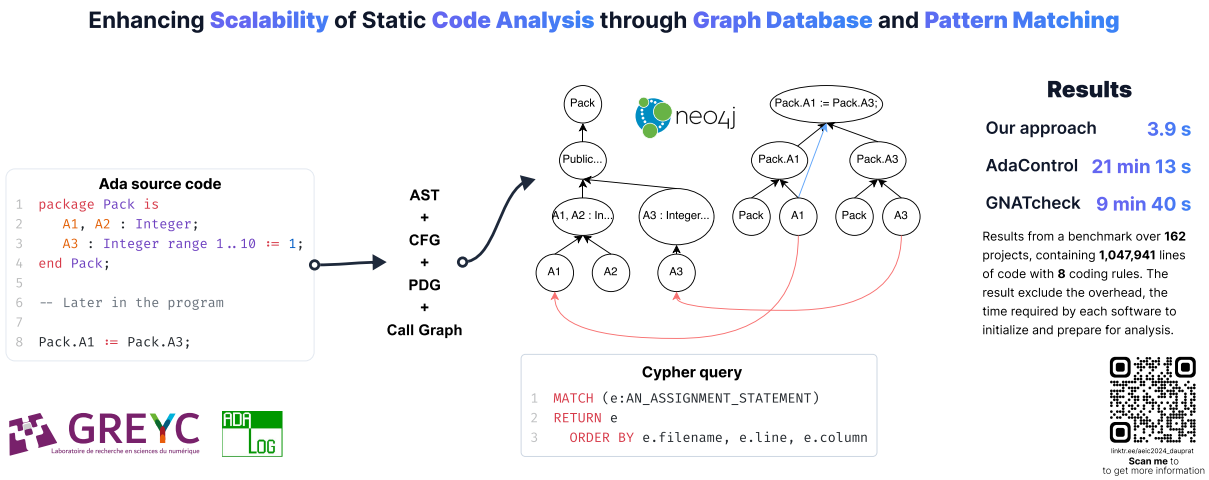
Enhancing Scalability of Static Code Analysis through Graph Database and Pattern Matching

Quentin Dauprat, Paul Dorbec, Gaétan Richard, Jean-Pierre Rosen

Graphical Abstract

Enhancing Scalability of Static Code Analysis through Graph Database and Pattern Matching

Quentin Dauprat, Paul Dorbec, Gaétan Richard, Jean-Pierre Rosen



Highlights

Enhancing Scalability of Static Code Analysis through Graph Database and Pattern Matching

Quentin Dauprat, Paul Dorbec, Gaétan Richard, Jean-Pierre Rosen

- Demonstrated that integrating a graph database and pattern matching queries can significantly improve the performance and scalability of static code analysis tools, with execution times reduced by up to 326 times compared to traditional tools.

- Introduced a methodology to represent source code as a Neo4j graph database, enabling efficient traversals and pattern matching queries for analysis.

- Evaluated approach on Ada codebases ranging from hundreds to hundreds of thousands of lines, demonstrating enhanced scalability.

- Showed potential to incorporate graph-based analysis into modern software engineering workflows, supporting improved code quality and developer productivity.

Enhancing Scalability of Static Code Analysis through Graph Database and Pattern Matching

Quentin Dauprat[a,b,*], Paul Dorbec[b], Gaétan Richard[b], Jean-Pierre Rosen[a]

[a]*Adalog, 2 rue du docteur lombard, Issy-les-Moulineaux, 92130, Île-de-France, France*
[b]*Université de Caen Normandie, Normandie Univ, UNICAEN, ENSICAEN, CNRS, GREYC, Caen, 14000, Normandie, France*

Abstract

This study aims to enhance the scalability of static code analysis tools by combining graph databases and pattern matching queries. Traditional tools often struggle with large codebases, leading to increasingly long analysis times. A significant portion of this time is attributed to the necessity of accessing all information pertaining to an object, which may be dispersed across numerous files, and so, into multiple Abstract Syntax Trees (ASTs). The proposed method stores and queries code representations in a graph database, improving efficiency and scalability. The study focuses on Ada, a language with complex code structures, and benchmarks the method against established tools such as AdaControl and GNATcheck. The results show that the proposed method outperforms conventional tools, demonstrating its potential to enhance static code analysis.

While graph databases have been used for code analysis in other languages, this study presents the first comprehensive application of this approach to Ada for coding rule verification.

First results obtained show a clear improvement in terms of performance, being 326 times faster than AdaControl and outshining GNATcheck, ranging from 57 to 148 times faster respectively in multithreaded (32 cores) and mono-thread configurations.

Keywords:
Ada, Graph Databases, Neo4J, Pattern Matching, Scalability, Static code analysis

1. Introduction

The increasing complexity and mission-critical nature of software systems necessitate robust measures for ensuring their dependability, security, and performance. Static code analysis (SCA), which scrutinizes software code for potential issues and vulnerabilities without execution, has surfaced as an indispensable technique in this domain [1]. Despite its utility, the effectiveness of traditional static analysis tools is often hindered by the expanding scale and intricacy of contemporary software architectures, resulting in prolonged analysis duration and restricted scalability [2].

A considerable amount of the analysis time in conventional tools can be attributed to the

*Corresponding author
Email addresses: dauprat@adalog.fr (Quentin Dauprat), paul.dorbec@unicaen.fr (Paul Dorbec), gaetan.richard@unicaen.fr (Gaétan Richard), rosen@adalog.fr (Jean-Pierre Rosen)

requirement of accessing comprehensive information about a code construct, which may be scattered across multiple files and Abstract Syntax Trees (ASTs). This fragmentation of information poses significant challenges for traditional AST-based methodologies, impeding their ability to efficiently conduct complex analysis tasks that necessitate navigating and integrating data from disparate sections of the codebase [3]. In response to these challenges, this study introduces a pioneering approach that integrates graph databases with pattern matching queries to enhance the scalability and efficiency of static code analysis.

The use of graph databases for representing and analyzing code offers a promising approach to address these challenges. Graph databases, with their ability to efficiently store and query highly connected data, can enable more intelligent and context-aware code analysis tools. By leveraging the rich semantic information captured in the graph representation, these tools can provide developers with valuable insights into code quality, maintainability, and potential vulnerabilities. Moreover, the scalability and performance characteristics of graph databases make them suitable for handling large and complex codebases.

This paper builds upon and significantly extends our preliminary work presented in a work-in-progress paper published in 2022 [4]. While the initial study introduced the concept of using graph databases for Ada static code analysis, the current research presents a comprehensive implementation, evaluation, and analysis of the approach. We have conducted benchmarking to validate the effectiveness of our graph-based static analysis technique.

This research extends the concept of Code Property Graphs (CPGs) introduced by Yamaguchi et al. [2] to the Ada programming language. We adapt and enhance the CPG model to capture Ada-specific semantics. This novel representation enables more efficient and expressive analysis of Ada code compared to traditional AST-based methods.

The principal aim of this research is to investigate the application of graph databases and pattern matching queries in static code analysis, notably for coding rule verification, with a particular emphasis on the Ada programming language. Ada is renowned for its intricate code structures and is predominantly employed in safety-critical systems where the quality and reliability of code are paramount. This research endeavors to advance static analysis techniques by developing a graph-based representation of Ada code and validating its effectiveness through rigorous benchmarking, thereby contributing to its application in real-world software development scenarios.

In the context of this research, we define scalability for static code analysis as the ability of a tool to efficiently analyze codebases of varying sizes, from small projects to large-scale systems, within reasonable time frames. Our primary focus is on reducing analysis time, which may potentially be achieved through various means, including but not limited to multi-core processing or distributed computing. However, the specific implementation of such optimizations is beyond the scope of this current study. The key objective is to develop an approach that maintains performance and accuracy across a wide range of codebase sizes, ensuring that the analysis remains practical and useful for both small-scale and large-scale software projects.

The proposed methodology involves the representation and querying of code within a graph database, specifically Neo4j, which facilitates efficient traversal and pattern matching. This graph-based representation encapsulates the semantic relationships among code elements, thereby enabling more expressive

and intuitive analysis queries compared to traditional AST-based methods. The efficacy of this novel approach is assessed through a comparative analysis with established static analysis tools for Ada, namely AdaControl [5] and GNATcheck [6].

The key contributions of this research are delineated as follows:

- The development of an Ada-specific graph-based code representation using Neo4j, extending the Code Property Graph concept to capture Ada's unique language features.
- A novel approach to coding rule verification leveraging graph pattern matching, specifically tailored for Ada's complex type system and modular structure.
- A comprehensive evaluation of the proposed method across various Ada codebases, demonstrating marked enhancements in analysis speed and scalability in comparison to conventional AST-based tools.
- Provision of insights regarding the potential of graph databases and pattern matching queries to augment coding rule verification, with broader implications for their application across different programming languages.

The structure of the remainder of this paper is outlined as follows: Section 2 provides a review of related literature on static code analysis and graph databases. Section 3 details the proposed graph-based representation of Ada code and the methodology employed for static analysis using pattern matching queries. Section 4 elucidates the results of the experimental evaluation, showcasing the efficiency and scalability of the proposed approach relative to traditional AST-based tools. Section 5 present the benchmark environments and results. Section 6 discusses the implications of these findings and identifies prospective avenues for future research. Finally, Section 7 concludes the paper by summarizing the major contributions and outcomes of the research, which indicate a significant performance enhancement, with the proposed methodology achieving speeds up to 326 times faster than AdaControl and surpassing GNATcheck by factors ranging from 57 to 148 in mono-thread and multithreaded configurations, respectively.

2. Related Work

2.1. Overview of Static Code Analysis

Static code analysis involves inspecting source code to uncover bugs, vulnerabilities, and quality issues without executing programs [7, 8]. As software grows in size and complexity, static analysis is critical for ensuring code quality and reliability [9]. Objectives include detecting security vulnerabilities, enforcing standards, improving maintainability, optimizing performance, and verifying correctness. Compared to dynamic analysis, static analysis provides complete coverage and savings by identifying issues without execution [7].

Early tools utilized heuristic pattern matching to catch common bugs [10]. Coding standards drove techniques like abstract interpretation for semantic approximation [11, 12], integration into IDEs and pipelines [9, 13], and machine learning to infer patterns [14]. Prevalent methods include data flow analysis, control flow analysis, pattern matching against rule catalogs, and metrics computation [15, 7, 16, 17, 18, 19]. However, challenges persist around scalability, false positives, and language support [9, 20, 8].

2.2. Graph Databases

Graph databases effectively model complex relationships [21]. Unlike relational databases, they are optimized for highly interconnected data via network traversal [22]. Core concepts are nodes for entities, edges for relationships, and properties. This flexible representation suits intricate networks across domains. Within software engineering, graph databases represent codebases, dependency trees, hierarchies, and call graphs [3].

Neo4j exemplifies graph databases, providing availability, scalability, and the Cypher query language. Prior work established representing code as graphs enabled more effective static analysis through nodes for artifacts and edges for connections [23, 2]. However, research gaps remain regarding graph databases' scalability and efficiency in static analysis. This work evaluates these techniques for Ada.

2.3. Use of graph databases for static code analysis

The application of graph databases to source code analysis is an ongoing field of research for a decade now. Several researchers have explored this approach for various programming languages and analysis tasks.

Urma and Mycroft [24] pioneered the use of graph databases for source code queries, demonstrating their potential for analyzing program structure and evolution. They implemented a prototype system using Neo4j and showed its scalability for multi-million-line programs. While their work laid important groundwork, it focused primarily on general code queries rather than specific static analysis tasks. Our research builds upon their findings but differs in several key aspects. Firstly, we focus specifically on Ada, a language with distinct challenges due to its strong typing and package structure. Secondly, our approach is tailored for coding rule verification rather than general code queries, addressing the unique requirements of Ada development practices.

Yamaguchi et al. [2] introduced the concept of Code Property Graphs (CPGs) for vulnerability discovery in C code. They combined abstract syntax trees, control flow graphs, and program dependence graphs into a single graph representation, enabling more sophisticated analysis. Our work extends this concept to Ada, adapting and enhancing the CPG model to capture Ada-specific semantics.

Ramler et al. [3] provided a comprehensive study on the benefits and drawbacks of using graph databases for representing and analyzing source code and software engineering artifacts. They explored various use cases, including dependency analysis and architecture recovery. Our research builds upon these findings but goes further by adapting the graph-based approach to the specific needs of Ada static analysis. We demonstrate how the unique features of Ada can be effectively represented and analyzed using a graph database, providing a novel solution for Ada developers and maintainers.

Rodriguez-Prieto et al. [25] presented an efficient platform for Java source code analysis using overlaid graph representations. Their work shares similarities with our approach in using graph structures for code analysis and demonstrated significant performance improvements over traditional AST-based methods. However, our research differs in its focus on Ada and the specific challenges of coding rule verification. We extend the concept of overlaid graphs to capture Ada-specific semantics and demonstrate its effectiveness in a language domain with distinct characteristics from Java.

The state of the art in this field is continuing to explore the vulnerabilities detection [26]. However, we can notice that Borowski et al. [27] proposed the Semantic Code Graph

(SCG), a comprehensive source code model that preserves direct relations to the code and capture semantics and dependencies between code entities. Implemented for Java and Scala, SCG supports a wide range of software comprehension tasks, including finding critical entities and visualizing dependencies. Chen [28] presented a variability-aware Neo4j approach for analyzing software product lines, lifting the Neo4j query engine to reason over the fact base of an entire product line and return variability-aware results. Despite the limited number of publications in this field, these approaches demonstrate the growing interest in leveraging graph databases and pattern matching for various aspects of software analysis, from security-focused applications to broader comprehension tasks and product line analysis.

While these previous works have established the potential of graph databases for code analysis, there remains a significant gap in their application to Ada programming language and coding rule verification. Ada and coding rule verification present specific challenges that have not been addressed by existing graph-based analysis approaches.

Our research fills this gap by providing a comprehensive adaptation of graph-based static analysis techniques to Ada. We not only address the language-specific challenges but also focus on the practical aspects of coding rule verification, which is crucial for maintaining code quality in Ada projects, particularly in safety-critical domains. By demonstrating significant performance improvements over existing Ada static analysis tools, our work provides a novel and practical solution to a pressing need in the Ada development community.

In summary, while our work builds upon the foundation laid by previous research in graph-based code analysis, it makes significant novel contributions in its application to Ada, its focus on coding rule verification, and its practical performance improvements. These advancements address a crucial need in Ada development and maintenance, particularly for large-scale, safety-critical systems where efficient and accurate static analysis is paramount.

3. Methodology

The proposed approach for enhancing static code analysis involves representing Ada code as a graph database using Neo4j. This graph-based representation captures the semantic relationships between code elements, enabling more efficient and expressive analysis compared to traditional AST-based methods.

3.1. Definition related to graph theory

A graph comprises vertices and edges, where vertices represent distinct entities and edges denote relationships between vertices [29]. Graphs may be directed, with edges possessing directionality, or undirected, lacking directionality.

Labeling assigns categorical identifiers to vertices and edges based on shared characteristics, enhancing organization and semantic structure [25].

Property graphs extend traditional graphs by incorporating additional descriptive information via key-value pairs associated with vertices as node properties and edges as edge properties [21]. This augmentation provides nuanced detail beyond connectivity. Furthermore, labels can be added to vertices and edges to help categorize elements.

Figure 1 depicts a fragment of a property graph representing the following Ada code:

```
procedure Test is
   Year : number;
begin
   Year := 2024;
```

```
end Test;
```

Listing 1: Example of an Ada code to illustrate Poperty Graph

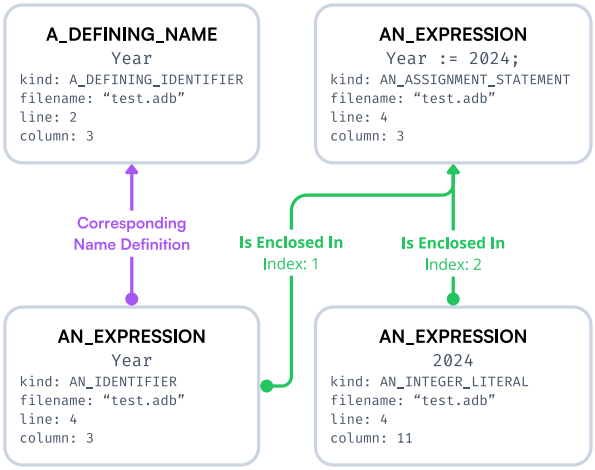


Figure 1: Example of a Property Graph

This property graph illustrates several key concepts:

- **Nodes**: The graph contains four nodes, each representing a different element of the Ada code.
- **Relationships**: The edges between nodes represent relationships, such as 'Corresponding Name Definition' and 'Is Enclosed In', which describe how the code elements are connected. The relationship 'Is Enclosed In' has a property 'Index'.
- **Labels**: Each node and edge has a label (e.g., A_DEFINING_NAME, AN_EXPRESSION, Is Enclosed In, Corresponding Name Definition) that categorizes the type of code element it represents.
- **Properties**: Nodes have properties such as 'kind' (specifying the exact nature of the node), 'filename', 'line', and 'column' (indicating the location in the source code). In the interests of clarity, the second line of Nodes refers to a 'content' property of the node.

For instance, the top-left node labeled 'A_DEFINING_NAME' represents the declaration of the 'Year' variable. It is connected to an 'AN_EXPRESSION' node (bottom-left) via a 'Corresponding Name Definition' relationship, indicating that from "AN_EXPRESSION" node of kind "AN_IDENTIFIER" this identifier is declared in "A_DEFINING_NAME" node in top-left.

The top-right node, also labeled 'AN_EXPRESSION', represents the assignment statement. It has two 'Is Enclosed In' relationships (which is the relationship with the parent), connecting it to the nodes representing the left-hand side (the 'Year' identifier) and the right-hand side (the integer literal '2024') of the assignment.

This property graph structure allows for rich representation of code elements and their relationships, enabling complex queries and analyses in static code analysis tools.

3.2. Definition related to static code analysis (SCA)

In order to define Code Property Graph (CPG), the key structure used as a backbone in our research, we will introduce key structures used in static code analysis, which are the building block of CPG.

Abstract Syntax Tree (AST). An AST is a tree representation of the abstract syntactic structure of a source code. Each node in the tree represents a construct occurring in the source code, such as variables, operators, and control structures. The AST abstracts away certain details and retains just enough information to help the compiler understand the structure of the code.

Control Flow Graph (CFG). A CFG is a graphical representation of the control flow. It is a directed graph where nodes represent basic blocks of code, and edges represent control flow paths between these blocks. CFGs are used in static analysis and compiler applications to represent the flow inside a program unit accurately. They show all the paths that can be traversed during program execution, making it easier to locate inaccessible code and identify syntactic structures like loops. CFGs are essential for reasoning about code and performing checks for errors, such as undefined variables [30, 31].

Call Graph (CG). A CG is a graphical representation of the relationships between different subprogram calls within a program. It illustrates how different methods in the program are connected, showing the flow between methods and the parts of third-party libraries that are used. Call graphs can be created manually or automatically, with automated methods, saving time and ensuring accuracy. There are two main approaches to creating call graphs: static analysis, which analyzes the source code without executing it, and dynamic analysis, which involves running the program and analyzing its behavior. Call graphs are useful for understanding the codebase, debugging, performance optimization, and refactoring.

Program Dependency Graph (PDG). A PDG is a directed graph that represents both the control and data dependencies of a program. In a PDG, nodes correspond to program statements, and edges indicate dependencies between these statements. This dual representation allows the PDG to capture the intricate relationships within the program, facilitating various compiler optimizations and program transformations [32, 33].

Code Property Graph (CPG). Pioneered by Yamaguchi [2], the CPG is a graph representation of source code that captures not only the code's syntax but also its semantics, control flow, and data flow. It is a combination of AST, CFG and PDG. This augmented representation enables more sophisticated analysis like vulnerability detection and comprehension. The CPG in addition with CG forms the backbone of this research by enabling systematic coding rule examination through a graph.

The augmented representation offered by CPG empowers a more nuanced and detailed analysis of software systems, facilitating tasks like program slicing, vulnerability detection, and code comprehension. Its value becomes particularly apparent in scenarios demanding a holistic understanding of software architecture, behavior, and interdependencies.

In the context of our study, we strategically leverage CPGs as the foundational backbone to explore the effectiveness of graph database representations in the domain of static code analysis. This strategic utilization allows us to delve into the systematic examination of coding rules, thereby enhancing the precision and efficiency of rule verification. By utilizing CPG as the cornerstone of our approach, we aim to provide a comprehensive exploration of code semantics, control flow, and data flow, ultimately contributing to improved code quality and adherence to coding standards. This research endeavor underscores the potential of CPGs in advancing program analysis and software engineering processes.

3.3. Selection of the Ada programming language as a case study

The Ada programming language was designed for reliability and maintainability. While the semantic analysis phase of a compiler indeed resolves ambiguities and provides detailed information about the code structure,

the challenge for static analysis tools lies in efficiently accessing and processing this information for large-scale codebases, especially when verifying complex coding rules.

For example, consider the expression $V := A(B);$. A static analysis tool checking coding rules might need to repeatedly access information across multiple Abstract Syntax Trees in order to control a coding rule like ”for each variable in a program, is the variable READ and / or WRITE?”

The difficulty arises when scaling this process to large codebases with millions of lines of code. Many coding rules require gathering information from various parts of the AST, potentially across multiple compilation units. This task is not so hard, but it is time consuming. This necessitates efficient navigation and querying of the code structure, which becomes increasingly challenging with traditional AST-based approaches as the codebase size grows.

Our graph-based approach aims to address these scalability challenges by providing a more efficient way to represent and query these complex relationships, allowing for faster and more flexible analysis of large Ada codebases.

3.4. Graph ”schema”

To capture the semantic information necessary for effective static code analysis, the graph representation includes several types of relationships between nodes. These relationships are designed to express the connections and dependencies among various code elements, enabling efficient querying and pattern matching.

The key relationships in the graph include:

- **CALLING**: Connects a subprogram node to the subprograms it calls, allowing for call graph analysis.
- **CORRESPONDING_ACTUAL_PARAMETER**: Links a parameter association node to the corresponding actual parameter value.
- **CORRESPONDING_ASSIGNATION**: Connects a variable node to the corresponding assignment statement node.
- **CORRESPONDING_FIRST_SUBTYPE**: Relates a subtype node to its corresponding first subtype declaration.
- **CORRESPONDING_FORMAL_NAME**: Links a parameter association node to the corresponding formal parameter identifier.
- **CORRESPONDING_INSTANTIATION**: Connects a node to the corresponding generic instantiation declaration.
- **CORRESPONDING_NAME_DEFINITION**: Relates an identifier, operator symbol, character literal, or enumeration literal node to its corresponding definition.
- **CORRESPONDING_PARAMETER_SPECIFICATION**: Links a parameter association node to its corresponding parameter specification.
- **CORRESPONDING_ROOT_TYPE**: Connects a derived type node to its original ancestor type.
- **CORRESPONDING_SPECIFICATION**: Relates a subprogram, package, task body declaration, or expression function declaration node to its corresponding specification, if available.
- **CORRESPONDING_TYPE_ DECLARATION_VIEW**: Links a type declaration node to its corresponding definition characteristics.

- **IS_ANCESTOR_OF**: Expresses the ancestor-descendant relationship between type declarations.
- **IS_ENCLOSED_IN**: Represents the parent-child relationship between nodes. This is the only relationship in an AST.
- **IS_OF_TYPE**: Connects a declaration node (e.g. component, constant, discriminant specification) to its corresponding type definition.
- **IS_PROGENITOR_OF**: Expresses the relationship between an interface type and its directly implementing types.

These relationships are carefully chosen to strike a balance between expressiveness and database size. Having too many types of relationships can lead to a significant increase in the size and creation time of the database, which may negatively impact performance. Therefore, it is important to limit the number of relationship types to those that are most essential for the desired static code analysis tasks.

The relationships added have been selected based on their semantic significance, particularly in how they facilitate the enforcement of specific coding rules. While it is true that not all coding rules benefit equally from these relationships, the chosen ones are instrumental for a subset of rules that are critical for assessing certain aspects of code quality and security. This targeted approach allows us to maximize the utility of the database while maintaining manageable size and performance.

It is worth noting that, currently, all of the above relationships are created regardless of whether they are used by the specific coding rules being applied. A potential future improvement could involve creating only the relationships required by the coding rules specified by the user. This optimization would help to further reduce the size of the database and improve its creation time, especially for larger codebases.

Added nodes in the graph have the following properties:

- **column**: Column location of the node in file.
- **element_kind**: Kind of the node, such as "A_DEFINING_NAME".
- **enclosing_unit**: Name of the unit where the node is located.
- **filename**: File where the node is located.
- **is_part_of_implicit**: True if the node is, or forms part of any implicitly declared or specified program Element structure. False otherwise.[1]
- **is_part_of_inherited**: True for any node that is, or that forms part of, an inherited primitive subprogram declaration. False otherwise.
- **is_part_of_instance**: True if the node is part of an implicit generic specification instance or an implicit generic body instance. False otherwise.
- **kinds**: All kinds of an element[2], from the less specific to the more specific, such as: Element, A_DEFINING_NAME, A_DEFINING_IDENTIFIER.
- **line**: Line location of the node in file.
- **node_id**: A unique identifier.

[1] Definition from `asis-elements.ads`

[2] ASIS defines multiple element kinds, with subkinds

In order to take advantage of indexes, each node have the equivalent of ”kinds” property in label. For example, one node have multiple labels like: Element, A_DEFINING_NAME, A_DEFINING_IDENTIFIER.

By carefully designing the graph schema and relationships, the proposed methodology aims to provide a rich and efficient representation of Ada code that enables powerful static analysis through pattern matching queries.

3.5. Selection of coding rules

Coding rule verification presents distinct challenges compared to vulnerability analysis or bug checking. While vulnerability analysis often focuses on specific patterns of dangerous code constructs, and bug checking looks for logical errors or undefined behaviors, coding rule verification must ensure adherence to a wide range of stylistic and structural guidelines. These rules can be highly project-specific or organization-specific, requiring a more flexible and customizable analysis approach. Furthermore, coding rules often involve complex inter-procedural and inter-module relationships that necessitate a more holistic view of the codebase. Our graph-based approach is particularly well suited to address these challenges, as it allows for efficient traversal of complex code relationships and flexible pattern matching for diverse coding rules.

In order to compare the current approach with existing tools (AdaControl [5], GNATcheck [6]), it is important to select rules that are present in both existing tools and show the same behavior.

Therefore, eight easy-to-analyze rules that require little or no deep exploration in the AST were selected to obtain the necessary information.

The following rules were selected:

- **Constructors:** Identify any declaration of a primitive function of a tagged type that has a controlling result and no controlling parameter. If a declaration is a completion of another declaration, then it is not reported.
- **Too_Many_Parents:** Identify any tagged type declaration, interface type declaration, single task declaration, or single protected declaration that has more than N parents, where N is a parameter of the rule. A parent here is either a (sub)type denoted by the subtype mark from the parent_subtype_indication (in the case of a derived type declaration), or any of the progenitors from the interface list (if any).
- **Abort_Statements:** Identify abort statements.
- **Abstract_Type_Declarations:** Identify all declarations of abstract types, including generic formal types. For an abstract private type, the full-type declaration is reported only if it is itself declared as abstract. Interface types are not reported.
- **Blocks:** Identify each block statement.
- **Renamings:** Identify renaming declarations.
- **Slices:** Identify all uses of array slicing.
- **Enumeration_Representation_Clauses:** Identify enumeration representation clauses.

These rules will provide us with an overview of coding rules to successfully conduct our benchmarks.

It should be noted that the selection of rules in this study is not comprehensive. Future research will incorporate more sophisticated

rules to extend the analysis. The rules currently selected cover common use cases that feature straightforward logic which may not be managed optimally using an Abstract Syntax Tree (AST). For instance, the rule **Blocks** require processing the entire AST to identify all block statements. In contrast, by employing a graph database, one can leverage labels (indices) to instantly retrieve results. A more intricate rule such as **Constructors** requires gathering multiple pieces of information from various locations within ASTs. It is important to highlight that there is no ambiguity concerning the queried information, as the data stored in the database represents a completely decorated AST, enriched with additional relationships as seen in section 3.4.

3.6. Selection of the graph database

This section details the selection process of the most suitable graph database for our research, outlining the critical criteria and evaluating various options.

Graph databases have gained prominence due to their ability to effectively store and manage complex relationships between data points. However, the abundance of graph database management systems (GDBMS) poses a challenge in selecting the most fitting one. To make an informed decision, we established a comprehensive evaluation framework based on mandatory and optional requirements, emphasizing the alignment of features and constraints.

Mandatory requirements, identified as GDBCs (Graph Database Constraints), were defined as fundamental criteria for effective database selection. These included: operating locally, supporting complex queries, being open-source, storing data persistently in the disk, adhering to a property graph structure, and enabling extensibility.

Optional requirements, termed OGDBCs (Optional Graph Database Constraints), incorporated additional considerations that could enhance the overall user experience and database capabilities. These included an easy-to-use query language, an easy-to-learn query language, a graphical user interface (GUI) for visual representation, comprehensive documentation, and readily available support and resources.

We evaluated various GDBMS against the established set of requirements, including DGraph [34], ArangoDB [35], OrientDB [36], Neo4j [37], Amazon Neptune [38], Microsoft Azure Cosmos DB [39], Aerospike [40], and JanusGraph [41]. Upon thorough analysis, only four databases met all mandatory requirements: ArangoDB, Neo4j, Aerospike, and JanusGraph.

Aerospike was deemed unsuitable due to its recent introduction at the start of our studies in 2020. JanusGraph, while satisfying all mandatory requirements, required a ”storage backend” to store data on the disk, which conflicted with our preference for a standalone, self-contained database.

Neo4j’s market leadership and our favorable assessment of its features and capabilities—including comprehensive documentation, a user-friendly query language, and extensibility—were decisive factors in this choice. In practical terms, any graph-oriented database supporting property graphs and equipped with a query language robust enough to handle complex queries would be adequate. For this study, we chose Neo4j, although other databases might also fulfill the requirements. Nevertheless, it is crucial to utilize a query language that can effectively express the semantics of the programming language under analysis.

Neo4j offers several features that are mandatory for our research:

- **Property Graph:** This feature allows the addition of properties to nodes and edges, which allows one to add information on nodes like the location in file, the type of node, etc.
- **Multiple Labels (Indices) on Nodes:** These labels facilitate targeted querying by enabling quick location and retrieval of nodes based on specific criteria.
- **Powerful Query Language:** Neo4j utilizes Cypher, a declarative graph query language that offers several advantages for static code analysis:
  - **Pattern Matching:** Cypher excels in expressing complex graph patterns, allowing us to efficiently query intricate code structures and relationships.
  - **Path Finding:** Cypher's path-finding capabilities enables us to traverse the graph efficiently.
  - **Aggregation and Sorting:** Cypher provides robust support for aggregation and sorting operations, allowing us to perform quantitative analyses on code structures. This is particularly useful for metric calculation such as the number of ancestors, for example.
  - **Subqueries and Complex Logic:** Cypher allows for the composition of complex queries using subqueries and conditional logic. This enables us to express sophisticated analysis rules that may involve multiple steps or conditional branching.
  - **Performance Optimization:** Cypher includes features like query profiling and execution plan visualization, which help in optimizing complex queries for large-scale code analysis tasks.
  - **Extensibility:** Neo4j allows for the creation of user-defined procedures in languages like Java, which can be called from Cypher queries. This extensibility can enable us to implement custom analysis algorithms when needed[3], while still leveraging Cypher's graph querying capabilities.

We would also like to underline that the practical advantages of Neo4j, and graph databases in general, are significantly pronounced because of their schemaless nature. This architecture allows for exceptional flexibility, facilitating easy extensibility and adaptation as project requirements evolve. The schemaless design not only supports rapid development iterations but also simplifies the process of integrating and adapting to changing data models without the need for extensive database redesigns. Thus, the schemaless approach of Neo4j offers a strategic advantage, enabling both the efficient management of current data and the seamless incorporation of new data types and relationships as research progresses. This flexibility is particularly beneficial in dynamic research environments where adaptation to emerging requirements is crucial.

3.6.1. Benefits of Semantic Relationships and Query Writing

Our graph-based representation of Ada code, with its semantic relationships, offers several unique benefits for static code analysis. By capturing the meaningful connections between code elements, such as proce-

[3]This feature is currently not used in our research

dure calls, variable references, and type dependencies, our approach enables more expressive and efficient querying compared to traditional AST-based approaches, we do not have to go back through (sub)ASTs several times to obtain the information we want during an analysis. The declarative nature, and ASCII-art syntax of the Cypher query language allows for a concise and intuitive formulation of patterns, making it easier to identify and extract relevant code information. However, it can be awkward or difficult to formulate some queries that require multiple steps [24].

One of the key contributions of our work is the adaptation and extension of the CPG concept to the Ada programming language. The CPG combines abstract syntax trees, control flow graphs, and program dependence graphs into a single, unified data structure.

Our implementation extends the CPG concept by incorporating Ada-specific and non-specific semantic relationships, such as package hierarchies, generic instantiations, tasking constructs, and links between a variable assignation and the variable declaration.

Moreover, the graph structure facilitates the traversal and exploration of code relationships, enabling precise pattern matching queries to extract information from the graph.

The extended CPG representation provides a robust foundation for advanced code analysis techniques, including data flow analysis. Unlike traditional compiler intermediate representations (IRs), which often require separate passes for different types of analysis, our CPG-based approach allows for seamless integration of control flow and data flow information. This integration enables more efficient and context-aware data flow analysis, as demonstrated by Yamaguchi et al. [2] in their work on vulnerability detection.

The query-based approach offers greater flexibility compared to traditional analysis frameworks. We can easily define custom patterns using Cypher queries, allowing for rapid prototyping of new analysis techniques without modifying the underlying analysis engine.

In summary, our graph-based representation, with its rich semantic relationships and query capabilities, provides a powerful and flexible foundation for static code analysis in Ada. It combines the strengths of CPGs with some extensions, offering a unique approach to analyzing Ada code that goes beyond traditional compiler-based techniques.

3.7. Selection and preparation of the dataset for the benchmark

To obtain a representative dataset, we used Alire [42], Ada's library manager. We downloaded all available libraries from the Alire repository. We then selected libraries that were compiled on our architecture (Linux x86_64) and contained at most Ada 2012 code; this restriction was necessary since AdaControl (and our approach) does not yet process Ada 2022 code.

Given that some libraries depend on another, local links were created to avoid cloning the same library multiple times. As a result, all projects compiled with dependencies were self-contained within a single code repository, facilitating reproducibility.

In total, 162 projects were used, containing 1 047 941 lines of code.

4. Implementation

4.1. Code Extraction and Pre-Processing

Our long-term objective is to develop a compiler-independent solution, thereby avoiding vendor lock-in and allowing users the freedom to select their preferred development stack. Consequently, we opted not to work directly with the GNAT frontend for source code

information extraction. Instead, we are capitalizing on our experience in developing static code analysis tools for Ada, which enables us to reduce the development time of the proof of concept for this research.

To this end, we started the implementation phase of this research by leveraging the existing codebase of AdaControl, a static analysis tool developed by Adalog. AdaControl is built upon the Ada Semantic Interface Specification (ASIS) [43], which is a standardized interface providing access to the syntactic and semantic content of Ada programs. ASIS facilitates the interaction with Ada's source code, which is crucial for static code analysis tools like AdaControl or GNATcheck.

ASIS is an interface between an Ada environment and tools that require information from this environment, such as code analyzers and formatters. It is designed to be independent of the underlying Ada environment implementations, promoting the portability of software engineering tools. While the ASIS standard officially supports versions of Ada up to the Ada 1995 standard, it is important to note that ASIS-for-GNAT, the implementation by AdaCore, has extended support to include Ada 2005 and Ada 2012 features. This extended implementation is used by a variety of tools for different purposes, including code monitoring, browsing, and testing. AdaControl relies on this extended version of ASIS-for-GNAT. However, this implementation has a number of weaknesses:

- Non-reentrant: ASIS-for-GNAT does not support concurrent tasks that concurrently explore the same code.
- Tree Swapping Issues: Each compilation unit is represented by an ASIS AST. However, in ASIS, the corresponding AST for each compilation unit encompasses the specifications of all dependent units, leading to the creation of large ASTs. Additionally, certain ASIS queries require loading multiple ASTs into memory. However, it can only load one AST at a time, resulting in timing issues when dealing with large codebases.
- End of Life support: AdaCore has discontinued the development and maintenance of ASIS-for-GNAT.

We were compelled to utilize ASIS as, at the commencement of our research, libadalang [44] was in its nascent stages and not sufficiently mature to handle the complexities required for our static analysis needs. Libadalang is a semantic engine for Ada that provides a high-level interface to analyze Ada sources. It is designed to be more efficient and flexible than ASIS, supporting newer Ada standards and offering a more modern API. However, at the time of our initial research, it was not yet fully developed to meet our specific requirements.

An alternative approach could have involved developing our own Ada parser; however, this would have required an excessive amount of work disproportionate to the scope of the current study.

While ASIS has been instrumental in this research as a proof of concept, there is an intention to transition to the newer library 'libadalang' for future work. Although libadalang does not respect the naming conventions defined in the Ada standard for naming and querying language elements, it now offers support for newer Ada standards, up to Ada 2022 (currently the latest version), which will allow for more comprehensive analysis of modern Ada code bases, and it is more efficient than ASIS.

4.2. *Graph Population and query development*

A significant modification to the AdaControl code base was the replacement of the rule verification system with a mechanism to populate a Neo4j graph database. This job is performed in a separate task to ensure that the main task remains focused on AST traversal. During this process, the main task fills a protected object with syntactic and semantic information, which is then read by the population task to generate JSON files containing Cypher commands. Figure 2 show how the program works.

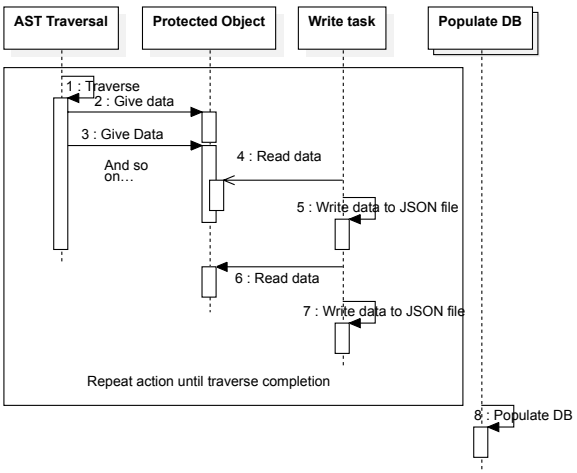


Figure 2: Sequence Diagram of how the program works

The generated commands are organized into files based on their function: '0_XXX' for database initialization, '1_XXX' for node creation, '2_XXX' for relationship creation, and '3_XXX' for database finalization. Initialization commands only creates custom labels and the index used for the database population, while finalization commands do this:

1. For each node, create an index property on the relationship to each child to create an order, based on the filename, line, and column.
2. Create the call graph.
3. Create ancestor relationships.
4. Remove the label used in DB population.
5. Remove constraints used in DB population.

Once the entire code has been explored, the files are read sequentially to populate the database (using an HTTP connection), except for files starting with "1" or "2", which are executed in parallel, with all the files starting with "1" being executed first, followed by all the files starting with "2". The figure 3 illustrate the previous explanation. We have adopted this approach because we have observed that by doing so, instead of real-time feeding, we achieve 100-fold improvement in the database population time.

The Neo4j Desktop GUI has been an essential tool during the development phase, providing immediate visual feedback on the graph structures created by the commands. This visual feedback is invaluable for ensuring that the commands produce the intended results.

The coding rules were developed to align with those of existing tools. Rules are written in Cypher query language. The following example shows the query related to "Too_Many_Parents" coding rule:

```
// Parameters of the query
:params { "minNbParents": 2 }
// The query
MATCH (typeDecl)<-[r:IS_PROGENITOR_OF|
    IS_ANCESTOR_OF]-(parent)
WITH typeDecl, count(r) as nbParents
WHERE nbParents >= $minNbParents
RETURN typeDecl, nbParents
  ORDER BY typeDecl.filename, typeDecl
    .line, typeDecl.column
```

Listing 2: Example of Cypher query with "Too Many Parent" coding rule

This Cypher query retrieves nodes that we call **typeDecl** which have at least two (set in

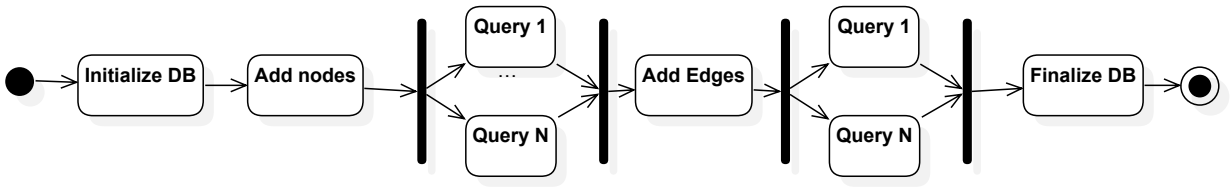


Figure 3: DB population diagram

parameters of the query) incoming relationships labeled with **IS_PROGENITOR_OF** or **IS_ANCESTOR_OF** from other nodes called **parent**. The final result set is ordered based on the filename, line, and column properties of each **typeDecl** node.

Many simple queries follow the same global structure query:

```
MATCH (e:<a-set-of-labels>)
RETURN e
  ORDER BY e.filename, e.line, e.
    column
```

Listing 3: Common structure of several simple Cypher query

Where **<a-set-of-labels>** is replaced by one or many labels. For example, for **Abort Statements** rule, we only control the "AN_ABORT_STATEMENT" label. And the same behavior is followed for the following rules: **Blocks**, **Slices**, **Enumeration Representation Clauses**. **Renamings** rule on the other hand, which also follows this structure, use multiple labels:

```
  MATCH (e:
    AN_OBJECT_RENAMING_DECLARATION|
    A_PACKAGE_RENAMING_DECLARATION|
    A_FUNCTION_RENAMING_DECLARATION|
    A_PROCEDURE_RENAMING_DECLARATION|
    AN_EXCEPTION_RENAMING_DECLARATION|
    A_GENERIC_PACKAGE_RENAMING_
    DECLARATION)
  RETURN e
   ORDER BY e.filename, e.line, e.
    column
```

Listing 4: "Renamings" coding rule in Cypher

As we can see, the code can be readily controlled. However, as shown in Listing 5, the query can be more complex as a function of the intrinsic complexity of the request.

```
// First, we match every function decl
    that return a specific type
MATCH (function:A_FUNCTION_DECLARATION
    |A_FUNCTION_RENAMING_DECLARATION|
    AN_EXPRESSION_FUNCTION_DECLARATION)
    <-[:IS_ENCLOSED_IN]-(:AN_IDENTIFIER
    )-[:CORRESPONDING_NAME_DEFINITION
    ]->()-[:IS_ENCLOSED_IN]->(typeDef)
    -[:
    CORRESPONDING_TYPE_DECLARATION_VIEW
    ]->(:
    A_TAGGED_RECORD_TYPE_DEFINITION|
    A_TAGGED_PRIVATE_TYPE_DEFINITION|
    A_TAGGED_INCOMPLETE_TYPE_
    DECLARATION|
    A_DERIVED_RECORD_EXTENSION_
    DEFINITION)

// Get the type of each parameter of
    the function, if exists
OPTIONAL MATCH (function)<-[:
    IS_ENCLOSED_IN]-(:
    A_PARAMETER_SPECIFICATION)<-[:
    IS_ENCLOSED_IN]-(:AN_IDENTIFIER)-[:
    CORRESPONDING_NAME_DEFINITION]->()
    -[:IS_ENCLOSED_IN]->(parmType:
    A_DECLARATION)

WITH collect(parmType) AS
    functionParams, function, typeDef
```

```
MATCH (function)
WHERE NOT apoc.coll.contains (
    functionParams, typeDef)

RETURN function
 ORDER BY function.filename, function.
    line, function.column
```

Listing 5: "Constructors" coding rule in Cypher

However, since Cypher provides a flexible way to query graph databases, it poses some challenges for expressing complex static code analysis queries [24]. Cypher adheres to a declarative, pattern matching paradigm rather than an imperative one. This can make it difficult to perform iterative, procedural operations common in static analysis. The lack of recursion and function definitions in Cypher limits its ability to concisely define and query recursive structures prevalent in abstract syntax trees. There are multiple solutions to overcoming these limitations, from custom (user defined) procedures from a custom plugin, integrated into Neo4J database, to a "simple" script that does tasks that cannot be done in a declarative language. But this requires verbose, multiple steps workarounds that can hinder understanding and maintenance of complex analysis queries. As such, while useful for simpler graph operations, Cypher's declarative nature restricts its expressiveness for intricate analysis tasks in the static code analysis domain.

In light of the limitations posed by Cypher's declarative nature, especially in handling complex static code analysis queries, we mainly have two potential solutions: a Domain-Specific Language (DSL) that transpile to a set of Cypher queries, as briefly explored by Urma and Mycroft [24], and the Gremlin language.

Gremlin is part of the Apache TinkerPop graph computing framework and supports both imperative and declarative querying styles. Unlike Cypher, Gremlin excels in expressing iterative and procedural operations, which are often essential for traversing and analyzing the recursive structures found in abstract syntax trees. Its ability to seamlessly handle recursion and define custom functions makes Gremlin particularly adept for intricate queries that require deep traversal or manipulation of graph data. This flexibility can significantly enhance the capability to perform detailed and complex static code analyses without the need for verbose workarounds, thereby simplifying the development and maintenance of analysis queries. Consequently, considering Gremlin as an alternative querying language could provide a more robust toolset for tackling the sophisticated demands of static code analysis in graph databases. However, this will mean using a GDBMS that supports this query language. In contrast, defining a DSL would provide an abstraction of the language actually used by the graph database, but requires a backend for each graph database query language. This could be solved by the implementation by GDBMS vendor of a standardised query language [45, 46].

5. Evaluation

This section presents the experimental results obtained using our approach and compares them with two existing static code analysis tools: AdaControl and GNATcheck. The evaluation metrics used are the time required to analyze the entire codebase for all coding rules.

Currently, the memory usage of our approach operates within the constraints of a typical personal computer and has not been identified as a bottleneck; hence, memory performance was not specifically evaluated. There are no significant memory-related issues, but

time efficiency remains a challenge.

Regarding the accuracy of error detection, our evaluation is based on a finite set of coding rules; it is not exhaustive. The aim was not to cover every possible scenario but to identify instances where certain coding rules might fail to detect specific errors.

5.1. Experimental setup

The benchmarking was performed on a computer with a Debian 12 operating system.

The computer specification:

- OS: Debian GNU/Linux 12 (bookworm) X86_64
- Host: MS-7E12 1.0
- Kernel: 6.1.0-17-amd64
- CPU: AMD Ryzen 9 7950X3D (32) 4.2 GHz
- GPU 1[4]: AMD ATI 19:00.0 Raphael
- GPU 2: AMD ATI Radeon RX 7900 XTX
- Memory: 64 GB
- Storage: Crucial P5 Plus 1 TB SSD using M.2 PCIe Gen 4 connection, up to 6600 MB/s in read operations and 5000 MB/s in write operations

Regarding the software, the setup is:

- GNAT Pro 24.0w (20230301-122): Ada compiler.
- GNATcheck 24.0w (20230301-122): static analysis tools. This version is based on libadalang.
- AdaControl 1.23b4: static analysis tools. This version is based on ASIS.
- GNAT Pro 21lts: for ASIS support.
- Deno JavaScript runtime 1.38.3 with v8 12.0.267.1 and typescript 5.2.2: for benchmark scripts.

The Neo4J setup operates on Neo4J Desktop version 1.5.9.106. The database utilizes engine version 5.12.0, complemented by the APOC plugin.

The benchmark repository is available on GitHub [47].

5.2. Evaluation metrics and Results

The benchmark was conducted on projects of varying sizes, from hundreds of lines of code to hundreds of thousands of lines. However, we noticed that AdaControl exhibited poor performance on large files due to ASIS tree swapping issues. For example, libadalang's largest file contains more than 100 000 lines of codes, and it has severely impacted AdaControl runtimes, with an overhead that exceeds 24 hours, for a project with more than 600 000 lines of codes. Hence, libadalang has been omitted.

It is important to clarify that AdaControl runs in monothreaded mode, while GNATcheck is run in monothreaded and multithreaded, 32 cores. Our approach consists of two main parts:

1. **Initialization:** This phase, which includes parsing the code structure and populating the database, primarily runs on a single core. However, the actual database population utilizes 32 cores to speed up the process. The initialization time is measured in the overhead, defined below.

[4]GPUs are mentioned purely for the sake of transparency. They are not used by any of the static code analysis tools in this study.

2. **Querying:** The most critical part of the analysis, which involves executing queries on the populated database, uses mainly one core [48].

To measure the performance of the static code analysis tools, we utilized the following evaluation metrics:

- **Overhead**: We measured the time required by each software to initialize and prepare for analysis. For AdaControl and GNATcheck, we created an empty rule that did nothing, while for our approach, we measured the time required to traverse all the code and populate the graph database.
- **Individual Rule Execution Time**: We executed each coding rule individually to obtain the execution time of each rule. This allowed us to analyze the performance of each rule in isolation.
- **One Run Execution Time**: In addition to evaluating the execution time of each rule individually, we also performed a "one run" analysis where all the coding rules were executed in a single run. This scenario represents a more realistic use case and allows us to assess the overall performance of the tools.

We then measured the execution time of each tool for rule by rule analysis and rules in one batch. The results are shown in the table 1.

As shown in Table 1, our approach demonstrates a substantially higher initialization overhead (2 h 15 min 17 s) compared to AdaControl (4 min 22 s) and GNATcheck (3 min 35 s) for single-threaded and (5 min 18 s) for multithreaded configurations. This considerable difference in overhead times is a notable aspect of our current implementation that warrants further discussion and improvement, and is detailed in section 6.7.

Regarding the rule by rule analysis, the results show that our approach consistently outperforms AdaControl and GNATcheck for single rule analysis. Even if GNATcheck's multithreaded results are not astonishing, we think that it should give much better results on very large projects, totaling numerous files. A future benchmark on a huge project (+1M Lines of Code (LoC)) will help to determine whether the problem of slow performance of GNATcheck in multithreaded mode is due to the loading of several small projects (as is currently the case in the benchmark) or not.

The results where rules are in one batch show that our approach is significantly faster than AdaControl (326 times faster) and GNATcheck (from 57 to 148 times faster) for one run that controls all rules. However, we see that AdaControl takes more time than GNATcheck, which is in contrast to the rule-by-rule tests. This could be explained due to tree swapping issues on ASIS lib used by AdaControl, and maybe a better management of rule control in GNATcheck. Regarding our approach, the time is identical to the rule-by-rule version, because it is the same analysis, and we can separate out the overhead, unlike other tools where it is calculated empirically by computing the average of 10 executions of these tools on the whole code with a custom rule that do nothing.

6. Discussion

6.1. Analysis of Results

The results from the research confirmed the initial hypothesis that employing a graph database can significantly enhance the performance of static code analysis tools. This also gives similar results to Rodriguez-Prieto's [25]

Rule	**Our approach**	**AdaControl**	**GNATcheck (monothread)**	**GNATcheck (multithread 32 cores)**
Constructors	935 ms	1 min 5 s 700 ms	1 min 54 s 513 ms	2 min 5 s 41 ms
Too many parents	200 ms	1 min 3 s 100 ms	1 min 45 s 978 ms	2 min 2 s 939 ms
Abort statements	100 ms	1 min 4 s 300 ms	46 s 434 ms	1 min 33 s 557 ms
Abstract type decl.	1 s 452 ms	1 min 5 s 100 ms	1 min 25 s 575 ms	1 min 52 s 851 ms
Blocks	230 ms	1 min 3 s 900 ms	46 s 507 ms	1 min 33 s 563 ms
Renamings	440 ms	1 min 3 s 900 ms	47 s 346 ms	1 min 33 s 125 ms
Slices	140 ms	1 min 12 s 600 ms	1 min 58 s 598 ms	2 min 4 s 714 ms
Enum. repr. clauses	410 ms	1 min 3 s 100 ms	46 s 683 ms	1 min 33 s 486 ms
Overhead lower is better	**2 h 15 min 17 s**	**4 min 22 s**	**3 min 35 s**	**5 min 18 s**
Overhead comparison	**slowest**	**30.98 × faster**	**37.75 × faster**	**25.52 × faster**
Total time analyzing rule one by one lower is better	**3 s 907 ms**	**9 min 49 s**	**10 min 11 s 638 ms**	**14 min 19 s 280 ms**
Time comparison rule one by one	**fastest**	**150.75 × slower**	**156.55 × slower**	**219.93 × slower**
Total time analyzing rule in one batch lower is better	**3 s 907 ms**	**21 min 13 s 864 ms**	**9 min 40 s**	**3 min 46 s**
Time comparison rule in one batch	**fastest**	**326 × slower**	**148.45 × slower**	**57.84 × slower**

Table 1: Performance of static analysis tools

research with Java. The use of a graph database allowed for a comprehensive analysis of a substantial codebase in a fraction of the time taken by traditional tools. However, an important revelation from the study was the potential for performance trade-offs on smaller codebases, where the overhead of database population may outweigh the analysis time benefits. This finding suggests that the proposed approach is more suitable for larger codebases or scenarios where the codebase is analyzed multiple times, allowing the initial cost of the database population to be amortized over analyzes.

6.2. Limitations and Approximations in Static Analysis

It is important to acknowledge that our proposed method, like all static analysis techniques, is subject to fundamental limitations due to the undecidability of certain program properties in the general case. Our current implementation relies on ASIS, which means that we inherit certain limitations that affect the scope and accuracy of our static analysis approach.

A significant limitation of ASIS is its inability to fully process certain dynamic aspects of Ada, such as dispatching calls. This means that for scenarios involving dynamic dispatch,

our analysis cannot determine the actual target of the call at compile time. Consequently, our approach does not currently address these scenarios, as they are inherently dynamic and therefore beyond the scope of static analysis.

Other potential areas of concern in static analysis, such as precise aliasing analysis, complex loop behaviors, and concurrency issues, are similarly constrained by the capabilities of ASIS. Our graph-based representation, while offering benefits in terms of query efficiency and scalability, is fundamentally limited by the information available through ASIS.

It is crucial to note that these limitations are not inherent to our graph-based approach, but rather stem from our current reliance on ASIS. By migrating to libadalang in a future work, it might provide more comprehensive information, potentially allowing our approach to handle a wider range of Ada language features and static analysis scenarios.

Despite these limitations, our approach still offers significant benefits for the static analysis tasks it can perform, particularly in terms of scalability and efficiency for large Ada codebases.

6.3. Benefits and Limitations of Graph Database Integration

The principal benefit of integrating a graph database for static code analysis is evident in the performance metrics. This integration allows for a faster analysis post-database population and can be particularly effective for repeated analyzes common in modern software development practices. However, the approach does introduce a performance trade-off for smaller codebases and also necessitates additional setup, which could be viewed as cumbersome for teams without experience in graph databases.

Another limitation is the need for further experimentation, particularly with complex queries, to comprehensively understand the performance benefits across various analysis types.

6.4. Practical Implications and Potential Applications

The implications for software developers and the industry are significant, with the potential for increased code quality and reduced development time. The improved analysis speed could lead to more frequent code checks, thereby enhancing code robustness and developer productivity. P. van de Laar [49] has already started down this path, with Renaissance-Ada [50].

However, before this approach can be seamlessly integrated into existing development workflows, the transition from ASIS to libadalang is necessary to ensure the support of the latest Ada standards. The user-friendliness of the approach will depend largely on the ease of installing and setting up the necessary graph database infrastructure.

6.5. Generalizability and Future Directions

The proposed approach of using graph databases and pattern matching for static code analysis, while demonstrated on the Ada programming language, can be adapted and extended to other programming languages and software engineering tasks. The key insights and contributions of this research that can be leveraged by other researchers include:

- **Graph-based code representation**: Encoding source code and its semantic relationships in a graph database allows for a more expressive and flexible representation compared to traditional ASTs. This enables the formulation of complex

static analysis rules that can capture intricate dependencies and patterns in the code [2, 3].

- **Pattern matching queries**: Expressing static analysis rules as pattern matching queries, such as those in the Cypher language, provides an intuitive and declarative way to specify code patterns of interest. This approach can be adapted to other query languages and graph databases, enabling researchers to leverage the power of pattern matching in their static analysis tools [2].

- **Scalability and performance**: The use of graph databases and optimized query execution can significantly improve the scalability and performance of static analysis tools, as demonstrated in this study. Researchers can build upon these findings to develop efficient and scalable tools for analyzing large codebases in various programming languages [1].

However, it is crucial for researchers to carefully consider the relationships they choose to include in the graph database. Adding too many relationships can lead to a significant increase in the size and creation time of the database, potentially impacting the overall performance of the static analysis tool. Researchers should prioritize the inclusion of relationships that are most relevant to the specific static analysis tasks they are targeting, striking a balance between the expressiveness of the code representation and the efficiency of the analysis process.

6.6. Comparison with Other Static Analysis Tools

While this research focuses on the comparison between traditional AST-based static analysis tools and the proposed graph-based approach, it is worth discussing other static analysis tools that employ graph-like structures, such as Semmle and CodeQL.

Semmle and CodeQL are query-based static analysis tools that use a custom query language, QL, to analyze code [51]. They represent code as a relational data model, which is conceptually similar to a graph structure. However, their underlying storage and querying mechanisms differ from those of native graph databases like Neo4j.

In contrast to the proposed approach, which directly stores and queries the code representation in a graph database, Semmle and CodeQL use a relational database as an intermediate layer. The code is first transformed into a relational representation, and then queries are executed on this relational model. While this approach enables powerful querying capabilities, it may introduce additional overhead in terms of storage and query performance compared to using a native graph database.

Moreover, the query languages used by Semmle and CodeQL, although expressive, are specific to their respective tools and require learning a new syntax. In contrast, the proposed approach leverages the standard Cypher query language, which is widely used in the graph database community and provides a more intuitive and declarative way of expressing graph patterns.

The focus of this research is to investigate the benefits of using a native graph database, Neo4j, for representing and querying code structures, and to compare its performance and scalability with traditional AST-based tools. While Semmle and CodeQL offer powerful static analysis capabilities, their underlying storage and querying mechanisms differ from the graph-based approach proposed in this study.

Future research could explore a more detailed comparison between the proposed

graph-based approach and tools like Semmle and CodeQL, analyzing factors such as query expressiveness, performance, and ease of use. Such a comparison would provide valuable insights into the trade-offs, and the benefits of different static analysis approach that leverage graph-like structures. However this will potentially require a lot of initial work, since CodeQL does not support the Ada language.

6.7. Analysis of Overhead and Future Optimizations

The considerable overhead in our approach, as highlighted in the results, is primarily due to our current strategy of extracting and storing information about all nodes in the AST, regardless of their relevance to the specific coding rules being checked. In contrast, AdaControl's more efficient overhead can be attributed to its selective extraction of information during the tree traversal, focusing only on the nodes required by the user-specified coding rules.

This difference in approach explains the stark contrast in initialization times. While our method provides superior performance during the actual analysis phase, the initial high overhead is a significant drawback.

To address this limitation, a key area of future work will be the development of a more intelligent and selective graph construction process, the migration to libadalang. We plan to implement a system that constructs the graph based on the specific coding rules requested by the user. This targeted approach in addition of migrating to libadalang, should significantly reduce the initialization overhead, potentially bringing it more in line with, or even improving upon, the overhead times of existing tools.

By combining this selective graph construction with our already superior analysis times, we aim to create a more balanced and efficient static code analysis tool that outperforms existing solutions in both initialization and analysis phases.

6.8. Future Research Directions

An important aspect of our future work in the short term will involve a comprehensive study of how our approach scales with increasing LoC. While the current research provides valuable insights into the performance of our method compared to existing tools, a detailed analysis of scalability across a wider range of codebase sizes is planned for later this year. This forthcoming benchmark will help us better understand the relationship between codebase size and analysis time, allowing us to further optimize our approach for large-scale projects.

The future research will involve including only the information needed to analyze the rules desired by the user in the database, in order to reduce overheads to a minimum. We also plan to implement an incremental update, in order to not have to fully fill the database with the whole project, when one file has been updated, so that we can only update nodes related to the user modification. We will also have to move from ASIS to libadalang for modern Ada language support, and to reduce the overhead. Likewise, we also have to conduct further experiments with complex queries. The integration of the approach into CI/CD environments and IDEs is planned, which could open up new areas of inquiry.

Advancements in graph databases and static code analysis tools could further enhance the applicability of this approach, potentially offering faster analyzes and enabling a broader spectrum of analysis types.

In conclusion, while the research has demonstrated a significant potential for the use of graph databases in static code analysis, it

has also highlighted the need for further investigation, particularly in terms of complex queries and codebase diversity. The integration with modern development practices and tools remains an area ripe for development, promising to usher in a new era of code analysis, efficiency and effectiveness.

7. Conclusion

In conclusion, this research has made a meaningful contribution to advance the field of static code analysis, notably in Ada. Our work has demonstrated the potential for graph databases to enhance the scalability and performance of analysis techniques for checking coding rules. Through rigorous experimental evaluation, our approach exhibited dramatically reduced analysis times (if the overhead is not taken into account) compared to traditional tools when processing large codebases. Moving to libadalang could be a good way in order to improve the overhead.

The utilization of a graph database architecture for static code analysis, as evidenced by our results, confers a notable advantage for software projects of substantial size and complexity where analysis efficiency can directly impact productivity and code quality. We have shown the initial costs of populating the graph database to be justified given the subsequent improvements in analysis speeds for more massive, complex and frequently inspected codebases.

However, we also acknowledge certain limitations and trade-offs associated with our method. For codebases of restricted scope, the overhead of establishing and populating the graph structure may not outweigh the performance benefits realized through analysis. We are convinced that transitioning from ASIS to libadalang in future work will significantly reduce the overhead, which will facilitate broader and faster applicability, as well as ensure compatibility with evolving Ada standards.

The practical implications of this work are apparent. By incorporating graph databases into static analysis workflows, development teams can leverage a powerful tool to enhance their code evaluation practices, culminating in higher-quality outputs and more efficient development cycles. Our approach is well positioned for integration within existing processes, with considerations for usability and judicious resource consumption warranting attention.

Several prospects exist to further strengthen this research domain. More comprehensive testing with sophisticated queries and across divergent programming languages presents an opportunity for continued betterment. Integration with additional software engineering tools and platforms, particularly within CI/CD pipelines and IDE environments, is an exciting prospect that could redefine present practices.

Fundamentally, this research has established a foundation for transformative change within static code analysis. As software systems increase in scale and intricacy over time, the necessity for scalable and efficient analysis techniques will become ever more imperative. Our work has taken a significant step toward satisfying this need and sets the stage for advances that will progressively redefine possibilities within software engineering.

More resources about this research can be found on our dedicated linktree [52].

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the authors employed ChatALL[5], a concurrent chat

[5]ChatALL has been used to encompass the following LLMs: Assistant of Poe.com, Bing Chat, Claude.ai,

assistance tool, primarily for the purpose of refining the writing style. The tool was utilized solely for enhancing the coherence, formality, and academic tone of the manuscript. After employing ChatALL, the authors thoroughly reviewed and edited the content as needed, ensuring accuracy and clarity in conveying the research findings. The authors take full responsibility for the content of the publication, recognizing that while the tool contributed to stylistic improvements, ultimate accountability rests with the human authors for the substantive and scholarly aspects of the work.

References

[1] M. Harman, P. O'Hearn, From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis, in: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2018. doi:10.1109/scam.2018.00009.

[2] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and Discovering Vulnerabilities with Code Property Graphs (May 2014). doi:10.1109/sp.2014.44.

[3] R. Ramler, G. Buchgeher, C. Klammer, M. Pfeiffer, C. Salomon, H. Thaller, L. Linsbauer, Benefits and Drawbacks of Representing and Analyzing Source Code and Software Engineering Artifacts with Graph Databases (2018) 125–148doi:10.1007/978-3-030-05767-1_9.

[4] Q. Dauprat, P. Dorbec, G. Richard, J.-P. Rosen, Use of graph databases for static code analysis, Ada User Journal 43 (3) (2022) 155–159.
URL https://adalog.fr/aeic-2022/dauprat2022.pdf

[5] Adalog, AdaControl.
URL https://www.adalog.fr/en/adacontrol.html

[6] AdaCore, GNATcheck.
URL https://www.adacore.com/static-analysis/gnatcheck

[7] B. Chess, J. West, Secure Programming with Static Analysis (Addison-Wesley Software Security Series), Addison-Wesley Professional, 2007.

[8] G. Fan, Practical static code analysis: challenges, methods, and solutions, Ph.D. thesis, The Hong Kong University of Science and Technology Library. doi:10.14711/thesis-991012818569403412.

[9] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, D. Engler, A few billion lines of code later, Communications of the ACM 53 (2) (2010) 66–75. doi:10.1145/1646353.1646374.

[10] S. C. Johnson, Lint, a C program checker, Bell Telephone Laboratories Murray Hill, 1977.

[11] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '77, POPL '77, ACM Press, 1977. doi:10.1145/512950.512973.

[12] O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, E. Goubault, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, M. Turin, Space Software Validation using Abstract Interpretation 1 (May 2009).

[13] M. Beller, G. Gousios, A. Zaidman, TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), IEEE, 2017. doi:10.1109/msr.2017.24.

[14] J. Wang, M. Huang, Y. Nie, J. Li, Static Analysis of Source Code Vulnerability Using Machine Learning Techniques: A Survey, in: 2021 4th International Conference on Artificial Intelligence and Big Data (ICAIBD), IEEE, 2021. doi:10.1109/icaibd51990.2021.9459075.

[15] F. E. Allen, Control flow analysis, Acm Sigplan Notices 5 (7) (1970) 1–19. doi:10.1145/390013.808479.

[16] S. Heckman, L. Williams, On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques, in: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, ACM, 2008. doi:10.1145/1414004.1414013.

[17] M. Jemli, J.-P. Rosen, A methodology for avoiding known compiler problems using static analysis, Ada. Letters. 30 (3) (2010) 23–30. doi:10.1145/1879097.1879073.

[18] U. P. Khedker, Data Flow Analysis, 1st Edition,

GPT-4, llama-2-70b, Pi.ai, vicuna-33b, You.com

Taylor & Francis Group, Baton Rouge, 2009.

[19] J.-C. V.-D.-H. Jean-Pierre Rosen, Using Ada's Visibility Rules and Static Analysis to Enforce Segregation of Safety Critical Components, Ada User Journal 37 (3) (2016) 146–149.

[20] A. Ponomarev, H. S. Nalamwar, R. Jaiswal, Source Code Analysis: Current and Future Trends Challenges 685 (2016) 877–880. doi:10.4028/www.scientific.net/kem.685.877.

[21] R. Angles, C. Gutierrez, Survey of graph database models, Acm Computing Surveys 40 (1) (2008) 1–39. doi:10.1145/1322432.1322433.

[22] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, D. Wilkins, A comparison of a graph database and a relational database, in: Proceedings of the 48th Annual Southeast Regional Conference, ACM, 2010. doi:10.1145/1900008.1900067.

[23] I. Robinson, Graph databases (2015).

[24] R.-G. Urma, A. Mycroft, Source-code queries with graph databases-with application to programming language usage and evolution, Science of Computer Programming 97 (2013) 127–134, special Issue on New Ideas and Emerging Results in Understanding Software. doi:10.1016/j.scico.2013.11.010.

[25] O. Rodriguez-Prieto, A. Mycroft, F. Ortin, An Efficient and Scalable Platform for Java Source Code Analysis Using Overlaid Graph Representations, IEEE Access 8 (2020) 72239–72260. doi:10.1109/access.2020.2987631.

[26] S. A. Atiiq, C. Gehrmann, K. Dahlén, K. Khalil, From Generalist to Specialist: Exploring CWE-Specific Vulnerability Detection (2024). arXiv:2408.02329.
URL https://arxiv.org/abs/2408.02329

[27] K. Borowski, B. Balis, T. Orzechowski, Semantic Code Graph—An Information Model to Facilitate Software Comprehension, IEEE Access 12 (2024) 27279–27310. doi:10.1109/access.2024.3351845.
URL http://dx.doi.org/10.1109/ACCESS.2024.3351845

[28] X. Chen, J. M. Atlee, Variability-aware Neo4j for Analyzing a Graphical Model of a Software Product Line, 2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS) (2023) 307–318.
URL https://api.semanticscholar.org/CorpusID:266197724

[29] R. Diestel, Graph Theory, Springer Berlin Heidelberg, 2017. doi:10.1007/978-3-662-53622-3.

[30] F. E. Allen, J. Cocke, A program data flow analysis procedure, Communications of the ACM 19 (3) (1976) 137. doi:10.1145/360018.360025.

[31] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd Edition, Pearson Addison-Wesley, Boston, 2007, rev. ed. of: Compilers, principles, techniques, and tools / Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. 1986.

[32] J. Ferrante, K. J. Ottenstein, J. D. Warren, The program dependence graph and its use in optimization, Acm Transactions On Programming Languages and Systems 9 (3) (1987) 319–349. doi:10.1145/24039.24041.

[33] A. Podgurski, L. A. Clarke, A formal model of program dependences and its implications for software testing, debugging, and maintenance, IEEE Transactions on Software Engineering 16 (9) (1990) 965–979. doi:10.1109/32.58784.

[34] D. Technology, Dgraph — GraphQL Cloud Platform, Distributed Graph Engine.
URL https://dgraph.io

[35] ArangoDB Inc., Ultimate Scalable Graph Database: ArangoDB for Real-World Use Cases.
URL https://arangodb.com

[36] OrientDB LTD, OrientDB.
URL https://orientdb.org/

[37] N. Inc., Neo4j Graph Database: The Fastest Path to Graph Success.
URL https://neo4j.com

[38] Amazon, Amazon Neptune.
URL https://aws.amazon.com/fr/neptune/

[39] Microsoft, Microsoft Azure Cosmos DB.
URL https://azure.microsoft.com/services/cosmos-db

[40] A. Inc., Aerospike: Multi-model NoSQL and graph database.
URL https://aerospike.com/

[41] JanusGraph, JanusGraph: Distributed, open source, massively scalable graph database.
URL https://janusgraph.org/

[42] Alire, Alire.
URL https://alire.ada.dev/

[43] S. Rybin, A. Strohmeier, V. Fofanov, A. Kuchumov, ASIS-for-GNAT: A Report of Practical Experiences (2000) 125–137doi:10.1007/10722060_13.

[44] AdaCore, libadalang GitHub repository.

URL https://github.com/AdaCore/libadalang

[45] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, F. Murlak, S. Plantikow, P. Selmer, H. Voigt, O. van Rest, D. Vrgoč, M. Wu, F. Zemke, Graph Pattern Matching in GQL and SQL/PGQ (Dec. 2021). doi:10.1145/3514221.3526057.

[46] Organization for Standardization, GQL: ISO/IEC 39075:2024.
URL https://www.iso.org/standard/76120.html

[47] Adalog, Ada static code analysis tools benchmark.
URL https://github.com/Adalog-fr/ada-static-code-analysis-tools-benchmark

[48] Neo4j Inc., Neo4j documentation: Runtime concepts.
URL https://neo4j.com/docs/cypher-manual/current/planning-and-tuning/runtimes/concepts/#runtimes-parallel-runtime

[49] A. M. Piërre van de Laar, Renaissance-Ada: Tools for Analysis and Transformation of Ada Code, Ada User Journal 43 (3) (2022) 165–170.

[50] TNO, Renaissance-Ada: Tooling for analysis and manipulation of Ada software.
URL https://github.com/TNO/Renaissance-Ada

[51] P. Avgustinov, O. de Moor, M. P. Jones, M. Schäfer, QL: Object-oriented Queries on Relational Data, in: S. Krishnamurthi, B. S. Lerner (Eds.), 30th European Conference on Object-Oriented Programming (ECOOP 2016), Vol. 56 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2016, pp. 41–66. doi:10.4230/LIPIcs.ECOOP.2016.2.
URL https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2016.2

[52] Adalog, More ressources about this research.
URL https://linktr.ee/aeic2024_dauprat