit.com` (visited on 04/25/2017).

[10] Pär Emanuelsson and Ulf Nilsson. “A comparative study of industrial static analysis tools”. In: *Electronic notes in theoretical computer science* 217 (2008), pp. 5–21.

[11] B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsborrow, N. J. Ward, and D. W. R. Marsh. “Industrial perspective on static analysis”. In: *Software Engineering Journal* 10.2 (1995), pp. 69–75. DOI: 10.1049/sej.1995.0010.

[12] Wikipedia. *List of tools for static code analysis*. URL: https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis (visited on 04/25/2017).

[13] Benjamin Livshits. “Improving software security with precise static and runtime analysis”. PhD thesis. Stanford University, 2006.

[14] Paul Anderson. “The use and limitations of static-analysis tools to improve software quality”. In: *CrossTalk: The Journal of Defense Software Engineering* 21.6 (2008), pp. 18–21.

[15] Alan Mathison Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* 2.1 (1937), pp. 230–265.

[16] David Flanagan. *JavaScript: the definitive guide*. "O’Reilly Media, Inc.", 2006.

[17] Charles Severance. “JavaScript: Designing a Language in 10 Days”. In: *Computer* 45 (2012), pp. 7–8. DOI: doi.ieeecomputersociety.org/10.1109/MC.2012.57.

[18] World Wide Web Consortium Community. *A Short History of JavaScript*. URL: https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript (visited on 04/26/2017).

[19] Node.js Foundation. *About Node.js®*. URL: https://nodejs.org/en/about (visited on 04/26/2017).

[20] npm Inc. *About npm*. URL: https://www.npmjs.com/about (visited on 04/26/2017).

[21] ECMA International. *Standard ECMA-262, 1st Edition*. URL: http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf (visited on 04/25/2017).

[22] Ralf S. Engelschall. *ECMAScript 6 — New Features: Overview & Comparison*. URL: http://es6-features.org (visited on 04/26/2017).

[23] PC Magazine. *Definition of: compiler*. URL: http://www.pcmag.com/encyclopedia/term/40105/compiler (visited on 04/26/2017).

[24] Rohit Kulkarni, Aditi Chavan, and Abhinav Hardikar. “Transpiler and it’s Advantages”. In: *(IJCSIT) International Journal of Computer Science and Information Technologies* 6.2 (2015), pp. 1629–1631. ISSN: 0975-9646.

[25] Kangax GitHub user. *ECMAScript 6 compatibility table*. URL: http://kangax.github.io/compat-table/es6 (visited on 04/26/2017).

[26] Microsoft Inc. *TypeScript – JavaScript that scales*. URL: https://www.typescriptlang.org (visited on 04/30/2017).

[27] Magnus Madsen, Benjamin Livshits, and Michael Fanning. “Practical static analysis of JavaScript applications in the presence of frameworks and libraries”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM. 2013, pp. 499–509.

[28] Benjamin Livshits and Salvatore Guarnieri. “Gulfstream: incremental static analysis for streaming JavaScript applications”. In: *Proceedings of Technical Report MSR-TR-2010-4, Microsoft* (2010).

[29] Simon Holm Jensen, Anders Møller, and Peter Thiemann. “Type analysis for JavaScript”. In: *International Static Analysis Symposium*. Springer. 2009, pp. 238–255.

[30] Ariya Hidayat. *Esprima: Editing Autocomplete*. URL: `http://esprima.org/demo/autocomplete.html` (visited on 04/26/2017).

[31] Ekaterina Prigara. *How WebStorm Works: Completion for JavaScript Libraries*. URL: `https://blog.jetbrains.com/webstorm/2014/07/how-webstorm-works-completion-for-javascript-libraries` (visited on 04/26/2017).

[32] IBM Inc. *IBM Graph*. URL: `https://www.ibm.com/us-en/marketplace/graph` (visited on 04/30/2017).

[33] ArangoDB GmbH. *ArangoDB - highly available multi-model NoSQL database*. URL: `https://www.arangodb.com` (visited on 04/30/2017).

[34] ArangoDB GmbH. *DataStax - always-on data platform | NoSQL | Apache Cassandra*. URL: `https://www.datastax.com` (visited on 04/30/2017).

[35] Neo Technology Inc. *Neo4j, the world's leading graph database*. URL: `https://neo4j.com` (visited on 04/30/2017).

[36] OrientDB LTD. *OrientDB - Distributed Graph/Document Multi-Model Database*. URL: `http://orientdb.com` (visited on 04/30/2017).

[37] DB-Engines. *Graph DBMS*. URL: `https://db-engines.com/en/article/Graph+DBMS` (visited on 04/26/2017).

[38] DB-Engines. *DB-Engines Ranking of Graph DBMS*. URL: `https://db-engines.com/en/ranking/graph+dbms` (visited on 04/26/2017).

[39] Neo Technology Inc. *Neo4j*. URL: `https://github.com/neo4j` (visited on 04/27/2017).

[40] Neo Technology Inc. *Using Neo4j in Open Source Software*. URL: `https://neo4j.com/open-source` (visited on 04/26/2017).

[41] Neo Technology Inc. *Neo4j Licensing*. URL: `https://neo4j.com/licensing` (visited on 04/26/2017).

[42] Neo Technology Inc. *Introduction, Casual Cluster*. URL: `https://neo4j.com/docs/operations-manual/current/clustering/causal-clustering/introduction` (visited on 04/30/2017).

[43] Neo Technology Inc. *Intro to Cypher*. URL: https://neo4j.com/developer/cypher-query-language (visited on 04/26/2017).

[44] Kangax GitHub user. *ECMAScript 5 compatibility table*. URL: http://kangax.github.io/compat-table/es5 (visited on 04/27/2017).

[45] Aarhus University. *TAJS - Type Analyzer for JavaScript*. URL: https://github.com/cs-au-dk/TAJS (visited on 04/27/2017).

[46] Aarhus University. *Static analysis for JavaScript*. URL: https://users-cs.au.dk/amoeller/talks/TAJS2.pdf (visited on 04/27/2017).

[47] Aarhus University. *TAJS: Type Analyzer for JavaScript*. URL: http://www.brics.dk/TAJS (visited on 04/27/2017).

[48] Facebook Inc. *Flow*. URL: https://github.com/facebook/flow (visited on 04/27/2017).

[49] Facebook Inc. *Flow: A Static Type Checker for JavaScript*. URL: https://flow.org (visited on 04/27/2017).

[50] Facebook Inc. *Flow Documentation*. URL: https://flow.org/en/docs (visited on 04/27/2017).

[51] Marijn Haverbeke. *Tern*. URL: http://ternjs.net (visited on 04/27/2017).

[52] Marijn Haverbeke. *Tern*. URL: https://github.com/ternjs/tern (visited on 04/27/2017).

[53] SonarSource SA. *SonarQube*. URL: https://sonarqube.com (visited on 05/12/2017).

[54] Shape Security Inc. *Shift AST*. URL: http://shift-ast.org (visited on 04/27/2017).

[55] Shape Security Inc. *A Technical Comparison of the Shift and SpiderMonkey AST Formats*. URL: http://engineering.shapesecurity.com/2015/01/a-technical-comparison-of-shift-and.html (visited on 04/27/2017).

[56] Ariya Hidayat. *Esprima*. URL: https://github.com/jquery/esprima (visited on 04/27/2017).

[57] FindBugs. *FindBugs™ - Find Bugs in Java Programs*. URL: http://findbugs.sourceforge.net (visited on 04/28/2017).

[58] Nick Rutar, Christian B Almazan, and Jeffrey S Foster. "A comparison of bug finding tools for Java". In: *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*. IEEE. 2004, pp. 245–256.

[59] buschmais GbR. *jQAssistant*. URL: http://jqassistant.de (visited on 04/28/2017).

[60] buschmais GbR. *jQAssistant User Manual*. URL: http://buschmais.github.io/jqassistant/doc/1.2.0 (visited on 04/28/2017).

[61] Free Software Foundation. *GNU General Public License, Version 3, 29 June 2007*. URL: https://www.gnu.org/licenses/gpl-3.0.html (visited on 04/28/2017).

[62] Clang Project. *Clang Static Analyzer*. URL: https://clang-analyzer.llvm.org (visited on 04/28/2017).

[63] Ted Kremenek. "Finding software bugs with the clang static analyzer". In: *Apple Inc.* (2008).

[64] Patrick Cousot and Nicolas Halbwachs. "Automatic discovery of linear restraints among variables of a program". In: *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1978, pp. 84–96.

[65] The MathWorks Inc. *Polyspace Static Analysis*. URL: https://www.mathworks.com/products/polyspace.html (visited on 04/28/2017).

[66] Synopsys Inc. *Coverity is now a part of Synopsys*. URL: http://www.coverity.com (visited on 04/28/2017).

[67] Eclipse Foundation. *Eclipse Public License - v 1.0*. URL: https://www.eclipse.org/legal/epl-v10.html (visited on 04/29/2017).

[68] Matt Asay. *Would closing the ASP loophole create more problems than it solves?* URL: https://www.cnet.com/news/would-closing-the-asp-loophole-create-more-problems-than-it-solves (visited on 04/29/2017).

[69] Gábor Szárnyas et al. *ingraph – Incremental evaluation of openCypher queries*. URL: https://github.com/ftsrg/ingraph (visited on 04/30/2017).

[70] JetBrains s.r.o. *Project Grizzly*. URL: https://grizzly.java.net (visited on 04/30/2017).

[71] Curl Community. *curl*. URL: https://curl.haxx.se (visited on 04/30/2017).

[72] Postdot Technologies Inc. *Postman | Supercharge your API workflow*. URL: https://www.getpostman.com (visited on 04/30/2017).

[73] Gábor Szárnyas. *neo4j-drivers*. URL: https://github.com/szarnyasg/neo4j-drivers (visited on 04/29/2017).

[74] Shape Security Inc. *Shift Java - Shift format ECMAScript AST tooling*. URL: https://github.com/shapesecurity/shift-java (visited on 05/01/2017).

[75] Dr. Axel Rauschmayer. *Exploring ES6: Upgrade to the next version of JavaScript*. 2016.

[76] JavaScript Jabber. *JavaScript Jabber with Matt Pardee, Charles Max Wood, Jamison Dance, Tim Caswell*. URL: https://devchat.tv/js-jabber//020-jsj-cloud9 (visited on 05/06/2017).

[77] Ruben Daniels. *Twitter post*. URL: https://twitter.com/javruben/status/233580129798991872 (visited on 05/06/2017).

[78] Scott Hanselman. *Integrating Office and the Open Web with Lucidchart's Brian Pugh*. URL: https://hanselminutes.com/371/integrating-office-and-the-open-web-with-lucidcharts-brian-pugh (visited on 05/06/2017).

[79] Mozilla Developer Networks. *export*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export (visited on 05/06/2017).

[80] Mozilla Developer Networks. *import*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import (visited on 05/06/2017).

[81] ECMA International. *Exports*. URL: http://www.ecma-international.org/ecma-262/6.0/#sec-exports (visited on 05/06/2017).

[82] ECMA International. *Imports*. URL: http://www.ecma-international.org/ecma-262/6.0/#sec-imports (visited on 05/07/2017).

Appendix

A Cypher Queries for Interconnecting the ASGs of Related Modules

A.1 exportAlias–importAlias

```
MATCH
// exporter.js: export { name1 as exportedName1 };
   (exporter:CompilationUnit)-[:contains]->(:ExportLocals)
      -[:namedExports]->(exportLocalSpecifier:ExportLocalSpecifier)
      -[:name]->(:IdentifierExpression)
      <-[:node]-(:Reference)
      <-[:references]-(:Variable)
      -[:declarations]->(declarationToMerge:Declaration)
      -[:node]->(:BindingIdentifier),

// importer.js: import { exportedName1 as importedName1 } from
    "exporter";
   (importer:CompilationUnit)-[:contains]->(import:Import)
      -[:namedImports]->(importSpecifier:ImportSpecifier)
      -[:binding]->(importBindingIdentifierToMerge:BindingIdentifier)
      <-[:node]-(declarationToDelete:Declaration)
      <-[:declarations]-(importedVariable:Variable)

   WHERE
   exporter.parsedFilePath CONTAINS import.moduleSpecifier
   AND exportLocalSpecifier.exportedName = importSpecifier.name

MERGE
   (importedVariable)-[:declarations]->(declarationToMerge)
      -[:node]->(importBindingIdentifierToMerge)

DETACH DELETE
   declarationToDelete
```

A.2 exportAlias–importDefault

```
MATCH
// exporter.js: export { name1 as default };
   (exporter:CompilationUnit)-[:contains]->(:ExportLocals)
      -[:namedExports]->(exportLocalSpecifier:ExportLocalSpecifier)
      -[:name]->(:IdentifierExpression)
      <-[:node]-(:Reference)
      <-[:references]-(:Variable)
      -[:declarations]->(declarationToMerge:Declaration)
      -[:node]->(:BindingIdentifier),

// importer.js: import defaultName from "exporter";
   (importer:CompilationUnit)-[:contains]->(import:Import)
      -[:defaultBinding]->(importBindingIdentifierToMerge:BindingIdentifier)
      <-[:node]-(declarationToDelete:Declaration)
      <-[:declarations]-(importedVariable:Variable)

   WHERE
   exporter.parsedFilePath CONTAINS import.moduleSpecifier
   AND exportLocalSpecifier.exportedName = 'default'

MERGE
   (importedVariable)-[:declarations]->(declarationToMerge)
      -[:node]->(importBindingIdentifierToMerge)

DETACH DELETE
   declarationToDelete
```

A.3 exportAlias–importName

```
MATCH
// exporter.js: export { name1 as exportedName1 };
   (exporter:CompilationUnit)-[:contains]->(:ExportLocals)
      -[:namedExports]->(exportLocalSpecifier:ExportLocalSpecifier)
      -[:name]->(:IdentifierExpression)
      <-[:node]-(:Reference)
      <-[:references]-(:Variable)
      -[:declarations]->(declarationToMerge:Declaration)
      -[:node]->(:BindingIdentifier),

// importer.js: import { exportedName1 } from "exporter";
   (importer:CompilationUnit)-[:contains]->(import:Import)
      -[:namedImports]->(:ImportSpecifier)
      -[:binding]->(importBindingIdentifierToMerge:BindingIdentifier)
      <-[:node]-(declarationToDelete:Declaration)
      <-[:declarations]-(importedVariable:Variable)

   WHERE
   exporter.parsedFilePath CONTAINS import.moduleSpecifier
   AND exportLocalSpecifier.exportedName =
       importBindingIdentifierToMerge.name

MERGE
   (importedVariable)-[:declarations]->(declarationToMerge)
      -[:node]->(importBindingIdentifierToMerge)

DETACH DELETE
   declarationToDelete
```

A.4 exportDeclaration–importAlias

```
MATCH
// exporter.js: export var name1;
   (exporter:CompilationUnit)-[:contains]->(:ExportDeclaration)
      -[:declaration]->
      (:FunctionDeclarationClassDeclarationVariableDeclaration)
      -[:declarators]->(:VariableDeclarator)
      -[:binding]->(exportBindingIdentifier:BindingIdentifier)
      <-[:node]-(declarationToMerge:Declaration)
      <-[:declarations]-(:Variable),

// importer.js: import { name1 as importedName1 } from "exporter";
   (importer:CompilationUnit)-[:contains]->(import:Import)
      -[:namedImports]->(importSpecifier:ImportSpecifier)
      -[:binding]->(importBindingIdentifierToMerge:BindingIdentifier)
      <-[:node]-(declarationToDelete:Declaration)
      <-[:declarations]-(importedVariable:Variable)

   WHERE
   exporter.parsedFilePath CONTAINS import.moduleSpecifier
   AND exportBindingIdentifier.name = importSpecifier.name

MERGE
   (importedVariable)-[:declarations]->(declarationToMerge)
      -[:node]->(importBindingIdentifierToMerge)

DETACH DELETE
   declarationToDelete
```

A.5 exportDeclaration–importName

```
MATCH
// exporter.js: export var name1;
   (exporter:CompilationUnit)-[:contains]->(:ExportDeclaration)
      -[:declaration]->
      (:FunctionDeclarationClassDeclarationVariableDeclaration)
      -[:declarators]->(:VariableDeclarator)
      -[:binding]->(exportBindingIdentifier:BindingIdentifier)
      <-[:node]-(declarationToMerge:Declaration)
      <-[:declarations]-(:Variable),

// importer.js: import { name1 } from "exporter";
   (importer:CompilationUnit)-[:contains]->(import:Import)
      -[:namedImports]->(importSpecifier:ImportSpecifier)
      -[:binding]->(importBindingIdentifierToMerge:BindingIdentifier)
      <-[:node]-(declarationToDelete:Declaration)
      <-[:declarations]-(importedVariable:Variable)

   WHERE
   exporter.parsedFilePath CONTAINS import.moduleSpecifier
   AND exportBindingIdentifier.name = importBindingIdentifierToMerge.name

MERGE
   (importedVariable)-[:declarations]->(declarationToMerge)
      -[:node]->(importBindingIdentifierToMerge)

DETACH DELETE
   declarationToDelete
```

A.6 exportDefaultDeclaration–importAlias

```
MATCH
// exporter.js: export default name1;
   (exporter:CompilationUnit)-[:contains]->(:ExportDefault)
      -[:body]->(:FunctionDeclarationClassDeclarationVariableDeclaration)
      -[:name]->(exportBindingIdentifier:BindingIdentifier)
      <-[:node]-(declarationToMerge:Declaration)
      <-[:declarations]-(exportedVariable:Variable),

// importer.js: import { name1 as importedName1 } from "exporter";
   (importer:CompilationUnit)-[:contains]->(import:Import)
      -[:namedImports]->(importSpecifier:ImportSpecifier)
      -[:binding]->(importBindingIdentifierToMerge:BindingIdentifier)
      <-[:node]-(declarationToDelete:Declaration)
      <-[:declarations]-(importedVariable:Variable)

   WHERE
   exporter.parsedFilePath CONTAINS import.moduleSpecifier
   AND importSpecifier.name = exportBindingIdentifier.name

MERGE
   (importedVariable)-[:declarations]->(declarationToMerge)
      -[:node]->(importBindingIdentifierToMerge)

DETACH DELETE
   declarationToDelete
```

A.7 exportDefaultDeclaration–importDefault

```
MATCH
// exporter.js: export default name1;
   (exporter:CompilationUnit)-[:contains]->(:ExportDefault)
      -[:body]->(:FunctionDeclarationClassDeclarationVariableDeclaration)
      -[:name]->(exportBindingIdentifier:BindingIdentifier)
      <-[:node]-(declarationToMerge:Declaration)
      <-[:declarations]-(exportedVariable:Variable),

// importer.js: import defaultName from "exporter";
   (importer:CompilationUnit)-[:contains]->(import:Import)
      -[:defaultBinding]->(importBindingIdentifierToMerge:BindingIdentifier)
      <-[:node]-(declarationToDelete:Declaration)
      <-[:declarations]-(importedVariable:Variable)

   WHERE
   exporter.parsedFilePath CONTAINS import.moduleSpecifier

MERGE
   (importedVariable)-[:declarations]->(declarationToMerge)
      -[:node]->(importBindingIdentifierToMerge)

DETACH DELETE
   declarationToDelete
```

A.8 exportDefaultDeclaration–importName

```
MATCH
// exporter.js: export default name1;
   (exporter:CompilationUnit)-[:contains]->(:ExportDefault)
      -[:body]->(:FunctionDeclarationClassDeclarationVariableDeclaration)
      -[:name]->(exportBindingIdentifier:BindingIdentifier)
      <-[:node]-(declarationToMerge:Declaration)
      <-[:declarations]-(exportedVariable:Variable),

// importer.js: import { name1 } from "exporter";
   (importer:CompilationUnit)-[:contains]->(import:Import)
      -[:namedImports]->(importSpecifier:ImportSpecifier)
      -[:binding]->(importBindingIdentifierToMerge:BindingIdentifier)
      <-[:node]-(declarationToDelete:Declaration)
      <-[:declarations]-(importedVariable:Variable)

   WHERE
   exporter.parsedFilePath CONTAINS import.moduleSpecifier
   AND importBindingIdentifierToMerge.name = exportBindingIdentifier.name

MERGE
   (importedVariable)-[:declarations]->(declarationToMerge)
      -[:node]->(importBindingIdentifierToMerge)

DETACH DELETE
   declarationToDelete
```

A.9 exportDefaultName–importAlias

```
MATCH
// exporter.js: export default name1;
   (exporter:CompilationUnit)-[:contains]->(:ExportDefault)
      -[:body]->(exportedIdentifierExpression:IdentifierExpression)
      <-[:node]-(:Reference)
      <-[:references]-(exportedVariable:Variable)
      -[:declarations]->(declarationToMerge:Declaration),

// importer.js: import { name1 as importedName1 } from "exporter";
   (importer:CompilationUnit)-[:contains]->(import:Import)
      -[:namedImports]->(importSpecifier:ImportSpecifier)
      -[:binding]->(importBindingIdentifierToMerge:BindingIdentifier)
      <-[:node]-(declarationToDelete:Declaration)
      <-[:declarations]-(importedVariable:Variable)

   WHERE
   exporter.parsedFilePath CONTAINS import.moduleSpecifier
   AND exportedIdentifierExpression.name = importSpecifier.name

MERGE
   (importedVariable)-[:declarations]->(declarationToMerge)
      -[:node]->(importBindingIdentifierToMerge)

DETACH DELETE
   declarationToDelete
```

A.10 exportDefaultName–importDefault

```
MATCH
// exporter.js: export default name1;
   (exporter:CompilationUnit)-[:contains]->(:ExportDefault)
      -[:body]->(exportedIdentifierExpression:IdentifierExpression)
      <-[:node]-(:Reference)
      <-[:references]-(exportedVariable:Variable)
      -[:declarations]->(declarationToMerge:Declaration),

// importer.js: import defaultName from "exporter";
   (importer:CompilationUnit)-[:contains]->(import:Import)
      -[:defaultBinding]->(importBindingIdentifierToMerge:BindingIdentifier)
      <-[:node]-(declarationToDelete:Declaration)
      <-[:declarations]-(importedVariable:Variable)

   WHERE
   exporter.parsedFilePath CONTAINS import.moduleSpecifier

MERGE
   (importedVariable)-[:declarations]->(declarationToMerge)
      -[:node]->(importBindingIdentifierToMerge)

DETACH DELETE
   declarationToDelete
```

A.11 exportDefaultName–importName

```
MATCH
// exporter.js: export default name1;
   (exporter:CompilationUnit)-[:contains]->(:ExportDefault)
      -[:body]->(exportedIdentifierExpression:IdentifierExpression)
      <-[:node]-(:Reference)
      <-[:references]-(exportedVariable:Variable)
      -[:declarations]->(declarationToMerge:Declaration),

// importer.js: import { name1 } from "exporter";
   (importer:CompilationUnit)-[:contains]->(import:Import)
      -[:namedImports]->(importSpecifier:ImportSpecifier)
      -[:binding]->(importBindingIdentifierToMerge:BindingIdentifier)
      <-[:node]-(declarationToDelete:Declaration)
      <-[:declarations]-(importedVariable:Variable)

   WHERE
   exporter.parsedFilePath CONTAINS import.moduleSpecifier
   AND exportedIdentifierExpression.name =
       importBindingIdentifierToMerge.name

MERGE
   (importedVariable)-[:declarations]->(declarationToMerge)
      -[:node]->(importBindingIdentifierToMerge)

DETACH DELETE
   declarationToDelete
```

A.12 exportName–importAlias

```
MATCH
// exporter.js: let name1 = "name1Value"; export { name1 };
   (exporter:CompilationUnit)-[:contains]->(:ExportLocals)
      -[:namedExports]->(:ExportLocalSpecifier)
      -[:name]->(exportBindingIdentifier:IdentifierExpression)
      <-[:node]-(:Reference)<-[:references]-(:Variable)
      -[:declarations]->(declarationToMerge:Declaration)
      -[:node]->(:BindingIdentifier),

// importer.js: import { name1 as importedName1 } from "exporter";
   (importer:CompilationUnit)-[:contains]->(import:Import)
      -[:namedImports]->(importSpecifier:ImportSpecifier)
      -[:binding]->(importBindingIdentifierToMerge:BindingIdentifier)
      <-[:node]-(declarationToDelete:Declaration)
      <-[:declarations]-(importedVariable:Variable)

   WHERE
   exporter.parsedFilePath CONTAINS import.moduleSpecifier
   AND exportBindingIdentifier.name = importSpecifier.name

MERGE
   (importedVariable)-[:declarations]->(declarationToMerge)
      -[:node]->(importBindingIdentifierToMerge)

DETACH DELETE
   declarationToDelete
```

A.13 exportName–importName

```
MATCH
// exporter.js: export { name1 };
   (exporter:CompilationUnit)-[:contains]->(:ExportLocals)
      -[:namedExports]->(exportLocalSpecifier:ExportLocalSpecifier)
      -[:name]->(exportBindingIdentifier:IdentifierExpression)
      <-[:node]-(:Reference)
      <-[:references]-(:Variable)
      -[:declarations]->(declarationToMerge:Declaration)
      -[:node]->(:BindingIdentifier),

// importer.js: import { name1 } from "exporter";
   (importer:CompilationUnit)-[:contains]->(import:Import)
      -[:namedImports]->(importSpecifier:ImportSpecifier)
      -[:binding]->(importBindingIdentifierToMerge:BindingIdentifier)
      <-[:node]-(declarationToDelete:Declaration)
      <-[:declarations]-(importedVariable:Variable)

   WHERE
   exporter.parsedFilePath CONTAINS import.moduleSpecifier
   AND exportBindingIdentifier.name = importBindingIdentifierToMerge.name
   AND NOT exists(exportLocalSpecifier.exportedName)
   AND NOT exists(importSpecifier.name)

MERGE
   (importedVariable)-[:declarations]->(declarationToMerge)
      -[:node]->(importBindingIdentifierToMerge)

DETACH DELETE
   declarationToDelete
```

B Cypher Queries of the Analyses

B.1 nonInitialisedVariable

```
MATCH
   (containingCompilationUnit:CompilationUnit)-[:contains]->
      (variableLocation:SourceLocation)<-[:start]-(:SourceSpan)
      <-[:location]-(variableReference:VariableReference)
      <-[:node]-(:Reference)
      <-[:references]-(subjectVariable:Variable)
      -[:declarations]->(:Declaration)
      -[:node]->(:VariableReference)
      <-[:binding]-(variableDeclarator:VariableDeclarator)

   WHERE NOT (variableDeclarator)-[:init]->()

RETURN
   'Non-initialized variable' AS message,
   subjectVariable.name AS entityName,
   containingCompilationUnit.parsedFilePath AS compilationUnitPath,
   variableLocation.line AS line,
   variableLocation.column AS column
```

B.2 unusedExport — exportName-exportAlias

```
MATCH
   (exporter:CompilationUnit)-[:contains]->(:ExportLocals)
      -[:namedExports]->(exportLocalSpecifier:ExportLocalSpecifier)
      -[:name]->(:VariableReference)
      <-[:node]-(:Reference)
      <-[:references]-(exportedVariable:Variable),
   (exportLocalSpecifier)-[:location]->(:SourceSpan)
      -[:start]->(exportLocation:SourceLocation)

   WHERE
   NOT (exportedVariable)-[:declarations]->(:Declaration)
      -[:node]->(:VariableReference)
      <-[:binding]-(:ImportSpecifier)

RETURN
   'Globally unused export' AS message,
   exportedVariable.name AS entityName,
   exporter.parsedFilePath AS compilationUnitPath,
   exportLocation.line AS line,
   exportLocation.column AS column
```

B.3 unusedExport — exportDefault-exportDefaultName

```
MATCH
   (exporter:CompilationUnit)-[:contains]->(exportDefault:ExportDefault)
      -[:body]->(:IdentifierExpression)
      <-[:node]-(:Reference)
      <-[:references]-(exportedVariable:Variable),
   (exportDefault)-[:location]->(:SourceSpan)
      -[:start]->(exportLocation:SourceLocation),

   (exporter:CompilationUnit)-[:contains]->(:ExportLocals)
      -[:namedExports]->(exportLocalSpecifier:ExportLocalSpecifier)
      -[:name]->(:VariableReference)
      <-[:node]-(:Reference)
      <-[:references]-(exportedVariable:Variable),
   (exportLocalSpecifier)-[:location]->(:SourceSpan)
      -[:start]->(exportLocation:SourceLocation)

   WHERE
   NOT (exportedVariable)-[:declarations]->(:Declaration)
      -[:node]->(:VariableReference)
      <-[:binding]-(:ImportSpecifier)

RETURN
   'Globally unused export' AS message,
   exportedVariable.name AS entityName,
   exporter.parsedFilePath AS compilationUnitPath,
   exportLocation.line AS line,
   exportLocation.column AS column
```

B.4 unusedExport — exportDeclaration

```
MATCH
   (exporter:CompilationUnit)-[:contains]->
      (exportDeclaration:ExportDeclaration)
      -[:declaration]->
      (:FunctionDeclarationClassDeclarationVariableDeclaration)
      -[*1..2]->(:BindingIdentifier)
      <-[:node]-(:Declaration)
      <-[:declarations]-(exportedVariable:Variable),
   (exportDeclaration)-[:location]->(:SourceSpan)
      -[:start]->(exportLocation:SourceLocation)

   WHERE
   NOT (exportedVariable)-[:declarations]->(:Declaration)
      -[:node]->(:VariableReference)
      <-[:binding]-(:ImportSpecifier)

RETURN
   'Globally_unused_export' AS message,
   exportedVariable.name AS entityName,
   exporter.parsedFilePath AS compilationUnitPath,
   exportLocation.line AS line,
   exportLocation.column AS column
```

B.5 divisionByZero-literal — restricted

```
MATCH
   (binaryExpression:BinaryExpression)-[:right]->
      (rightValue:LiteralNumericExpression)
      -[:location]->(:SourceSpan)
      -[:start]->(locationStart:SourceLocation)
      <-[:contains]-(containingCompilationUnit:CompilationUnit)

   WHERE
   binaryExpression.operator = 'Div'
   AND rightValue.value = 0

RETURN
   'Division by zero' AS message,
   '' AS entityName,
   containingCompilationUnit.parsedFilePath AS compilationUnitPath,
   locationStart.line AS line,
   locationStart.column AS column
```

B.6 squareRootNegativeArgument-literal — restricted

```
MATCH
   (containingCompilationUnit:CompilationUnit)-[:contains]->
      (callExpression:CallExpression)
      -[:callee]->(memberExpression:StaticMemberExpression)
      -[:object]->(variableReference:VariableReference),
   (callExpression)-[:arguments]->(unaryExpression:UnaryExpression)
      -[:operand]->(:LiteralNumericExpression),
   (callExpression)-[:location]->
       (:SourceSpan)-[:start]->(entityLocation:SourceLocation)

   WHERE
   variableReference.name = 'Math'
   AND memberExpression.property = 'sqrt'
   AND unaryExpression.operator = 'Minus'

RETURN
   'Square_root_called_with_negative_argument' AS message,
   '' AS entityName,
   containingCompilationUnit.parsedFilePath AS compilationUnitPath,
   entityLocation.line AS line,
   entityLocation.column AS column
```

B.7 divisionByZero-variable — transitive

```
MATCH
   (binaryExpression:BinaryExpression)-[:right]->(rightValue:Expression)
      -[:location]->(:SourceSpan)
      -[:start]->(locationStart:SourceLocation)
      <-[:contains]-(containingCompilationUnit:CompilationUnit),
   (rightValue)-[:_qualifier]->(equalsZero:EqualsZero)

   WHERE
   binaryExpression.operator = 'Div'

RETURN
   'Division_by_zero' AS message,
   '' AS entityName,
   containingCompilationUnit.parsedFilePath AS compilationUnitPath,
   locationStart.line AS line,
   locationStart.column AS column
```

B.8 squareRootNegativeArgument-variable — transitive

```
MATCH
   (containingCompilationUnit:CompilationUnit)-[:contains]->
      (callExpression:CallExpression)
      -[:callee]->(memberExpression:StaticMemberExpression)
      -[:object]->(variableReference:VariableReference),
   (callExpression)-[:arguments]->(argument:Expression)
      -[:_qualifier]->(negativeNumeric:NegativeNumeric),
   (callExpression)-[:location]->
       (:SourceSpan)-[:start]->(entityLocation:SourceLocation)

   WHERE
   variableReference.name = 'Math'
   AND memberExpression.property = 'sqrt'

RETURN
   'Square root called with negative argument' AS message,
   '' AS entityName,
   containingCompilationUnit.parsedFilePath AS compilationUnitPath,
   entityLocation.line AS line,
   entityLocation.column AS column
```

B.9 unreachableCode-exception — transitive

```
MATCH
   (containingCompilationUnit:CompilationUnit)-[:contains]->
      (statement:Statement)-[:_qualifier]->(:ExceptionThrown),
   (statement)-[:_next]->(unreachableStatement:Statement),
   (unreachableStatement)-[:location]->(:SourceSpan)
      -[:start]->(entityLocation:SourceLocation)

RETURN
   'Unreachable code' AS message,
   '' AS entityName,
   containingCompilationUnit.parsedFilePath AS compilationUnitPath,
   entityLocation.line AS line,
   entityLocation.column AS column
```

C Cypher Queries of the Qualifier System

C.1 Initialising the Qualifier System

```
MERGE (qs:QualifierSystem)

MERGE (qs)-[:_instance]->(:Qualifier:EqualsZero)
MERGE (qs)-[:_instance]->(:Qualifier:NegativeNumeric)
MERGE (qs)-[:_instance]->(:Qualifier:ExceptionThrown)
```

C.2 Tagging literals with `EqualsZero`

```
MATCH
   (literalNumericExpression:LiteralNumericExpression),
   (qs:QualifierSystem)-[:_instance]->(equalsZero:Qualifier:EqualsZero)

   WHERE
   literalNumericExpression.value = 0

MERGE
   (literalNumericExpression)-[:_qualifier]->(equalsZero)
```

C.3 Tagging `throw` statements with `ExceptionThrown`

```
MATCH
   (throwStatement:ThrowStatement),
   (qs:QualifierSystem)-[:_instance]->(exceptionThrown:ExceptionThrown)

MERGE
   (throwStatement)-[:_qualifier]->(exceptionThrown)
```

D Cypher Queries for Qualifier Propagation

D.1 Propagation along function calls

```
MATCH
   (callExpression:CallExpression)-[:callee]->(:Expression)
      -[:_qualifier]->(qualifier:Qualifier)

MERGE
   (callExpression)-[:_qualifier]->(qualifier)
```

D.2 Propagation along function declarations

```
MATCH
   (qualifier:Qualifier)<-[:_qualifier]-(functionDeclaration:FunctionDeclaration)
      -[:name]->(bindingIdentifier:BindingIdentifier)

MERGE
   (bindingIdentifier)-[:_qualifier]->(qualifier)
```

D.3 Propagation along function return statements

```
MATCH
   (function:Function)-[:body]->(:FunctionBody)
      -[:statements]->(:ReturnStatement)
      -[:expression]->(:Expression)
      -[:_qualifier]->(qualifier:Qualifier)

MERGE
   (function)-[:_qualifier]->(qualifier)
```

D.4 Propagation along throw statements in functions

```
MATCH
   (function:Function)-[:body]->(:FunctionBody)
      -[:statements]->(:ThrowStatement)
      -[:_qualifier]->(qualifier:Qualifier)

MERGE
   (function)-[:_qualifier]->(qualifier)
```

D.5 Propagation along variable declarations

```
MATCH
   (variable:Variable)-[:declarations]->(:Declaration)
      -[:node]->(:BindingIdentifier)
      -[:_qualifier]->(qualifier:Qualifier)

MERGE
   (variable)-[:_qualifier]->(qualifier)
```

D.6 Propagation along variable declaration statements

```
MATCH
   (variableDeclarationStatement:VariableDeclarationStatement)
      -[:declaration]->(variableDeclaration:VariableDeclaration)
      -[:declarators]->(variableDeclarator:VariableDeclarator)
      -[:binding]->(:BindingIdentifier)-[:_qualifier]->(qualifier:Qualifier)

MERGE
   (variableDeclarationStatement)-[:_qualifier]->(qualifier)
MERGE
   (variableDeclaration)-[:_qualifier]->(qualifier)
MERGE
   (variableDeclaration)-[:_qualifier]->(qualifier)
```

D.7 Propagation along variable initialisations

```
MATCH
   (expression:Expression)-[:_qualifier]->(qualifier:Qualifier),
   (expression)<-[:init]-(:VariableDeclarator)-[:binding]
      ->(:BindingIdentifier)<-[:node]-(:Reference)
      <-[:references]-(variable:Variable)

MERGE
   (variable)-[:_qualifier]->(qualifier)
```

D.8 Propagation along variable references

```
MATCH
   (variable:Variable)-[:_qualifier]->(qualifier:Qualifier),
   (variable)-[:references]->(:Reference)
      -[:node]->(variableReference:VariableReference)

MERGE
   (variableReference)-[:_qualifier]->(qualifier)
```

E Selected Open-Source Repositories for the Evaluation

Apart from the `tresorit/webclient` — which is a closed-source, security-oriented industrial project from the cloud-security company Tresorit —, every source code repository has been selected and downloaded from GitHub[1], a popular web-based repository hosting service for the Git version control system. The below format denotes `owner/repository`.

E.1 Repository and Graph Data

	JavaScript SLOC excl. comments	Number of JavaScript source file	Number of nodes in the ASG	Number of relationships in the ASG
initialstate/silent-doorbell	15	2	686	2,306
babel/example-node-server	17	2	573	1,900
bradtraversy/rxjs_boiler	19	2	340	1,104
tj/deferred.js	29	3	1,152	3,927
karma-runner/gulp-karma	32	4	988	3,232
scotch-io/node-web-scraper	34	1	1,559	5,326
brettlangdon/jsnice	36	1	1,460	4,976
facundoolano/promise-log	41	1	533	1,816
jinzhe/vue-editable	41	1	1,479	5,092
callmecavs/gotem	44	2	1,172	3,998
Heydon/forceFeed	48	1	2,205	7,574
varHarrie/Dawn-Blossoms	50	2	2,369	8,120
kmewhort/pointer_events_polyfill	54	1	1,595	5,402
bodil/eslint-config-cleanjs	55	1	973	3,238
Verba/jquery-readyselector	71	2	2,754	9,378
scotch-io/node-api	74	2	2,782	9,376
kolodny/wavy	79	2	3,186	11,068

[1] `https://github.com`

	JavaScript SLOC excl. comments	Number of JavaScript source file	Number of nodes in the ASG	Number of relationships in the ASG
bas2k/jquery.appear	81	1	3,158	10,896
bevacqua/trunc-html	89	2	3,061	10,452
tj/node-trace	89	2	3,429	11,690
louisondumont/facematch	93	6	3,174	10,692
bilbof/purser	97	4	6,232	21,200
markdalgleish/react-themeable	112	2	3,299	11,160
OutSystems/AutoAnimations	125	1	3,965	13,802
sindresorhus/grunt-sass	145	3	4,185	14,154
dissimulate/Tearable-Cloth	184	1	6,708	23,369
ebidel/appmetrics.js	307	6	13,244	45,578
BohemianCoding/sketch-image-compressor	367	2	13,825	47,879
eduardomb/scroll-up-bar	457	7	17,129	59,224
mewo2/naming-language	463	1	11,398	39,044
angular-ui/ui-codemirror	580	6	19,027	65,344
angular-app/Samples	3,310	123	118,811	402712
mzabriskie/axios	3,863	76	146,178	498,722
alvin198761/web-os	5,922	67	205,115	707,597
reactjs/redux	6,036	158	56,750	192,018
joyent/node-workflow	6,143	20	191,449	653,107
facebookincubator/create-react-app	6,855	141	97,933	335,169
freeCodeCamp/freeCodeCamp	11,823	177	163,070	551,500
vuejs/vue	12,982	188	81,619	282,253
tresorit/webclient	34,546	609	1,346,776	4,576,319

E.2 Measurement Results

	Duration of synchronisation	Duration of interconnection	Duration of running the Qualifier System	Duration of the analyses
initialstate/silent-doorbell	1,497,553 µs	13,367 µs	2,650,105 µs	87,580 µs
babel/example-node-server	932,053 µs	2,702,617 µs	2,171,579 µs	3,091,490 µs
bradtraversy/rxjs_boiler	1,202,026 µs	3,577,111 µs	4,531,478 µs	6,583,445 µs
tj/deferred.js	5,539,918 µs	15,038 µs	3,294,333 µs	5,358,816 µs
karma-runner/gulp-karma	1,521,805 µs	2,663,639 µs	2,123,288 µs	6,016,907 µs
scotch-io/node-web-scraper	1,991,224 µs	11,180 µs	2,181,329 µs	64,417 µs
brettlangdon/jsnice	1,909,302 µs	7,215 µs	1,943,869 µs	43,765 µs
facundoolano/promise-log	7,116,37 µs	2,515,415 µs	1,586,458 µs	54,952 µs
jinzhe/vue-editable	2,669,613 µs	8,086 µs	2,635,767 µs	2,409,739 µs
callmecavs/gotem	2,763,192 µs	1,033,612 µs	7,960,501 µs	2,625,625 µs
Heydon/forceFeed	3,146,354 µs	7,280 µs	2,222,943 µs	52,337 µs
varHarrie/Dawn-Blossoms	3,672,140 µs	2,450,689 µs	2,107,809 µs	53,592 µs
kmewhort/pointer_events_polyfill	2,164,437 µs	7,563 µs	1,996,131 µs	104,837 µs
bodil/eslint-config-cleanjs	1,445,878 µs	26,147 µs	3,276,381 µs	41,178 µs
Verba/jquery-readyselector	5,290,469 µs	3,585,861 µs	3,501,627 µs	6,432,633 µs
scotch-io/node-api	3,777,201 µs	8,681 µs	4,124,129 µs	37,368 µs
kolodny/wavy	4,570,584 µs	5,468 µs	1,994,231 µs	54,605 µs
bas2k/jquery.appear	5,448,897 µs	10,726 µs	2,269,582 µs	5,208,812 µs
bevacqua/trunc-html	4,327,909 µs	11,221 µs	1,986,631 µs	6,669,424 µs
tj/node-trace	11,879,062 µs	12,639,721 µs	6,610,741 µs	7,725,408 µs
louisondumont/facematch	4,626,696 µs	5,967 µs	2,161,497 µs	64,844 µs
bilbof/purser	8,663,335 µs	2,668,585 µs	2,390,193 µs	3,090,801 µs
markdalgleish/react-themeable	5,162,753 µs	2,189,079 µs	2,393,060 µs	6,524,218 µs
OutSystems/AutoAnimations	6,269,249 µs	2,653,444 µs	2,098,536 µs	5,944,605 µs
sindresorhus/grunt-sass	5,483,316 µs	2,017,179 µs	2,122,158 µs	2,968,374 µs
dissimulate/Tearable-Cloth	12,913,704 µs	240,211 µs	2,804,645 µs	2,172,038 µs
ebidel/appmetrics.js	21,597,123 µs	2,657,964 µs	2,590,861 µs	5,978,452 µs
BohemianCoding/sketch-image-compressor	30,141,715 µs	2,128,170 µs	3,716,391 µs	5,902,471 µs
eduardomb/scroll-up-bar	24,928,994 µs	2,664,088 µs	2,214,006 µs	85,223 µs
mewo2/naming-language	50,654,490 µs	2,900,949 µs	4,030,616 µs	4,063,752 µs
angular-ui/ui-codemirror	79,015,546 µs	3,832,991 µs	3,574,924 µs	6,571,692 µs
angular-app/Samples	221,836,333 µs	6,233,693 µs	5,411,927 µs	8,201,479 µs
mzabriskie/axios	345,069,568 µs	2,894,648 µs	7,629,029 µs	5,965,491 µs
alvin198761/web-os	1,126,669,966 µs	3,167,234 µs	38,086,892 µs	5,852,797 µs
reactjs/redux	115,551,087 µs	7,887,990 µs	7,517,116 µs	7,368,536 µs
joyent/node-workflow	1,036,041,011 µs	5,412,719 µs	5,729,655 µs	7,454,727 µs
facebookincubator/create-react-app	200,879,187 µs	3,702,773 µs	3,512,440 µs	3,223,522 µs
freeCodeCamp/freeCodeCamp	243,149,247 µs	3,074,393 µs	3,997,613 µs	5,563,976 µs
vuejs/vue	273,768,964 µs	1,862,202 µs	5,625,412 µs	4,905,929 µs
tresorit/webclient	4,651,047,877 µs	26,838,424 µs	36,416,152 µs	5,588,438 µs