Reliability of the Type System in TypeScript in Software Development

Abstract. The article examines the TypeScript type system as a critical element influencing its use in software development. The primary objective of the study is to analyze the characteristics of the type system and identify methods for improving the accuracy of type checking. The article explores the principles underlying the type system, its vulnerabilities such as the use of the any type, type assertions, issues with object and array indexing, and proposes approaches to enhancing system reliability.

The methodology includes an analysis of TypeScript's structure, the principles of its compiler, and the application of tools for static code analysis. The study references academic articles available in open-access online repositories, providing a broad perspective on the topic. Additionally, examples of code presented in the work illustrate key concepts and methods for working with type systems. Results demonstrate that configuring the compiler, avoiding the any type, and leveraging libraries for data validation improve the quality of type safety.

The findings, based on the analysis of relevant sources, will be useful for programmers and corporate professionals. **This review paper is a guide for software developers to understand the essence and the reliability of the Type System in TypeScript. This is a requirement to the scientific community as it guides software developers with the understanding and methods for efficiency in Type Checking. Principles, vulnerabilities and practice in code analysis are explicitly enumerated.**

Keywords: TypeScript, type system, static typing, type safety, reliability, compiler, static analysis, programming.

Introduction

The article examines the TypeScript type system, its role in software development, and methods for improving type safety and soundness. TypeScript's popularity stems from its well-designed type system and a robust compiler. However, the type system has limitations that reduce its reliability. The use of the any type, type assertions and indexing operations leads to errors in the code, thus weakening type control. These aspects emphasize the need to analyze the system's capabilities, identify its weaknesses, and subsequently improve its efficiency. In this context, it is necessary to understand how different programming languages address these challenges. By studying the approaches of the programming languages used, it is possible to identify the best methods to improve type handling.

The growing use of TypeScript in web application and software system development underscores the relevance of this topic. The reliability of the type system is a critical criterion for selecting programming languages and developer tools, as it directly impacts development time, code quality, and application security. Research on methods to optimize TypeScript's type system contributes to the creation of more robust software solutions.

The advantages of static typing in programming languages include:

- Identifying errors during the development phase, which reduces development costs and enhances the quality of the final product.
- Enhancing the capabilities of development tools, particularly code editors (IDEs). The type system enables development tools to better understand the written code, automating scenarios such as code editing. For example, renaming a variable automatically updates all instances of its usage within the program.
- Facilitating the division of large programs into smaller modules by defining interaction contracts through the type system. This improves project scalability, enabling the development of large software systems without compromising quality.

The objective of this study is to explore the reliability of the TypeScript type system and to identify methods for improving the efficiency of type checking during TypeScript program compilation.

Materials and Methods

Recent studies on the application of TypeScript cover various areas, including code quality improvement, type automation, data security, and information flow analysis. These topics address several key aspects that influence the efficiency of development and the functionality of software systems.

One of the reasons for choosing TypeScript is its ability to enhance code quality through static typing. Emmanni P. [1] demonstrated that the use of TypeScript reduces the number of errors during development, improves code structure, and simplifies the comprehension of program logic. This is particularly relevant for JavaScript applications that require type control and clear data structure definitions.

The study by Rani M. [2] highlights how strict typing facilitates debugging and refactoring processes. Type annotations make error detection easier, minimize the occurrence of errors during code modifications, and accelerate the development process.

Works by Merkel M. [4] and Bogner J., Merkel M. [5] provide a comparative analysis of code written in TypeScript and JavaScript. Their results confirm that using TypeScript enhances code readability and maintainability while reducing errors. This demonstrates the creation of more stable solutions.

The work by Wu Z. et al. [3] focuses on performance, examining methods for optimizing TypeScript programs. For instance, technologies such as inline caching are proposed to increase execution speed, along with improvements to type hierarchies that reduce overhead associated with type checking, thereby accelerating program execution.

TypeScript is employed to address tasks related to data security and information flow analysis. In the article by Chadalawada A. [6], the use of static typing for security analysis is discussed. The proposed approach allows for the detection of data leaks at early stages of development, preventing them before the final assembly of the application.

The work by Seidel L. et al. [8] explores type inference for data flow analysis in TypeScript. The results indicate that strict typing aids in analyzing data dependencies, improving processing and enhancing security.

The article by Cristiani F. and Thiemann P. [7] describes the process of automatically generating declarations for TypeScript based on existing JavaScript code. This approach simplifies the integration of third-party libraries, improving the security and reliability of interactions with external components.

The data from source [9] on the papl.cs.brown.edu website outlines a theoretical approach to type safety. It discusses concepts such as type safety and soundness, which are fundamental to ensuring program correctness.

Source [10], found on the www.executeprogram.com website, describes the practical aspects of ensuring type safety in TypeScript development. It explores techniques and approaches for achieving type safety, as well as potential errors that may arise if the type system is used incorrectly.

A review of scientific studies demonstrates that TypeScript enhances development quality, simplifies code management, and improves program performance. This is especially significant for projects requiring a high degree of code structuring. However, several issues remain unresolved. Additionally, the integration of TypeScript with non-standard JavaScript libraries and methods for ensuring type safety in large dynamic applications have been insufficiently studied. These topics require further research.

The methodology of this study includes an analysis of TypeScript's architecture, the study of compiler principles, and the use of tools for static code analysis.

Results and Discussion

The type systems of modern programming languages ensure strict data validation, reducing the likelihood of errors during program development and execution.

When discussing type safety, many operations in a language are partial: they are defined over a certain domain, accepting some but not all elements of that domain. A safe language provides developers with an important guarantee: no operation will execute on nonsensical data [1,2]. This guarantee is achieved through a multi-level type checking process, encompassing both static and runtime checks, as illustrated in Figure 1 below.

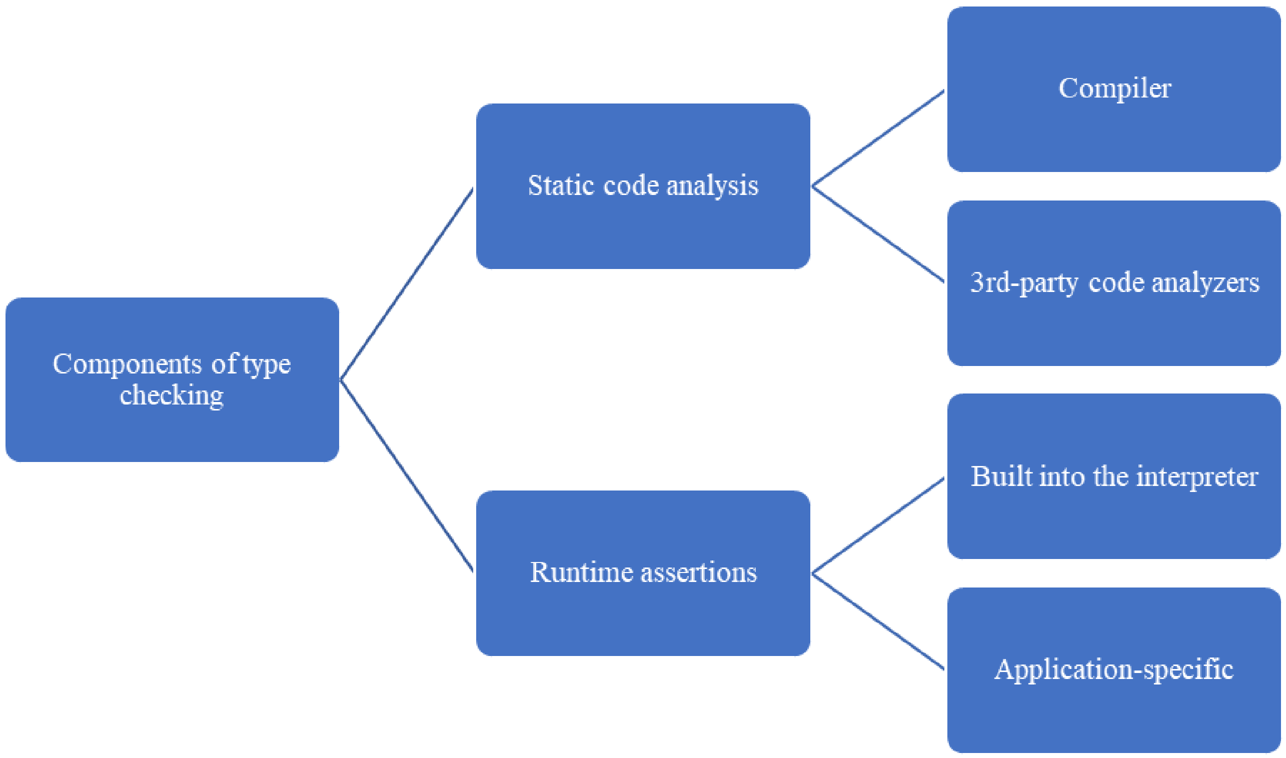

Fig.1. Components of type checking

The reliability of a language's type system lies in the fact that the type environment binds identifiers to types, while the interpreter environment binds identifiers to values or locations. A type-checking mechanism compresses potentially infinite sets of values into types, while the interpreter processes individual elements of those sets differently. The primary result desired for this type of system is referred to as soundness.

The standard method for proving this theorem is to divide it into two parts, known as progress and preservation. Progress states that if a term passes type-checking, it can take a computational step (unless it is already a value); preservation states that the result of this step will retain the same type as the original. When either of these properties fails—such as an array index is out of bounds—the program lacks a meaningful type. Therefore, every type of soundness theorem implies a set of documented, permissible exceptions or error conditions that may arise. A developer using the type system implicitly agrees to accept this set [9].

TypeScript offers features that provide development flexibility and software reliability. It is designed as a statically typed language. However, its type system cannot be considered fully sound. The Figure 2 below illustrates key approaches for improving TypeScript's type system reliability.

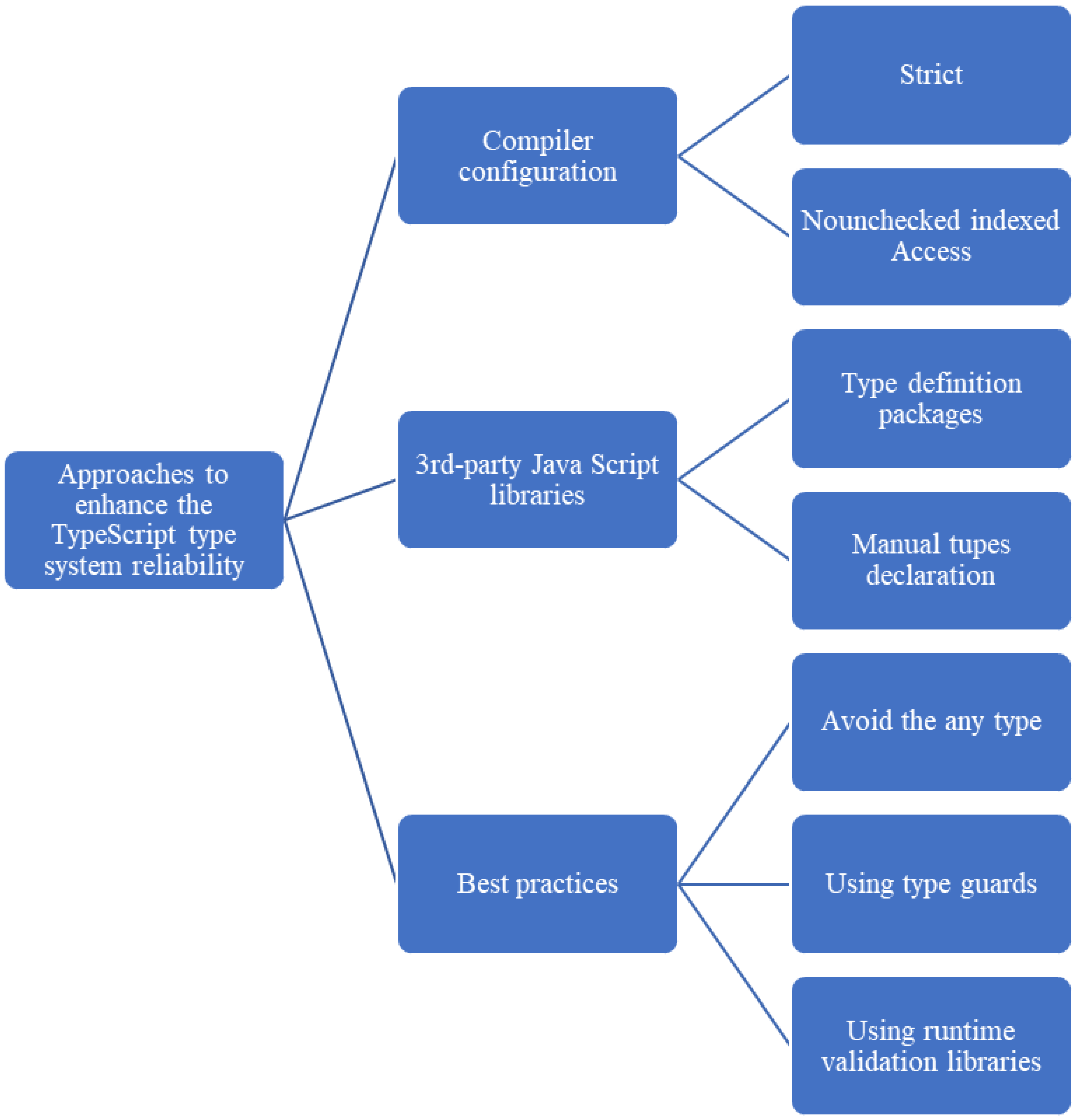
Approaches to enhance the TypeScript type system reliability
Compiler configuration
3rd-party Java Script libraries
Best practices
Strict
Nounchecked indexed Access
Type definition packages
Manual tupes declaration
Avoid the any type
Using type guards
Using runtime validation libraries

Fig.2. Approaches to enhance the TypeScript type system reliability

Unlike many other statically typed languages (e.g., C, Java, C#), TypeScript is built as a superset of JavaScript, which means it must support all JavaScript constructs and features. This fundamental design choice leads to certain vulnerabilities in TypeScript's type system. These vulnerabilities are deliberately introduced by the language's authors to ensure that TypeScript can work seamlessly with JavaScript code. JavaScript, being a dynamically typed language, was never designed with type systems in mind. Table 1 below outlines the reasons why the TypeScript type system cannot be considered entirely sound.

Table 1. The reasons why the type system is not reliable [10].

Reason	Description
Iterative development of the language led to overcomplication of the compiler's configuration	The compiler has been evolving since 2012, gradually improving the reliability of the type system. To maintain backward compatibility with existing projects, many new type checks are included in the compiler through configuration files and are disabled by default.
Decisions made to simplify learning for new developers and increase the language's popularity	An absolutely reliable type system would require developers to put significantly more effort into describing types during program development. This entails learning new design patterns, gaining a deeper understanding of type theory, and writing substantial additional code.
The necessity to account for JavaScript's dynamic typing to ensure compatibility with a large number of existing JavaScript libraries	This significantly reduces the cost of transitioning from JavaScript to TypeScript in existing projects and accelerates the development of new products.

The main vulnerabilities in the TypeScript type system are as follows:

The any type. Its vulnerability lies in the concept that a variable can accept values of any type without any checks from the compiler. While useful in situations where the type of a value cannot be predicted in advance, it simultaneously reduces the type safety provided by TypeScript. Using this type effectively disables the type system, as the variable can be assigned any value without causing compilation errors. This can lead to unexpected results during runtime, as type errors are only discovered at runtime rather than at the compilation stage [4,5]. Below is a code example:

```
// Example of using the `any` type
function processData(data: any): void {
  // Runtime error if `data` does not have a 'name' property
  console.log(data.name);
}

const user = { name: "Alice", age: 30 };
processData(user); // Works fine because `user` has a 'name'
property

const notUser = { age: 40 };
processData(notUser); // Runtime error: 'name' property does not
exist
```

Given these vulnerabilities, implementing strict control over the any type usage becomes essential for maintaining type system integrity. The static analysis capabilities provided by the typescript-eslint library enable systematic identification and elimination of the any type occurrences throughout the codebase. This methodology ensures type safety not only in application code but also extends to interactions with the standard library and third-party dependencies, ultimately achieving type system reliability comparable to established statically typed languages such as Java, Go, and Rust.

Type Assertions and Custom Type Guards in TypeScript represent a mechanism that allows developers to inform the compiler that an object or variable has a specific type, even if TypeScript cannot automatically guarantee this. Type Assertions are particularly useful when the developer is confident in the type of a value, but the compiler lacks sufficient context to infer it. However, it is important to note that this mechanism is merely a way to "convince" the compiler and does not affect runtime type checking [4, 3, 10]. Below is an example of code illustrating Type Assertions:

```
// Example using Type Assertion
function getElementById(id: string): HTMLElement {
  const element = document.getElementById(id);
  // By default, document.getElementById returns the type
HTMLElement | null
  // We assert that the element definitely exists (is not null)
  return element as HTMLElement;
}

const element = getElementById("myElement");
console.log(element.innerHTML);
```

A more robust approach to type safety involves implementing Type Guards. This methodology enables developers to provide the necessary context for the compiler while moving some of the checks to the runtime. In TypeScript Type Guards represented by functions and operators that perform runtime checks. For instance, a type guard can verify object type membership using the instanceof operator. Below is an example of code illustrating Type Guards:

```
// Example using Type Guard
function getElementById(id: string): HTMLElement {
  const element = document.getElementById(id);
  if (!(element instanceof HTMLElement)) {
    // Runtime error indicating an incorrect developer's
assumption about the element type
    throw new Error('The element has unexpected type');
  }
  // The element now has the type HTMLElement
  return element;
}
```

The implementation of Type Guards should be approached selectively within the application architecture. They should be implemented in scenarios where TypeScript's static code analysis proves insufficient for ensuring comprehensive type safety. The key scenarios where Type Guards prove most valuable are presented in Table 2.

Table 2. Scenarios Requiring Type Guards.

Reason	Description
Providing the compiler with context about the execution environment	TypeScript's static analysis capabilities are limited in determining the complete runtime context of an application. This limitation is particularly evident when dealing with Document Object Model (DOM) structures, where the compiler cannot perform static analysis of dynamic markup elements during the compilation phase.
Integration with untyped third-party JavaScript libraries.	Integration of third-party JavaScript libraries into TypeScript applications requires providing explicit type information to the compiler. This type information facilitates static analysis and ensures type safety during the integration process. However, a significant number of JavaScript libraries lack comprehensive type declarations in their native implementation.
Handling untyped data from I/O operations, including network requests, disk reads, and user input.	External data sources typically transmit data in an unstructured binary format that requires parsing into language-specific data structures. For instance, when an application receives a JSON-formatted string, it must parse the string using the JSON.parse() method before accessing the data. When working with TypeScript, developers need to add type definitions to this parsed data so the compiler can check for potential type errors during static code analysis.

Third-party JavaScript library integration in TypeScript development commonly uses type definition packages distributed as separate modules. These type definition packages follow the naming convention of using the @types/ namespace prefix. A notable implementation of this approach is demonstrated by the lodash utility library, which is complemented by its corresponding @types/lodash package containing comprehensive type annotations for the library's API surface. Once installed, the TypeScript compiler automatically identifies these type definition packages and enables type-checking for the library usages throughout the application.

In scenarios where type definition packages are unavailable, developers can implement type annotations at the application level using TypeScript's Module Augmentation functionality. The following example demonstrates the implementation of type declarations for the math library's sum method:

```
// Module Augmentation for the "math" library
declare module "math" {
    // Defining exported functions and their type signatures
    export function sum(a: number, b: number): number;
}
```

For processing unstructured data from I/O operations, it is essential to use TypeScript-compatible data validation libraries such as zod, superstruct, or valibot. These libraries provide schema-based validation to ensure data structures match their expected types at runtime. This methodology enables developers to provide type information to the compiler in a declarative and efficient manner, enhancing the compiler's static analysis capabilities. The following example demonstrates data validation utilizing the zod library:

```
import { z } from 'zod';

// Describe the schema for the external data
const personSchema = z.object({
  name: z.string(),
  age: z.number(),
});
```

```

// Infer the type
type Person = z.infer<typeof personSchema>;

function parsePerson(data: string): Person {
  // Safely parse and validate the data structure
  return personSchema.parse(JSON.parse(data));
}
```

The vulnerability of "Object and Array Lookups" in TypeScript arises when developers attempt to access object properties or array elements using loosely typed keys or without proper type validation during compilation. These types are often too generalized and do not strictly enforce the structure of the data, which can lead to runtime errors if developers assume an object has a particular structure or an array has a specific length and elements. For instance, accessing objects or arrays using arbitrary strings or indices can result in unauthorized operations or runtime errors if the actual data structure does not match these assumptions. Effectively, such operations generalize all objects and arrays, which reduces the type safety of their usage [1,5,7]. Below are examples illustrating this vulnerability:

```
// Example of vulnerability with objects
function getUserInfo(user: object): string {
  // Error if the object does not have a "name" property
  return user["name"].toUpperCase();
}

const user = { name: "Alice" };
console.log(getUserInfo(user)); // Works correctly because the
"name" property exists

const unknownUser = {};
console.log(getUserInfo(unknownUser)); // Runtime error: "name"
property does not exist

// Example of vulnerability with arrays
function getElementAtIndex(arr: unknown[], index: number): any {
  return arr[index]; // Does not check if the element exists at
the given index
}

const numbers = [1, 2, 3];
console.log(getElementAtIndex(numbers, 2)); // Returns 3
console.log(getElementAtIndex(numbers, 5)); // Returns undefined
but does not trigger a compile-time error
```

To address this vulnerability, TypeScript provides the noUncheckedIndexedAccess compiler option. When enabled, this option enforces stricter type checking for object and array access, making the type system more sound by requiring explicit handling of potential undefined values.

When discussing error detection during compilation, TypeScript helps eliminate the following defects:

- Incorrect assignments.
- Use of invalid arguments in functions.
- Access to non-existent object properties.

```
interface Person {
  name: string;
  age: number;
}

// The compiler reports a type mismatch.
const person: Person = { name: "Alice", age: "25" };
```

Such errors are addressed before runtime, preventing crashes during execution. Type checking allows for safe data handling using conditional checks, eliminating the need for complex additional validations. Example:

```
function getUserInfo(user: Person | null): number | undefined {
  return user?.age;
}
```

TypeScript enables defining contracts between software components, which is particularly critical in collaborative development. Example:

```
interface UserModule {
  getName: () => string;
  getEmail: () => string | undefined;
}
```

This approach enhances consistency among developers and simplifies integration. Despite its many advantages, TypeScript has some limitations that should be considered. Since TypeScript is transpiled into JavaScript, certain errors can only surface during runtime [2, 4, 9]. For instance:

```
const info= JSON.parse('{"name": "Alice"}');
info.age.toString(); // The error is detected during runtime.
```

Thus, the type system cannot catch errors related to algorithms, as they fall outside the scope of data validation. TypeScript provides developers with a set of tools that enhance software reliability through strict typing and extensive type-related capabilities. Despite its limitations, the language remains an effective choice for creating modern software solutions.

Conclusion

The analysis of the TypeScript type system has confirmed its importance in the development of software solutions. This study examined the theoretical foundations of type systems, identified vulnerabilities in the type handling mechanism, and proposed methods for addressing these issues. The results demonstrated that TypeScript, with its flexibility, remains an effective tool for improving code quality and security.

TypeScript enables the detection of errors at early stages of development, simplifies the maintenance of software systems, and facilitates project scaling. However, constructs such as the any type, type assertions, and indexing operations reduce the reliability of the system. To enhance type safety and minimize risks, the proposed methods include compiler configuration, static code analysis, and the use of libraries for data validation.

In conclusion, TypeScript offers a well-designed type system and a robust compiler. However, type checking can be made significantly more reliable through compiler options, incorporating static code analysis with tools like typescript-eslint, and utilizing libraries for validating data structures.

COMPETING INTERESTS

Authors have declared that they have no known competing financial interests OR non-financial interests OR personal relationships that could have appeared to influence the work reported in this paper.

Disclaimer (Artificial intelligence)

Option 1:

Author(s) hereby declare that NO generative AI technologies such as Large Language Models (ChatGPT, COPILOT, etc.) and text-to-image generators have been used during the writing or editing of this manuscript.

Option 2:

Author(s) hereby declare that generative AI technologies such as Large Language Models, etc. have been used during the writing or editing of manuscripts. This explanation will include the name, version, model, and source of the generative AI technology and as well as all input prompts provided to the generative AI technology

Details of the AI usage are given below:

1.

2.

3.

References

1. Emmanni, P. The Role of TypeScript in Enhancing Development with Modern JavaScript Frameworks // International Journal of Science and Research (IJSR). - 2021. - Vol. 10 (2). - pp.1738-1741.
2. RaniM. Intelligent Coding Using TypeScript // Interantional journal of scientific research in engineering and management. - 2023. - Vol. 7 (11). - pp. 1-3.
3. Wu Z. et al. Hidden inheritance: an inline caching design for TypeScript performance //Proceedings of the ACM on Programming Languages. – 2020. – Vol. 4. – no. OOPSLA. – pp. 1-29.
4. Merkel M. Do TypeScript applications show better software quality than JavaScript applications?: a repository mining study in GitHub : dis. – 2021.

5. Bogner J., Merkel M. To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github //Proceedings of the 19th International Conference on Mining Software Repositories. – 2022. – pp. 658-669.

6. Chadalawada A. Static Taint Analysis via Type-checking in TypeScript. – 2023.

7. Cristiani F., Thiemann P. Generation of typescript declaration files from javascript code //Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes. – 2021. – pp. 97-112.

8. Seidel L. et al. Learning Type Inference for Enhanced Dataflow Analysis //European Symposium on Research in Computer Security. – Cham : Springer Nature Switzerland, 2023. – pp. 184-203.

9. Safety and Soundness. [Electronic resource] Access mode: https://papl.cs .brown.edu/2014/safety-soundness.html#:~:text=There%20is%20a%20good%20reason%20to%20be%20suspicious%20of%20a%20type%20system%2C%20beyond%20general%20skepticism.%20There%20are%20many%20differences%20between%20the%20way%20a%20type%20checker%20and%20an%20interpreter%20work%3A(accessed date: 10.12.2024).

10. Everyday TypeScript: Type Soundness. [Electronic resource] Access mode: https://www.executeprogram.com/courses/everyday-typescript/lessons/type-soundness (date of application: 10.12.2024).