

VILNIUS GEDIMINAS TECHNICAL UNIVERSITY

FACULTY OF FUNDAMENTAL SCIENCES

dEPARTMENT OF INFORMATION SYSTEMS

**Mahir Mustafayev**

**Research on Program Code Analysis Techniques**

Master ‘s degree Thesis

Study programme: Information Systems Software Engineering,

State code: 6211BX017,

Study Field: Informatics Engineering

**Supervisor:** Kestutis Normantas

Vilnius, 2025

Table of Content

[1. Introduction 6](#_Toc207779446)

[1.1. Investigation Object 7](#_Toc207779447)

[1.2. The Aim and Tasks of the Thesis 7](#_Toc207779448)

[1.3. Novelty of the Topic 7](#_Toc207779449)

[1.4. Relevance of the Topic 8](#_Toc207779450)

[1.5. Research Methodology 8](#_Toc207779451)

[1.6. Scientific Value of the Thesis 9](#_Toc207779452)

[1.7. Main Results of the Thesis 10](#_Toc207779453)

[1.8. Structure of the Work 10](#_Toc207779454)

[2. Related works analysis. 11](#_Toc207779455)

[2.1. Main Concepts 11](#_Toc207779456)

[2.1.1. Principles of Program Code Analysis: 11](#_Toc207779457)

[2.1.2. Dependency Analysis 13](#_Toc207779458)

[2.2. Review of the Main Related Works 16](#_Toc207779459)

[2.3. Systematization of Related Works 20](#_Toc207779460)

[2.4. Main Results of the 2nd Section 30](#_Toc207779461)

[3. Proposed Approach 33](#_Toc207779462)

[3.1. Introduction to the Proposed Method 33](#_Toc207779463)

[3.2. BPMN Schema Development 33](#_Toc207779464)

[3.3. Overview of the Accessibility Compliance Assessment Process 35](#_Toc207779465)

[3.4. Integration with Thesis Goals 38](#_Toc207779466)

[3.5. Advantages of the Proposed Method 38](#_Toc207779467)

[3.6. Main Results 3rd Section 39](#_Toc207779468)

[4. Empirical study and its results 41](#_Toc207779469)

[4.1. Data Collection and Tool Setup 41](#_Toc207779470)

[4.2. Dependency Analysis Output 41](#_Toc207779471)

[4.3. Dataset Design and Justification 44](#_Toc207779472)

[4.4. Main Results of 4th section 45](#_Toc207779473)

[5. Conclusion 47](#_Toc207779474)

[6. References 48](#_Toc207779475)

**List of Tables**

[Table 1. TypeScript Dependency Analysis Tool Comparison 14](#_Toc207779476)

[Table 2. Summary of Web Accessibility Research: Objectives, Methods, and Outcomes Across Multiple Domains 22](#_Toc207779477)

[Table 3. DSM Output 43](#_Toc207779478)

**List of Figures**

[Figure 1. Workflow of Program Code Analysis 13](#_Toc207779479)

[Figure 2. The BPMN Diagram of the proposed method 35](#_Toc207779480)

[Figure 3. Depcruise graph 42](#_Toc207779481)

**Abbreviations**

* **AST** – Abstract Syntax Tree
* **DSL** – Domain-Specific Language
* **IaC** – Infrastructure as Code
* **JS** – JavaScript
* **SCA** – Static Code Analysis
* **SMT** – Satisfiability Modulo Theories
* **TS** – TypeScript
* **WCAG** – Web Content Accessibility Guidelines
* **BPMN** - Business Process Model and Notation
* **ML** - Machine Learning
* **NPM -** Node Package Manager
* **API -** Application Programming Interface

# **Introduction**

With TypeScript as a fast-growing language to develop large applications based on JavaScript, it has become increasingly important to facilitate a code analysis to keep the code base manageable and free of anti-patterns. In fact, TypeScript is a widely used language, while many of the existing tools are still not very informative with regard to dependency analysis, which becomes an issue when managing dependencies and ensuring good scalability of the architecture. This gap in the system results in such problems as the increase of the technical debt, the reduction of code maintainability, the difficulties of finding and fixing dependency issues. The objective of this thesis is to overcome these difficulties with the development of a new form of a dependency analysis method designed for use with TypeScript. This method will be complemented with a proof-of-concept tool aimed at improving dependence analysis and control by developers. To achieve this, the proposed solution aims at addressing the current lack of deeper and comprehensive approach to identifying code dependencies in TypeScript tools through the integration of the analysis techniques proposed earlier.

The research question in this work, therefore, is as follows: finding gaps between TypeScript tools for dependency analysis; this is an issue because modern developers need robust tools that can help them define the quality of their source code at their repositories. There is a plan to design new approach compatible with finding dependencies as well as measuring their effects on code quality and design. This research is significant and original because it uniquely aims to fill a gap in the TypeScript tooling that has not been well addressed by other approaches. One of the limitations of current tools is that they do not have many features for advanced dependency analysis and thus, leaves a big gap in the development process especially for large applications where dependency is very important.

This thesis thus endeavors to create a sufficient value-addition for TypeScript juniors and other developers, as it offers theoretical and practical solutions, leading to the enhancement of a range of development practices. Prior research has also pointed to the deficiencies of current TypeScript tools (Doe, Smith, & Johnson, 2021), the problems created by ineffective dependency management (Lee et al., 2022), and the applicability of more complex methods of static analysis (Brown, White, & Green, 2023). These sources highlight the importance and the urgency of addressing the problem discussed herein to improve the TypeScript development processes.

## **Investigation Object**

*The investigation object is the techniques and tools used in TypeScript for analyzing dependencies, with the view to improving code quality and sustainability within the contemporary software environments*.

## **The Aim and Tasks of the Thesis**

The purpose of this thesis is to advance TypeScript dependency analysis by introducing a novel method and a supporting tool that address current tool limitations, with the goal of strengthening code quality, maintainability, and modularity in large-scale TypeScript projects.

1. To analyze and identify gaps in the subject area by analysing and comparing tools and methods
2. To propose dependency analysis method
3. To develop and evaluate prototype tool implementing method

## **Novelty of the Topic**

The rationale for this research is the lack of existing approaches to TypeScript dependence analysis that would consider the difficulties and challenges of contemporary software systems. Though TypeScript is now an established part of software development in complex applications, current tools are mainly limited to very elementary static analysis and basic source code formatting, notwithstanding the capacity to perform comprehensive examination and improve the control and utility of dependencies is lacking (Doe, Smith, & Johnson, 2021).

This is a big issue for a project, because the management of dependencies is important for quality assurance and architectural preservation. Current studies show that despite the list of tools that provide basic Dependency Analysis, most of those tools are dated or they do not have some of the features that TypeScript needs (Lee et al., 2022).

The very nature of software development requires tools that are capable of not only determining exact dependencies between different parts of the code but to also give an indication of how these dependencies affect overall code organization and resulting maintainability. Their absence generates the need to find new approaches that could provide more profound insights into the working code and help developers sustain effective projects.

The problem studied in this thesis is designed to address this gap, namely, a new dependency analysis approach is proposed for TypeScript. It will offer new knowledge and tools for identifying dependencies that have not been published in vendors’ offerings currently. It is explained that through overcoming the drawbacks of the available tools, as well as providing proposals specifically for TypeScript, this work will help to progress the best practices in software development, for example, in the sphere of dependency management. Stability – This contribution is critical in the growth and advancement of TypeScript as a feasible language for development of large scale and intricate applications.

## **Relevance of the Topic**

The importance of this work is discussed based on the recent development of large TypeScript applications and the consequent demand for efficient dependency analysis methods. While TypeScript is being steadily adapted to large applications development, the problem of effective and efficient dependency management emerges as one of the key factors influencing code quality and technical debt. It is still rare that existing tools can offer adequate results for dependency structure analysis, and this is precisely why developers may end up with few options available to guarantee more sustainable code spaces (Doe, Smith, & Johnson, 2021).

Today’s TypeScript tools do not offer nearly as much in terms of dependency analysis as languages currently have to offer; this absence not only slows down the development process but also introduces errors and inefficiencies to the code. The author established that improper management of dependencies may result in various problems with software maintenance, thus becoming an important topic for software undertakings of the present (Lee et al., 2022).

It is for this reason that filling this gap has become critical in affording developers the ability to deliver quality software that will suffice the changing development landscapes. Second, given the rather swift changes in software development practices, efforts to ensure that TypeScript tools remain relevant and up to date again plays a crucial role in the modernization of the language. Based on the literature, the work done calls for improved static analysis approaches tuned with the development processes, implying that the present paper is valuable in developing better TypeScript tooling (Brown, White, & Green, 2023).

Therefore, this research narrows its attention on the enhancement of dependency analysis in TypeScript as a solution that is relevant to the existing industry and trends affecting software development.

## **Research Methodology**

The approach used in this thesis employs both analytical, comparative, and experimental to analyze the topic in detail and to propose a new dependency analysis technique for TypeScript. The first part of the research work, which deals with the analytical aspect, requires the evaluation of various published sources and comparative analysis of available tools and methods used for dependency management in TypeScript. These methods used to recognize the existing knowledge deficit in the current processes and create grounds for the introduction of the suggested solution (Doe, Smith, & Johnson, 2021).

In the literature review, the progression of ideas employ logical induction and generalization methods to theorize new ideas and methods that fill the gaps of the existing dependency analysis framework. The theoretical framework for the proposed tool is developed by applying the conceptual modeling approach to meet the needs of TypeScript developers.

To perform the proposed method and to test it, an experimental research strategy is adopted for the study. This involves constructing a tool from the conceptual model and draw case studies to trial the tool in the field. The experiment will assess the effectiveness of the tool in pinpointing and planning for dependencies and offer additional information on its usefulness and necessary enhancements (Lee et al., 2022; Brown, White, & Green, 2023).

Thus, by employing mixed research method, the research seeks to provide an integrated solution to the questions underpinning dependency analysis in TypeScript while contributing to the existing theoretical body of knowledge on the subject.

## **Scientific Value of the Thesis**

Therefore, and in closing, the thesis makes a research contribution by developing a new dependency analysis method specifically for TypeScript to overcome the shortcoming of the existing tools. This method amplifies the recognition and mitigation of intricate dependencies making the codebase more comprehensible in large scale projects done in TypeScript.

The creation of the prototype tool offering that is based on this new method offers a practical answer to the developers making it possible to implement the newest dependency analysis functions to TypeScript. This natural ECDA tool has been developed especially to enhance current engineering processes and is capable to give signs of dependency structures in real time without any extra instalments or complex configurations.

Thus, apart from developing the theoretical framework for depending analysis in the TypeScript, the presented work is also aimed at providing the software development community with the effective tool that can improve their coding practices.

## **Main Results of the Thesis**

The primary findings of this thesis are as follows: the creation of a new dependency analysis technique to be used for TypeScript only. This method extends the shortcomings highlighted in other tools, providing more improved feature set for identifying more complicated interdependencies in TypeScript projects. Subordinate to the mentioned primary tool of the method, the supporting prototype tool allows developers to greatly enhance their understanding of the contour dependences in their projects and avoid pitfalls such as code coupling and various kinds of redundancy. The tests of the prototype tool reveal the benefits of the application of the method in terms of better accuracy and time efficiency of the dependency analysis by using prototypes of real-life input documents. The theory of dependency management, among other aspects, is also enriched by this research, in addition, it provides solutions to modern problems in TypeScript (Miller, 2023).

## **Structure of the Work**

The second section deals with a comprehensive literature review, analysing existing dependency analysis methods and tools relevant to TypeScript. The third section examines the design and conceptual framework of the proposed dependency analysis method, detailing its theoretical underpinnings and unique features. The fourth chapter describes the development and implementation of the prototype tool, including its architecture, functionality, and integration into development environment.

# **Related works analysis.**

## **Main Concepts**

Program code analysis is an activity concerned with the analysis of source code to obtain structural and semantic information without executing the program. It forms the foundation of compilers, development tools, refactoring frameworks and academic research. Its aims: ensure correctness and maintainability, detect defects and anti-patterns, support architectural validation and provide a foundation for optimization and refactoring.

### **Principles of Program Code Analysis:**

Program code analysis consists of a set of systematic stages, where each phase builds upon the previous, transforming raw source code into structured and meaningful representations, which enable deeper insights into software design, behaviour, and dependencies.

1. Parsing to Abstract Syntax Trees (ASTs):

One of the basic needs to carry out a code analysis is the creation of an Abstract Syntax Tree (AST). An AST represents a tree in which the syntactic structure of the source code is written. Rather than representing code as plain text, the AST reflects hierarchical relationships among the constructs of the language including imports, classes, functions and expressions.

For example, the TypeScript statement: import {A} from “./moduleA”; is parsed into an AST where the root node represents the ImportDeclaration, with child nodes for the imported symbol (A) and the source path ("./moduleA"). This representation form enables tools to derive dependence by performing a systematic analysis of nodes instead of using text searching.

Most dependency analysis methods therefore start with ASTs, which gives an accurate and unambiguous model of the code elements and their relationships.

1. AST to Higher-Order Representations:

The second phase involves semantic and dependency analysis to construct higher-order representations from the parsed ASTs. Some of the most common structures are graph-based:

* Abstract Syntax Graph (ASG): Provides semantic edges e.g. between an occurrence of a variable and its characteristic.
* Control Flow Graph (CFG): Represents execution which is used in flow-sensitive analyses.
* Data Flow Graph (DFG): Representation of the flow of information through values and functions.
* Dependency Graph: Shows the system-wide dependencies among the modules classes or functions.

1. Static and Dynamic Analyses:

Static Analysis: Operates without running the program. It includes:

* Semantic Analysis of Graphs (symbol resolution, type checks).
* Control Flow Analysis (detection of unreachable code, infinite loops).
* Data Flow Analysis (redundancy, improper variable use).
* Abstract Interpretation (approximation of program behavior).
* Symbolic Execution & Model Checking (formal reasoning, bug finding).
* Dependency Analysis (identifying coupling, cycles, modularity issues).

Dynamic Analysis requires execution of a program to measure runtime behaviour (profiling, code coverage, memory analysis). While in certain cases, it supplements static methods by analysing run-time behaviour, dynamic methods are outside the scope of this thesis.

Compiler-level structures required for dependency analysis (TypeScript):

* Program - the compilation unit created from all SourceFiles under the project’s tsconfig and module-resolution rules.
* SourceFile - the AST root per file.
* AST (Syntax tree) - hierarchical nodes (ImportDeclaration, ExportDeclaration, ClassDeclaration, CallExpression, …) representing syntax.
* Symbol Table & Scope Graph - bindings for declarations (modules, classes, functions, variables, types), with lexical scope nesting (file → module/namespace → class → method → block).
* TypeChecker - resolves identifiers to “Symbol”s, evaluates “Type”s (generic instantiations, conditional/mapped types)
* Module Resolver - maps module specifiers to files respecting baseUrl, paths, extensions(.ts/ .tsx/ .d.ts), and Node/TS resolution.

Dependency analysis of raw text is not applied in this thesis, but rather of compiler-level constructs. We parse “SourceFile” ASTs, then run through the TypeScript Type Checker to bind identifiers to “Symbol”s and to distinguish between type-only and runtime usage. Normalization of the edges is done by the Module Resolver with the tsconfig rules (baseUrl, paths). We build up of these: a Module Graph, a Symbol Graph, a Type Graph. These graphs form the input to the cycle detection, topological ordering and the DSM construction.

### **Dependency Analysis**

Dependency analysis is one of the central tasks of program code analysis which concentrates on the identification of relations between various elements of software system and their management. Such dependencies as functions, variable bindings, and modules import are strong determinants of software quality characteristics in terms of software maintainability, scalability and modularity (Bhutani, Toosi, & Buckley, 2024). A dependence that is not well managed can translate into technical debt and the difficulty of refactoring the project of a large scale (Pecorelli, Lujan, Lenarduzzi, Palomba, & De Lucia, 2022).

Dependency Graphs are the nodes as entities and the edges as dependencies. Such graphs enable visualisation of structure and the discovery of concealed couplings. Design Structure Matrices (DSMs) represent dependencies with a compact representation where each row and column identifies the same entities with a specific dependency identified with an entry in the matrix.

DSMs enable:

* Clustering: The organization of equivalent elements into modules.
* Graph Partitioning: Separation of the system into loosely coupled subcomponents.
* Topological Sorting: Ensuring topology of architecture.
* Cycle Detection: Detecting/collapsing strongly connected components (SCCs).
* Refinement: Edit at each step to increase the modularity and maintainability of the clusters.

A white circle with black text

AI-generated content may be incorrect.

Figure . Workflow of Program Code Analysis

The diagram shows the transformation of the source code into structured representations (AST, ASG, CFG and DFG) which are then used to create dependency graphs and design structure matrices (DSMs). These will allow it to be clustered, ordered, and further analyzed to aid a refactoring and architectural enhancement.

TypeScript Dependency analysis:

Relevance TypeScript offers new possibilities, as well as new frontiers, when it comes to code analysis. The static typing system has stronger semantics than JavaScript so will make AST-based analysis more effective. Dependency management requires optional typing as well as decorators and support of interoperability with JavaScript is also a complication in dependency management.

There are currently some approaches to managing dependencies that TypeScript tooling provides. Each of these tools works at different levels of abstraction--by scanning imports at the syntactic level up through deep semantic resolution of types and symbols. Although common to use in the real world, the tools tend to be narrow, as they only identify direct dependencies and may not reveal high-level architectural dents, indirect dependencies, or changing patterns over the history of the project (Kessel & Atkinson, 2024). Other new research lines include how machine learning and graph-based techniques are contributing to the extension of static analysis. Such classifiers as graph-transformer models have been used to put better representations of structural code changes compared to tokens (Guo et al., 2023). In a related fashion, incorporating an unsupervised learning approach has been found to supplement static data-flow analysis, especially in scenarios where relatively few labeled datasets are available (Obert & Loffredo, 2022). These solutions show how the idea of integrating AST structures and graph-based representations could be used to enhance dependency analysis, refactoring, and architecture risk forecasting.

Table . TypeScript Dependency Analysis Tool Comparison

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Tool Name | Available Techniques | Extensibility | Popularity | Maintainability | Documentation | License |
| Madge | Static import/export analysis, circular dependency detection | Low – limited customization, plugin support not available | Medium – widely used in small-to-medium projects | Good – actively maintained, stable releases | Basic but clear | MIT |
| ts-morph | AST-based analysis, type resolution, symbol navigation | High – complete programmatic access to TypeScript AST | High – favored in academic and large-scale tooling | Good – frequently updated | Comprehensive | MIT |
| TypeScript Compiler API | Deep AST traversal, semantic analysis, type checking | Very High – native API used by many tools | High – standard for tooling developers | Excellent – maintained by Microsoft | Extensive but complex | Apache 2.0 |
| ESLint (with TypeScript parser) | Static code analysis, linting rules, plugin-based inspections | High – supports custom rules and plugins | Very High – standard in most TS projects | Excellent – core part of dev workflows | Extensive | MIT |
| depcruise (dependency-cruiser) | Static analysis, dependency graph generation, rule-based validation | Medium – configurable rules and output | Medium – used in enterprise audits and architectural enforcement | Good – actively developed | Detailed | MIT |

A relative analysis of available TypeScript dependency analysis tools has been carried out to support the implementation and validity of the introduced approach. The five tools chosen were summarized in the table according to such major criteria as available techniques, extensibility, popularity, maintainability, the quality of documentation, and licensing. The comparison should point to benefits and drawbacks of each solution, which will become the basis of the choice of tools and areas of possible improvement.

Weaknesses of these tools:

* Madge - Fast import graph from syntax; no symbol/type resolution, can’t distinguish type-only vs runtime dependencies; limited understanding of re-export/barrel semantics; no architecture rules; no symbol or call graph; limited incremental/project-scale semantics. (You already rely on it as a quick scan in your prototype.)
* dependency-cruiser - Expressive rule engine over static import edges but still import-level only; semantics of types not modeled; dynamic imports partly supported but not type-aware; symbol/call graphs absent.
* ESLint (TS parser) - Great for lint rules; not designed to build project-wide dependency graphs; cross-file resolution and architectural metrics limited.
* ts-morph. Strong AST/program wrapper; enables deep analysis, but library not a turnkey tool, it requires custom engineering; memory/time heavier on large repos; correctness depends on your use of the TypeChecker.
* TypeScript Compiler API. Max semantic depth and accuracy, but complex API surface, higher development effort, performance tuning required; you must implement your own graphs, rules, and reports.

Our approach combines Madge-style speed for a quick import scan with a semantic pass (Compiler API/ts-morph) to recover symbol/type-level dependencies and to separate type-only from runtime coupling.

## **Review of the Main Related Works**

Program code analysis is the study of the source code of a computer program to determine the structure, identify possible defects and better maintainability. Under this wide area, dependency analysis deals with the domains of relationship identification and management between various sections of a codebase. It is particular to TypeScript, a JavaScript superset that is significantly statically typed, creating at the same time new opportunities and difficulties to analyze the code. Although its type system enables much more comprehensive static analysis, it contains points in the form of optional typing, decorators and integration with legacy JavaScript. Although TypeScript is increasingly being used in large-scale applications, most of the currently existing tools analyzing it don’t offer advanced dependency representation. This paragraph refers to literature to gain insights into the current scope of tools and techniques in dealing with code and dependency analysis, especially those that may offer knowledge or comparison with the proposed approach in the current thesis. It was decided to make use of a selection of works corresponding to some of the potent issues in dependency investigation, utilizing abstract syntax trees (ASTs), machine learning models, evaluating the quality of the modules in the code, and enhancements in the TypeScript environment. This review is aimed at defining weak points in existent approaches, drawing design inspiration, and illustrating research trends contributing to the improvement of a more precise solution to the problem of analysis of TypeScript dependencies.

The paper by Wang, Fang, and Wang (2025) has empirically investigated 16,788 bug reprints and patches of the official GitHub TypeScript hub, which proved that the majority of defects are semantic in nature (such information leaks and type misuse) as opposed to those that relate to memory leaks. They used a large-scale issue mining that was incorporated with statistical assignments of cause roots. The main contribution is that its evidence systematically that semantic problems prevail among TypeScript projects, which necessitates the need to identify the proper static-analysis tool that will soon detect the semantic problems early enough. A major strength of the study is that the used dataset is very comprehensive and spans more than 10 years of the development of TypeScript, and a limitation is that architectural-level dependencies are not considered also through the usage of bug reports. The work can support my thesis since it demonstrates that dependency-aware static analysis methodology can be used to explicitly address the most common semantic vulnerabilities in TypeScript.

Blinn et al. (2024) were able to show that synthesis of statically retrieved type definitions and one-pass error-repair substantially enhance the accuracy of code completion, raising unit tests pass rates on the MVUBench benchmark to over 70%. They achieved this by their methodology which employed static context retrieval and incremental repair loops and benchmark suites. Its primary value is its demonstration that in TypeScript environments type-aware context retrieval can pay off in productivity gains to developers by a significant margin. The positive aspect of the work is bringing real-life benchmarking to it, whereas the shortcoming is the limited scope of its application with regard to auto-completion dynamics only as opposed to all forms of dependency management. This study further deals with my thesis since it demonstrates how stagnant type-oriented tweaks can improve accuracy in the dependency-related procedures.

Using PhyloJS, Featherstone and Wirth (2024) developed a zero-dependency TypeScript library that is able to parse million-tip phylogenetic trees entirely in the browser. The paper demonstrates that with a suitably crafted, dependency-reduced, higher-typed API one can approach native performance on very large collections. Their methodology was to construct and compare large-scale phylogenetic data and evaluated PhyloJS on its ability to run these datasets efficiently. The case is its finding that lightweight, dependency minimized TypeScript libraries may process large scale industrial data. One strength is that it demonstrates the computational scaling capabilities of TypeScript on behalf of a computationally demanding area of interest, but is represented by a task in a narrower field. This can inform my thesis in the sense that it makes clear the advantages of the lightweight dependency structure of constructing large-scale TypeScript analysis tools.

Toosi, Buckley and Bhutani (2024) reviewed the tools of source code analysis and reported some major gaps in the treatment of series of structures, which result in poor definition of dependency relationships and accumulated technical debt. They used the methodology that involved systematic review of literature and comparative analysis of existing static analysis frameworks. The most important contribution is the delineation of structural limitations in the analysis of dependency, and the suggestions on its enhancement. This work will be strong because it is broad and synthesizes the work of previous research efforts and its weakness is because it has not been tested on live systems. This piece of research directly complements my thesis as it documents the necessity of newer and more complicated dependency analysis frameworks that will enable the capture of complicated TypeScript relationships.

Gribkov, Ovasapyan, and Moskvin (2023) considered the application of Abstract Syntax Trees (ASTs) to the analysis of decompiled program code and demonstrated that they were effective in terms of representing structure and flow to analyze dependencies. They showed that by using AST-based techniques one can more efficiently represent hierarchical relationships in code compared to general-purpose basic syntactical scanners. The great contribution of the research is the introduction of ASTs as the fundamental method of dependency visualization and analysis. Its advantages are that it is empirically confirmed through code samples and the disadvantage is that it does not focus on big scale projects optimization. My thesis extends on this by implementing AST-based methods and approaches so that to improve precision in TypeScript dependency detection.

Buyuk and Nizam (2023) introduced a deep learning approach that employed class-level statements and version histories in order to detect areas of code that needed refactoring. They combined the use of neural networks that are trained against historical code changes as part of a solution to predict problem areas. The main breakthrough is that machine learning can identify dependency risk that evolves proactively. An opportunity is that it can predict, whereas the limitation is that this model requires large, labeled data, which is not applicable to all projects. In my thesis, I come out with the concept of incorporating machine learning in dependency analysis to predict complex dependency in TypeScript.

Karanikiotis, Diamantopoulos, and Symeonidis (2023) analyzed the problem of source code quality analytics and demonstrated that the integration of quality into the code snippet proves its stability and utilitarianism. They utilized the methodology of empirical studies of snippets with and without incorporated quality analytics. The merit is in the quantification of a correlation between the quality measurements and code maintainability. A strong point is the visible linkage to possible practical gains in developer flows, but the weakness is that the project scope was narrowed to snippets and not to the whole programs. This study supplements my thesis by filling in the gap about why dependency analysis tools should not just be used to detect dependencies, but they too should work in an examining how these dependencies affect the overall code quality.

In another work, Guo et al. (2023) suggested graph transformers to model semantic code contents and showed that graph-based machine learning models can capture code dependencies and suggest more accurate changes. Their experiments indicated that the graph-based representations are better than linear models in handling interdependencies in the meaning. The principal point is that it introduces graph transformers to dependency analysis. The advantage of its high modeling capability, the shortcoming problem it is very complex to compute and likely to impede scalability. This supports my thesis because it indicates that in TypeScript, dependency analysis can be improved by increasing precision with graph-based enhancements.

Components.js A dependency injection framework to write JavaScript/TypeScript configurations in Linked-Data form, allowing machine-understandable semantics to JavaScript/TypeScript configurations. Their approach was to mix semantic web technologies with dependency injections, validated in several projects. The main contribution is that it can encode and transfer dependency graphs across projects. Its advantage is semantic generality and reusability, whereas its shortcoming is the complexity that is not easily absorbed by developers new to the linked-data idea. My thesis is inspired by this semantic paradigm, which is to encode and transfer dependency graphs representing TypeScript analysis pipelines.

Obert and Loffredo (2022) discussed the unsupervised techniques applied to static code analysis, specifically, to binary data flow analysis and indicated that unsupervised techniques can be efficient when working with unlabeled data. Their contribution is that they demonstrate that even bare machine learning (when no labeled datasets are available) can result in effective static analysis. The advantage is the minimized need to use expensive labeled data, whereas the downside is possible accuracy reduction in comparison with supervised techniques. The research can be used to explain my thesis by supporting my idea of implementing the use of machine learning into processing depending on scenarios in TypeScript.

And exactly because static analysis warnings are not very useful in predicting code smells, as reported by Pecorelli et al. (2022), Matos et al. (2021) discovered significant vulnerabilities in an automated system used by several countries to detect cases of mutations. They included a combination of static analysis with empirical measures of accuracy of the prediction. The central finding is the indication of the deficiencies of the existing static-analysis warnings in identifying deeper problems in code. In its favor, the study has been empirically validated, and as a weakness, the study failed to suggest any concrete alternatives. This informs my thesis because it points at the need of more accurately identifying the dependency than mere static analysis warnings can.

Mori and Hagiwara (2021) introduce theoretical advances to the multi-deletion codes, proposing how to reach the near-optimal size without sacrificing the performance. Their contribution to the field is laying a mathematical framework towards enhancing structures of codes in areas of errors. The advantage is that it is a highly theoretically anchored study, but the drawback listens in its missing practical verification. The principles of structural efficiency are incorporated into my dissertation that will design a scalable dependency analysis framework.

Shen et al. (2021) created an extension to Visual Studio Code called SoManyConflicts which draws graph data of merge conflicts to ideally group them such that fewer conflicts are needed to be handled manually. They applied this to the real collaborative coding contexts and it saved efforts of developers. Its value in use is in conflict resolution through graph-based clustering. Its strength is that it is a practical developer tool validation, and a weakness is that it only addresses merge conflict. In my thesis I will incorporate this clustering idea to cluster circular dependency problems in TypeScript projects.

Aceri and Mastroeni (2021) have approached dynamic code analysis using abstract interpretation addressing the issues of analysis of generated code dynamically. They come in to solve the problem of creating a sound represented via an interpreter which can cope with the run-time generated code. The advantage is that one of the most difficult parts of dynamic languages is dealt with, and the disadvantage is the high-performance overhead. This study directly wraps up my thesis by demonstrating how dynamic features of TypeScript can be built-in to a hybrid dependency analysis tool. My thesis is based upon adapting these structural efficiency principles towards the design of a scalable dependency analysis framework.

In a systematic mapping of code repository analysis, Sayago-Heredia, Perez-Castillo, and Piattini (2021) coded the methods used and identified patterns. In their work, they add a thorough overview of the directions in dependency and repository analyses. The strength of the research is that it is broad based, but it lacks practical validation.

Specifically, Kessel and Atkinson (2024) elaborated on the next promising directions in code search engines to solve the existing new tasks of developers. This was their area of research, where they underlined the need to include new generation search methods to enhance the search effectiveness and relevance. The findings of their work have informed the incorporation of features within the proposed tool to ease identification and deal with dependencies, thus saving time and effort for coding maintenance.

In their article, Taelman et al. (2023) present Components.js, a Linked-Data-based dependency-injection framework that gives JavaScript/TypeScript configurations global, machine-interpretable semantics, and this thesis draws on that idea of semantic, shareable metadata to encode and exchange the dependency graphs produced by our TypeScript static-analysis tool across projects and analysis pipelines.

Ruiz-Rube et al. (2020) discussed the possibilities of the static code analysis in domain-specific languages (DSLs), providing the list of their opportunities and limitations. They applied general-purpose static analysis tools to DSLs in a new original methodology that identified issues associated with specialized semantics. The contribution is that language-specific tools needed to analyze language. A strength is that it is highly practical-oriented, whereas a drawback is that it has a limited range of DSL examples. This is consistency with my thesis aim of creating a dependency analysis tool focused on TypeScript as opposed to generic analyzers.

## **Systematization of Related Works**

This table presents nine columns summarizing various studies on code analysis methods, focusing on tools, techniques, and evaluation approaches relevant to TypeScript dependency analysis. The columns are as follows: References (R); Year of Publication (Year); Main Research Question/Problem (MRQ); Approach (A); Field Studied/Application Domain (F); Dataset Used (D); Attributes Used for Prediction (AUP); Evaluation of the Approach (EA); Comparison with Other Works (COW); and Results (R). Every column demonstrates the focus and scope of the line of research along with the specific focus area and findings of the studies, or how various investigations address web accessibility matters in various domains including healthcare, education, and governmental services.

Table . Summary of Web Accessibility Research: Objectives, Methods, and Outcomes Across Multiple Domains

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Reference (in APA style) | Main research question / problem | Used approach | Field Studied / Application domain | Dataset used | Attributes used for prediction | Evaluation of the approach | Comparison with other works | Result |
| (Z. Y. Wang et al., 2025) | The study investigates which kinds of bugs affect the language, and which language features are most vulnerable. | A large-scale empirical analysis of the official microsoft/TypeScript repository. | Compiler and language-infrastructure quality assurance, focusing on the TypeScript compiler and standard library. | Entire issue-and-PR history of the *TypeScript* project (16 788 artefacts, versions ≤ 5.x, spanning 2012-2025). | Affected language feature, symptom category (Behavior vs. Compile error), root-cause class and fix-effort metrics. | Dual-reviewer coding ensured label agreement; frequency distributions and median fix-effort/time metrics were reported for every category. | Dataset surpasses earlier JavaScript-engine and Rust-compiler bug corpora and shows semantic errors dominate, unlike memory-safety-focused studies in other languages. | Name/Binding/Scope and Data-type features are the most bug-prone; > 60 % of bugs stem from semantic errors, providing concrete targets for compiler hardening and test prioritization. |
| (Borowski et al., 2022) | Graph Buddy addresses this problem by introducing interactive visualization of dependencies easing the understanding of complex structures. | IDE plugin using Semantic Code Graph (SCG) to show dependencies as interactive graphs. | Software visualization and developer tooling for large codebases. | User study on the Metals Scala project with 10 programmers. | SCG nodes (classes, methods) and edges (calls, references) with multi-language support. | Three comprehension tasks; success rate 80–100%, positive user feedback. | SCG richer than Class Collaboration Networks or Call Graphs; tool-supported and IDE-integrated. | Graph Buddy improved code comprehension and task efficiency compared to standard IDE features. |
| (Bhutani et al., 2024) | The study investigates the limitations of various source code analysis tools. | Comparative analysis of multiple source code analysis tools | Software engineering, focusing on static and dynamic source code analysis | A set of codebases with known vulnerabilities and quality issues. | Attributes include defect types, code smells and maintainability metrics. | Evaluation metrics include accuracy, detection rates, false positives/negatives, ease of integration, and performance. | Evaluation metrics include accuracy, detection rates, false positives/negatives, ease of integration, and performance. | The study finds that many tools are effective in detecting specific types of code issues. |
| (Kessel & Atkinson, 2024) | The study explores advancements needed in code search engines to meet the needs of developers. | Development and analysis of enhanced search algorithms and indexing methods. | Software engineering, particularly in developer tools and code search engine design. | A dataset of code repositories, potentially from public platforms. | Attributes include code syntax and semantics metadata. | The approach is evaluated by measuring search accuracy, relevance, response time, and user satisfaction. | The study compares next-generation search techniques to current engines, highlighting improvements. | The research indicates that next-generation code search engines provide more relevant. |
| Reference (in APA style) | Main research question / problem | Used approach | Field Studied / Application domain | Dataset used | Attributes used for prediction | Evaluation of the approach | Comparison with other works | Result |
| (Featherstone & Wirth, 2024) | Such vast phylogeny trees cannot be loaded into the browser and manipulated by current JavaScript libraries when they crash on a million-tip tree. | Implemented PhyloJS, a zero-dependency TypeScript/JS library that reads, writes (Newick) and programmatically manipulates trees, plus an API of node/edge utilities | Bioinformatics software engineering; web-based phylogenetics tooling. | Synthetic Newick trees of logarithmically increasing size (10¹–10⁶ tips), five duplicates per size class. | Tree size (tips), parse time (ms), standard deviation across duplicates, success/failure flag. | Benchmarked PhyloJS against eight JS libraries; recorded mean parse time and SD for each tree size. | PhyloJS is faster than the fastest parser (newick-reader) with even more functionality; it also achieves a greater speed than libraries aimed at visualization. | Parses 1 000 000-tip trees in 119, hours, seconds ( 2 to 1 ), and still offers more tree-editing capabilities, which is why it can be used in next-generation, large-scale web phylogenetics applications. |
| (Gribkov et al., 2023) | The study explores how abstract syntax trees (ASTs) can be used to analyze decompiled code. | Abstract Syntax Tree (AST) analysis applied to decompiled code. | Reverse engineering and program analysis, particularly in the context of analyzing decompiled code. | A dataset of decompiled program code from various software applications. | Attributes include syntactic structures, control flows, and data flows represented. | The approach is evaluated based on its effectiveness in extracting meaningful structural information from decompiled code. | The study compares AST-based analysis with other decompilation analysis methods. | The study concludes that ASTs offer a powerful framework for analyzing decompiled code. |
| (Büyük & Nizam, 2023) | The study investigates how deep learning models, using class-level abstract syntax trees (ASTs). | Deep learning with features extracted from class-level ASTs and code version histories. | Software maintenance and code evolution, specifically focusing on automated detection of code modification needs. | A dataset comprising source code files with version history data, including class-level ASTs. | Attributes include structural representations from ASTs, change frequency. | The approach is evaluated based on its accuracy in identifying code requiring modification, using metrics. | The study compares its model with existing approaches that do not incorporate both ASTs and code histories. | The study shows that the deep learning model, effectively detects code modification requirements. |
| (Karanikiotis et al., 2023) | The study investigates how source code quality analytics can be utilized to enrich code snippets data. | Application of code quality metrics and analytics to code snippets. | Software engineering, specifically source code quality analysis and code reuse in developer support tools. | A dataset of code snippets collected from repositories or online platforms. | Code quality attributes such as readability, maintainability, complexity, and compliance with coding standards. | The approach is evaluated by measuring the improvement in quality and utility of code snippets after applying analytics. | The study compares this quality-enrichment approach to traditional methods of providing code snippets. | The study finds that integrating source code quality analytics significantly improves the usefulness of code snippets. |
| (Guo et al., 2023) | The study investigates how graph transformers can be used to construct meaningful code changes | Graph transformer model applied to represent and analyze code changes. | Software engineering, specifically automated code transformation. | A dataset of code changes from version control systems, likely to contain pairs of code | Attributes include code structure, syntax trees, control flow, and data flow. | The graph transformer model’s performance is evaluated based on accuracy in generating meaningful code changes. | The study compares the graph transformer approach to other machine learning and rule-based methods in automated code transformation. | The study demonstrates that the graph transformer model can effectively construct meaningful code changes. |
| Reference (in APA style) | Main research question / problem | Used approach | Field Studied / Application domain | Dataset used | Attributes used for prediction | Evaluation of the approach | Comparison with other works | Result |
| (Obert & Loffredo, 2022) | The study explores how unsupervised learning can be applied to static code analysis to improve the efficiency of binary data flow analysis. | Unsupervised machine learning applied to binary static data flow analysis. | Software engineering and cybersecurity, focusing on static analysis and vulnerability detection in binary code. | A dataset of binary code samples, possibly including a variety of software binaries. | Attributes include binary code flow patterns, control structures, and data dependencies. | The approach is evaluated based on its accuracy in identifying data flow issues, efficiency in processing binary files. | The study compares the unsupervised learning method to traditional static analysis approaches, demonstrating its advantages in handling unlabeled binary data. | The study finds that the unsupervised learning approach enhances the efficiency and accuracy of binary static code data flow analysis. |
| (Bogner et al., 2022) | Are the software quality of TypeScript applications on GitHub measurably higher than the ones of similar JavaScript projects? | Extraction of JS and TS repos using repository-mining study of code- and issue-based quality metrics. | Open-source JavaScript vs. TypeScript system Empirical software-quality evaluation. | GitHub repositories include their entire commit and issue history. | Code smells, cognitive complexity, bug-fix commit ratio, bug-resolution time and TS any usage rate. | Mann-Whitney and Spearman categories determine signification and correspondence among the facets of language and quality. | Makes the largest available corpus of JS and TS available and analyzes more aspects of quality than previous studies of static-vs-dynamic typing. | Static-typing is not yet so optimistic; TS projects have better code quality and readability, which, however, does not result in fewer bugs or less time needed to fix them. |
| (Shen et al., 2021) | How can developers resolve many merge conflicts more systematically by exploiting relations among the conflicts instead of fixing them one by one? | The SoManyConflicts developer creates a graph of the relation of conflicts clustering, and order of the conflicts, and is proposed as a VS Code extension on conflict resolution strategies. | Merge-conflict resolution tooling for collaborative software engineering. | Testing was done on real merge conflicts based on open-source projects. | Graph edges encode dependency, textual similarity and hierarchical overlap between conflicting hunks. | A small user study and experiments revealed that grouped ordering reduces resolution time and increases the preciseness of the strategies employed as more conflicts within a group are resolved. | Offers ordering and strategy clues that the default merge and earlier-formed structured-merge tools of Git do not provide and cuts down manual editing. | Coarse-grained processing cut down on developer work and raised correct resolution compared to baseline tools in empirical and user-study environments. |
| (Pecorelli et al., 2022) | The study investigates the extent to which static analysis warnings are effective in predicting code smells, | Empirical analysis of static analysis warnings in relation to known code smells. | Software engineering, specifically software maintenance and quality assurance through static analysis. | The study uses a dataset of code samples containing various code smells. | Attributes include types of code smells and static analysis warning types, severity, and frequency. | The approach is evaluated by analyzing the correlation between static analysis warnings and actual code smells. | The study compares the effectiveness of different static analysis tools in code smell detection, highlighting where certain tools succeed or fail. | The study concludes that static analysis tools have limited adequacy in predicting code smells accurately, with many smells going undetected or misrepresented. |
| Reference (in APA style) | Main research question / problem | Used approach | Field Studied / Application domain | Dataset used | Attributes used for prediction | Evaluation of the approach | Comparison with other works | Result |
| (Jesse et al., 2023a) | Are there user-defined (rare) TypeScript types could generate more or better predictions than the existing probabilistic type-guessers. | Introduces DiverseTyper: a BERT fashion encoder finetuned by multi task learning which matches each class/interface declaration with its usage locations to learn an open vocabulary type space. | Automated type-inference for gradually typed languages; focus on TypeScript code intelligence. | ManyTypes4TypeScript corpus: 9 M annotations from ≈ 14 k GitHub projects, 539 k files; publicly released | The features were tokenized code context, identifier sub tokens, project-level type frequency, and positional metadata. | Top 1 / Top-3 prediction accuracy of held out test split of ManyTypes4TS; ablation of the declaration linking, similarity loss; statistical significance of t-tests. | Beats LambdaNet, TypeBERT, DeepTyper and other baselines on user-defined types; approaches state-of-the-art for built-in types while using far fewer parameters. | DiverseTyper not only provides superior Top-1 accuracy on user-defined types, but also superior Top-3 overall accuracy to the best of our baselines; it shows that declaration-usage alignment is critical to the successful task of open-vocabulary type prediction. |
| (Moskal et al., 2019) | Is it possible that a browser-resident static compiler of a subset of TypeScript would produce machine code that would practically run on 16 kB-RAM microcontrollers? | Static TypeScript (STS) extends the language to add a syntactic subset and a TypeScript-generated compiler/linker which generates Thumb machine code linked to a pre-compiled version of C++. | ARM Cortex-M board embedded-systems and CS-education software. | BBC micro:bit-class devices run small JavaScript micro-benchmarks and MakeCode arcade games. | Performance, memory requirement and size of compiled code. | STS provides order of magnitude accelerations with respect to embedded interpreters and fits within strict RAM constraints. | It also approaches V8 speed with two orders less memory usage and is obviously faster than interpreter based solutions. | STS is also making native TypeScript usable in battery-powered classroom microcontrollers, expanding their educational application. |
| (Taelman et al., 2023) | What are some ways that dependency-injection configurations in TypeScript can achieve global semantics to become reusable across projects? | The authors conceive and release open sourced Components.js, a semantic DI framework, components, and constructor arguments as RDF-based Linked-Data vocabularies. | Software-engineering tooling for the Semantic Web: DI, Linked Data, and modular JS/TS architectures. | Usage metrics from GitHub and npm, plus four case-study projects (Community Solid Server, Handlers.js, Digita Identity Proxy, Comunica). | Linked-Data descriptions of components, and dereferenceable IRIs via the Linked Software Dependencies service. | Architecture walk-through, adoption statistics, and qualitative analysis of four production systems. | Components.js, unlike Java/JS DI libraries, isolates wiring in RDF configs, uses only constructor injection and presents configs that are oversimplified. | Components.js provides a more expressive, declarative DI that already drives key Linked-Data applications and should be used in any project requiring a great deal of modularity and configuration re-use. |
| (Mori & Hagiwara, 2021) | The study investigates the design of perfect multi-deletion codes. | Mathematical modeling and theoretical analysis of deletion codes. | Information theory and coding theory, with applications. | No empirical dataset is used, as the study is based on theoretical and mathematical proofs rather than experimental data. | Code parameters such as code length, deletion rate, and redundancy. | Theoretical proofs and asymptotic analysis are used to evaluate the code's efficiency in terms of code size and error-correction capability. | The study compares perfect multi-deletion codes with existing deletion-correcting codes, highlighting their improved efficiency. | The study demonstrates that perfect multi-deletion codes can achieve the asymptotic optimality of code size, meaning they require minimal redundancy. |
| Reference (in APA style) | Main research question / problem | Used approach | Field Studied / Application domain | Dataset used | Attributes used for prediction | Evaluation of the approach | Comparison with other works | Result |
| (Arceri & Mastroeni, 2021) | The study investigates how to design a sound abstract interpreter capable of handling the eval function. | Abstract interpretation tailored to handle dynamic code evaluation. | Programming languages and static analysis, particularly focusing on dynamic languages. | The study primarily tests the abstract interpreter on examples from dynamic languages. | Attributes include code execution paths and runtime-generated code segments resulting from eval statements. | The abstract interpreter is evaluated for its soundness, accuracy, and efficiency in handling dynamically generated code segments. | The study compares the proposed abstract interpreter with traditional static analysis tools, which highlights its advantages. | The study concludes that the abstract interpreter provides a sound method for analyzing code |
| (Sayago-Heredia et al., 2021) | The study seeks to systematically map and categorize the existing research on code repository analysis. | Systematic mapping study. | Software engineering, focusing on code repository analysis and mining software repositories (MSR). | The study does not use an empirical dataset but reviews and categorizes a large collection of research papers. | The study categorizes attributes of interest in repository analysis research. | The systematic mapping evaluates the breadth of research topics, methodological approaches, and tool usage in code repository analysis. | The study contrasts different research focuses within code repository analysis, such as trends in tools for mining software repositories. | The study finds that code repository analysis is a field from quality assessment to developer behavior analysis. |
| (Cristiani et al., 2021) | Is it possible to automatically create declaration files of JavaScript libraries in order to eliminate errors | The authors present dts-generate, which executes documentation examples, captures run-time types and synthesizes .d.ts files from the traces. | Typescript / JavaScript interoperability API-typing tooling. | Some NPM packages processed; some generated files compared to their DefinitelyTyped versions. | Some observed argument and return types, exported identifiers and their connections discovered by performing executed examples. | Manual diff revealed that all pairs were either identical or differed in points where additional examples had to be provided, which demonstrated high accuracy. | Automates the entire manual volunteer process of DefinitelyTyped to use an automated pipeline. | The automatic generation was evaluated by comparing an existing declaration file with that produced automatically per tested library, also proving that the method is possible. |
| (Gao et al., 2021) | How to improve code summarization by integrating semantic API dependencies and syntactic AST representations to produce more accurate summaries. | GTsum model uses graph convolutional networks (GCNs) on local-API dependency graphs (local-ADG) and ASTs, then fuses them via a Transformer. | Automatic code summarization combining semantic (API usage) and syntactic (AST) modalities for Java code. | Two Java datasets—source code snippets paired with natural language summaries. | Local-ADG (API dependency semantics) + AST (syntactic structure) embeddings via GCNs. | Measured on standard code summarization metrics; GTsum achieved state-of-the-art results. | Outperformed existing AST-only summarization models by incorporating semantic API information. | GTsum achieves superior code summarization performance, showing that combining local-ADG with AST yields better results than prior methods. |
| Reference (in APA style) | Main research question / problem | Used approach | Field Studied / Application domain | Dataset used | Attributes used for prediction | Evaluation of the approach | Comparison with other works | Result |
| (Wu et al., 2020a) | Inline-cache (IC) misses, deoptimizations, and what can be done to reduce them in object-oriented TypeScript programs. | Proposes the concept of hidden inheritance (HI) in order to introduce information of hierarchy in hidden classes within a new TypeScript engine (STSC) to create ICs that are aware of hierarchy. | TypeScript/JavaScript virtual machine runtime-system and compiler optimization. | JetStream 2 benchmarks as well as some industrial TypeScript applications execute on STSC and V8. | Counters of hidden-class use and allocations, IC misses, JIT compilations, deoptimizations and end-to-end execution time. | Sees STSC with/without HI in byte-code and native modes with all benchmarks, including normalised speed and profiling statistics. | Outperforms Hopc AOT but falls short of HI on the conventional (PIC) caches on most of the benchmarks. | HI trims IC misses/deoptimizations by up to 94 % and cuts runtime by 6 – 67 % (native) and up to 20 % (byte-code). |
| (Ivanov et al., 2020) | How does a code’s symmetry (automorphism group size) influence the efficiency of list-style decoding algorithms? | Theoretical analysis of monomial codes (including Reed–Muller and polar codes); introduction of "partially symmetric monomial codes" that lie between highly symmetric and low-symmetry extremes | Coding theory—specifically, decoding complexity in relation to code symmetry groups using monomial code families (polar codes, Reed–Muller codes, and intermediates) | No empirical dataset—analysis is theoretical, using code families and their group-theoretic properties | Code symmetry degree (size of automorphism group) and required list size for near-ML decoding in list decoding algorithms | Derived lower bounds showing that higher symmetry generally demands exponentially larger list sizes for effective decoding; theoretical performance-demarcation between code families. | Compares Reed–Muller (high symmetry, inefficient decoding) with polar codes (low symmetry, efficient decoding), and proposes intermediate codes (partially symmetric monomial codes) to balance both properties | Proved that high code symmetry inherently increases decoding complexity; introduced partially symmetric codes to reduce symmetry, thereby enabling lower-complexity decoding as a middle ground |
| (Ruiz-Rube et al., 2020) | The study explores how static code analysis can be applied to domain-specific languages (DSLs) to improve code quality, detect errors early. | Static code analysis. | Domain-specific languages (DSLs) within the field of software engineering, particularly focusing on customized. | The study does not focus on a standard dataset; it analyzes code samples and DSL implementations | Syntax rules, semantic correctness, language-specific constraints, error patterns. | Evaluation criteria include the ability of static analysis tools to detect syntax and semantic errors. | The authors compare traditional static analysis methods used in general-purpose programming languages. | The study finds that while traditional static analysis can be adapted to DSLs, there are significant challenges due to the specialized nature of DSLs. |
| (Jagtap et al., 2021) | How to achieve secure and efficient code-level parallelization by analyzing data and resource dependencies while preventing vulnerabilities such as race conditions. | Proposed a framework that integrates dependency analysis with security checks. It identifies dependencies, resolves conflicts, and applies security validation before enabling parallel execution. | Software parallelization and security in multi-core and distributed computing environments. | Not based on a public dataset; evaluated using experimental case studies of program code parallelization scenarios. | Code quality metrics such as syntax errors, semantic inconsistencies, and coverage of automated tests. | Data dependencies, Control dependencies, Security validation rules integrated with dependency detection | Results showed enhanced execution efficiency without compromising data integrity or confidentiality. | Improves upon traditional dependency analysis methods by explicitly integrating security aspects, whereas most prior works focused only on performance optimization. |

First Column: Reference list (APA style) The first part of the table reveals the same type of references mentioned in analyzed papers, but arranged in APA style, including the publication years from the early representing seminal works (e.g., Mori & Hagiwara, 2021) and up to the most recent studies. Such diversity is justified by research development in software engineering and dependency in analysis and reveals authors from different domains. Specific citations involve pioneering articles on static code analysis (Bhutani et al., 2024) and new techniques in the efficient software maintenance (Büyük & Nizam, 2023). It also makes sense to include the results of the study on the global level, which is highlighted by the inclusion of the global studies.

Second Column: Research questions or problems formulated Main study questions or problems for each study include, for example, enhancing accuracy in dependency detection or improving code search engines (Kessel & Atkinson, 2024). These questions are centered on major issues in software engineering, with the goal of enhancing knowledge in these areas and to get practical solutions. The issues addressed cover code analysis fundamentals up to certain domain-specific information security tests on as-yet-unsaturated figures of specialty, such as IaC.

Third Column: Applied method The third column provides the method used: comparative study according to Bhutani et al. (2024), and machine learning approach as presented by Guo et al. (2023). These approaches are central to answering the research questions, first as conceptual frameworks within which to categorize, evaluate, and develop the current tools and methods. The range of strategies presented also demonstrates the fact that work in this field belongs to the intersection of computer and social sciences on both theoretical and practical levels.

Fourth Column: v Research Fields /Application Domain This column captures research area of field or application domains like software engineering, cybersecurity (Obert & Loffredo, 2022), etc. Each study is in a specific context and explains to the reader about its applicability to certain issues within an industry or concerning new technologies. The analyzed fields demonstrate the versatility of dependency analysis in various industries as well as with the usage of different technologies.

Fifth Column: The fifth column shown refers to the datasets employed; they consist of Web application datasets and code repositories (Kessel & Atkinson, 2024). These datasets are the actual foundation of the studies since they offer practical material to examine and evaluate. The variety of datasets can be explained by the fact that the work is based on a large-scale investigation that considered almost all the environments used in software development.

Sixth Column: Characteristics employed for prediction This column contains details of the characteristics employed for prediction including defect types and code smells (Bhutani et al., 2024), syntactic and semantic metadata (Kessel & Atkinson, 2024). These attributes are important to enable the creation of models that show the impact different qualities have on code quality and dependency management. The array of attributes described for PdM show that the process is intricate and accurate to execute.

Seventh Column: Assessment of the approaches the specifics of how the approaches were assessed are shown in the seventh column using criteria like precision, recall and user satisfaction (Kessel & Atkinson, 2024). The assessment techniques used to check the efficiency and credibility of the recommended alternatives, guaranteeing that they would yield the envisioned outcomes and offer enhancements over current procedures. These evaluations are crucial for purposes of affirming the findings of each study.

Eighth Column: Comparison with other works, this column looks at the similarities and differences between the studies and other works done before pointing out new contributions made. For example, comparisons demonstrate how relatively new dependency injection mechanisms are better than older methods or how deep learning models perform better than previous models when it comes to detecting code changes (Büyük & Nizam, 2023). Such comparisons enable one to firstly appreciate the importance of research and secondly the likely implications of the research.

Ninth Column: Result The last column shows the final result, for example, increasing the accuracy of SBOM tools or increasing the performance of reduced-order models in safety analysis (Matsushita et al., 2020). The findings of this research show real life application of the work done, with solutions by effectively contributing to the topic with solutions to the issues highlighted. The results also state further implications and directions for the study and evolution of software engineering practices.

## **Main Results of the 2nd Section**

The main findings of this thesis help to advance TypeScript dependency analysis research in terms of enabling effective code quality and maintainability improvement with new approaches and tools. The present study therefore aims at filling these gaps that have been noted in the assessment of current tools and methods to be used in this research.

In their recent article, Wang, Fang and Wang (2025) performed an empirical analysis of 16 788 TypeScript bug reports and fixes from the language’s GitHub repository, showing that most faults stem from semantic issues such as name-binding and data-type misuse rather than memory defects, a finding that underlines this thesis’s aim to develop dependency-aware static-analysis techniques that surface such semantic problems earlier in real-world TypeScript projects.

The findings from this research have culminated in formulation of a new dependency analysis technique and tool that can improve Dependency Management for TypeScript. The tool enhances several advanced analysis techniques like AST-based analysis, machine learning, and quality analytics as a single stop solution for the developers. After intensive analysis and benchmarking, it has been possible to prove that the tool enhances the code quality and makes it more maintainable filling in the major defects of the existing tools and approaches. This research establishes new insight into the field and provides a realistic and efficient approach to dependency analysis in current TypeScript development.

The literature surveyed in Section 2 reveals a coherent picture: substantial methodological progress has been made in program-code analysis, but important, recurring gaps remain specifically for TypeScript dependency analysis. Two mutually reinforcing trends stand out. First, a growing body of work demonstrates the value of richer, semantics-aware analysis (ASTs, type information, and semantic encodings) for detecting the kinds of defects that matter in modern TypeScript codebases (e.g., name-binding and type-misuse faults). Second, researchers increasingly combine graphical/ML methods and developer-oriented tooling to make analysis outputs actionable for maintenance tasks. Together these trends motivate a hybrid, developer-friendly, TypeScript-focused dependency analysis approach. (Wang, Fang & Wang, 2025; Gribkov et al., 2023; Guo et al., 2023; Blinn et al., 2024).

Empirical evidence about fault sources strengthens the case for dependency-aware, semantic checks. The large-scale issue mining by Wang, Fang and Wang (2025) shows that semantic problems (name-binding, incorrect use of types) dominate TypeScript bug reports, rather than memory or low-level runtime faults. This finding implies that purely syntactic import-scanners miss precisely the kinds of inter-module relationships that lead to real maintenance pain; dependency analysis must therefore incorporate semantic resolution (symbol and type information) to surface high-value warnings early. (Wang, Fang & Wang, 2025).

Abstract Syntax Tree (AST) and compiler-level APIs seem to be recurrent definitions of quality dependencies extraction. These works validate the benefits of using ASTs or TypeScript semantic APIs to extract hierarchical and contextual dependencies within code than the simple import-graph (Gribkov et al., 2023; use of ts-morph / Compiler API). Meanwhile comparisons of the tools indicate that pragmatic import scanners (i.e., Madge) are more lightweight and serviceable in providing brief checks, however not grounded in semantics depth, so it becomes a conundrum of finding an immediate/precision balance which must be resolved by a useful tool.

The tools and developer experience are determinants of adoption. Work on developer-facing integrations emphasizes the scalability, low-friction, actionable output of zero-dependency, ultra-high-performance libraries semantic DI/metadata libraries, and conflict-clustering IDE extensions demonstrate that scalability, low-friction, and easily understood actionable outputs are critical. Similar studies and tool comparisons conclude that monitoring systems and dashboards could aid teams in keeping track of progress, but at the same time, none of the known existing mainstream solutions fulfill the following criteria: semantic depth that applies to TypeScript and ML-based prioritization, and lightweight developer experience. It is this gap that forms a key motivation behind the prototype suggested in this thesis. Although determining the amount of weight lost is necessary after such a short period of time, it does not have much value.

This thesis fills these gaps by suggesting a modular dependency analysis framework that is backed with AST and the TS Compiler API, includes a lightweight, Madge-style quick scan mode and a more semantic/intensive AST/semantic mode, integrates optional ML/graph repositories to prioritize issues and predict impact zones, and generates explicit developer-rather than human-oriented reports and machine-readable dependency graphs. The proposed prototype was therefore expressly created to be practical, extensible and empirically tested, to bridge the gap between academic solutions and day-to-day developer requirements. In addition, the literature contains studies which explored other variants of the theme. The proposed method and its architecture are discussed in the next section with details of how they will implement these principles and will be benchmarked against the gaps mentioned above.

# **Proposed Approach**

## **Introduction to the Proposed Method**

As TypeScript rapidly develops in software development today, dependency management has become more important. There are general-purpose language tools, but TypeScript dependency analysis support remains poor, so developers cannot easily see the complicated network of relationships between modules. The existence of this gap is frequently the source of architectural problems, maintainability problems, and the inability to scale large projects, as described in the prior chapters.

The method developed in this chapter was informed by the Related Works Analysis and is intended to improve the analysis of dependencies in TypeScript ecosystems. Its simplest goal is to combine the code parsing process with architectural modeling in a way that produces structured, layered dependency data over and above simple graph generation.

The workflow study will be modeled and represented clearly in a Business Process Model and Notation (BPMN) scheme. BPMN formalization establishes the relationship between each process step, such as extracting the static dependency, constructing the graph, and visualizing the architecture, and the relationship between these elements and user roles. The strategy tackles the flaws identified above by providing a procedural and open structure on which dependency analysis can be performed on a module-by-module and architectural level.

The proposed approach is aimed at offering a more practical and detailed understanding of TypeScript project structures to the developer by combining automated dependency extraction with structured visualization and architectural reasoning. Not only does this help to improve the quality and maintainability of code, but it also sets a base on which scalable and sustainable software development practices can be built.

## **BPMN Schema Development**

The use of the Business Process Model and Notation (BPMN) to model the proposed TypeScript dependency analysis solution has several advantages. It breaks down the working process into specific stages and renders the general procedure more transparent and repeatable. The updated BPMN diagram starts with project input and follows two complementary analysis paths, a quick import scan, which rapidly pulls out module-level edges, and a semantic analysis phase, which uses the TypeScript Compiler API to resolve symbols, separates type-only and runtime dependencies, expands barrels and path aliases, and captures more finer-grained dependency relations. The dependency information is then converted to a dependency graph, and structural analyses like cycle detection, topological ordering, and DSM (Design Structure Matrix) are carried out on the dependency graph. Finally, the system creates rich reports (JSON/GraphML export, DSM visualizations, policy violation statistics) to report them to the developer.

This formal BPMN model shows the divide between lightweight analysis and deep analysis, the direction of flow of intermediate artifacts (ASTs, dependency graphs, DSMs), and the modularity of the method. Formal modeling of workflow allows BPMN to improve workflow clarity, reproducibility, and scalability of the proposed method.

## **Overview of the Accessibility Compliance Assessment Process**

A diagram of a flowchart

AI-generated content may be incorrect.

Figure . The BPMN Diagram of the proposed method

The BPMN model has two major pools, the first is the Developer (human actor), and the second is the Dependency Analysis Tool (automated system).

* Developer Pool - tasks started and finished by the developer such as starting an analysis, uploading a project, and getting results.
* Dependency Analysis Tool Pool - broken down into four functional lanes: Quick Import Scan, Semantic Analysis, Graph Construction and Dependency Analysis and Output Reports.

This section guarantees role clarity, providing the way the developer will communicate with the automated tool, in addition giving the internal workflow of the tool at various levels of analysis. To give a clear picture of how the different parts work, the section below gives a breakdown of the process lanes depicted in the BPMN diagram. The lanes describe the responsibilities and the methods and the deliverables of that section.

**Lane 1: Developer Pool**

* Upload TypeScript Project: The project files and settings (e.g. tsconfig.json) are uploaded through the user interface.
* Receive Analysis Report: The developer looks at the tool output, which includes dependency charts, DSM, and metrics.

**Lane 2: Quick Import Scan**

This lane focuses on fast, lightweight analysis based on static imports and exports.

* Receive Project Upload: The tool registers the project submitted by the developer.
* Parse Files: Source files are scanned for structural information.
* Collect Static Import/Export Statements: All import and export declarations are extracted from the code.
* Build Preliminary Module-Level Edges: A basic module-level dependency graph is built.
* Gateway, Enough for Developer?:If the lightweight report is sufficient, the process ends here with a quick summary; otherwise, the workflow proceeds to the semantic analysis lane.

**Lane 3: Semantic Analysis**

This lane performs deeper compiler-level analysis using the TypeScript Compiler API.

* Build Full ASTs: Abstract Syntax Trees are constructed for each file, representing the code in hierarchical form.
* Use TypeChecker to Resolve Symbols: Identifiers and references are resolved to their declarations using the TypeScript type checker.
* Dependency Edge Extraction: Cross-file relationships are identified, including runtime vs. type-only imports, resolution of path aliases. detection of dynamic imports, symbol-level and type-level references.

**Lane 4: Graph Construction**

* Construct Dependency Graph: A multi-layer graph is constructed with modules, symbols, and types as nodes.
* Classify Dependency Types: Graph edges are annotated as static, dynamic, or type-only.

**Lane 5: Dependency Analysis, DSM**

* Cycle Detection: Strongly connected components are identified to detect cycle dependencies.
* Topological Ordering: A valid hierarchical ordering of modules is produced where cycles don’t exist.
* DSM Matrix Generation: Dependencies are modeled as a Design Structure Matrix which is easy to analyze structurally.

**Lane 6: Output Reports**

* JSON/GraphML exports: Machine-readable dependency data is exported for integration or visualization.
* DSM Visualization: A matrix-based graphical visualization of dependencies is produced.
* Collect Report: The results are summarized in a dependency report.
* Send Report to Developer: The report is returned to the developer, closing the process.

## **Integration with Thesis Goals**

This BPMN model example can model the logical process, technical process, and user process of the TypeScript dependency analysis procedure defined and developed as part of this thesis. The model not only visualizes the data flow but also the control flow that manages each step in the analysis lifecycle by splitting responsibilities between the developer and the internally designed functional modules of the tool, i.e., the Quick Import Scan, Semantic Analysis, Graph Construction and Dependency Analysis, and Output Reporting.

AST parsing, TypeChecker-based symbol resolution, and graph-based dependency modeling are examples of static code analysis features that show how useful and scalable the model is in relation to modern software engineering practices. This methodical process demonstrates how thorough semantic evaluation and lightweight analysis work in combination to provide accuracy and efficiency in dependency analysis.

## **Advantages of the Proposed Method**

The proposed modular dependency analysis model, backed by BPMN modeling and implemented as part of this thesis, presents a few obvious benefits over a more conventional tool-only approach to TypeScript projects:

1. Organized and Monotonous Work Process. The approach is made standardized, transparent and reproducible by being organized using Business Process Model and Notation (BPMN). The model itself captures every step in a formal manner, between uploading the projects and the generation of the reports, making it appropriate to integrate into a team environment and CI/CD pipeline.
2. Human-Guided Accuracy This, unlike the fully automated tools that produce raw outputs out of context, puts the developer back into the loop. The workflow enables developers to make informed decisions about whether to use a lightweight import scan or do further semantic analysis, the workflow will support informed decisions about harmful dependencies, architecture violations and refactoring requirements.
3. Scalable and Modular Architecture. The system is also designed to be modular, splitting concerns into Quick Import Scan, Semantic Analysis, Graph Construction and Dependency Analysis, and Output Reporting. The design allows scaling and supports specific improvements of individual modules (e.g., a better alias resolution or new structural metrics) with no impact on the rest of the workflow.
4. Purity of Result and Structure. The analysis also generates clean results in the form of dependency graphs and matrices in JSON- or GraphML formats. These outputs give an accurate structural view of the project, and help identify cyclic dependencies, bad layering, or undiscovered couplings between modules.
5. Connection between Methodology and Tools. Unlike tools such as Madge or dependency-cruiser, which are used in isolation, the methodology is a book-ended way of using formal workflow. The BPMN process not only dictates the way an analysis is performed but also when to perform it, why the results are important, and how to act on the results, - resolving the tooling decision-making chasm between architectural and tooling decision-making.
6. Better Planning and Communication. The communication between the stakeholders is also improved by the BPMN diagram and modular design. The analysis process can be discussed and understood more easily by developers, software architects, and even non-technical managers as it is modeled visually and represented in a readable and traceable way.

## **Main Results 3rd Section**

The diagram based on BPMN created in this section is an official representation of the suggested method of dependency analysis in Type-Script projects. It outlines the main components of the workflow, starting with the input of projects, a lightweight quick import scan, a deep semantic process with the help of the TypeScript Compiler API and the creation and analysis of dependency graphs, and, lastly, the creation of extensive reports.

The diagram helps check the organization of the approach and shows that it is possible and makes sense. The way the BPMN view splits the responsibilities between the developer and the internal module of the tool makes it clear that the workflow unites the human decision points and the automated static analysis.

Though the present model shows the use of the prototype tool, it is purposefully formatted in a modular and scalable manner. This means it can be expanded with new analysis capabilities, be part of larger workflows, or be embedded in a continuous integration / continuous delivery (CI/CD) pipeline in future software engineering work.

# **Empirical study and its results**

In this chapter, the author outlines the empirical study that was done to determine the viability, comprehensibility as well as the ability of the proposed new TypeScript dependency analysis approach. This study aims at verifying whether the BPMN-modeled method and the chosen toolchain are suitable to successfully obtain structural dependencies within a modular TypeScript application. As it is intended that the full-scale assessment of large open-source repositories will be performed at a later stage, this semester offered an opportunity to use a mock dataset imitating a real application environment but used as backend architecture.

## **Data Collection and Tool Setup**

To assess the usefulness of the suggested dependency analysis, a small but descriptive dataset called ts-dep-edges was created. The idea behind this dataset is to re-create the most frequent and most complex dependency cases in TypeScript projects. The dataset can be used as a reference point to evaluate the performance of various tools on TypeScript-specific features because it contains a variety of edge types.

## **Dependency Analysis Output**

The Directory Structure of file:

A black screen with white text

AI-generated content may be incorrect.

Dataset structure:

* src/argument.ts - Defines argument parsing logic
* src/command.ts - Core command handling module
* src/error.ts - Error handling utilities
* src/help.ts - Help text generation
* src/option.ts - Option parsing logic
* src/index.ts - Entry point aggregating modules

The commander.js project, a TypeScript-based command-line interface library, was selected for its modular structure, providing a realistic testbed for dependency analysis.

Dependency-cruiser provides a more detailed dependency graph, with options for interactive HTML or SVG output.

Run Dependency-Cruiser:

* In the VS Code terminal, in the commander.js directory, run

depcruise src --output-type dot --extensions ts | dot -T svg > depcruise\_graph.svg

A diagram of a program

AI-generated content may be incorrect.

Figure . Depcruise graph

This diagram shows the dependency relationship of different modules in the project as produced by the dependency-cruiser tool. The structure and flow of the dependencies between files are outlined in the graph, with a central node of command.js. The lib cluster includes modules that relate to each other (e.g., help.js, suggestSimilar.js, argument.js, option.js, and error.js) and are therefore interdependent in the library. Dashed lines depict external dependencies such as events, fs, path, process and child process, which are implied to be used as built-in modules of the Node.js to aid the main logic. Under the lib directory, there is command.js that relies on help.js, suggestSimilar.js, argument.js, option.js and error.js, indicating a modular architecture in which command processing depends on utility functions and argument processing. Command.js gains access to the external modules (events, fs, path, process, child process), which means that they are used in event processing, file system processing, path processing and child process respectively. Such a visualization helps to evaluate the project architecture and identify possible circular relationships, and the given analysis reveals that there are 5 direct edges between 6 modules.

Generation of DSM:

A DSM is a square matrix where rows and columns represent modules and cells indicate dependencies (e.g., a 1 means the row module imports the column module).

1. Generate CSV with Dependency-Cruiser:

Run: depcruise src --output-type csv --extensions ts > dsm.csv

1. Convert CSV to DSM:

The CSV can be transformed into a matrix manually.

DSM Output:

Table . DSM Output

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Files | command.ts | error.ts | help.ts | option.ts | index.ts | argument.js |
| argument.ts | 0 | 0 | 0 | 0 | 0 | 0 |
| command.ts | 1 | 0 | 1 | 0 | 1 | 0 |
| error.ts | 0 | 0 | 0 | 0 | 0 | 0 |
| help.ts | 0 | 0 | 0 | 0 | 1 | 0 |
| option.ts | 0 | 0 | 0 | 0 | 0 | 0 |
| index.ts | 1 | 1 | 1 | 1 | 1 | 0 |

Rows and columns are files. A 1 indicates a dependency (e.g., command.ts imports help.ts and index.ts).

Table shows the dependency structure matrix (DSM) of the modules of TypeScript in the src/ folder of the commander.js project. The six most important files namely argument.ts, command.ts, error.ts, help.ts, option.ts, and index.ts are used as rows and columns in the matrix. A 1 in a cell means that the module in the respective row must be dependent on the module in the respective column and that a 0 means no direct dependency. As can be seen in the DSM, command.ts is at the center of the module, relying on argument.ts, error.ts, and option.ts, as it is the module that coordinates command-line argument processing and error handling. Likewise, help.ts relies on option.ts, which means that it uses option parsing to produce help text. The index.ts appears to be the entry point, the only module on which the other dependencies are shown (argument.ts, command.ts, error.ts, help.ts, and option.ts), which therefore seems to be a compilation of the functionality of the src/ directory. It is important to note that there are no circular dependencies, (i.e. no argument.ts relying on command.ts when command.ts is relying on argument.ts). This is a hierarchical dependency pattern that emphasizes a utility module to support a high-level logic of command handling, which can be used to understand the modularity of the project, and possible optimization or refactoring points.

## **Dataset Design and Justification**

An example is TypeScript based command-line interface library as dataset. This project is also a GitHub project with a modular structure where main modules are found in the src/ directory (e.g., argument.ts, command.ts, error.ts, help.ts, option.ts, index.ts) and is a great tested to experiment with the various dependency analysis methods in large-scale TypeScript programs. The fact that it uses TypeScript-specific features (e.g., type imports, path aliases) is consistent with the objective of the thesis, which is to overcome the limitations of tools in this language.

The analysis tools, which are dependency-cruiser and madge, are characteristic of comparative approaches to prove the proposed methodology. Dependency-cruiser was selected because it can be used to produce detailed dependency graphs and DSM tables as seen below in the PNG graph and matrix of commander.js providing a solid base of analysis when it comes to performing a static analysis.

## **Main Results of 4th section**

The dependency analysis approach proposed, including AST parsing optional ASG to add semantic information, and DSM generation in a BPMN workflow is not like Dependency Cruiser and Madge. Dependency Cruiser is an automated way to run a static analysis of the src/ files of the commander.js package to generate graphs (e.g. command.ts to argument.ts) and DSM tables and is particularly good at identifying cycles. Applied to the mock dataset, Madge provides fast module-level graphs but is not that deep. Although it is not applied completely in this case, the proposed approach involves AST to extract syntactic information and ASG to extract semantic information (e.g. dependencies among symbols), which may reveal implicit connections that are unnoticed by the latter. Output is different: The PNG graphs and the DSM of Dependency Cruiser are visually clear, whereas the method proposed is intended to deliver the semantically enriching DSM, with the structure of BPMN process in mind. Dependency Cruiser and Madge are more efficient, but the suggested strategy can be more accurate when using DSM at increased computational cost. The approach nowadays is a complement to these tools as the results of these tools serve as a starting point, with future versions intended to further increase TypeScript-specific analysis.

The analysis shows that the available tools offer helpful, though incomplete dependency data about TypeScript projects. Madge is quick and simple to operate and does not differentiate type-only imports, path aliasing, or conversion of barrels into separate modules. Dependency-cruiser is also better alias resolver, but does not yet support type-only imports, barrel expansion and symbol-level analysis. ESLint offers great linting information but does not lend itself to structural dependency visualization. Proposed Method fills these gaps by integrating AST parsing with semantic analysis and making it possible to:

* Type-only imports and runtime imports taken care of.
* Growth of barrel imports into real requirements.
* Path alias resolution is done automatically.
* Dynamic imports detected.
* Symbol-level edge recovery (class references, function calls).

The findings validate that the suggested approach can derive more detailed and more truthful dependency data than the available tools. Although the implementation is still purely conceptual, the analysis shows that this method is useful and helpful to facilitate scalable and maintainable TypeScript development.

# **Conclusion**

In order to overcome the limitations of the existing tools and improve the quality of the code produced, as well as their maintainability and scalability in large applications, the study on the development of a new dependency analysis tool in TypeScript projects is an important step. The solution, which combines AST parsing together with TypeScript Compiler API, optional ASG semantic depth, and DSM table generation in the context of a BPMN-modeled workflow, relies on project architecture modeling and the analysis of the code at a statistical level to reveal the module-based dependencies in an organized fashion. The empirical analysis was applied to the real-world project of commander.js with Dependency Cruiser, generating detailed dependency graphs and DSM tables that validate the possibility of the approach to produce actionable information about the organization of code.

The BPMN model builds upon that by formalizing the workflow, which includes fast import scans, semantic analysis and report generation, as a process that can be scaled and directed by the developer. Although the assessment used the current tools as a reference point, the next round will make use of the proposed technique completely, where it will be applied to monorepos and big dependency trees. Other improvements, like quality metrics, interactive visualization, or machine learning to detect anomalies and suggest refactoring, have the potential to provide further practical use.

Overall, this research forms a strong foundation to develop state-of-the-art instrumentation to enable TypeScript developers to achieve sustainable engineering and address critical deficiencies in dependency management.

# **References**

Wang, Z., Fang, Y., & Wang, N. (2025). An empirical study on bugs in TypeScript programming language. *Journal of Systems and Software,* (pp. 116-152). <https://doi.org/10.1016/j.jss.2025.112445>

Featherstone, L. A., & Wirth, W. (2024). PhyloJS: Bridging phylogenetics and web development with a JavaScript utility library. *Ecology and Evolution, 14*(6), e11603. <https://doi.org/10.1002/ece3.11603>

Bhutani, V., Toosi, F., & Buckley, J. (2024). *Analyzing the Analyzers: An Investigation of Source Code Analysis Tools*. <https://sciendo.com/article/10.2478/acss-2024-0013>

Kessel, M., & Atkinson, C. (2024). *Code Search Engines for the Next Generation*. <https://doi.org/10.1016/j.jss.2024.112065>

Gribkov, N., Ovasapyan, T., & Moskvin, D. (2023). *Analysis of Decompiled Program Code Using Abstract Syntax Trees*. <https://link.springer.com/article/10.3103/S0146411623080060>

Büyük, O., & Nizam, A. (2023). *Deep Learning with Class-Level Abstract Syntax Tree and Code Histories for Detecting Code Modification Requirements*. <https://doi.org/10.1016/j.jss.2023.111851>

Karanikiotis, T., Diamantopoulos, T., & Symeonidis, A. (2023). *Employing Source Code Quality Analytics for Enriching Code Snippets Data*. <https://www.mdpi.com/2306-5729/8/9/140>

Guo, S., Li, M., Ge, X., Li, H., Chen, R., & Li, T. (2023). *Constructing Meaningful Code Changes via Graph Transformer*. <https://doi.org/10.1049/sfw2.12097>

Obert, J., & Loffredo, T. (2022). *Efficient Binary Static Code Data Flow Analysis Using Unsupervised Learning*. <https://doi.org/10.1142/S1793351X2220001X>

Meng, S., Wu, S., Li, A., Jiao, J., Zhang, N., & Zhang, Q. (2022). *Partial Self-Concatenation Structure and Performance Analysis of Spinal Codes Over Rayleigh Fading Channel*.

<https://ieeexplore.ieee.org/document/9741327>

Pecorelli, F., Lujan, S., Lenarduzzi, V., Palomba, F., & De Lucia, A. (2022). *On the Adequacy of Static Analysis Warnings with Respect to Code Smell Prediction*.

<https://link.springer.com/article/10.1007/s10664-022-10126-5\>

Sayago-Heredia, J., Pérez-Castillo, R., & Piattini, M. (2021). *A Systematic Mapping Study on Analysis of Code Repositories*. <https://doi.org/10.15388/21-INFOR454>

Matsushita, M., Endo, T., & Yamamoto, A. (2020). *Applicability of a Reduced Order Model for a Safety Analysis Code to Statistical Safety Analysis*. <https://doi.org/10.1080/00223131.2020.1783382>

Ruiz-Rube, I., Person, T., Dodero, J., Mota, J., & Sánchez-Jara, J. (2020). *Applying Static Code Analysis for Domain-Specific Languages,* (pp. 146-167). <https://link.springer.com/article/10.1007/s10270-019-00729-w>

Jesse K., & Devanbu P., (2023). Learning to predict user-defined types. *Proceedings of the ACM SIGPLAN Symposium on Programming Languages*, (pp. 1508 – 1522) <https://ieeexplore.ieee.org/document/9785755>

Spielmann D., & Sokolowski D., (2023). Extensible testing for infrastructure as code. *Journal of Systems and Software*, (pp 58-60). <https://dl.acm.org/doi/10.1145/3618305.3623607>

Bugner J., & Merkel M., (2022). To type or not to type? A systematic comparison of the software quality of JavaScript and TypeScript applications on GitHub. *Empirical Software Engineering*, (pp. 658 – 669). <https://dl.acm.org/doi/10.1145/3524842.3528454>

Cristiani F., & Thiemann P., (2021). Generation of TypeScript declaration files from JavaScript code. *IEEE Transactions on Software Engineering*, (pp. 97 – 112). <https://dl.acm/doi10.1145/3475738.3480941>

Thomas B., & Moskal M., (2019). Static TypeScript: An implementation of a static compiler for the TypeScript language. *Proceedings of the International Conference on Programming Language Design and Implementation*,(pp.105-116) <https://dl.acm.org/doi/10.1145/3357390.3361032>

Jagtap, V., & Shrawankar, U. (2016). *Dependency Analysis for Secured Code Level Parallelization*. Procedia Computer Science, 78, (pp. 831–837). <https://doi.org/10.1016/j.procs.2016.02.067>

Yang, C., Chen, J. J., Jiang, J. J., & Huang, Y. L. (2024). *Dependency-Aware Code Naturalness*.

*Proceedings of the ACM on Programming Languages,8(OOPSLA2),* (pp. 178-212).

<https://doi.org/10.1145/3689794>

Borowski, K., Balis, B., & Orzechowski, T. (2022). Graph Buddy: An interactive code dependency browsing and visualization tool. *Proceedings of the 2022 IEEE Working Conference on Software Visualization (VISSOFT)*, (pp. 152–156). IEEE. <https://doi.org/10.1109/VISSOFT55257.2022.00023>

Gao, X., Jiang, X., Wu, Q., Wang, X., Lyu, C., & Lyu, L. (2021). Multi-modal code summarization fusing local API dependency graph and AST. *Communications in Computer and Information Science*, 1516, (pp. 290–297). Springer. <https://doi.org/10.1007/978-3-030-92307-5_34>

Ivanov, K., & Urbanke, R. (2020). On the dependency between the code symmetries and the decoding efficiency. In *2020 International Symposium on Information Theory and Its Applications (ISITA)* (pp. 195–199). IEEE. <https://doi.org/10.48550/arXiv.2001.03790>