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

Graph-Based Source Code Analysis of JavaScript Repositories

Master's Thesis

Author:

Dániel Stein

Supervisors:

Gábor Szárnyas
Ádám Lippai
Dávid Honfi

2016

Contents

Kivonat Egyre több, egyre inkább komplex szoftver vesz körül minket, amelyek gyakran kritikus rendszereket vezérelnek. Az ilyen rendszerek fő jellemzője, hogy a legapróbb hibáik is komoly következményekkel járhatnak. A forráskód statikus analízise egy, a kritikus szoftverrendszereknél általánosan elfogadott megközelítés, amely a hibák mihamarabbi megtalálását célozza meg. A statikus analízis már a fejlesztési folyamat korai szakaszaiban is alkalmazható, mivel nincs szükség a kód fordítására és futtatására az ellenőrzés véghezviteléhez. A megközelítést számos eszköz megvalósítja, amelyek képesek visszajelzést adni a potenciális hibahelyeken túl arról is, hogy a forráskód megfelel-e a kódolási szabályoknak és követelményeknek.

Habár több statikus analízis eszköz is elérhető általános célú nyelvek elemzéséhez, és ezek gyakran a folytonos integráció részét képzik, JavaScript esetén ez nem mondható el annak dinamikus jellege miatt. A dinamikusan tipizált nyelvek sajátosságai miatt csak pár eszköz érhető el JavaScript forráskódok kódtárszintű statikus analíziséhez, illetve az eddig ismert ilyen eszközök nem nyújtanak egyszerre megoldást alaki és globális ellenőrzésre, futási utak meghatározására és folytonos integrációval történő alkalmazásra.

Jelen dolgozatomban egy olyan, a folytonos integráció kiegészítésére képes keretrendszert tervezek, valósítok meg és értékelek, amely képes nagyméretű és gyakran változó JavaScript forráskódtárak konfigurálható statikus analízisére. A keretrendszer alapjául szolgáló újszerű megközelítésnek köszönhetően az eddig megszokott megoldások helyett a felhasználók egyszerűbb módon fejezhetik ki az ellenőrzésre szánt követelményeket és képesek a több forráskódon átívelő követelményeket hatékonyabban ellenőrizni.

Abstract We are surrounded by more and more complex software that operate in mission-critical systems. Even small errors in these software can lead to serious consequences that may be too expensive to let happen. Static analysis is a proven approach for detecting mistakes in the source code early in the development cycle. Since static analysis does not compile or run the code, it can be applied at an early state of development. With static analysis it is possible to check whether the software conforms to the coding rules and requirements, and to locate potential errors.

While multiple static analysis tools exist for general purpose programming languages and these are generally part of the continuous integration systems, this is not the case with JavaScript. Due to the dynamically typed nature of this language there are only a few tools available for JavaScript codebases. Also, there are currently no tools available jointly providing lower level and global static analysis, finding control flows, and providing integration points for continuous integration systems.

In this thesis I design, implement and evaluate a framework extending the continuous integration workflow of large and frequently changing JavaScript repositories with configurable static analysis tools and techniques. Due to the novel approach of the framework, its users can express requirements easier and they are able to check global level requirements more efficiently.

Chapter 1

Introduction

1.1 Context

Quality control plays an important role in large-scale software development. Software systems are getting more complex and more versatile. In particular, the size of the code repository (measured in lines of code) has been shown increase the number of errors in the software [1].

To ensure the quality of the source code, and at the same time help developers with their tasks, there is a growing need for a solution that allows continuous checks for the code. Such checks include code reviews, searching for mistakes, and enforcing conventions. [2]

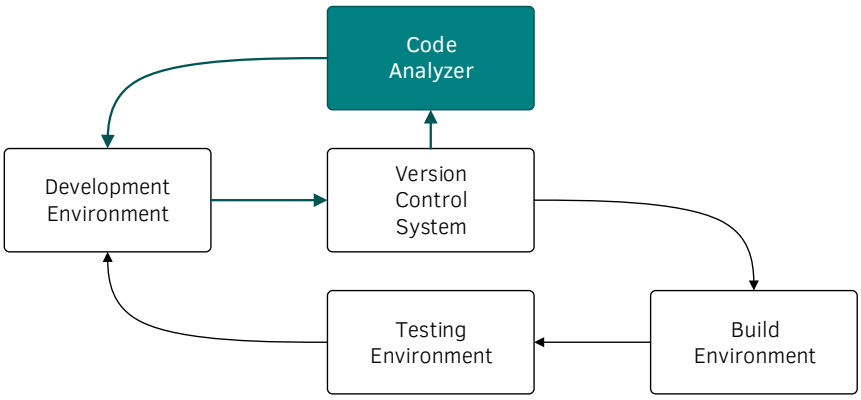

Figure 1.1 Continuous integration workflow extended with static analysis.

Version control systems (VCS) and continuous integration (CI) [3] solutions are widely used tools of modern software developers. Figure 1.1 shows an extended version of the generally used continuous integration workflow. The basic workflow consists of the following steps:

1) the developer makes modifications to the codebase using their *Development Environment*, 2) the modifications are committed into a *Version Control System*, and 3) this commit triggers the *Build Environment* to build the project. Then 4) the *Testing Environment* can perform runtime tests on the latest build of the project, and 5) the results — build and test logs — are presented to the developer.

These logs help the developers discover bugs and failures before the software is released for manual testing or production purposes. Producing this information often and as early as possible — thus making sure the software is working as intended — is vital for agile development.

A proven method of enhancing software quality is utilizing static program analysis techniques extending the basic CI workflow. During this process the code is analyzed without executing the application. In practice it is usually employed to reveal problems undetectable with testing and thus enabling the developer to create higher quality software.

1.2 Problem Statement

Static analysis methods verifying that the code is compliant with coding conventions is often time-consuming and resource-intensive in practice. The size of the codebase may necessitate a scalable solution, especially for continuous integration purposes, since the entire verification process needs to be carried out on the whole codebase every time it is modified.

A temporary solution to tackle this problem is to process the changes in batches. This way — to save resources — static analyses are carried out for a joined group of changes, rather than for every individual commit.

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

1.3 Objectives and Contributions

My main objective is to provide a solution for reducing the time required for a global, codebase-level reevaluation of static analysis after a change occurs.

In this work I create a framework that transforms the whole source code repository into a graph representation and maintains it subsequently. The proposed approach is suitable for performing code convention compliance checks, for executing built-in static analysis tests and arbitrary transforming extensions written by the user.

In order to speed up the static analysis, the presented framework uses incremental processing. Thus it is able to process a subset of the repository, e.g. only the modifications introduced by

the latest commit, then integrate the changes into the maintained representation. This way the system can process the modifications for each commit incrementally. After the initial query evaluation and report generation, consecutive runs can be executed significantly more efficiently.

The framework relies on two substantial technologies: 1) a source code parser, and 2) a database solution. Also, the framework provides interfaces making integration possible with external tools, such as version control systems and integrated developer environments.

1.4 Structure of the Thesis

This thesis is structured as follows. Chapter 2 introduces the previously mentioned background technologies selected to build an incremental static analyzer. Chapter 3 details the various approaches and related works. Chapter 4 shows the overview of my approach and details the main components of its architecture. Chapter 5 presents the implementation of the framework, and discusses the steps of the analysis. Chapter 6 demonstrates and evaluates the performance of the framework. Chapter 7 reveals future visions and possible ways of further improvements. Chapter 8 concludes the thesis.

Chapter 2

Preliminaries

In this chapter I present the conceptual foundations and related technologies of my work. Also, I discuss the building blocks required to design a static analyzer framework for JavaScript.

2.1 JavaScript

Ohe most used dynamic languages in the world is JavaScript. According to [4], JavaScript is the most utilized client-side programming language for web applications, with over 94% estimated usage. In this section I briefly summarize the history of JavaScript, present recent advancements and discuss why it is timely to utilize static analysis techniques for the language.

2.1.1 From Glue Language to a Full-Fledged Language

This section follows [5]. The language written in 10 days by Brendan Eich in April 1995 — originally called LiveScript — was one of the first attempts at bringing dynamic behavior to the web. It was supposed to look like Java, leaving its complexity behind and appealing for developers looking for easy scripting solutions on the web.

The initial goal — to create a language for portable applications — has been seemingly achieved by the time of writing this thesis. While it has been gaining popularity due to its simpler syntax, the language, its tooling, and its community had time to evolve, making it the most used web programming language. JavaScript has shifted from simpler scripts to complex systems on the web, desktops, and servers.

2.1.2 ECMAScript

After its release, JavaScript was submitted to and standardized by the Ecma International industry association, resulting in the first release of the ECMAScript language specifica-

tion (of the ECMA-262 standard [6]) in June 1997. Apart from JavaScript, there are several implementations, e.g., *Chakra*[1], *JScript*[2], and *V8*[3].

There are several reasons why the standard and the usage of ECMAScript is getting more and more popular. Besides the improvement of the developer tools, developer community, and being already popular, conscious and regular development of the language also makes the language more appealing. At the time of writing this thesis there are 8 editions of the standard, 6 of them published. The last published edition, ES7 was published in June 2016, one year after the previous edition, ES6.

The initial language specifications were indulgent, and only best practices were guiding the developers. The challenges of analyzing such dynamic, untyped language — able to express one thing in several different ways — may be one of the reasons why there are only a few tools present even today implementing static analysis for ECMAScript.

The syntactic sugars added to the language across the editions of ECMAScript made it possible to express the most used code parts in an easier manner, while being more concise. Encouraging the developers to use these constructs makes it easier to interpret the source code — both manually and with source code analysis.

2.2 Static Analysis

The idea of static analysis is almost half a century old. A paper from 1995 states that "The idea that computer software should be used to analyze source programs rather than compile them, has a history of at least 25 years." [7]

Source code analysis can be used to discover facts about a particular program. Two basic automated analysis methods exist for this purpose:

- *Static analysis* is performed by parsing the source code and analyzing it without evaluating the statements or executing the program.
- *Dynamic analysis* is performed by executing the program and evaluating its output for given input sequences.

While high-level language (e.g., C++, Java) source codes are checked at least by the compiler, JavaScript is usually not compiled before it is published. Thus a specific static analysis tool for JavaScript should aim to discover unwanted traits of the source in ways a generic compiler would not be able to, resulting in better code quality. These traits, *code smells* are usually perceptible while running the code. Another way to locate these bugs is to write and run tests, dynamically testing the program. [7]

[1] `https://github.com/Microsoft/ChakraCore`
[2] `https://msdn.microsoft.com/library/hbxc2t98.aspx`
[3] `https://github.com/v8/v8`

The two analysis methods are complementing each other. They discover different subsets of problematic constructions. While static analysis can discover syntactical problems (like the lack of the default case in a switch), dynamic analysis may catch behavioral problems (such as error handling and timing errors). Information discovered using static analysis may be used later in dynamic testing, resulting in a hybrid technique.

2.2.1 Use Cases

Static analysis tools employ diverse levels of abstraction. *Formatters* are able to ensure that the source code complies with a predefined style guide. *Linters* check for stylistic and programming errors, thus indicating suspicious programming constructs. *Formal verification*, on the other hand, utilizes formal mathematical methods to prove statements about a piece of source code and its behavior.

For dynamic languages static analysis has even more use cases. For example, it allows finding previously undefined property reads, catching invocation of non-functional variables [8], detecting dead code.

2.2.2 Advantages and Disadvantages

Since static analysis tools deliberately do not evaluate the source code, there are fundamental limitations to what problems they can discover. In the context of my work, static analysis has the following trade-offs.

Advantages

No Need for Execution Since there is no need for execution, the hardware and software requirements of the analysis software do not need to match the requirements of the application. There is also no need to emulate or mock its dependencies, making the analysis more portable and cost efficient.

Early Detection of Possible Errors Static analysis may catch problems even before the whole software is complete or even runnable, potentially allowing the developers to fix the problems earlier in the development process. [9]

Thorough While dynamic testing executes the manually written or automatically generated test cases covering a portion of the source code, static analysis systematically explores every possible execution scenarios. Thus it may achieve higher coverage and detect more problems in the code. [9]

Disadvantages

Speed Static analysis trades CPU time and memory for better code quality. By design it may be multiple orders of magnitude slower than compilation. Its speed depends not only on the underlying data structure and algorithms, but also the level of analysis. [10]

However, with a given limitation of granularity (e.g., considering a single file as the unit of processing), in case of a source code modification, previous results can be reused. There is a possibility that only the modified — and other affected — parts need to be processed again. The incremental approach of static analysis may speed up the process by orders of magnitude [11].

False Positives Static analysis can not prove the correctness of a source code. It rather warns in case there is a possibility of a problem. Thus static analysis tools can introduce false positive warnings and flag code parts as problematic even if they behave correctly. To reduce the number of false positive warnings, one usually introduces more precise, specified rules and more thorough analysis. [10]

2.2.3 Source Code Processing and Analysis

The source code of a program is a sequence of instructions formulated in a programming language as a text. Grammars of formal languages are a set of rules describing what the compiler considers a valid input—how to create valid instructions or a set of instructions from the alphabet of the language *(syntax)*. A source code processing entity (transformer, or hereafter *compiler*) assigns a meaning *(semantics)* and transforms the instruction to another language (generally an intermediate language or bytecode). [12]

What input data the compiler considers useful information depends on the semantics of the language the compiler is built for. Source codes contain a much wider variety of data than a compiler requires for transforming, analyzing the application: comments, function declaration order, indentation, line breaks all help the reader (and writer) of the code, but carry no additional information.

Figure 2.1 shows the general process for processing source code and transforming a stream of characters into a data structure with *meaning* (semantic information).

Lexical Analysis

A *lexer*, *tokenizer*, or *scanner* forms the first phase of a parsing process. It scans the input character stream and segments them into sequence of groups, tokens, *strings with a "meaning"*. It also categorizes these groups into various token classes, token types. Processing the raw input into a value (converting the string `"2"` into the number 2) can also happen in this phase. Figure 2.2 shows how an expression string can be tokenized.

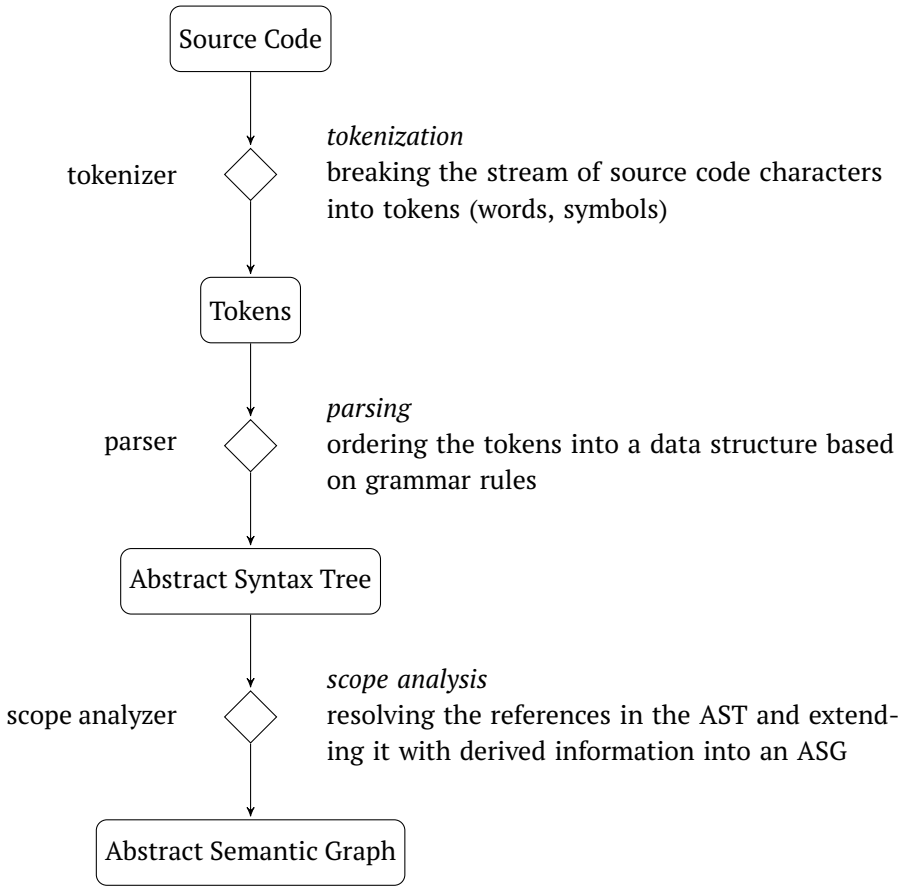

Figure 2.1 Processing the source code.

```
foo = 1 / 0
```

Token	Token type
`foo`	`IDENTIFIER (Ident)`
`=`	`ASSIGN (Punctuator)`
`1`	`NUMBER (NumericLiteral)`
`/`	`DIV (Punctuator)`
`0`	`NUMBER (NumericLiteral)`

Figure 2.2 Character stream and tokenization result with token type information.

Parser

A *parser* forms the second phase of a parsing process, using the output token stream of the *lexer*. It takes the input data and builds a hierarchical data structure (a *parse tree* or an *abstract*

syntax tree) representing the input. If the input does not comply with the syntax rules, and a tree can not be built, the source code is syntactically incorrect.

Figure 2.3 shows a sentence and its s-expression representation. S-expressions (for "symbolic expression") are a notation for tree-structured nested list, mainly used in Lisp.

```
The quick brown fox jumps over the lazy dog
```

```
(Sentence
  (Word The)
  (Word quick)
  (Word fox)
  (Word jumps)
  (Word over)
  (Word the)
  (Word lazy)
  (Word dog)
)
```

Figure 2.3 A sentence and its s-expression representation.

Abstract Syntax Tree An *Abstract Syntax Tree* (AST) is a tree representation of the syntactic structure of the result of a parsing along a grammar. The nodes of the tree denote a construct occurring in the input, while the edges represent the connection between these nodes based on the grammar rules.

As mentioned in Section 2.2.3, source code contains much more detail (e.g., indentation, whitespace, comments) than is required and restrained during the parsing process. Thus this representation is a more compact, *abstract* syntax tree. It may be transformed based on transformation rules and source code can be generated from ASTs. Without restraining the layout information and reusing it during the code generation, the resulting text can show differences in indents, whitespaces and expression formulation compared to the initial source code.

Figure 2.4 shows an example AST (in black), where a one-statement JavaScript file is parsed. The content of the file was only the following line:

```
var foo = 1 / 0;
```

Scope Analyzer

A *scope analyzer* or *context analyzer* produces a data structure representing the scoping information of a program, extending the information in the AST or ASG. An element representing a scope in this data structure may contain, inter alia, metadata about the scope itself, the AST node related to the scope, available assets (variables, functions, etc.) in that scope. [13]

Abstract Semantic Graph An *Abstract Semantic Graph* (ASG) differs from an Abstract Syntax Tree in two essential concepts: ASGs 1) are not necessarily trees, and 2) express more than the syntactic information; ASGs carry semantic information expressed with additional edges. An example of this type of edge connects variable references to their declarations. [14]

An ASG is a graph representation of a parse result; the nodes represent subterms of an expression. Shared subterms can occur, having more than one node linked to the same, common subterm. Compilers generally work on ASGs internally, since not only the syntactical, but the behavioral information is also represented in them.

Figure 2.4 shows the difference between an AST and an ASG (in turquoise).

Type Inference

In order to make sure that a system behaves correctly according to the specification, a broad range of *formal methods* can be used. Besides *model checkers*, *run-time monitoring*, the most popular formal methods are *type systems*. [15] "A type system is a syntactic method for automatically checking the absence of certain erroneous behaviors by classifying program phrases according to the kinds of values they compute." [15]

Type inference refers to the deduction of the data types of expressions, statements in the source code, usually executed during compilation and static analysis. By utilizing type inference, deducing types as *interfaces*, and checking contracts — e.g., preconditions for the types of arguments, postconditions for the types of return values — between various parts of the software can yield a more consistent and bug-free code.

Type systems also help developers write the source code with context-aware assistance. With implicitly typed languages the ability to infer data types can make prototyping and developing programs occur in a more agile fashion, compared to explicitly typed languages. Omitting the type annotations eliminates the need to propagate changes in the source code in case of a refactor, while still executing interface checks prevent type errors at runtime.

Typed JavaScript Derivations There are several languages with type systems that compile to JavaScript, such as TypeScript[4], Dart[5], Elm[6], and Flow (see Section 3.1.5).

[4] `https://www.typescriptlang.org/`
[5] `https://www.dartlang.org/`
[6] `http://elm-lang.org/`

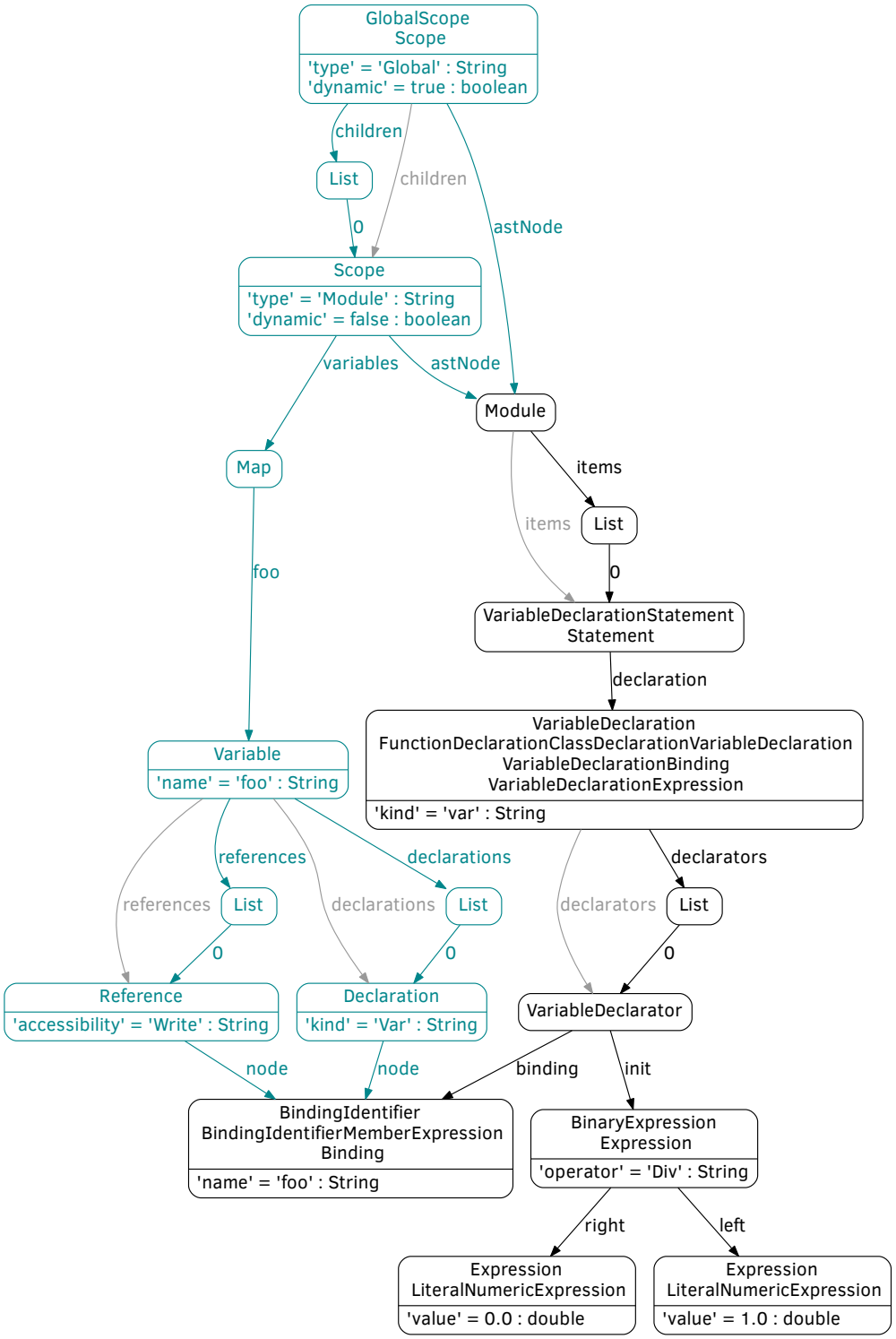

Figure 2.4 AST (in black) with additional edges (ASG, in turquoise) and derived helper edges (in gray).

2.3 Handling Large Interconnected Data

In numerous use cases, interconnected data sets can be represented and processed as a graph. *Graph databases* provide a way to efficiently store graphs and evaluate graph queries. In this section I introduce the concept of graph databases, and graph pattern matching. I also present different types of graph database implementations from which I selected the most appropriate technology for the approach.

The advancements in hardware components – the ever increasing amount of processing power, and memory and storage speed and sizes – and the analogous growth of data to be stored and processed during the last more than fifty years yielded various solutions.

Based on historic evolution, these solutions can be categorized in three main categories:

- *Navigational* database management systems (DBMS) were mainly used in the era of magnetic tape based storages, in which the *records* contained references to other records allowing the system to fast-forward, *navigate* there and load additional data.

- *Relational* DBMSs organizes data in a *relational model* [16], where one or more *relations* contain unique entries (*records* or *tuples*).

  Relational databases leverage precise mathematical background (see *relational algebra* and *relational calculus*), have diverse implementations, mature tooling, and data access security by authentication and authorization. There are also disadvantages; due to their data structure, relational databases may have scalability and performance issues. They are also typically optimized for transactional processing and not data analysis (there are exceptions, see *data warehouses*).

- *Post-relational* databases is a vague collective name for every database system that abandons the strictness and burden of the relational data model and the Structured Query Language (SQL).

  Since the turn of the millennium, the struggle with storing and processing huge amounts of data using relational technologies spawned a diverse palette of new database management systems using simpler, more scalable data models. These systems are consequently called *non SQL*, NoSQL databases, and are increasingly used in real-time and big data applications.

  NoSQL systems are a heterogeneous set of systems, with very different approaches. Categories of these systems based on their data models include, but are not limited to: *key-value stores*, *wide column stores*, *document stores*, *graph DBMSs*, *RDF stores*.

2.3.1 On Graph Computing

The mathematical concepts of graphs are well-known and widely used in computer sciences. Numerous graph technologies have evolved, each with their advantages and disadvantages.

From graphs themselves to physical and virtual worlds, many scenarios can be represented as graphs and stored in graph databases with their respective data model. This section loosely follows [17, 18].

Simple, *textbook-style* graphs can be extended in several different ways. To describe the connections in more detail, one may add directionality to edges (*directed graph*). To allow different connections, one may label the edges (*labeled graph*). *Typed graphs* introduce types for vertices. *Property graphs* add even more possibilities by introducing properties. Each graph element, both vertices and edges can be described with a collection of properties. The properties are key–value pairs, e.g. `type = 'Person'`, `name = 'John'`, `age = 34`.

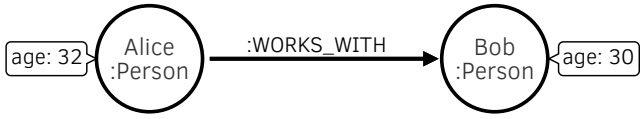

Figure 2.5 Typed and labeled property graph example.

My approach utilizes typed (labeled) property graphs (see Figure 2.5).

Graph Computing Technologies

The practice of data storage and processing is encumbered with space and time trade-off. This trade-off is also present in various graph computing technologies. This section discusses the categorization of these technologies and mentions a few technologies of which the most popular ones are detailed in Section 2.3.3.

The landscape of graph storing and processing solutions is populated and diverse (see [19]). The basic categorization of software graph database solutions is the following. Graph computing technologies can be divided into two groups: 1) on-line, real-time, and 2) global analyzing, batch-processing graph databases. The former can be divided into in-memory, and persistent databases.

In-Memory Graph Toolkits The challenge of big data problems shed light on that the existing disk-based systems can not offer timely response due to the latency of hard disks. The role of storage shifted from the hard drive to the memory of the system. In-memory graph databases are constrained to graphs that fit into the main memory. Thus these systems are single-user systems that are oriented towards low-latency graph analysis.

The locality of data allows the usage of rich algorithm libraries and the choice of the adequate graph representation with respective space-time trade-off. The constraint of space allows large graphs (with millions of edges) to be stored and processed, but this may not be sufficient in all cases.

Example in-memory graph toolkits: *Apache Giraph*[7], *Microsoft Graph Engine*[8] (formerly *Trinity*), *Apache Spark GraphX*[9], *WhiteDB*[10].

Persistent, Real-Time Graph Databases Persistent graph databases are the prevalent group of graph computing technologies. Unlike in-memory graph tools, graph databases persist data on hard drives, thus are able to store billions of edges on a single machine and distributed systems can handle hundreds of billions of edges. These databases can provide multi-user concurrency, transactional semantics and eventual consistency.

Since global graph algorithms are not feasible, these systems are optimized for local neighborhood analysis and concurrent access. Global graph analytics are inefficient due to the communication overhead and the computational overhead, e.g., ensuring ACID transactional semantics.

Example for persistent, real-time graph databases: *InfiniteGraph*[11], *Neo4j* (see Section 2.3.3), *OrientDB*[12], and *Titan*[13].

Batch-Processing Graph Frameworks In case there is no real-time requirement for an analysis that accesses the whole dataset, batch-processing graph frameworks can be used for global analytics. Since there is also no requirement for quick response time, the applied algorithm can even scan data multiple times and leverage sequential reads from the disk. These systems can be used to periodically process data and feed back the results into real-time graph databases. Most of these frameworks utilize Hadoop for storage (HDFS) and processing algorithms implemented using the MapReduce paradigm.

Example batch-processing graph frameworks: *Apache Hama*[14] and *Apache Giraph*[15].

2.3.2 Evaluating Queries on a Data Structure

Numerous strategies exist for defining and executing queries over data structures. They can be defined in an imperative manner with programming languages like a navigation described in *Gremlin*[16] over a graph database. Declarative solutions also exist, where the query plan is

[7] http://giraph.apache.org
[8] https://www.graphengine.io
[9] https://spark.apache.org/graphx/
[10] http://whitedb.org
[11] http://www.objectivity.com/products/infinitegraph/
[12] https://orientdb.com
[13] http://titan.thinkaurelius.com
[14] https://hama.apache.org
[15] http://giraph.apache.org
[16] http://gremlin.tinkerpop.com

calculated from the query formalized in a declarative language (like *SQL*[17], *SPARQL*[18], and *Cypher*) and execution is performed by a query framework (such as *4store*[19] and *Neo4j*). Graph pattern matching – one of the declarative querying solutions, supplemented by imperative logic – forms the foundation of my approach.

Graph Pattern Matching

"Graph patterns are a declarative, graph-like formalism representing a condition (or constraint) to be matched against instance model graphs. The formalism is useful for various purposes in model-driven development, such as defining model transformation rules or defining general purpose model queries including model validation constraints. A graph pattern consists of structural constraints prescribing the interconnection between nodes and edges of a given type.

A match of a pattern is a tuple of pattern parameters that fulfill all the following conditions:

1. have the same structure as the pattern,
2. satisfy all structural and attribute constraints,
3. and does not satisfy any negative application condition (NAC) describing cases when the original pattern does not hold.

" [20]

2.3.3 Graph Databases

Since my approach is based on graph-based data handling, it is essential to employ the most suitable technique. Related to my approach suitable refers to having the following traits: being fast, flexible, versatile, and easy to use and deploy. In this section I give an overview on the most promising candidates and justify why I have chosen Neo4j as the foundation of my approach.

Neo4j

Neo4j is the most popular [21] and most mature NoSQL graph database, developed by *Neo Technology*. It is open-source, well-documented and continuously-developed. Neo4j is available in two editions: a free *Community Edition* and a paid *Enterprise Edition*. [22]

[17] Structured Query Language
[18] SPARQL Protocol and RDF Query Language
[19] `https://github.com/garlik/4store`

Architecture It can be deployed two ways: in *server mode* the database is started separately and listens for queries on its HTTP REST and Bolt interface; where in *embedded mode* it runs in the same JVM as the only client application.

Data Model The graph model of Neo4j is an implementation of a labeled property graph. The relations are labeled and the nodes can hold multiple labels; the nodes and relations can both hold properties.

Sharding Although the *Enterprise Edition* of Neo4j has a high availability solution and supports clustered replication and cache sharding, it does not support sharded data storage over a cluster of devices. This results in the advantage of low latency (clustering provides scale out capabilities for read), the ability to handle transactions with ACID (Atomicity, Consistency, Isolation, and Durability) semantics. [23]

Query Language Neo4j provides two methods for data queries out-of-the-box: an object-oriented native Java API for graph navigation and Cypher, a graph pattern description and query language with declarative and imperative traits. Additional interfaces may be loaded as a plugin, like the imperative Gremlin query interface [24].

One of Cypher's best features is the readability of its syntax (see Section 5.3 for examples). It provides an intuitive way to describe patterns of nodes and relations in the graph and also connections between subpatterns using bound identifiers for nodes and relations. Cypher also manages the indexes and constraints of the graph database. Negative application conditions (NAC) can be also expressed, describing constraints that should not hold for the matches.

Other interesting and useful features of Cypher include:

- Transitive closure over a set of edge labels, with optional repetition binding.
- Java procedures can be stored in the server as a plug-in and called from inside the queries. This way an arbitrary logic with multiple queries can be stored on the Neo4j server and executed with a Cypher query. This can solve cases where it is not possible to express the query in Cypher; ranging from duplicating a node (with parameters and labels) to inferring the metamodel (from the nodes and relations already in the database).
- Parameterized queries may be utilized for faster evaluation.

Alternative Graph Databases

Although other database solutions, like Titan [25] or OrientDB might serve as an alternative, the rapid development and feature set of Neo4j ruled them out as a candidate for my approach.

For rapidly changing, albeit smaller datasets, incremental querying using VIATRA [26] Query over EMF [27] models can be a solution. Since VIATRA stores the database in-memory, it is not suitable for my approach.

2.4 Integrated Development Environment (IDE)

Apart from the continuous integration workflow, static analysis tools are usually employed by developers within the integrated development environment they use. In this section I introduce one of the many freely available IDEs, and detail how static analysis tools can be integrated with it.

2.4.1 Visual Studio Code

Visual Studio Code [28] is Microsoft's take on a lightweight, yet powerful Integrated Developer Environment for modern programming languages. It is available for free for Windows, macOS and Linux. It comes with built-in support for JavaScript, TypeScript and has a growing ecosystem of extensions for other languages, theming, and developer support [29].

Debugging is also made easy, as the editor can be attached to the running code and the developer can add break points, look at call stacks and evaluate statements with an interactive console. With the relatively small package, Git support comes built-in: reviewing changed lines, staging files, making commits can be made right in the IDE.

IntelliSense

Visual Studio Code's syntax highlighting and autocomplete system is called IntelliSense, that also provides better completion based on variable types, function definitions, and imported modules. IntelliSense provides syntactical features like *format on type*, *outlining*; and also language service features like *peek*, *go to definition*, *find all references* and *rename symbol*.

To make these smarter functions possible, JavaScript service relies upon the TypeScript language service to handle JavaScript source code. It uses the same type inference system as TypeScript to determine the type of a value. (It recognizes the *"ES3-style"* class declaration.) Explicit JSDoc annotations can also be used, in case the type inference does not provide the desired type information. For major libraries it is also possible to download an import a type definition file.

Extensions

Visual Studio Code is built with extensions in mind. Extensions make it possible to add new languages, themes, debuggers, and to connect to additional services. The framework runs them in a separate process, ensuring they will not slow the editor down.

Every extension uses the same model to describe its contribution (how it is registered in the framework), activation (when it is loaded) and the same way to access the VS Code extension API. There are two special type of extensions: language servers and debuggers, which have their own additional protocols.

Extensions are the building blocks of VS Code. When activated, every extension runs in a shared host process, separate from the IDE. This ensures that the IDE itself can remain responsive even if an extension is resource-heavy or not well-written.

An extension is a package of source code, resources, and configuration files. They have support for:

- Activation – it is possible to specify when an extension is loaded: when a specific file type exists in the workspace or is opened; or when a command (described in the configuration) is executed via the *Command Palette* or the key combination.
- Editor – the extension can read and manipulate the editor's content.
- Workspace – the extension can access working files, modify the content of the status bar and show information messages (and more).
- Eventing – it is also possible to subscribe and react to the life-cycle events of the editor such as: open, close, change events of the editor (and more).
- Evolved editing – rich language support can be provided, including IntelliSense services, peek, hover and diagnostic (info, warning and error messages).

Language Servers The language server framework and its sample implementation helps developers create a dedicated process for resource-heavy language server applications. It is the better design choice if the extension may slow down other extensions while working. Its possibilities are limited, as custom communication between the client extension and the language server needs modification in the underlying communication protocol handler.

Debuggers Connecting an external debugger written for any language to VS Code is also possible through the VS Code Debug protocol.

2.4.2 Alternative IDEs

There are several alternatives to Visual Studio Code. *Atom*[20], *Eclipse*[21], and *WebStorm*[22] can all be extended with a plugin adding extra features for a given language. Since VSC is one of the

[20] `https://atom.io`
[21] `https://eclipse.org`
[22] `https://www.jetbrains.com/webstorm/`

most actively developed lightweight IDEs aiming for JavaScript (and TypeScript) development, I have developed an extension integrating the IDE with my framework (see Section 6.3).

Chapter 3

Related Work

In this chapter I enumerate the most notable similar systems and approaches, while discussing the related scientific research.

3.1 Static Analysis Frameworks

This section briefly introduces the various static analysis approaches and available tooling for JavaScript: what they aim for, and with what kind of limitations.

3.1.1 Tern

"Tern is a stand-alone code-analysis engine for JavaScript. It is intended to be used with a code editor plugin to enhance the editor's support for intelligent JavaScript editing. Features provided are:

- Autocompletion on variables and properties
- Function argument hints
- Querying the type of an expression
- Finding the definition of something
- Automatic refactoring

Tern is open-source (MIT license), written in JavaScript, and capable of running both on node.js [30] and in the browser." [31]

The Tern suite is a modular, extendable stand-alone system. Editor plugins communicate with the Tern server module, connected to the Acorn parser (introduced in Section 3.2.1) and the inference engine. Third-party plugins can introduce implementation environmental or

behavioral information for the system, for example ECMAScript module loading rules, or node.js specific variables. [32] It uses the AST structure of Acorn (detailed in Section 3.2.1), extends the in-memory representation with type information and propagates it in the graph.

3.1.2 TAJS

Type Analyzer for JavaScript (TAJS) [33] is a dataflow analysis tool infering type information and call graphs developed by the programming language research group at Aarhus University, with contributions from Universität Freiburg.

The current version (as of 2016) can model scripts of ECMAScript 3; it also contains model of the standard library and partial model of the HTML DOM and browser API. [34] The initial aim of TAJS was to warn programmers about the following problematic cases. This enumeration follows [8].

- invoking a non-function value as a function
- reading an absent variable
- accessing a property of `null` or `undefined`
- reading an absent property of an object
- writing to variables or object properties that are never read
- implicitly converting a primitive value to an object
- implicitly converting `undefined` to a number
- calling a function object both as a function and as a constructor or passing function parameters with varying types
- calling a built-in function with an invalid number of parameters or with a parameter of an unexpected type

3.1.3 TRICORDER

TRICORDER [35] is a pluggable program analysis platform used internally at Google, helping developers and reviewers notice possible problems with code changes. The system mainly supports C++, Go, and Java codes, but it has support for JavaScript too.

Related researches show that static analysis tools are either not used or ignored, when not configured correctly and take more time from the user than necessary. “High false positive rates, confusing output, and poor integration into the developers’ workflow all contribute to the lack of use in everyday development activities [36, 37].

TRICORDER introduces an effective place to show warnings. Given that all developers at Google use code review tools before submitting changes, TRICORDER's primary use is to provide analysis results at code review time. This has the added benefit of enabling peer accountability, where the reviewer will see if the author chose to ignore analysis results." [35]

3.1.4 jQAssistant

"jQAssistant [38] is a QA tool which allows the definition and validation of project specific rules on a structural level. It is built upon the graph database Neo4j and can easily be plugged into the build process to automate detection of constraint violations and generate reports about user defined concepts and metrics.

Example use cases:

- Enforce naming conventions, e.g. EJBs, JPA entities, test classes, packages, maven modules etc.
- Validate dependencies between modules of your project
- Separate API and implementation packages
- Detect common problems like cyclic dependencies or tests without assertions

" [39]

jQAssistant mainly aims at processing and validating Java projects during the continuous integration process. Its approach is similiar to mine, it also uses the same database backend — Neo4j — for storing the source code representation. However, instead of analyzing the semantics of the source code, it concentrates on the structural details and connections within the codebase.

3.1.5 Facebook Flow

Flow [40] is an open-source static type checker for JavaScript, written in OCaml, utilizing annotation-based language extension and type inferencing. The implicit infered types can be corrected by explicit type annotations helping the framework. Exported variable, function, and class declarations have to be explicitly annotated.

Since Flow requires a modified language, a new compilation step is required in order to omit the annotations and produce a source code in pure JavaScript. It is also possible to provide interface files for Flow, enabling to import third party libraries.

Flow utilizes control flow analysis, allowing property access detection on null or undefined values. It also executes analyses in an incremental fashion, storing the intermittent results in a persistent server and updating it in the background when the source code is changed.

3.1.6 JSNice

JSNice [41] is a scalable prediction engine based on Nice2Predict [42], a learning framework for program property prediction. The novel approach of JSNice utilizes machine learning; first it learns a probabilistic model from existing, prepared data, then predicts properties for unseen programs based on this model. Besides predicting names of identifiers, it also predicts type annotations of variables, thus producing both syntactic and semantic information. [43]

3.1.7 Infernu

Infernu [44] is a type checker for JavaScript, written in Haskell. "Infernu's type system is designed for writing dynamic-looking code in a safe statically type-checked environment. Type annotations are not required [...] instead, Infernu infers the types of expressions by examining the code. If the inferred types contradict each other, Infernu reports the contradiction as an error." [44]

Its type system is based on Damas-Hindley-Milner type system, and it places restriction on the elements and expressions that can be expressed, thus the grammar of Infernu is a subset of JavaScript's.

3.1.8 Comparison

Although several tools are available, they are not widely used. This thesis aims to find out why and whether a graph pattern matching based approach can solve the issues and act as an universal framework. Table 3.1 concludes some of the common features of the previously mentioned tools.

	Flow	**jQA**	**Tern**	**TAJS**	**TRICORDER**	**My approach**
Open-source	●	●	●	●	○	●
Linting	●	◐	◐	○	●	●
Handles multiple files	●	●	●	○	●	●
Dead code detection	○	◐	○	○	◐	●
Type inferencing	●	◐	●	●	◐	◐
Languages	JS*	Java	JS	ESX	JS, Go, Py	JS

Table 3.1 Comparison of static analysis frameworks.

3.2 JavaScript Parsers

In this section I showcase the most used, trending JavaScript parser technologies and justify why I have chosen the Shape Security Shift family as the parser and additional toolset for my approach.

3.2.1 Acorn

Acorn [45] is an open-source, small JavaScript written in ECMAScript 6 (see Section 2.1.2). It is up-to-date, able to parse ECMAScript version 3, 5, 6, 7, and the newest one, 8. The resulting AST structure of the parser conforms the ESTree specification [46].

It is also able to parse multiple files into a single AST, connected with a `Program` node. To analyze and navigate in the resulting AST, Acorn provides a walker interface, to be used with a visitor pattern based algorithm.

The parser is also configurable with several options, `locations` being one of the most useful for my approach. Setting this option stores the location of the represented source snippet in the AST node. There is also an error-tolerant version of the parser enabling parsing unfinished or syntactically incorrect sources.

3.2.2 Esprima

Esprima is also an open-source, ECMAScript standard-compliant parser. It fully supports ES7, and produces ESTree models. It has a great user-base and several tools depend on it. It also has experimental support for JSX, an XML syntax extending JavaScript for React [47], “a declarative, efficient, and flexible JavaScript library for building user interfaces” [48].

3.2.3 Shift

The Shift [49] family consists of several tools. Besides the parser, there is a scope analyzer, a code validator, fuzzer, and a code generator, besides others.

Shift AST

The reason behind the number of tools is due to the fact that Shift does not conform the ESTree specification. In 2014, Shape Security, the company behind Shift announced a new JavaScript AST specification [50]. This specification was developed with ECMAScript 6 in mind, along with analysis and transformation.

The specification describes interface for an AST syntax that can represent the structure of an ECMAScript source code. According to Shape Security, a “good AST format…

– minimizes the number of inhabitants that do not represent a program.

– is at least partially homogenous to allow for a simple and efficient visitor.

– does not impede moving, copying, or replacing subtrees.

– discourages duplication in code that operates on it.

” [51]

Shift Scope Analyzer

“The Shift Scope Analyser produces a data structure called a scope tree that represents all of the scoping information of a given program. Each element of the scope tree represents a single scope in the analysed program, and contains many pieces of information, including:

– the scope type (there are 12 of them!)

– the AST node associated with the scope

– variables declared within that scope, each of which points to its declarations and references

– whether the scope contains a with statement or direct call to eval, making it dynamic

” [13]

Additional Notes

The Shift family has other interesting members and features as well:

– Besides JavaScript, most of the Shift family is also available for Java projects. This makes it easier to integrate it with projects and tools only available for Java.

– Shape Security has a project, Bandolier [52] for packaging projects with ES6 modules into a single JavaScript file, imitating the export-import mechanism, related to my approach.

– Shape Security is also developing a semantic transformer [53] for ECMAScript ASTs, also related to my approach.

3.2.4 Comparison of Parser Technologies

In order to find a best parser and related technologies, I had to compare them: measure their speed, investigate their parameters and output model, and transform their extra functions into potential features of my approach.

Speed

Although speed is not the most important property of a system aiming to make sure no errors are present, swift response can boost the performance of the user. Table 3.2 shows the time difference between parsers processing various source codes repositories. The benchmark [1] was run on a personal computer. Its sole purpose is to get a rough comparison between the different technologies available running in a JavaScript environment.

It is visible that Shift NEE[2] is one of the fastest parsers available.

Source	Esprima[3]	UglifyJS2	Traceur	Acorn[4]	Shift	Shift (NEE)
jQuery.Mobile[5]	154.0 ±22.3%	244.6 ±8.4%	304.6 ±15.1%	215.3 ±16.9%	480.7 ±13.1%	119.9 ±11.9%
Angular[6]	125.5 ±16.3%	212.2 ±11.2%	254.1 ±20.7%	146.3 ±18.6%	452.7 ±12.5%	94.6 ±18.2%
React[7]	134.7 ±10.8%	221.6 ±8.9%	258.5 ±13.4%	176.9 ±15.6%	496.4 ±11.6%	116.1 ±14.2%
Total	**414.3 ms**	**678.4 ms**	**817.2 ms**	**538.5 ms**	**1429.8 ms**	**330.6 ms**

Table 3.2 Speed comparison of JavaScript parsers.

Metamodel and Precision

For analysis and transformation purposes it is important to have a model with as much and as precise information as possible. If the parser produces a model conforming a detailed metamodel, it is easier to differentiate seemingly similar cases.

Based on the comparison [51] between the ESTree and the Shift AST specification, it is visible that Shift is a better choice if more details are required.

[1] `http://esprima.org/test/compare.html`
[2] Early error checking disabled. NEE – No Early Errors
[3] Esprima version 2.7.2
[4] Acorn version 2.4.0
[5] jQuery.Mobile version 1.4.2
[6] Angular version 1.2.5
[7] React version 0.13.3

Chapter 4

Overview of the Approach

In this chapter I introduce the high-level architecture overview of the proposed framework. The chapter also discusses each component in detail including details on how they cooperate.

4.1 Architecture

Figure 4.1 shows the overview of the framework's architecture. Since the novelty of the approach is how the source code representation is handled — stored, transformed, and queried — the essence of the approach is visualized on the right half of the figure. This is embedded and utilized in the framework itself, and integrated into the continuous integration circle and user-facing systems.

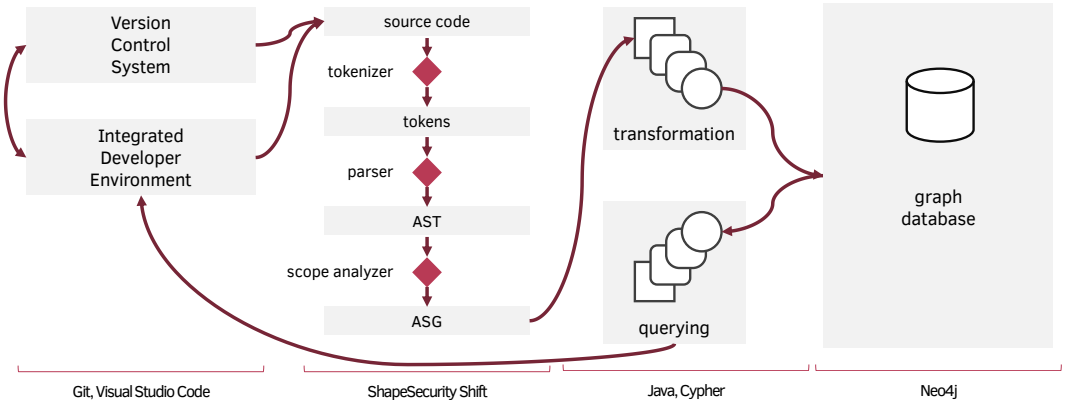

Figure 4.1 Architecture overview of the approach.

4.2 Main Components

4.2.1 Workflow Integration Points

The proposed framework is intended to be integrated with at least two types of development tooling. First, connecting to the *Version Control System* makes it possible to extend the continuous integration workflow and provide the users of the system with current information about the codebase they are working on. Second, connecting to the *Integrated Development Environment* empowers their users with feedback on the current modifications.

Version Control System (VCS)

Version control systems are change management systems for files (e.g., documents, software code). They store revisions of the current set of files managed by the system. Each revision is differentiated with a timestamp and also the person performing the changes are associated with a revision.

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

Integrating a VCS into the architecture makes it possible to extend the workflow with the features of the framework. By automatically calculating the changeset and forwarding this information to the framework, it is possible to keep an up-to-date representation of the version controlled data source.

The most known implementations are Git [54], Subversion (SVN) [55], and Mercurial (Hg) [56].

Integrated Development Environment (IDE)

An IDE is an application for (software) developers that integrates several tools making it easier to write, compile, and test the product. Integrated development environments are detailed in Section 2.4.

In the architectural overview, the IDE also represents the working set of the software and its dependencies available on the developer's computer. This working set also contains the developer's modifications that are not yet transferred to the shared VCS.

4.2.2 Transforming the Source Code

One of the most important third party components of the approach is the source code parser. This component is used to transform a given source file, a *compilation unit* into a model

representation of the syntax and semantics of the source code. The process itself is detailed in Section 2.2.3.

Since *Shape Security Shift* (detailed in Section 3.2.3) suits my approach the best (see comparison of JavaScript parsers in Section 3.2.4), I have used the Java implementation of the *Shift Parser*, and *Scope Analyzer*.

4.2.3 Graph Maintenance

The novelty of my approach is how it processes, stores and connects the source code representation in a graph database. In this section I discuss how the subgraphs of the source code files are prepared and then connected to each other constructing a connected graph representing the whole source code repository.

The process can be summarized in 4 concise sentences:

1. The repository is transformed one file-by-file.
2. The ASG *model* is transformed into a *property graph*.
3. After every file is processed, the ASG subgraphs are interconnected using graph transformations.
4. In case a file is added/modified/deleted, the corresponding subgraph is also added/removed and reprocessed/removed from the graph.

This process and the algorithm for graph maintenance is detailed in Section 4.3.3.

4.3 Steps of Processing

The following enumeration presents the basic algorithm for processing, transforming, storing and analyzing the source code repository.

4.3.1 Initial State

The graph representation of the repository follows the development of the source code. Thus initially both the graph database and the repository are empty. When the database is initialized, metadata and initial database structure can be inserted, so the queries executed later can build on the existence of this structure.

4.3.2 Calculating the Changes to Propagate

Once there are modifications published in the repository or the IDE, it is the integrating tool's responsibility to notify the framework. This can be an event hook for the IDE or the CVS, for example. The changes to a file at a given path may be the following:

– *Addition.* In case a new file is added, it is processed, and the subgraph representation is stored in the database.

– *Modification.* Since the approach is incremental with a file-level granularity, the modification of a file's contents requires the removal of the whole old and addition of the new subgraph.

– *Removal.* The removal of a file causes the removal of the subgraph too.

4.3.3 Maintaining the Graph

The file changes are processed one-by-one. The integrating tool notifies the framework, specifying the path and content of the file along with additional metadata. The content is then preprocessed using the parser resulting in a model representation.

Transforming the Instance Model

This representation of the parser conforms the metamodel based on the syntax and semantics of the source language. But the property graph database can not import the data in this form, thus it needs to be transformed. Figure 4.2 presents the transformation of a Java object structure into a graph based on the basic rules detailed in this section.

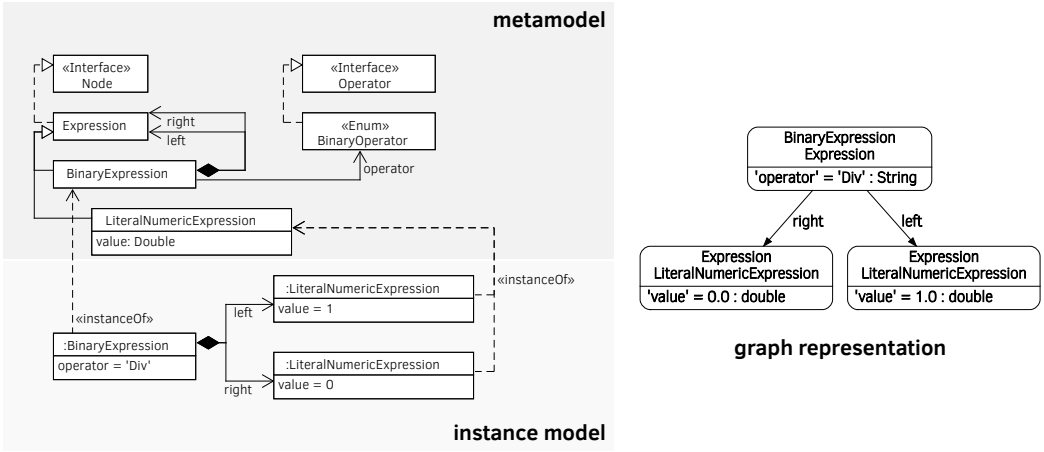

Figure 4.2 Transformation of a Java object structure into a graph.

In order to acquire information from this model and transform it, one either knows its metamodel and iterates over every element or — if the programming language of the parser allows — uses reflection. In my approach I use the combination of the Shift parser written in Java — thus using the reflective approach — and the Neo4j labeled property graph database.

Typed Nodes Every AST or ASG node is a node in the graph. The graph node is typed with the class, superclasses and interfaces of the represented model entity.

Node Properties Every property of a model entity — regardless of whether they are its own or inherited from its supertypes — are also transferred to the graph node. These properties are also stored with their basic type: *double*, *string* or *boolean*.

Labeled Relations References in the model are represented in the graph as relations, directed edges. The relations are labeled after the name of the references.

Collections The metamodel also contains several reference collections: maps, lists and tables. Maps and tables are represented in the graph as a new node, with appropriately labeled relations to the referred nodes.

Lists are transformed similarly. In order not to lose ordinal information, the items are related to the referrer with two routes (see Figure 4.3).

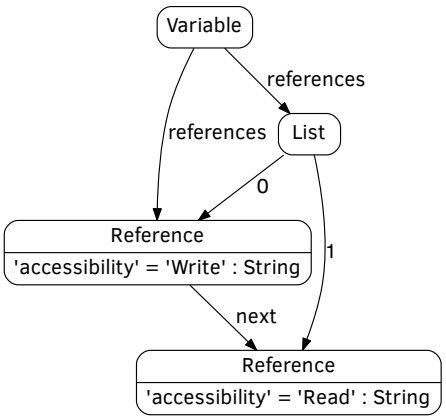

Figure 4.3 Graph representation of a list.

The first route is direct; the referrer has a relation to every item of the list (labeled with the name of the list). The second route has an additional *List* node between the two, and its relations are labeled with the index of the item. The sequential items also have a direct, chaining relation.

Source Code Location To be able to report precise location for a problem, the graph also contains line and character information for the beginning and the end of every model entity.

Storing the Graph Representation

If the parser successfully processes the source code, the resulting ASG is transformed using the aforementioned rules. In a writing transaction, the transformed graph is serialized and stored in the database.

Metadata Besides the ASG, additional properties are stored. For example, for every source file, a node exists in the graph connected to nodes belonging to the particular ASG. This enables handling nodes of a file as a whole.

Multilayered Graph With more complex rules it is also feasible to store multiple versions of the same file in the database. This yields the ability to maintain multilayered versioning for multiple users.

If the VCS supports branching the code development, the framework could also accommodate this behavior. For example if two developers work on the *master* development branch in their own IDE, there could be layers for all of them.

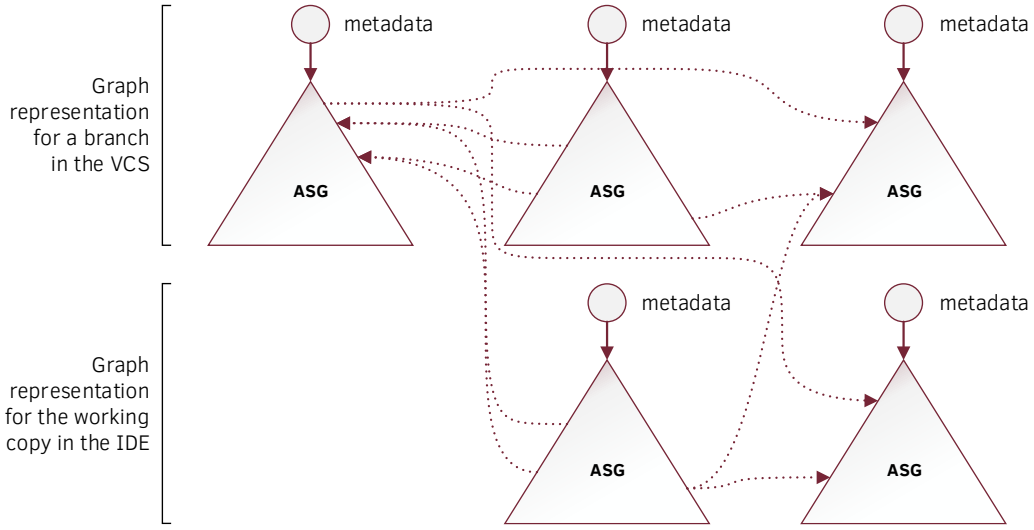

Figure 4.4 Multilayered graph.

One layer contains every ASG for the *master* branch. One layer for each user, containing only the ASGs of their modified files. ASGs in these are connected if both are present in the same layer. If not, they are connected to the corresponding file in the layer of the *master* branch (see Figure 4.4).

Transforming the Graph

After the data has been stored in the database, it can be freely transformed with framework- or user-defined rules. This step may be utilized for transforming the set of subgraphs into one connected graph based on the export and import rules of the language (detailed in Section 5.4).

Neo4j allows the execution of in-place transformations, where querying with pattern matching and manipulations may be declared in the same query. The possibilities of Neo4j are detailed in Section 2.3.3. Transformation examples are presented and visualized in Chapter 5.

4.3.4 Executing Graph Queries

By utilizing the built-in pattern-matching abilities of Neo4j, it is easier and more user-friendly to write pseudo-graphic, declarative patterns to find a structure in the graph — compared to the generally used visitor patterns. [20] An example graph pattern query is detailed in Section 5.3.

If the query language of Neo4j, Cypher is not powerful enough to express logic in a standalone query, one may also employ arbitrary Java code. This code may be used inside the query or even command multiple queries and aggregate the result.

Chapter 5

Elaboration of the Workflow

In this chapter I demonstrate the various steps of the static analysis workflow in detail. I employ small working source code examples in order to demonstrate the steps of the source code transformation. A prototype implementation of the proposed workflow is available as an open-source project at `https://github.com/FTSRG/codemodel-rifle`.

5.1 Transforming the Source Code Into an AST

As mentioned in Section 4.2.2, this transformation is carried out by the Shape Security Shift toolkit. The source code is passed to the parser along with the parameters for the parsing algorithm. Considering the new constructs introduced in ES6 (Section 2.1.2), I've chosen to parse every file as a *Module*. This affects the grammar used in the parser and the resulting data model.

The parse result of the one-line code snippet in Source 5.1 was previously presented in Figure 2.4.

```
var foo = 1 / 0;
```

Source 5.1 Basic example source code.

5.2 Storing the ASG in the Graph Database

Once the source code is parsed and the ASG is returned, the framework updates the stored graph representation. Since Section 4.3.3 details how the Java Object structure is transformed into a graph, this section only describes the queries used for the maintenance.

When a new file is added to the repository, there is no need to prepare the database. A new metadata node is created with the name and path of the file, and optionally the session identifier of the IDE it has been created and sent in to the framework. The subgraph representing the ASG is then inserted in a database transaction.

The newly created metadata node acts as a subgraph selector, since it is connected to every node belonging to the file (of the given session). This makes it easy to remove the file upon the removal of the file it represents.

In case the file has been modified, the nodes connected to its metadata node are removed, then the new graph is inserted.

5.3 Division by Zero

As one of the most basic and easy-to-discover mistakes, division by zero should be reported to the developer. In the context of mathematics, division by zero is undefined for the real numbers. In JavaScript, it may result in `NaN` or `Infinity`.

Without dynamic testing or symbolic execution it is rather hard to find this kind of expression, since the right side of the division may come from a variable. On the other hand, finding the cases where the right side is a literal is trivial.

Taking a look at Figure 5.1, the graph representation of the previous example, it is relatively easy to find the problematic nodes. A natural language definition of the problem would sound like this: „we are looking for binary expressions that have a literal zero on the right side”.

One way to formalize this declarative definition in Cypher can be seen in Source 5.2. Executing this query on the AST of the source code would 1) find matching nodes with the right type, 2) bind them to the corresponding names, and 3) check whether the required relations are present, resulting in a state like in Figure 5.2. Finally, the result is the `BinaryExpression` node. By querying the source code location (also stored in the graph connected to the node) this query can report the precise location of the expression in the source code as problematic.

5.4 Handling Import and Export

Since the beginning, JavaScript projects have grown tremendously. With the size of the average project, the language has also matured for handling bigger, more complex code repositories. With ES6, a module syntax has been introduced to spread the codebase into files, even directories. (Even before ES6 there have been several module systems, but ES6 described a standard for it.)

As described previously, I instruct the parser to process source files as modules. An ES6 *module* differs in two ways from a *script*: it is automatically interpreted as a *strict-mode code*,

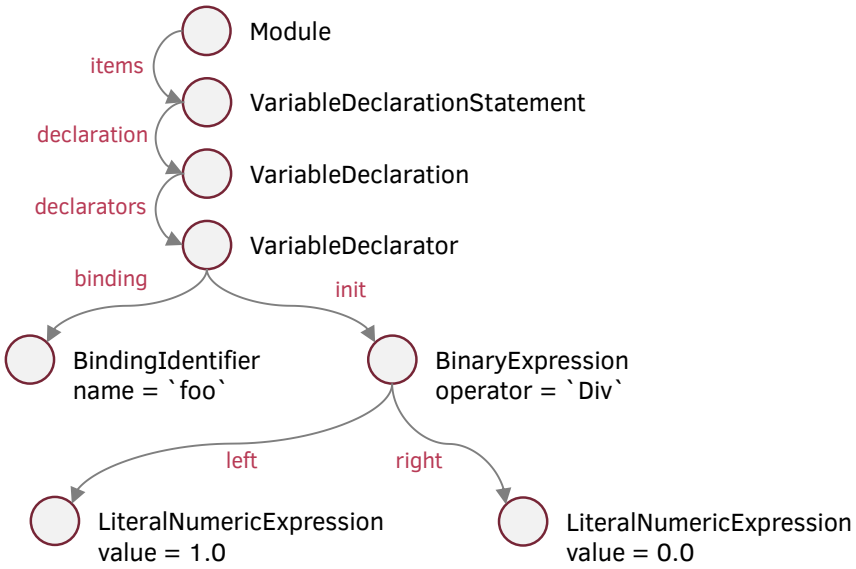

Figure 5.1 Graph representation of the code snippet of Source 5.1.

```
MATCH   (binding:BindingIdentifier)
        <-[:binding]-()-->
        (be:BinaryExpression)
        -[:right]->(right:LiteralNumericExpression)
WHERE      be.operator = 'Div'
       AND right.value = 0.0
RETURN binding
```

Source 5.2 Graph Pattern Matching Division by Zero.

and one can use `import` and `export` statements in them.

This section follows [57, 58, 59] and collects the most important details about the specifications.

5.4.1 Export

Everything declared inside a module belongs to the scope of the module. In order to let other modules to access them, they need to be explicitly exported.

Source 5.3 lists the various ways declaring a feature export. Just to list a few combinations; one may, for example, export expressions, variables, function and generator declaration statements either named, or using an alias. Or have default export expressions, even in an

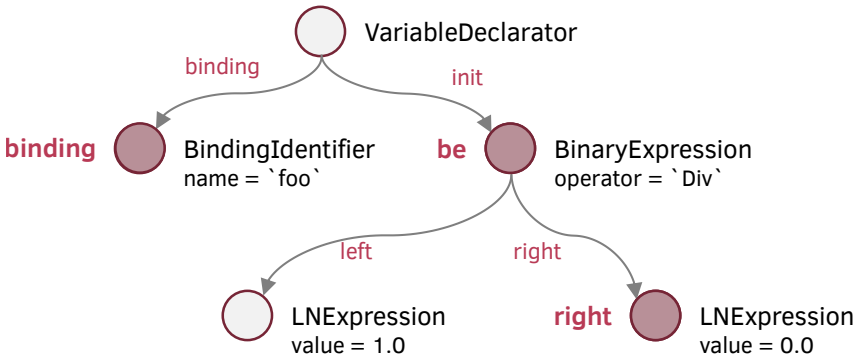

Figure 5.2 Graph representation of the query describing division by zero.

export statement.

```
export { name1, name2, . . . , nameN };
export { variable1 as name1, variable2 as name2, . . . , nameN };
export let name1, name2, . . . , nameN; // also var
export let name1 = . . . , name2 = . . . , . . . , nameN; // also var, const

export default expression;
export default function . . . () { . . . } // also class, function*
export default function name1. . . () { . . . } // also class, function*
export { name1 as default, . . . };

export * from . . . ;
export { name1, name2, . . . , nameN } from . . . ;
export { import1 as name1, import2 as name2, . . . , nameN } from . . . ;
```

Source 5.3 ES6 export statement examples from MDN.

Parsing these export statements result in different representation in the instance model and the resulting graph. Figure 5.3 shows an example inline default export statement for a function named `foo`: `export default function foo() {}`. Note that the export statements are listed in the `Module`, but not in any of the `Scopes`.

5.4.2 Import

Like there are many ways for exporting features in a module, there are also several ways for importing them. Source 5.4 lists these.

The imported features are placed in the module-level global scope as a `Variable` without their

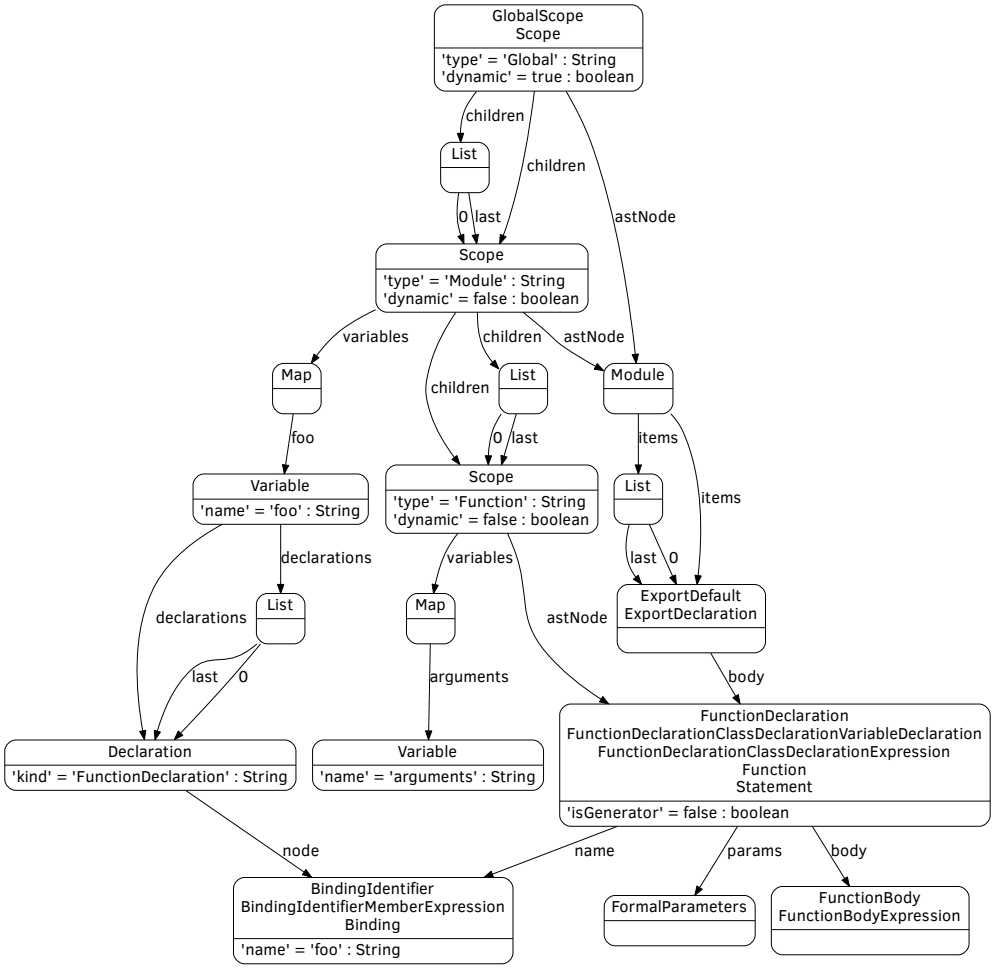

Figure 5.3 ES6 export statement example.

```
import defaultMember from "module-name";
import * as name from "module-name";
import { member } from "module-name";
import { member as alias } from "module-name";
import { member1 , member2 } from "module-name";
import { member1 , member2 as alias2 , [...] } from "module-name";
import defaultMember, { member [ , [...] ] } from "module-name";
import defaultMember, * as name from "module-name";
import "module-name";
```

Source 5.4 ES6 import statement examples from MDN.

declaring node, and the import statements are also present in the AST. Figure 5.4 shows the ASG of a module importing and then using a function declaration (exported in the previous module): `import foo from "export"; foo();`.

5.4.3 Connecting Imports to Exports

Since there are several ways to both export and import features, there are even more combinations. This thesis does not aim to cover all of these, but to show that it is possible to connect graph module representations based on simple rules.

Executing the following steps emulates the resolution made by the interpreter of the source codes and connects the usages of the imported feature to the declaration of the exported one:

1. Find the imported `IdentifierExpression` in the `items` list of the `GlobalScope` node.
2. Find the connected upstream `Variable` node.
3. Find the `Declaration` for the `Export` that has a `Node` with the same `BindingIdentifier` as the import.
4. Connect the `Variable` on the import side to the `Declaration` on the export side with a `declarations` relation.

The Cypher query in Source 5.5 manages to connect the exact type of imports and exports mentioned previously. Note that this query does not search for a file exporting the statement. For the correct match it should also check the metadata, whether the found node was declared in a file with matching absolute path.

After executing the query in Source 5.5, the two ASG subgraphs are connected with a new edge (see Figure 5.5). This enables executing global-level queries addressing multiple modules. These newly added relations are always removed when either the source or the destination is

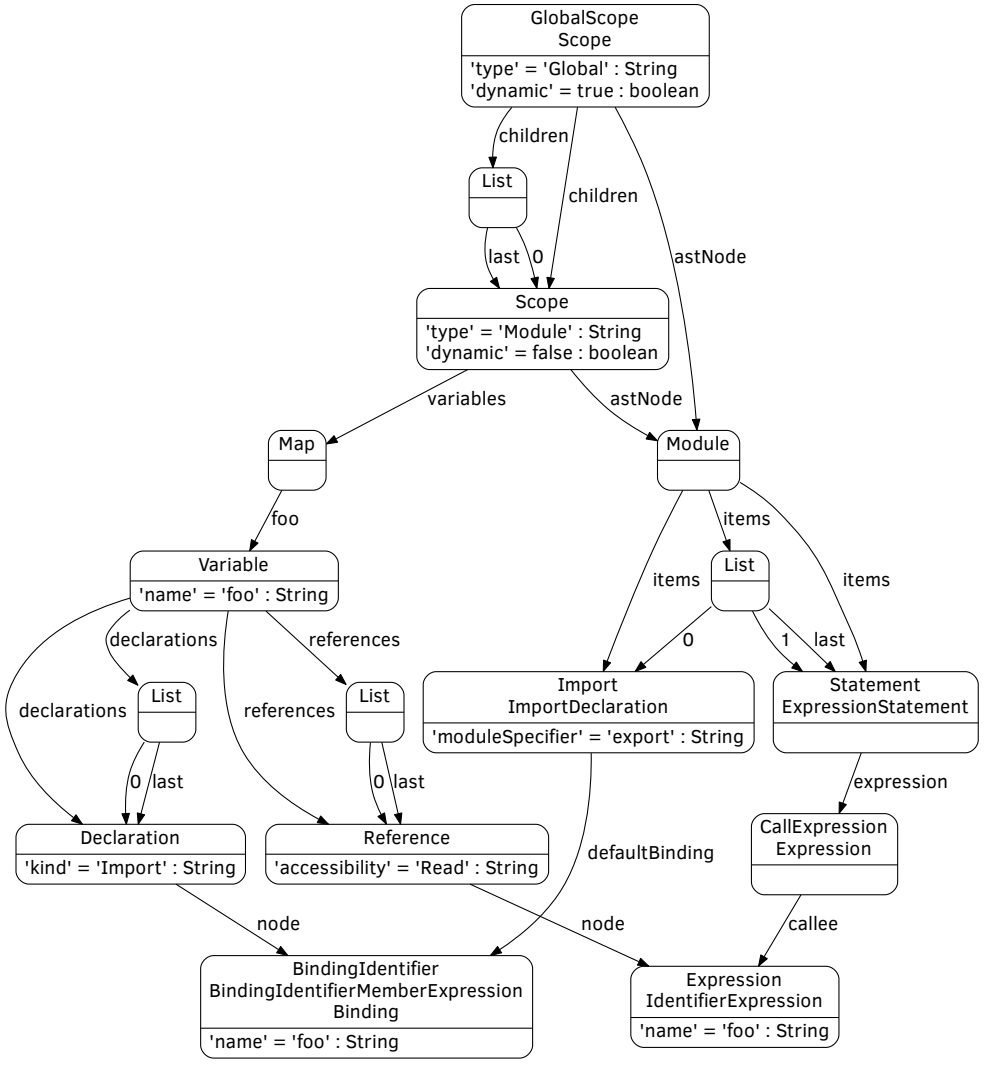

Figure 5.4 ES6 import statement example.

```
MATCH
    (:ImportDeclaration)-[*]->(importIdentifier:BindingIdentifier)
MATCH
    (:GlobalScope)-[:through]->(:HashTable)-[]->(reference:Reference)
    -[:node]->(:Either)-[:data]->(identifier:IdentifierExpression)
MATCH
    (:ExportDeclaration)-[:declaration]->(:Node)-[:name]
    ->(exportIdentifier:BindingIdentifier)

MATCH
    (exportIdentifier)<-[:node]-(declaration:Declaration)
MATCH
    (reference)<-[:references]-(variable:Variable)

WHERE
    importIdentifier.name = identifier.name AND
    exportIdentifier.name = importIdentifier.name

MERGE
    (variable)-[:declarations]->(declaration)
```

Source 5.5 Cypher query for connecting import and export statements.

removed, in case a module is modified. Thus it is optimal to run this query only when all of the files are already processed and the chance of modification in the near future is low. (This might happen after a modification is sent to the VCS or the developer saves the files in the IDE.)

5.5 Dead Code Search

Code that is written, but not used in the whole codebase is often called *dead code*. Without symbolic execution it is a complex problem to decide what part of the code is reachable, but (excluding dynamic evaluation, e.g., `eval` in JavaScript) it is feasible to find function declarations in a file which are possibly not referenced in the local scope.

5.5.1 Search Algorithm as a Graph Query

If a function declaration is not exported from the module and no exported declaration references it, it is highly likely that the declaration is unreachable and contains *dead code*. With the following steps and the Cypher query described in Source 5.6 it is possible to locate dead code in a file.

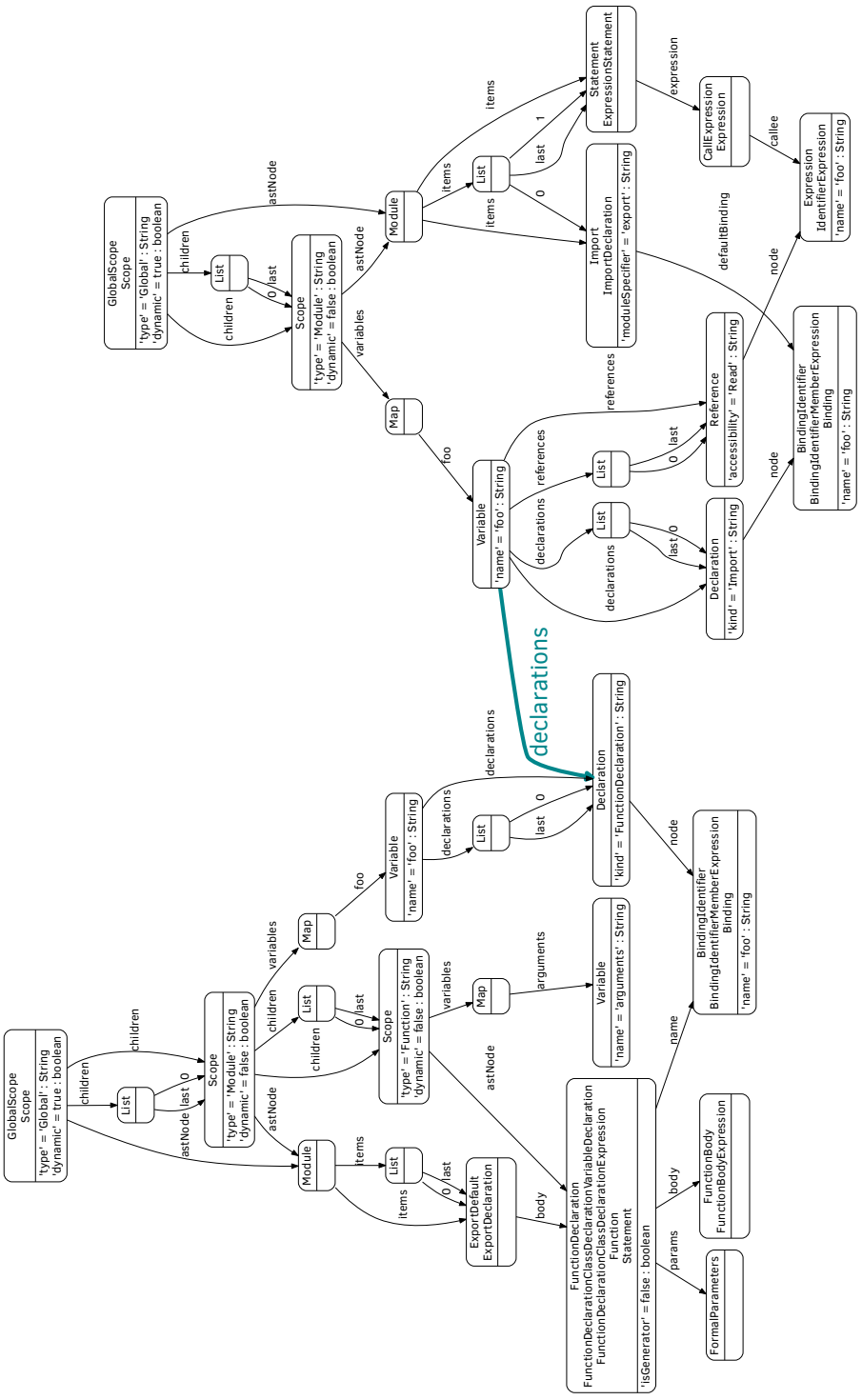

Figure 5.5 Connected ASG subgraphs.

1. Find the called `FunctionDeclarations` target for every `CallExpression`.
2. Find every call from the body of a function.
3. Create a `calls` relationship between the caller and the callee `FunctionDeclaration`.
4. Find the exported `FunctionDeclarations` that can be entrance points.
5. Find every `FunctionDeclaration` that should be available through the entrance points.
6. Return every `FunctionDeclaration` that are not in this list.

5.5.2 Evaluating the Result

For the sake of conciseness, I present the query for a simplified version of this algorithm, which only works with a single module in the database.

Compared to other commercially available software, this solution is able to detect more dead code scenarios: e.g., circular references without incoming calls (see Figure 5.6. Instead of reporting the problems one-by-one or layer-by-layer, it reports the clique, without user input.

5.5.3 Transformation in a Transaction

This query utilizes several possibilities only available in Neo4j. Since the query modifies the database, but the modifications are derived information, they should not be stored in the database. Database transactions that can be discarded, but still query the modified database are highly utilized here.

It is not possible to declare transitive closure[1] over arbitrary node type and edge label sequence in one Cypher query. This can be tackled by utilizing the *trick* of starting a new transaction, writing the database and immediately querying the modified created data, then discarding the transaction. One can match and replace every sequence with a new, dedicated labeled edge and declare a transitive closure over it, but this requires the database to be modified (see Figure 5.7).

5.6 Control Flow Graph (CFG)

Control Flow Graphs (CFG) are graph representations of the computation and control flows in the program. In this graph the nodes represent statements that are not interrupted by control changes. Directed edges represent possible flow of control between the two nodes. Nodes can have multiple incoming and outgoing edges. [60] Figure 5.8 presents an example CFG.

[1] Transitive closure is marked with an asterisk in Cypher.

```
// Prepare call edges
MATCH
    // Match called FunctionDeclarations for every CallExpression
    (call:CallExpression)-[:callee]->(:IdentifierExpression)
    <-[:node]-(:Reference)<-[:references]-(:Variable)
    -[:declarations]->(:Declaration)-[:node]->(:BindingIdentifier)
    <-[:name]-(fd:FunctionDeclaration)
MATCH
    // List every call from a function body
    shortestPath(
      (fun:FunctionDeclaration)-[*]->(call:CallExpression)
    )

MERGE
    // Create a calls relationship between the caller
    // FunctionDeclaration and the called FunctionDeclaration
    (fun)-[:calls]->(fd)
;

// Get not used FunctionDeclarations
MATCH
    // Find the exported FunctionDeclaration that
    // may be an entrance point
    (main)-[:items]->(:ExportDeclaration)
    -[:declaration]->(fd:FunctionDeclaration)

MATCH
    // Find every FunctionDeclaration that should
    // be available through the entrance points
    (find:FunctionDeclaration)

WHERE
    // List the ones that are not reachable from the
    // entrance nodes (thus are not the entrance nodes "<>").
        ( NOT (fd)-[:calls*]->(find) )
    AND ( find <> fd )
    AND ( main:Script OR main:Module )

RETURN
    find
```

Source 5.6 Cypher queries for finding dead code.

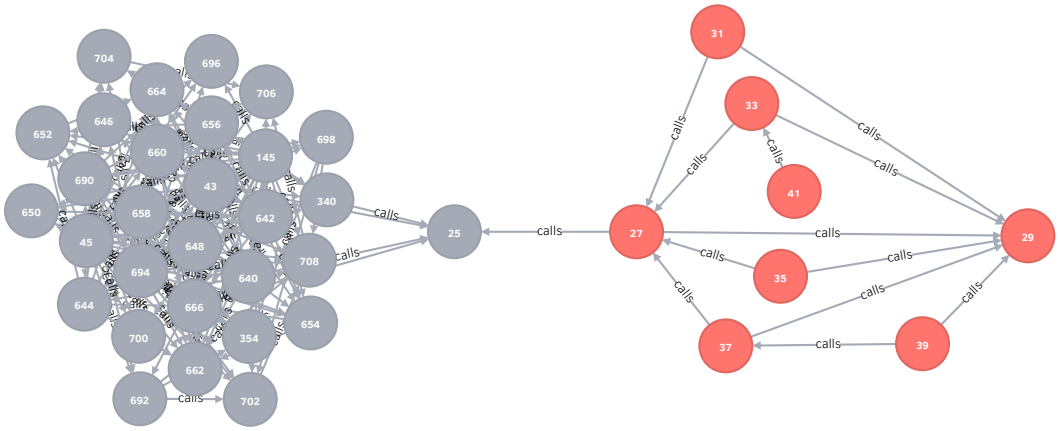

Figure 5.6 Call graph with self-referencing clique in red.

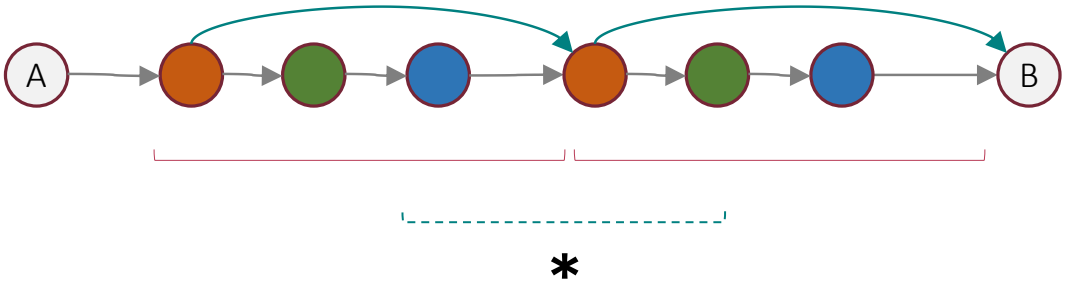

Figure 5.7 Transitive closure over arbitrary node type and edge label sequence.

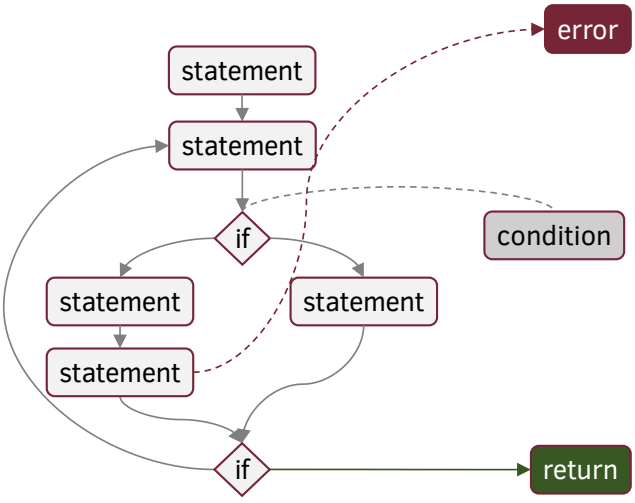

Figure 5.8 Example control flow graph.

In this section I demonstrate how to transform the basic ASG structures into a CFG, and to utilize the result for static analysis.

5.6.1 Theoretically Possible Paths

As previously mentioned, nodes can have multiple incoming and outgoing edges. At execution time when the state of the program is at a given CFG node — based on the dynamic state of the execution — it may continue with any statement of the corresponding subsequent nodes.

Without running or interpreting the source code it is only possible to find and map every scenario in the ASG and transform them into nodes and edges in the CFG. A possible execution path is a path in the graph from one of the entry nodes to one of the exit nodes. The constraints on this path may be unsatisfiable, thus an execution of the source code may not iterate over this path. The CFG thus contains every feasible, and some infeasible execution paths.

5.6.2 Transforming by Node Type

There are several ways to transform the ASG into a CFG. Since my approach utilizes a graph database, the framework transforms the graph one smaller subgraph structure at a time, utilizing on pattern matching.

Self-Containing Transformations

The key to this approach is widely used: every node has a *start* and *end* node that acts as a connection point for other nodes — transformed at another time. These transformations will eventually connect the whole graph and represent the CFG.

This approach enables parallelization and thus speeds up the transformation, while generalizing the transformation patterns. The following sections present how different ASG information may be transformed.

Sequencing

Knowing the order of the statement sequences is one of the most important information for CFG transformations. Although it can be derived from the indexes of the list nodes, it is a rather compute-heavy operation. The graph thus redundantly contains the sequence information by storing the list elements both chained and indexed.

Data Structures

Data structures, like uninterrupted list of statements (statement blocks, such as the body of a function) may contain zero or more elements. Like previously detailed, lists are chained and the last element is also marked.

If the list contains one or more elements, the ASG structure is transformed as follows:

1. The *start* node (the ASG node itself) is connected to the *start* node of the first element.
2. The *end* node of every element except the last is connected to the *start* node of the next element.
3. The *end* node of the *end* node of the list itself.

If the list does not contain any elements, its *start* node is connected to its *end* node.

Statements

As the main building blocks of a language, the various statement types represent different behaviors when interpreted. There are several statement types in the *Shift* metamodel, and in this section I introduce a few of these for the purpose of building a small example program.

VariableDeclarationStatement A variable declaring statement is parsed into a structure of multiple graph nodes. These are `VariableDeclarationStatement`, `VariableDeclaration`, and `VariableDeclarator`. On the right side of the variable declaration statement is one `Expression`.

It is the framework's job to transform this ASG segment into a CFG. The transformation of any other element inside of the subtree of the root `VariableDeclarationStatement` node is executed at another time.

The assigned behavior to this node type and structure is the following:

1. The `Expression` is evaluated.
2. The `VariableDeclaration` is executed.

IfStatement This branching statement is parsed into one graph node. This node has two or three references. It must have one condition that decides the branching direction. A (positive) consequence is also required, but the alternate path reference is optional. In order to optimally transform these two scenarios, two transformation patterns are necessary.

1. Both scenarios evaluate the `Expression`.
2. The positive consequence is connected with the `true` path, and the end node of the consequence node is connected to the end of the `IfStatement`.
3. If there is an alternate branch, its start node is connected with the `false` path from the conditional `Expression` and its end node is also connected to the end node of the `IfStatement`.

This transformation creates a diamond-shaped flow. If there are more alternative branches, the ASG represents them in chained containment. Thus eventually every branching will be connected, one-by-one.

FunctionDeclaration Function declaration structures are rather simple. The node has references to its parameters and body. The parameters — if necessary — are evaluated upon calling. The `FunctionBody` contains a list of directives and a list of statements.

The transformation pattern ignores the directives and immediately connects the `FunctionDeclaration` to the list of `Statements`, both ways (as after the function should also end after the last statement has been executed).

Expressions

There are 28 expression implementations in the Shift metamodel, for example literal expressions for strings, booleans, numerals, null, infinity, and regular expressions. There are also expressions representing class declarations, arrays, array functions. Unary and binary operators are also parsed into corresponding unary or binary expressions.

The prototype of the framework contains graph transformation patterns for literal and call expressions. Regarding the control flow graph, literals are immediately evaluated, so their end node is connected directly. Call expressions try to find the declaration of the called function and set it as the next block of the control flow.

The end node of the function declaration is also connected to the end node of the call expression. If the function is called more than once, its end node will have multiple outgoing edges. Thus when writing graph pattern queries over the CFG, one has to declare constraints matching the ingoing and outgoing edges between the `CallExpression` and the `FunctionDeclaration`.

In case the call expression provides parameters, they are also transformed into the CFG and evaluated in the given order.

5.6.3 Transformation Challenges

Besides the effort required for writing the transformation rules for every class and structure in the metamodel, there are also other challenges. Helping graph structures introduced for easy querying may slow down or even make impossible to correctly query given patterns.

Queries for dead code search (detailed in Section 5.5) for example require the list ordering and CFG end nodes to be removed, or the transitive closure slows down the query. This would make it unsuitable for near real-time user feedback.

Another challenge occurs, when a dynamic object or its member is passed as a parameter to a function call. When evaluated, the function declaration references it with an alias; the

static analysis, however, may only guess the current value or reference. Substituting all the possible uses for a given reference and collecting dynamically added members may be a partial solution, but this is a future work of the research.

5.6.4 Test Generation

As previously mentioned in Section 5.6.1, the CFG contains every feasible, and some infeasible execution paths. In Cypher it is also possible to find the shortest paths or every path between two nodes. Knowing the entrance node of the CFG and selecting an arbitrary node in the CFG it is thus possible to list every path from the entry node to the selected one. This node can represent, e.g., a troublesome, unwanted or unexpected state.

With the transformation of the statements and the conditions of the if statements and other branching structures into a satisfactory problem it may be possible to decide whether there are program inputs that take the execution of the program to the selected node. This topic is subject to future work.

5.7 Type Inference

As detailed in Section 2.2.3, type inference refers to the deduction of the data types of expressions, statements in the source code. My approach to finding the possible types of an expression or variable is similiar to the technique introduced in Section 5.6, transforming the ASG into a CFG-like structure, as the data flow closely follows the control flow.

Type inference is a particularly difficult problem, even for statically typed languages. For dynamic languages, such as JavaScript, it is even more problematic and only a few tools are available. The most popular ones are listed in in Chapter 3.

Following the approach of one of these tools, Tern (introduced in Section 3.1.1), first I assign type information to given nodes in the ASG, then propagate this information based on constraints and rules, detailed in this section.

5.7.1 Type System

In order to represent the types in the graph, structures are introduced and stored in the same graph database. The database has a singular `TypeSystem` collector node, connected to every `Tag`, representing a type. These `Tag`s are attached to a given ASG node, meaning the node can contain a value of the given type.

Literal Types

Since the literal expression nodes in the ASG are explicitly annotated with their type, it is the easy to mark these with a `Tag`: `Boolean`, `RegExp`, `Infinity`, `Numeric`, `Null`, and `String`.

Classes

Handling classes and composite structures requires new `Tags` to be introduced to the type system. The `Tags` should contain information about properties and methods, along with their possible types. The topic of developing the proper representation and transformations collecting and maintaining these structures is subject to future work.

Functions

Functions take parameters and based on inner logic (that could even consider the type of the parameters and other variables) may return any type of result. If the function declaration body returns a literal, an already tagged variable reference, or a new instance of a class, it is easily tagged.

The representation of function types is also subject to future work. I outline two possible representations:

1. Maintain multiple type declarations for a given function, with every combination of parameter types.
2. Store only a composite declaration with a list of possible values for every parameter and the return value.

5.7.2 Propagation Strategy

As the program runs, the values and thus their respective types are moving from variable to variable through variable reads, operators, expressions, statements, and variable writes. While this process follows the control flow of the program, during static analysis there is no way to find the exact order of execution. Just like the CFG in my approach contains every possible flow, the type annotations in the graph will also contain every possible type that a given node may have.

The basic algorithm for propagating type information is the following:

1. Identify the literals and class instaces, tag these accordingly.
2. Apply the propagation rules considering the constraints and propagate the tags so a given node can only have one reference to a given tag.
3. Repeat step 2 until there is no modification in the graph — or in case of an endless loop stop after given amount of iterations.

Expressions

There are 28 expression types in the Shift metamodel for JavaScript. Some of these have discriminator values, that may result in different *meaning* for different input types. `BinaryExpressions` have an `Expression` on their left and rigth side each with their type information, and the operator gives *meaning* (semantics) to what the resulting value can be.

Table 5.1 shows the various semantics of a simple addition (a `BinaryExpression` with a `+` sign as a discriminating `BinaryOperator`). [61]

Left expression	**Operator**	**Right expression**		**Result**	**Meaning**
Number	+	Number	⟶	Number	*(addition)*
Boolean	+	Number	⟶	Number	*(addition)*
Boolean	+	Boolean	⟶	Number	*(addition)*
Number	+	String	⟶	String	*(concatenation)*
String	+	Boolean	⟶	String	*(concatenation)*
String	+	String	⟶	String	*(concatenation)*

Table 5.1 Semantics of the addition operator in JavaScript.

Logical operators also behave differently compared to other languages, e.g., Java. The logical and operator (`&&`) returns the left expression if it is so-called *falsy* (values that can be converted to `false`, such as `null`, `0`, `undefined`). If the left side expression is *truthy*, it returns the right side value. Thus the possible return *type* of the expression can be types of both expression and not necessarily `Boolean`.

Apart from `UnaryExpressions` and `BinaryExpressions`, there are several more. Most importantly the `CallExpression`, that returns the value of the called function. A value with possible *types* that the function may return with. Just like in case of the CFG transformations, several rules have to be written in order to cover the various expressions of the language.

Since the base of the transformation is the ASG, where the expression is parsed in the correct precedence, the transformation will also respect this order and compute the possible types of the whole expression.

Statements

There are also more than 20 `Statement` types in the metamodel of Shift. Some of these introduce new, temporary variables in the scope, such as `ForStatement` or its specific, *syntactic sugar* version, `ForOfStatement`, that iterates over the values of an iterable type. In these cases the transforming graph pattern has to extract the inner type information from the type of the iterable and propagate it to the variable.

Variable References

In JavaScript one may think of a variable identifier as a label, a name, a reference to a value. These variables are represented in the ASG and (most of) their static access references are also represented.

When an `Expression` accesses the value of a `Variable`, a reading `Reference` is present in the ASG. The same goes for the `VariableDeclarationStatements` and `ExpressionStatements` with `AssignmentExpression` inside, assigning value to a variable, represented by a writing `Reference`.

These `References` are only present for static scope variable accesses. New transformation rules have to be created to cover the cases when an `Object` is dynamically introduced to a scope — e.g., passed as a parameter for a function. This is also subject to future work.

Figure 5.9 shows how the type information may propagate in the graph. After the `Literals` have been tagged (1), the type information propagates to the `Variable` (2) through the `VariableDeclarator` and the `BindingIdentifierMemberExpression`. This type information is read by the `IdentifierExpression` (3) and the final type of the `BinaryExpression` based on the rules of adding a number to another number will be calculated and assigned (4).

5.7.3 Limitations and Challenges

For my thesis I have only investigated whether it is possible to build the foundations of a type inferencing system on graph pattern matching and transformations. The type inferencing system in Tern (introduced in Section 3.1.1) has already shown that it is possible to do so on in-memory AST representations with visitor patterns.

In order to achieve better language coverage, it is subject to future work to develop an appropriate representation for compound and complex data structures, handle built-in functions and methods of data types. Challenges also include handling the `this` references in functions and class methods, anonymous functions, and method calls like `forEach`, `map`, or `filter`.

With the right constraints it is possible to approximately infer the type of a given expression and help developers with context-aware information and warnings for type-related problems.

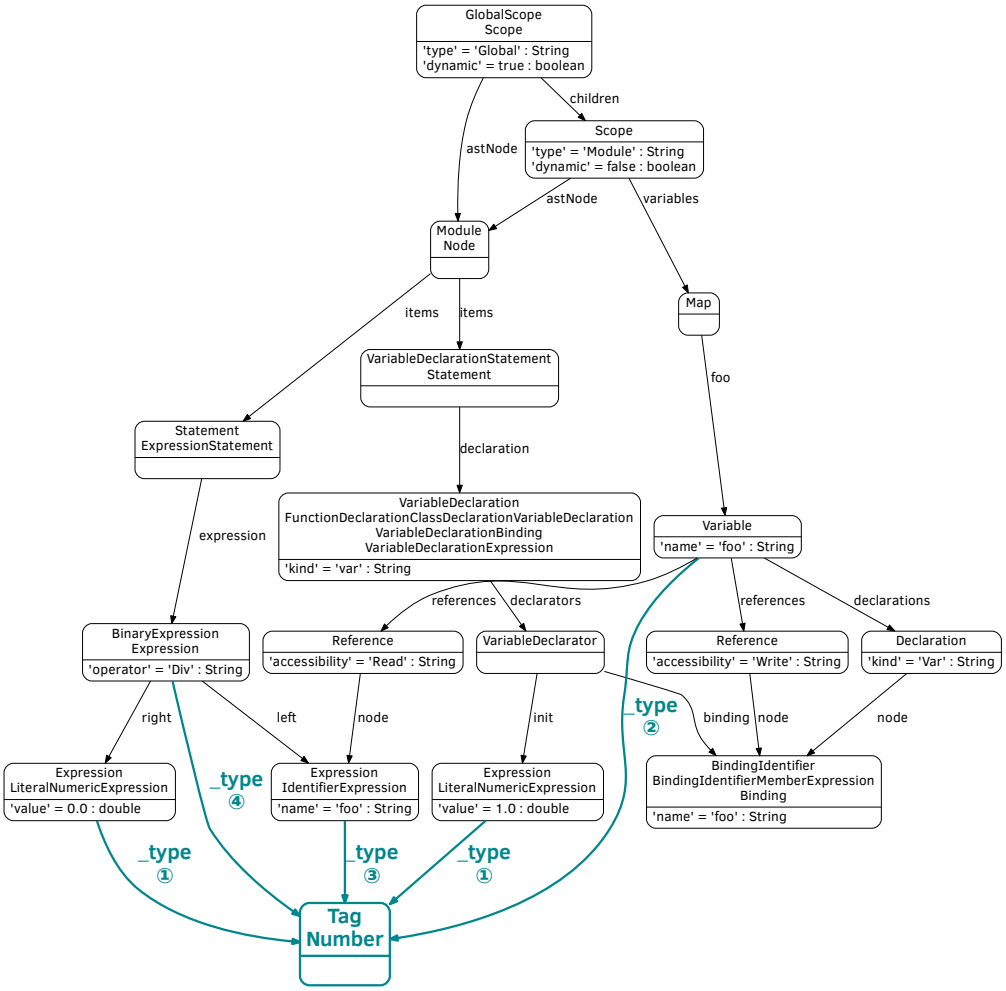

Figure 5.9 Type information propagation.

Chapter 6

Evaluation of the Prototype

In this chapter I present the measurements of various system functions and the runtime characteristics of the prototype framework.

Due to the underlying approached, the presented framework has its limitations and trade-offs. Processing one file at a time makes the approach incremental with file-level granularity, potentially saving time on the whole, but takes a large amount of time integrating the parser into the system. The approach also requires less memory during the parsing phase, but takes more time once every file was processed and the connecting phase takes place.

6.1 Benchmarking Environment

In order to make sure that the measurements are reproducible, and are not affected by user input or other environmental events, the measurements were performed in the cloud. In this section I detail the hardware and software configurations for the benchmarks.

6.1.1 Virtual Machine Configuration

The benchmarks were carried out in a Microsoft Azure virtual machine [62]. Since the approach utilizes a persisted graph database with in-memory caching, I have chosen a configuration with moderate amount of memory for a server and high IO performance.

The D-series virtual machines were designed with data-intensive use-cases in mind, like Big Data and Analytics. [63] The virtual machine instance was located in the West Europe region.

The *Standard DS3_v2* configuration consists of the following:

- 4 CPU cores (Intel(R) Xeon(R) CPU E5-2673 v3 @ 2.40GHz — *reported*)
- 14 GB memory
- 8 data disks

- 12800 maximum IOPS
- 28 GB local SSD

6.1.2 Software Configuration

The results of the benchmarks can also be affected by the software configuration. I have selected the preconfigured *Ubuntu Server* virtual machine with the following properties and modifications:

- Ubuntu 16.04.1 LTS (GNU/Linux 4.4.0-42-generic x86_64)
- Oracle Java(TM) SE Runtime Environment (build 1.8.0_111-b14)
- Java Virtual Machine with 2 GB minimum and 12 GB maximum heap space
- bash benchmarking script with curl

6.1.3 Framework Dependencies

The prototype of the framework was based on changing and rapidly developed dependencies. Both Neo4j and Shift had version and API changes, so I chose to freeze the versions at a working state and use them for the measurements.

Neo4j was freezed at the first released 3.0 version: 3.0.0. At the time of writing this thesis it is at version 3.0.6, with 3.1 being available as a beta. Shift was freezed at 2.2.0 with custom modifications.

I needed to perform some modifications on the Shift source code, which were later merged[1] to the Shift repository.

6.2 Benchmark Cases

In this section I iterate over the various benchmark cases, present them in detail, introduce the results of the measurements and evaluate the results.

6.2.1 Graph Database Initialization

The framework uses the Neo4j server in embedded mode (instead of a standalone configuration). This means that the server is started with the framework and at the first usage it needs initialization.

[1] `https://github.com/shapesecurity/shift-java/pull/101`

While the following benchmarks are prepared with initialized databases, I have measured the time required to prepare an empty database. This is measured by deleting the database folder, restarting the application and executing a simple query counting the nodes.

The database requires on average 996, median 1015 milliseconds to initialize[2].

6.2.2 Source to Graph Transformation

After the database has been initialized, it can receive content. This transformation process is one of the most time-consuming workflows. Here the source code of the file is read from the disk or the memory, transferred to the server. It is then parsed using the Shift parser, extended with the scope analyzer and stored in the database while iterating over every node.

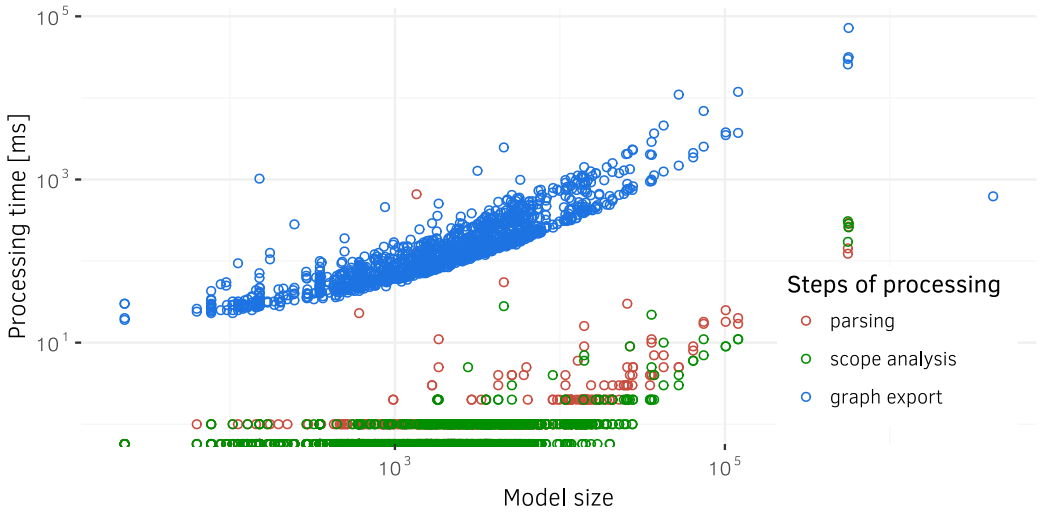

Figure 6.1 The characteristics of the import steps.

Figure 6.1 shows the characteristics of these three steps. Since even the longest source codes can be written in one line, instead of the source lines of code I have chosen the number of nodes in the transformed subgraph (model size) as the horizontal axis. The vertical axis represents the time (in milliseconds) required to perform the given transformation step. Note that both axes are logarithmic.

In order to present an accurate and wide-range measurement, I have selected the repository of the Tresorit[3] web client [64]. The current version of this repository contains 780 JavaScript files, with 75 907 lines of actual code in total. This results in 8 437 838 graph nodes.

[2] 936, 945, 1010, 1019, 1022, and 1042 milliseconds in the 6 runs, respectively

[3] Tresorit is a secure online cloud storage service with syncing and sharing features.

Since the source code contains language elements not yet standardized, I translated the source code to conform ES5 before it is parsed by the Shift parser. This step is not calculated in the benchmark. The resulting files are then imported into the database one-by-one, sequentially, thus the parallel optimizations are not present in this measurement.

Based on Figure 6.1 the time requirement of the parsing and scope analysis steps are negligible compared to the third step. Iterating and storing the elements of the ASG in the graph database shows polynomial correlation with the size of the graph. It is also visible that most of the files are parsed under one second, which indicates that it can be employed in continuous and everyday development.

These results are based on two separate, sequential, full import of the source code repository, containing 780 files.

6.2.3 Dead Code Search

Searching for dead code snippets consists of two phases: first, the call graph has to be prepared, then the graph is queried for unreachable code. Since the transformation rules and queries are far from complete and are not covering every code model scenario, this measurement can only report the characteristics of the current implementation.

Figure 6.2 The characteristics of dead code search.

Figure 6.2 shows at least two horizontal clusters for both steps. On the bottom, there are files that probably contain none or only a few nodes processed by the queries. Based on the figure and manual testing, the characteristics of the upper cluster is to be expected when the transformation rules provide full coverage, resulting in constant runtime for most cases.

The results are based on one-time, file-by-file import and dead code search of the source code repository, containing 780 files.

6.2.4 ASG to CFG Transformation

In order to measure the characteristics of the CFG transformation, I have imported the source files of the Tresorit web client one-by-one and executed the CFG transformation queries at a time.

Figure 6.3 The characteristics of building the CFG.

Figure 6.3 shows that the measurements are clustered into three parts. There are at least two explanations for this:

- The number of CFG transformations implemented in the prototype framework is low. Since there are measurements in every cluster for a great amount of model sizes, it is possible that the composition of the files are different in each cluster.

  This would mean that the top cluster — implementing business logic — contains more nodes to transform, while the files in the bottom cluster — mostly describing interfaces and proxies — contain less.

- The transformation is executed in a parallel manner. If two transformations cause a deadlock in the database, they are canceled and tried again later. It is also possible that subgraphs containing more transformable nodes by the framework are processed slower. The slower the transformations are, the more deadlocks may happen, resulting in slower overall performance.

It is also unexpected to have the top two clusters show no correlation with the size of the resulting graph size. This phenomenon can be explained with the reasons above or with the way Neo4j executes declarative transformations.

The results are based on one-time, file-by-file import and transformation of the source code repository, containing 780 files. Since the CFG transformation in the framework is far from finished, deeper examination of the results is subject to future work.

6.2.5 Incremental Processing

Since the framework processes the changes file-by-file, it is possible to execute the analysis in an incremental fashion with file-level granularity. After every file has been processed and a new changeset is present, it is only required to update and process the modified file and the affected graph parts.

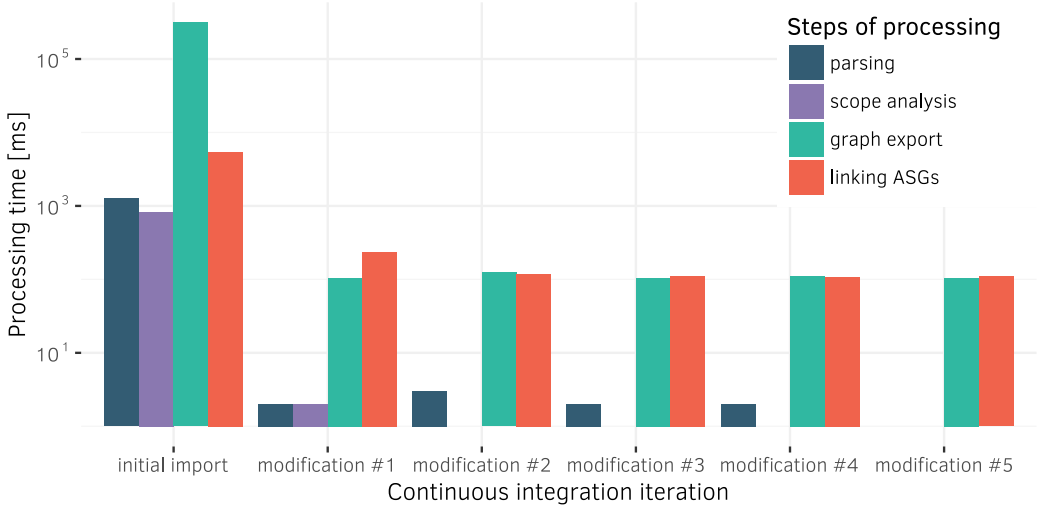

Figure 6.4 Runtime measurements of the execution of the analysis for sequential modifications.

Figure 6.4 details the execution times for each step during the initial and the subsequent change propagations. These all consist of the same steps: 1) the source code is parsed, then 2) scope analysis is applied, and 3) the resulting ASG is stored in the graph database, finally 4) the inserted subgraph is linked to the already stored nodes, imitating the JavaScipt module import-export resolution.

The initial import was executed first, represented by the first set of bars on Figure 6.4. After all 780 files have been processed one-by-one, the import-export linking was executed once. Subsequent modifications were simulated by removing and reprocessing a file with one of the

most `import` statements (thus resulting in more work for the linking transformation). The linking transformation was applied after every modification.

Since the processing time of the parsing and scope analysis steps are negligible in every iteration. The most time-consuming operation is storing the graph in the database, which closely relates to the size of the graph to be stored. Thus reducing this resulted in visible difference. The incremental approach for the complete process is faster by three orders of magnitude. By reusing the already stored and linked ASG subgraphs, the incremental linking step itself is faster by one order of magnitude.

Please note that along with the previous figures, Figure 6.4 is logarithmically scaled on the vertical axis.

6.3 IDE Integration

Besides executing measurements, I also created a plugin for Visual Studio Code (introduced in Section 2.4.1) that sends the content of the open file to the framework, when the file is modified and saved. Then it requests the results of the dead code search, and conviniently displays it using the API provided by Visual Studio Code.

```
1
2
3   function jpegParsingError(code) {
4     return {
5       type: 'JpegParsingError',
6       code: code
7     };
8   }
9   This function is never used or not accessible.
10  (parameter) len: any
11  function buf_ensure(bytes, ptr, len) {
12    if (!buf_has(bytes, ptr, len)) throw jpegParsingError('RangeOutOfBounds');
13  }
14
15  function buf_has(bytes, ptr, len) {
16    return ptr + len < bytes.length;
17  }
18
```

Figure 6.5 Proof-of-concept plugin integration in an industrial case study.

Figure 6.5 shows the proof-of-concept plugin integration displaying dead code warnings for a file in an industrial case study. The previously introduced results in Figure 5.6 show that the approach was able to find circular references without incoming calls and report the clique, without user input. To validate the results, I have manually checked the presented warnings.

6.4 Threats to Validity

Although I carefully designed and executed each measurement, there might have been factors beyond control that influence the results yielded. In this section, I try to list the possible mistakes and also discuss the steps taken to mitigate their effects.

6.4.1 Benchmarking in the Cloud

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

My mitigation strategy is to run the benchmarks multiple times and treat their median as the representative value, or import a larger codebase with file sizes varying from a few to hundreds of lines.

6.4.2 Methodological Mistakes

It is possible that I made mistakes while implementing the approach. It may not adhere to the specification correctly, perform the transformations correctly or measure correctly.

To check the validity of the results, I checked the results manually and with others tools.

Chapter 7

Future Vision

My goal was to create the foundation of a versatile framework capable of doing even more than originally planned. Based on my approach there are several use-cases it makes possible:

- Existing linters do a good job at analyzing linting rules. However, extending them with new, complex rules is difficult. My approach allows tool developers to formalize rules more intuitively with graph patterns.
- With connecting the ASGs there are new possibilities in static analysis that were not possible or available before. This might be used for finding usages of source code elements, helping refactors and finding problematic structures reaching across files.
- Having several files processed in the same database also enables comparing them, potentially allowing executing sophisticated graph-based plagiarism searches.
- Transforming the ASG into CFG not only allows path searches, but combining it with other tools and techniques may result in automated test generation. This can result in higher code coverage and the more unresolved references discovered.
- Based on the CFG and the ASG, basic type inferencing algorithms may produce typing information in the untyped source code, allowing even more ways to write queries and constraints on the source code.

Since the transformation rules in the framework are far from finished, writing more, more precise, optimized, and general transformation rules and examining the benchmark results is subject to future work.

Chapter 8

Conclusions

My main objective was and still is to provide a solution for reducing the time required for a global, codebase-level reevaluation of static analysis after a change occurs.

The elaborated framework can transform a rather large source code repository as a whole into a graph representation and maintain it subsequently. It is found that the approach is suitable for performing code convention compliance checks and for executing static analysis tests on the graph representation.

This approach also utilizes incremental processing with file-level granularity, speeding up the static analysis. Based on my measurements the framework is fast enough to help its users with fast changing code repositories.

Once the framework contains enough transformations and queries for handling the language, it can extend the everyday toolkit of developers.

8.1 Summary of Contributions

I presented an extensible proof of my novel concept to perform incremental static analysis on dynamic JavaScript source code repositories based on graph transformations. My proposal is based on software code modeling, graph transformation and graph pattern matching. The feasibility of the approach was evaluated using manual validation and benchmarking.

8.1.1 Scientific Contributions

I have achieved the following scientific contributions:

– Proposed an architecture for building an incremental static analyzer using freely available components.

- Proposed an approach to transform JavaScript source code repository into a connected graph model.
- Provided an algorithm to update the graph data model incrementally.
- Presented a working method for finding problematic code parts using graph pattern matching.
- Provided a transformation approach for creating CFG[1] from ASG[2].

8.1.2 Practical Accomplishments

I have also achieved the following practical accomplishments:

- Created an extensible incremental static analysis framework.
- Developed a tool for transforming JavaScript source code into graph data model.
- Designed a benchmark evaluating the approach.
- The prototype framework has been published as an open-source project[3].

8.2 Novel Results

A report about my approach was also presented [65] at the annual Scientific Students' Associations Conference[4] in 2016. Compared to this report, this thesis has the following novel contributions:

- I investigated the feasibility of a type inferring system on the graph structure using graph transformation.
- I have implemented a prototype transformation simulating a JavaScript module loader.
- To show the advantages of incremental processing, I have measured the time required to process a complete initial project import and subsequent modifications.

Compared to my earlier work in static analysis [66, 11, 67], this thesis has the following novel contributions:

- This approach utilizes a property graph for storing the transformed model of the source code (as opposed to EMF[5] and RDF[6] models).

[1] Control Flow Graph
[2] Abstract Semantic Graph
[3] `https://github.com/ftsrg/codemodel-rifle`
[4] In Hungarian: Tudományos Diákköri Konferencia
[5] Eclipse Modeling Framework
[6] Resource Description Framework

– My approach tackles the problem of carrying out static analysis for dynamic languages (e.g., JavaScript), instead of static languages (e.g., Java).

– Besides static analysis, I proposed an algorithm for transforming the ASG into a variant of the CFG.

– This approach was tested on an industrial case study, provided by Tresorit Kft. Tresorit is a secure online cloud storage service with syncing and sharing features, on multiple platforms. The Tresorit Web Client is based on JavaScript.

– I implemented a proof-of concept IDE integration with Visual Studio Code that is able to detect dead code with quick response times and display corresponding warnings to the source code developers.

Acknowledgments

First, I would like to thank Dr. István Ráth and Zoltán Ujhelyi for providing the scientific foundations for my research and insight into software model verification.

I would like to thank my supervisors Ádám Lippai, Dávid Honfi, and Gábor Szárnyas for their friendly advice and enthusiasm. Also, I wish to express my gratitude to all my colleagues at Tresorit and the Fault Tolerant Systems Research Group who provided numerous valuable observations and suggestions. Especially Soma Lucz and Tamás Hegedűs for their input, energy and time.

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

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

Microsoft Azure for Research Award This thesis was supported by Microsoft Azure by providing cloud resources under grant `#CRM:0518531`.

New National Excellence Program This thesis was supported by the ÚNKP-16-2-I. New National Excellence Program of the Ministry of Human Capacities.

Ministry
of Human Capacities

References

[1] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L. Bleris. "Code quality analysis in open source software development". In: *Inf. Syst. J.* 12.1 (2002), pp. 43–60. DOI: 10.1046/j.1365-2575.2002.00117.x.

[2] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. "Quality and Productivity Outcomes Relating to Continuous Integration in GitHub". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. ACM, 2015, pp. 805–816. DOI: 10.1145/2786805.2786850.

[3] Martin Fowler. *Continuous Integration*. URL: http://www.martinfowler.com/articles/continuousIntegration.html (visited on 10/26/2016).

[4] *Usage Statistics of JavaScript for Websites, October 2016*. URL: https://w3techs.com/technologies/details/cp-javascript/all/all (visited on 10/24/2016).

[5] Charles Severance. "JavaScript: Designing a Language in 10 Days". In: *Computer* 45 (2012), pp. 7–8. DOI: 10.1109/MC.2012.57.

[6] *Standard ECMA-262*. URL: http://www.ecma-international.org/publications/standards/Ecma-262.htm (visited on 10/24/2016).

[7] B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsborrow, N. J. Ward, and D. W. R. M. Marsh. "Industrial perspective on static analysis". In: *Software Engineering Journal* 10.2 (1995), pp. 69–75.

[8] Simon Holm Jensen, Anders Møller, and Peter Thiemann. "Type analysis for JavaScript". In: *Static Analysis*. Springer, 2009, pp. 238–255. URL: http://link.springer.com/chapter/10.1007/978-3-642-03237-0_17 (visited on 04/20/2016).

[9] Yichen Xie. "Static Detection of Software Errors". PhD thesis. Stanford University, 2006. ISBN: 978-0-542-70855-8.

[10] *Clang Static Analyzer*. URL: http://clang-analyzer.llvm.org/ (visited on 10/24/2016).

[11] Dániel Stein. "Incremental Static Analysis of Large Source Code Repositories". Bachelor's thesis. Budapest University of Technology and Economics, 2014.

[12] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-10088-6.

[13] *Shift Scope Analyser*. URL: http://shift-ast.org/scope.html (visited on 05/08/2016).

[14] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. "Dex: A semantic-graph differencing tool for studying changes in large code bases". In: *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. IEEE, 2004, pp. 188–197. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1357803 (visited on 05/07/2016).

[15] Benjamin C Pierce. *Types and programming languages*. MIT Press, 2002.

[16] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (1970), pp. 377–387. DOI: 10.1145/362384.362685.

[17] Gábor Szárnyas. "Superscalable Modeling". Master's thesis. Budapest University of Technology and Economics, 2013.

[18] Marko A. Rodriguez. *On Graph Computing*. URL: https://markorodriguez.com/2013/01/09/on-graph-computing/ (visited on 06/06/2016).

[19] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. "In-Memory Big Data Management and Processing: A Survey". In: *IEEE Transactions on Knowledge and Data Engineering* 27.7 (2015), pp. 1920–1948. DOI: 10.1109/TKDE.2015.2427795. (Visited on 06/06/2016).

[20] Zoltán Ujhelyi, Ákos Horváth, Dániel Varró, Norbert István Csiszár, Gábor Szőke, László Vidács, and Rudolf Ferenc. "Anti-pattern Detection with Model Queries: A Comparison of Approaches". In: *IEEE CSMR-WCRE 2014 Software Evolution Week*. IEEE. IEEE, 2014. DOI: 10.1109/CSMR-WCRE.2014.6747181.

[21] *DB-Engines Ranking - popularity ranking of graph DBMS*. URL: http://db-engines.com/en/ranking/graph+dbms (visited on 10/27/2016).

[22] *Neo4j: The World's Leading Graph Database*. URL: http://neo4j.com/ (visited on 05/02/2016).

[23] *Neo4j Graph Database: Unlock the Value of Data Relationships*. URL: http://neo4j.com/product/ (visited on 05/02/2016).

[24] *neo4j-contrib/gremlin-plugin: A Plugin for the Neo4j server add Tinkerpop-related functionality*. URL: https://github.com/neo4j-contrib/gremlin-plugin (visited on 05/02/2016).

[25] *Titan: Distributed Graph Database*. URL: http://thinkaurelius.github.io/titan/ (visited on 05/04/2016).

[26] *VIATRA Project*. URL: http://www.eclipse.org/viatra/ (visited on 05/08/2016).

[27] *Eclipse Modeling Project*. URL: http://www.eclipse.org/modeling/emf/ (visited on 10/18/2014).

[28] *Visual Studio Code - Code Editing. Redefined*. URL: https://code.visualstudio.com/ (visited on 05/06/2016).

[29] *Extending Visual Studio Code*. URL: https://code.visualstudio.com/docs/extensions/overview (visited on 05/06/2016).

[30] *Node.js*. URL: https://nodejs.org/en/ (visited on 10/26/2016).

[31] *Tern*. URL: http://ternjs.net/ (visited on 10/26/2016).

[32] *Tern Reference Manual*. URL: http://ternjs.net/doc/manual.html (visited on 10/26/2016).

[33] *Type Analysis for JavaScript*. Aarhus University. URL: http://www.brics.dk/TAJS/ (visited on 10/26/2016).

[34] *cs-au-dk/TAJS: Type Analyzer for JavaScript*. URL: https://github.com/cs-au-dk/TAJS (visited on 04/25/2016).

[35] Caitlin Sadowski and Jeffrey van Gogh and Ciera Jaspan and Emma Soederberg and Collin Winter. "Tricorder: Building a Program Analysis Ecosystem". In: *International Conference on Software Engineering (ICSE)*. 2015.

[36] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. "Why don't software developers use static analysis tools to find bugs?" In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 672–681.

[37] Lucas Layman, Laurie Williams, and Robert St Amant. "Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools". In: *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE. 2007, pp. 176–185.

[38] *jQAssistant*. URL: https://jqassistant.org/ (visited on 12/02/2016).

[39] *jQAssistant Manual*. URL: http://buschmais.github.io/jqassistant/doc/1.1.4/ (visited on 12/02/2016).

[40] *Flow | A static type checker for JavaScript*. URL: https://flowtype.org/ (visited on 11/27/2016).

[41] *Software Reliability Lab | ETH | Jsnice*. ETH Zürich | Department of Computer Science | Software Reliability Lab. URL: http://www.srl.inf.ethz.ch/jsnice.php (visited on 12/02/2016).

[42] *eth-srl/Nice2Predict: Learning framework for program property prediction*. URL: https://github.com/eth-srl/Nice2Predict (visited on 12/02/2016).

[43] Veselin Raychev, Martin T. Vechev, and Andreas Krause. “Predicting Program Properties from "Big Code"”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 2015, pp. 111–124. DOI: 10.1145/2676726.2677009.

[44] *sinelaw/infernu: Type inference and checking for a safer JavaScript*. URL: https://github.com/sinelaw/infernu (visited on 12/02/2016).

[45] *ternjs/acorn: A small, fast, JavaScript-based JavaScript parser*. URL: https://github.com/ternjs/acorn (visited on 10/26/2016).

[46] *estree/estree: The ESTree Spec*. URL: https://github.com/estree/estree (visited on 10/26/2016).

[47] *A JavaScript library for building user interfaces - React*. URL: https://facebook.github.io/react/ (visited on 10/26/2016).

[48] *Tutorial: Intro To React - React*. URL: https://facebook.github.io/react/tutorial/tutorial.html (visited on 10/26/2016).

[49] *Shift AST*. URL: http://shift-ast.org/ (visited on 10/26/2016).

[50] *shapesecurity/shift-spec: Shift AST Specification*. URL: https://github.com/shapesecurity/shift-spec (visited on 10/26/2016).

[51] Michael Ficarra. *A Technical Comparison of the Shift and SpiderMonkey AST Formats*. Shape Security. (Visited on 10/26/2016).

[52] *shapesecurity/bandolier: Bandolier - bundler for ES2015 modules*. URL: https://github.com/shapesecurity/bandolier (visited on 10/26/2016).

[53] *shapesecurity/shift-semantics-java: an Abstract Semantic Graph (ASG) for ECMAScript programs and a way to generate one from a Shift AST*. URL: https://github.com/shapesecurity/shift-semantics-java (visited on 10/26/2016).

[54] *Git*. URL: http://git-scm.com/ (visited on 10/26/2016).

[55] *Apache Subversion*. (Visited on 10/26/2016).

[56] *Mercurial SCM*. URL: https://www.mercurial-scm.org (visited on 10/26/2016).

[57] Jason Orendorff. *ES6 In Depth: Modules * Mozilla Hacks – the Web developer blog*. URL: https://hacks.mozilla.org/2015/08/es6-in-depth-modules/ (visited on 10/16/2016).

[58] *import - JavaScript | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import (visited on 10/16/2016).

[59] *export - JavaScript | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export (visited on 10/16/2016).

[60] Tim Teitelbaum. “Introduction to Compilers – Lecture 24: Control Flow Graphs”. 2008. URL: http://www.cs.cornell.edu/courses/cs412/2008sp/lectures/lec24.pdf.

[61] *Arithmetic operators - JavaScript | MDN*. URL: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators (visited on 12/03/2016).

[62] *Virtual Machines—Linux and Azure virtual machines | Microsoft Azure*. URL: https://azure.microsoft.com/en-us/services/virtual-machines/ (visited on 10/23/2016).

[63] Kenaz Kwa. *New D-Series Virtual Machine Sizes | Blog | Microsoft Azure*. URL: https://azure.microsoft.com/en-us/blog/new-d-series-virtual-machine-sizes/ (visited on 10/23/2016).

[64] *Tresorit Web Access*. URL: https://web.tresorit.com/ (visited on 10/23/2016).

[65] Dániel Stein. *Graph-Based Source Code Analysis of Dynamically Typed Languages*. Scientific Students’ Association Report. Budapest University of Technology and Economics, 2016.

[66] Dániel Stein. *Incremental Static Analysis of Large Source Code Repositories*. Scientific Students’ Association Report. Budapest University of Technology and Economics, 2014.

[67] Dániel Stein, Gábor Szárnyas, and István Ráth. “Java Refactoring Case: a VIATRA Solution”. In: *8th Transformation Tool Contest*. 2015.