GAST: A Generic AST Representation for Language-Independent Source Code Analysis

Jason Leiton-Jimenez[1], Luis Barboza-Artavia[2], Antonio Gonzalez-Torres[3], Pablo Brenes-Jimenez[4], Steven Pacheco-Portuguez[5], Jose Navas-Su[6], Marco Hernández-Vasquez[7], Jennier Solano-Cordero[8], Franklin Hernandez-Castro[9], Ignacio Trejos-Zelaya[10] and Armando Arce-Orozco[11]

***Abstract*—Organizations use various programming languages to develop their systems. These aim to take advantage of the most appropriate features of each language for a given domain and require programmers to command different languages and also to face the growing complexity of software development and maintenance. So, they need tools to help them analyze programs to identify relationships between their internal elements, uncover patterns, and calculate quality metrics. However, most tools have limited support for parsing multiple programming languages and high acquisition costs. Therefore, there is a need for new methods to analyze code written in multiple programming languages. This article describes the design of a method to automatically transform the syntax of various programming languages into a universal language with a generic syntax. The function of the generic language is to encapsulate the specificities of each specific language, so that the analysis of programs is facilitated in a single programming syntax and not in multiple syntaxes. The advantage of this approach is that only one analysis engine is required, not multiple code analyzers, to study the programs.**

***Keywords* - Code transformation, Generic Abstract Syntax Tree, Generic Language, Code Analysis.**

[1] J. Leiton-Jimenez works at the Department of Computer Engineering at the Costa Rica Institute of Technology (e-mail: jleiton@tec.ac.cr). https://orcid.org/0000-0002-6271-6595.

[2] L. Barboza-Artavia works at the Department of Computer Engineering at the Costa Rica Institute of Technology (e-mail: labarboza@tec.ac.cr). https://orcid.org/0009-0000-7524-8068.

[3] A. Gonzalez-Torres works at the Department of Computer Engineering at the Costa Rica Institute of Technology (e-mail: antonio.gonzalez@tec.ac.cr). https://orcid.org/0000-0001-5427-0637.

[4] P. Brenes-Jimenez is a masters student at the School of Computing at the Costa Rica Institute of Technology (e-mail: pablobrenes@estudiantec.cr). https://orcid.org/0000-0001-8394-3853.

[5] S. Pacheco-Portuguez is a masters student at the School of Computing at the Costa Rica Institute of Technology (e-mail: stpacheco@ic-itcr.ac.cr). https://orcid.org/0000-0001-7505-1644.

[6] J. Navas-Su works at the School of Computing at the Costa Rica Institute of Technology (e-mail: jnavas@tec.ac.cr). https://orcid.org/0000-0003-3431-0122.

[7] M. Hernandez-Vasquez works at the Department of Computer Engineering at the Costa Rica Institute of Technology (e-mail: marco.hernandez@tec.ac.cr). https://orcid.org/0000-0002-9432-721X.

[8] J. Solano-Cordero works at the Department of Computer Engineering at the Costa Rica Institute of Technology (e-mail: jensolano@tec.ac.cr). https://orcid.org/0000-0002-0983-6512.

[9] F. Hernandez-Castro works at the School of Industrial Design at the Costa Rica Institute of Technology (e-mail: franhernandez@tec.ac.cr). https://orcid.org/0000-0003-3589-4588.

[10] I. Trejos-Zelaya works at the School of Computing at the Costa Rica Institute of Technology (e-mail: itrejos@tec.ac.cr). https://orcid.org/0000-0003-4361-8444.

[11] A. Arce-Orozco works at the School of Computing at the Costa Rica Institute of Technology (e-mail: arce@tec.ac.cr). https://orcid.org/0000-0001-5005-5745.

Manuscript Received: April 19, 2023.
Revised: May 19, 2023.
Accepted: August 25, 2023.
DOI: https://doi.org/10.29019/enfoqueute.957

***Resumen*—Las organizaciones usan varios lenguajes de programación para desarrollar sus sistemas. Estas utilizan las características mas apropiadas de cada lenguaje para un dominio determinado. Por su parte los programadores deben tener dominio de diferentes lenguajes para hacer frente a la creciente complejidad del desarrollo y mantenimiento del software. Así que necesitan herramientas que les ayuden a realizar esas tareas. Esas herramientas deben ser capaces de analizar los programas para identificar las relaciones entre sus elementos internos, ayudar a descubrir patrones y calcular métricas de calidad. Sin embargo, la mayoría tienen soporte limitado para analizar diversos lenguajes de programacion y altos costos de adquisición. Por lo que existe la necesidad de contar con nuevos métodos para analizar el código escrito en múltiples lenguajes de programación. Este artículo describe el diseño de un método para transformar automáticamente la sintaxis de varios lenguajes de programación en un lenguaje universal con una sintaxis genérica. La función del lenguaje genérico es encapsular las especificidades de cada lenguaje concreto, de manera que se facilite el análisis de programas en una sola sintaxis de programación y no en múltiples sintaxis. La ventaja de este enfoque es que solo se requiere un motor de análisis, no varios analizadores de código, para estudiar los programas.**

***Palabras Clave* - Transformación de código, Árbol de sintáxis abstracta genérica, lenguaje genérico, análisis de código.**

I. Introduction

Due to the ever-evolving nature of software engineering and the continuous emergence of new languages, dialects, and language versions, the precise number of programming languages in current use remains uncertain. According to a study by Nanz and Furia, which examined $7,087$ programs addressing 745 distinct issues, the most popular programming languages were found to be C, Java, C#, Python, Go, Haskell, F#, and Ruby [1].

In the realm of modern software application development, artifacts originating from a variety of programming languages are often used, particularly in large-scale projects [2]. The objective is to take advantage of specific aspects of each language to create more comprehensive, efficient, and effective systems. However, the complexity of programming tasks and the demand for engineers proficient in multiple programming languages pose challenges that make development more intricate. The intricate nature of programming lies in the unique syntax of each language.

In order to aid development and maintenance activities, development teams heavily rely on the utilization of tools. Assessing the structure and interconnections among system components becomes challenging, especially when programmers need to evaluate programs written in different programming languages. Various techniques, such as examining source code for patterns and interdependencies, computing quality metrics (e.g., complexity, cohesion, direct and indirect coupling and logical coupling), as well as identifying clones and defects, are employed [3].

In this context, it is crucial to recognize that software quality is a fundamental attribute that distinguishes software and companies. The ISO / IEC 9126 standard, along with its successor, the ISO/IEC 25000 series, defines a multidimensional model for evaluating software quality based on factors such as functionality, reliability, maintainability, efficiency, usability, and portability. Additionally, developers often rely on tools to facilitate development and maintenance processes.

Methods for source code analysis in various programming languages, which adhere to different paradigms and possess distinct syntaxes, typically necessitate the development of unique analyzers for each language grammar. However, given the vast array of existing languages and the continuous emergence of new ones, creating a dedicated analyzer for every language is impractical.

An alternative approach is to implement a single analyzer that operates on a generalized representation of languages. Such an analyzer would collect information for metric calculation, internal software analysis, and investigation of interconnections among system components. This approach can contribute to cost reduction and minimize the need for language-specific analyzers.

To achieve this, all syntactic aspects of the represented languages must be considered and the representation itself should be scalable to accommodate the inclusion of additional languages. Furthermore, it should be able to adapt to the evolving characteristics of languages. The aim of this project is to develop a technique to automatically convert the syntax of a specific language into a generic syntax, capturing the unique traits of each language to enable software project analysis.

The methods for translating a language into a generic format that are currently available are limited in number and replete with drawbacks. These methods require separate tools for each language, such as SonarQube [4] and Moose [5]), to analyze the source code.

This article presents the findings of the implementation of a Generic Abstract Syntax Tree (GAST) that possesses a unified structure across multiple programming languages. The efficacy of this method is demonstrated through two experiments, which showcase the successful transformation of diverse languages into the GAST and the ability to conduct various types of structural analysis on it.

The subsequent sections of the article are structured as follows. Section II provides an overview of related works related to techniques for the transformation of source code. Section III outlines the design and structure of the GAST, together with the validation method used for language-specific transformations. Section IV delves into the results and analyses derived from the experiments conducted, which substantiate the equivalence between specific languages and the GAST. Finally, sections V and VI present the conclusions drawn from the study and outline potential avenues for future research.

II. Related Work

The program source code is often translated from one language to another using transpilers. These tools enable code to be written once and then translated into multiple target languages, allowing translated scripts to be executed across various platforms [6]. Transpilers commonly employ a syntax processing module, linear mappings, and code generation as integral components [7].

Various strategies are employed by transpilers, including machine learning techniques, translation rules, and the use of Abstract Syntax Trees (AST). The general principle underlying AST techniques involves creating an AST representation of the source code units and then mapping its components into an AST representation of the target language.

Semantic Program Trees (PST)[1] transpilers follow the following procedures [8] to convert the source code of the program from one language to another:

1) Analyze the source code of the original program to determine its PST.
2) Collect the libraries and dependencies utilized by the original program.
3) Create a second PST with appropriate references for the destination program.
4) Utilize the second PST and the grammar of the target language to generate the source code for the final program.

Kijin et al. propose a cross-platform strategy based on translation rules [9]. This approach uses linear mappings, transformations, and translation rules to establish equivalences across syntactic elements, with the aim of automating software translation between languages. Other approaches attempt to convert pseudocode to source code by constructing an intermediary model that incorporates a metamodel to represent pseudocode in a more structured manner [10].

CRUST employs an intriguing technique, a transpiler that converts C / C++ programs to Rust[2] code. In the CRUST conversion process, a set of compact syntactic analyzers called *Nano-Parsers* [11] is utilized. These *Nano-Parsers* are designed to handle specific grammars and cooperate with other analyzers, including a *Master Parser*, to handle complex text inputs.

The CRUST architecture comprises two key components: the syntax analyzer module and the code generator. The *Nano-Parsers*, constituting the syntactic processing element, employ a matching function that is activated when a regular expression identifies a valid pattern corresponding to Rust code. However, this strategy has two notable limitations: the need to develop parsers for each language and the requirement to specify regular expressions for multiple programming languages, including the complex RPG language, which is responsible for generating millions of lines of legacy code.

A technique known as tree-to-tree encoding and decoding uses parse trees and deep neural networks to convert source code from one language to another [12]. This approach includes a training phase to enhance the encoding process. The input is

[1]A Semantic Program Tree (PST) is a structure similar to that an AST but includes the semantics of the program.

[2]https://www.rust-lang.orgRust is a language created by Mozilla.

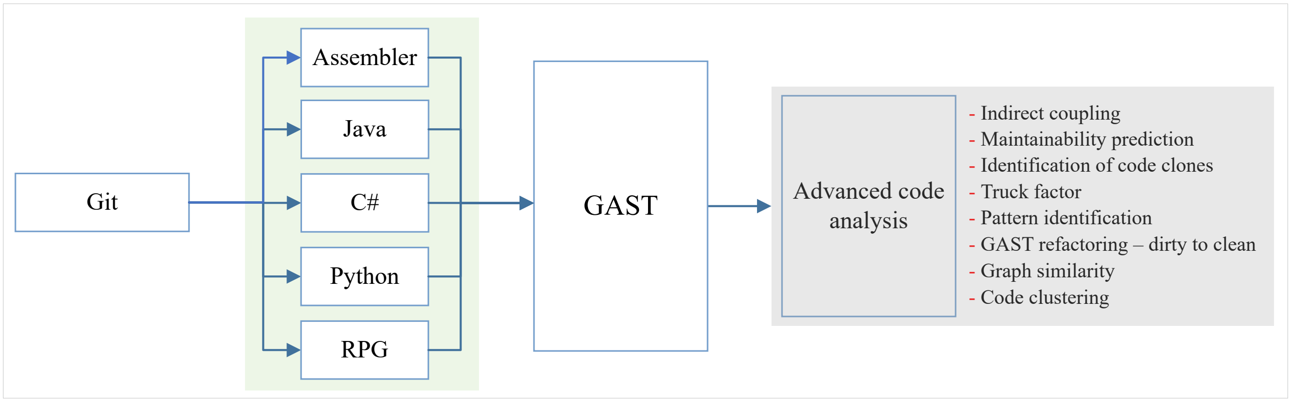


Fig. 1. GAST process.

encoded using a list of symbols, which assesses the likelihood of elements and selects the one with the best fitness value from the set. The subsequent decoding stage involves constructing nodes in the target programming language, starting from the root and generating offspring. Neural networks are used to implement a tree-to-tree encoding and decoding model, maintaining consistency with this concept [13].

Machine learning (ML) techniques have been employed in numerous research projects for automatic source code translation [14]. While supervised methods are commonly used, ML techniques can be classified as unsupervised, supervised, or reinforced. Many studies leverage source code from GitHub projects to train supervised learning algorithms. Since ASTs exhibit some degree of equivalence across languages, this type of research is particularly effective for languages with a similar level of abstraction.

Some research efforts use machine learning (ML) to translate source code [14] automatically. ML methods are classified as supervised, unsupervised, and reinforced, although the most common approaches use supervised methods. Much research uses source code from GitHub projects to train supervised methods. This type of research operates correctly with languages of a similar level of abstraction because the ASTs have some equivalence with each other.

Deep learning is often employed for language translation tasks. For example, Pengcheng and Graham utilize decoders and encoders to construct a neural network architecture that generates code from an AST [15]. Recurrent Neural Networks (RNNs) are employed in the decoders to simulate the sequential creation of a predefined AST, while Long Short-Term Memory (LSTM) networks are utilized in the encoders to generate a set of words. Token generation can utilize a predefined vocabulary or directly copy from the language input. Their study aims to produce an AST by employing grammatical model actions.

Another promising approach is the adoption of an abstract syntax network, although this method is prone to creating unstructured mappings during code production [16]. The model architecture is based on a hierarchical encoder-decoder. The decoder represents and constructs outputs in the form of ASTs using a modular structure, as opposed to a dynamic decoder that simultaneously develops the output tree structure. The HEARTHSTONE benchmark yielded favorable results for code generation, achieving a BLEU (Bilingual Evaluation Understudy) score of 79.2 % and an accuracy rate of 22.7 % for precise matches.

Another approach to translating source code into different languages is Statistical Machine Translation (SMT). Oda et al. utilized this method to convert Python code into pseudocode [17]. Their approach involves examining the source code file word-by-word to determine the best output based on the specified model. The code is then structured using an AST, from which the pseudocode is generated. Although this method is fast and automated, it does not guarantee semantic validity. Therefore, human evaluation is necessary to assess the output's correctness.

Similarly, Nguyen et al. employed SMT to convert Java source code for Android and C# for Windows Phone [18]. The concept behind their approach is to utilize SMT to infer translation rules by leveraging already migrated code as a baseline, rather than manually defining additional rules. They generate a set of annotations by training a model with the ASTs extracted from the source code. Subsequently, further training is conducted to construct lexemes, which are combined to produce the final C# source code.

III. Generic Abstract Syntax Tree

The GAST is designed to serve as a representation of language-specific ASTs (SASTs) in order to facilitate source code analysis across multiple programming languages. The definition of the GAST is based on the Meta Object Facility Specification (MOF) [19]. Figure 1 provides an overview of the GAST-supported process and its interaction with other components of the analysis framework. The general steps of this process are as follows.

1) Obtain the source code of the desired language from a Git software repository. The GAST currently supports several languages, including Assembler, Java, C#, Python, and RPG.
2) Map the corresponding GAST component to each syntactic part of the SAST for each language (see Figure 2)).
3) The Advanced Code Analysis Engine (ACAE) utilizes the GAST format as input to perform various types of analysis.

Figure 2 specifically illustrates the process of integrating a specific language into the GAST. The input for this process is the source code of the target program, while the validated GAST representation serves as the output. This process consists of three steps:

1) During the parsing phase, the source language and source code are provided to obtain the SAST.
2) Certain syntactic elements of the SAST are transformed into corresponding constituents of the GAST.
3) The mapping's validity is verified by comparing the structures of the GAST and SAST to ensure their similarity.

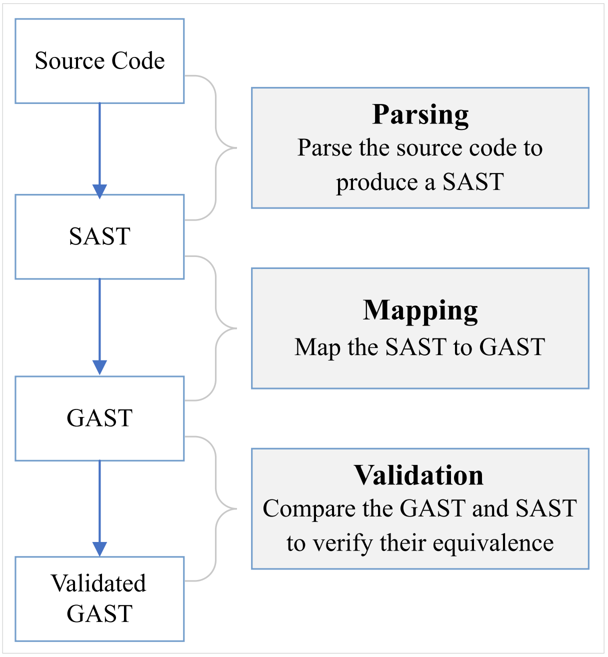

Source Code
SAST
GAST
Validated GAST
Parsing
Parse the source code to produce a SAST
Mapping
Map the SAST to GAST
Validation
Compare the GAST and SAST to verify their equivalence

Fig. 2. Stages for the conversion of source code into the GAST.

To generate each unique Abstract Syntax Tree (AST), a parser generator or specialized tool is necessary. In this study, ANTLR is used for languages other than Java, where the Java Development Toolkit (JDT) is employed for parsing Java source code and obtaining the corresponding AST [20].

The mapping stage determines the equivalence between syntactic components of the SAST and the corresponding structures in the GAST. The mapping rules that establish the connection between SAST and GAST elements are specified. *MapStruct* [21] is utilized to define these rules and perform the mapping of elements. In addition to supporting linear mappings between objects, *MapStruct* is capable of recursive processing during element mapping.

Once the parsing and structure mapping stages are completed, the validation phase begins. This phase ensures that the mapper has correctly linked all syntactic components from the SAST into the GAST tree. The validation phase is discussed in more detail in Section III-A.

Figure 3 depicts the high-level package diagram illustrating the structure of the GAST. The primary package, denoted as ASTMCore, encompasses three sub-packages: ASTMSemantics, ASTMSyntax, and ASTMSource. Each of these packages serves a distinct purpose within the GAST framework, as follows:

ASTMSemantics: The GAST structure is not concerned with semantic elements. However, it requires some elements to establish connections between syntactic elements, such as variable scopes within code blocks, is crucial. Considering the possibility of nested blocks, such as conditionals and nested loops, it is important to incorporate recursion in the class that models the scope.
To determine the validity scope of a component, various syntactic element scopes are implicitly modeled. For example, a variable defined within an if statement can only be used within the block of instructions associated with that if statement and not within an else statement.
The ASTMSyntax package of the GAST structure is responsible for emulating the syntactic components of programming languages and encompasses a significant number of classes within the GAST framework. The main components of this package are outlined in Figure 3.

ASTMSyntax: The ASTMSyntax package of the GAST structure is responsible for emulating the syntactic components of programming languages and encompasses a significant number of classes within the GAST framework. The main components of this package are outlined in Figure 3.

Declarations and Definition: This package includes all syntactic elements that involve the declaration or definition of variables, functions, or data. It provides modeling capabilities for these elements.

Expressions: The Expressions package represents composed instructions that relate to other valid expressions. It includes binary operations, conditionals, type conversions, aggregations, function calls, and arithmetic operations.

Statement: This package closely resembles expressions as it employs expressions to control the flow of execution for each instruction. It encompasses statements such as while, if, for, return, or break. For instance, an if statement consists of the then and else parts, each containing instructions that may have their own scope. These instructions are often associated with an expression that determines the flow of execution, thereby establishing connections with classes in the expressions package.

Types: The Types package encompasses both primitive types and built-in types. It is relevant to the Declarations and Definitions package as it models named, aggregate, function, and namespace types, as well as formal parameter types.

ASTMSource: This package focuses on the compilation unit, which serves as an abstraction of a source file. The class contains attributes such as language, package, scope, and import list, which define the fundamental structure of object-oriented code. The class representing a compilation unit also models additional data such as the file's location, the position of its lines of code, and references to other files.

After mapping the instructions from a SAST into the GAST, the accuracy of the result is validated.

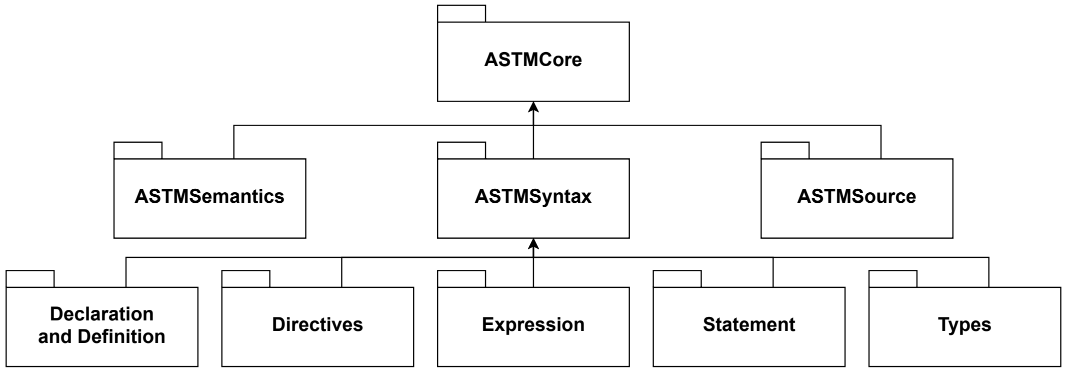


Fig. 3. High-level package structure diagram of the framework.

A. GAST validator

Following the completion of the mapping process, an essential automated task involves validating and verifying the accuracy of the mapping. Name equivalences play a key role in this process, as dictionaries are employed to establish connections between the two structures. The validator not only obtains the names of the leaf nodes but also establishes the corresponding nodes between the GAST and the SAST, ensuring the information contents of these nodes are equivalent.

The objective of this task is to verify the effectiveness of the mapping by ensuring that every element present in the SAST is also accurately represented in the GAST. Successful completion of this task confirms that the attributes of the analyzed files have been correctly mapped, thus demonstrating the GAST's fidelity as a representation of the original program.

The GAST validator generates a report that highlights the differences between the mapping of the SAST and the GAST. It identifies the file paths where non-conformances are located, providing insights into any inconsistencies.

The flowchart in Figure 4 illustrates the algorithmic process of the validator. It operates after generating the SAST and GAST for the given file. The initial phase of the algorithm involves retrieving the methods of a node. Subsequently, it compares the approaches of the two syntactic trees to determine their equivalence.

The dictionary containing the equivalences between the structures is utilized to facilitate the comparison. The algorithm then traverses the trees, ensuring that all nodes and leaf nodes in both trees are equivalent. Given that variables, constants, or modifiers are crucial components in both the SAST and GAST, their names can be used as values for comparison in the leaf nodes. If any discrepancies arise in the values of these elements between the two trees, the validator generates a report accordingly.

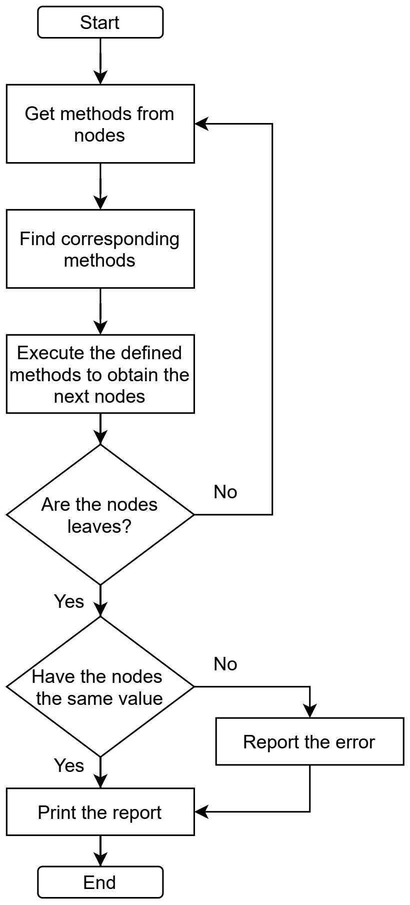

Start
Get methods from nodes
Find corresponding methods
Execute the defined methods to obtain the next nodes
Are the nodes leaves?
No
Yes
Have the nodes the same value
No
Report the error
Yes
Print the report
End

Fig. 4. Validation algorithm for the SAST and GAST structure.

IV. Results and analysis

This section presents and analyzes the results of two experiments conducted to substantiate the following:

1) The feasibility of employing a unified AST to represent programs written in multiple programming languages. This is demonstrated in Section IV-A where the mapping of different SASTs to the GAST is discussed.
2) The adoption of the GAST representation allows for the development of a unified source code analyzer capable of analyzing diverse programming languages. Sections IV-B and IV-C validate the use of the GAST for the analysis of source written in multiple languages.

Furthermore, we showcase the structural capabilities of the GAST by applying various metrics to the GAST of the JDT project.

A. Mapping SAST to GAST

The aim of this experiment is to evaluate the GAST's ability to accurately represent the syntactic elements of multiple SASTs and assess the feasibility of establishing a linear mapping between their corresponding elements. To ensure equivalence between the created SASTs and the GAST, RPG, C#, and Java were selected as target languages for testing. The projects selected for this experiment are listed in Table I. The assessment was carried out through the following steps:

1) The experiment proceeds as follows:Create the GAST structure for each project. 2) Verify the correct mapping between all SAST elements and the corresponding elements in the GAST. 3) Generate of a report highlighting the mapping differences between the SAST and the GAST. Compile statistics on the execution times of the project's code transformations.

1) The experiment proceeds as follows:Create the GAST structure for each project.
2) Verify the correct mapping between all SAST elements and the corresponding elements in the GAST.
3) Generate of a report highlighting the mapping differences between the SAST and the GAST.
4) Compile statistics on the execution times of the project's code transformations.

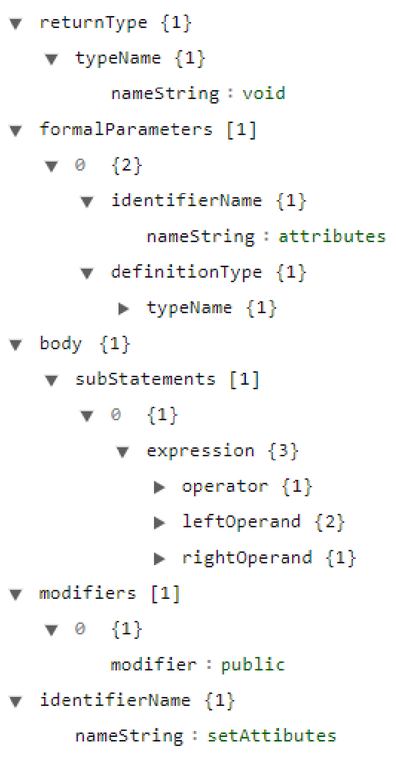
TABLE I
PROGRAMS USED TO VALIDATE THE MAPPING

Project	**Number of files**	**Language**
Arduino	279	Java
JDT	8365	Java
Java Design Patterns	1478	Java
NTLR v4	188	C#
ShareX	742	C#
Maui	6336	C#
Company project (confidential)	92	RPG

The GAST is a tree-based structure that serves as a representation of the source code. In this context, the code fragment depicted in Figure 5 can be directly correlated with the corresponding tree representation displayed in Figure 6.

```java
public void setAttributes(SimpleAttributeSet
     attributes) {
    this.attributes = attributes;
  }
```

Fig. 5. Source code fragment in Java.



```
▼ returnType {1}
    ▼ typeName {1}
        nameString : void
▼ formalParameters [1]
    ▼ 0  {2}
        ▼ identifierName {1}
            nameString : attributes
        ▼ definitionType {1}
            ▶ typeName {1}
▼ body {1}
    ▼ subStatements [1]
        ▼ 0  {1}
            ▼ expression {3}
                ▶ operator {1}
                ▶ leftOperand {2}
                ▶ rightOperand {1}
▼ modifiers [1]
    ▼ 0  {1}
        modifier : public
▼ identifierName {1}
    nameString : setAttibutes
```

Fig. 6. Source code mapping into the GAST.

The code fragment depicted in Figure 5 exemplifies a public method called setAttributes. In the GAST, this method is represented within the modifier tag with a value of public. Additionally, the GAST represents the return type of the method as void within the returnType tag. The method name, setAttributes, is represented as a leaf node in the GAST within the identifierName branch.

In Figure 6, the formalParameters branch of the GAST displays a single offspring representing the parameter of the function described in the code fragment, which is also illustrated in the same Figure. The parameter's name in the source code, identifierName, is represented as a leaf node within the GAST under the attributes section of the function. Additionally, the function body contains an expression involving left and right operand operators, both of which are listed under the subStatements branch in the GAST.

The code fragment serves as an illustrative example of mapping source code to the GAST structure, and the aforementioned statements facilitate a manual examination of the syntactic components of the setAttributes function. However, relying solely on manual verification is time-consuming, error-prone, and may lead to overlooking certain issues. To address this limitation, the technique incorporates a mapping validator that automates the verification process.

Figure 7, similar to Figure 8, depicts the source code mapped to the corresponding GAST structure. The same tests conducted on previous examples were also applied to this specific sample, ensuring consistency and enabling a comprehensive evaluation of the GAST's ability to accurately represent the syntactic components and structures of the source code.

Although the process of syntactic verification for the two trees is time-consuming, it is essential to establish the equivalence between the SASTs of each language and the GAST (as shown in Table II). The lack of parallelism in the procedure significantly contributes to the prolonged duration of the study. Currently, the analysis is performed in serial mode, sequentially checking each file, which consumes a considerable amount of time.

TABLE II
TRANSFORMATION OF JAVA, C# AND RPG SOURCE CODE

Language	Project	Time with verifier	Time without verifier
Java	Arduino	15.130	9.223
	Design Patterns	47.121	19.832
	JDT	646.253	97.544
C#	ANTLR v4	26.404	14.216
	ShareX	15.059	8.070
	Maui	2390.123	1118.999
RPG	Files	78.308	42.406

```
public abstract bool Matches(int symbol, int
    minVocabSymbol, int maxVocabSymbol);
```

Fig. 7. Source code of Transition.cs.

However, the results demonstrate the effective conversion of source code in the supported languages into the GAST, even for large projects like JDT, which encompasses over 8,000 files.

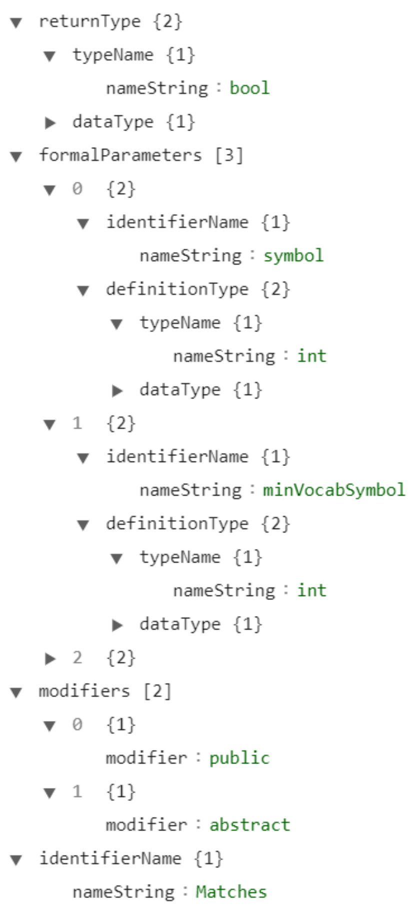
B. Homogenizing the analysis

This experiment aims to demonstrate the feasibility of standardizing the analysis of program constituents and grammatical structures across different programming languages. The experimental procedure is outlined as follows:

1) Develop two applications, one in Java and another in C#, with identical functionality.
2) Create Specific Abstract Syntax Trees (SASTs) for each program, corresponding to Java and C#.
3) Verify the similarity between the resulting Generic Abstract Syntax Trees (GASTs) generated for Java and C#.
4) Evaluate the results of code clone analysis when applied to the generated GASTs.

To illustrate this experiment, a representative chess game code was implemented in both Java and C# to showcase the analysis of the representation rather than the original syntax of the programs. The objective is to transform equivalent programs written in different programming languages into a generic syntax, enabling their analysis.

The chess program's architecture incorporated abstract classes and object arrays, utilizing class inheritance and association. This design choice adds complexity to the transformed



```
▼ returnType {2}
    ▼ typeName {1}
        nameString : bool
    ▶ dataType {1}
▼ formalParameters [3]
    ▼ 0  {2}
        ▼ identifierName {1}
            nameString : symbol
        ▼ definitionType {2}
            ▼ typeName {1}
                nameString : int
            ▶ dataType {1}
    ▼ 1  {2}
        ▼ identifierName {1}
            nameString : minVocabSymbol
        ▼ definitionType {2}
            ▼ typeName {1}
                nameString : int
            ▶ dataType {1}
    ▶ 2  {2}
▼ modifiers [2]
    ▼ 0  {1}
        modifier : public
    ▼ 1  {1}
        modifier : abstract
▼ identifierName {1}
    nameString : Matches
```

Fig. 8. Mapping of the ANTLR project's file Transition.cs.

GAST structure and enhances its ability to accurately represent real-world development programs.

For the purpose of comparison, the clone detection metric, capable of distinguishing Type I, II, and III clones [22] (with our research focusing on evaluating Type II clones), was implemented. Figures 9 and 10 present an excerpt of a clone found in the Java version, corresponding to the C# version of the program.

Table III lists the significant clones identified by the clone detector, excluding set and get methods. All identified clones were discovered in both the C# and Java versions of the Chess project using the GAST representation. The source code for both versions and the identified clones can be found at https://github.com/JasonLeiton/Ajedrez.

The results demonstrate that the GAST representation of both programs effectively identified clones, yielding reliable outcomes. This outcome supports the notion that analysis can be standardized by utilizing a universal structure to generate equivalent representations across various programming languages. As such, the development of a unified analyzer capable of operating with multiple programming languages and facilitating cross-language comparisons becomes a feasible endeavor with the utilization of this abstract syntax.

```
public boolean ValidateRightDiagonalDown(int x1, int y1, int x2, int y2) {
  boolean flag = true;
  while (x1 + 1 <= x2 && y1 + 1 <= y2) {
    if (board[x1+1][y1+1].isTaked()) {
      flag = false; break;
    }
    x1++; y1++;
  }
  return flag;
}
```

Fig. 9. First method from the class Board of the chess project that was detected as a clone.

```
public boolean ValidateRightLeftDown(int x1, int y1, int x2, int y2) {
  boolean flag = true;
  while (x1 - 1 >= x2 && y1-1 >= y2) {
    if (board[x1-1][y1-1].isTaked()) {
      flag = false; break;
    }
    x1++; y1++;
  }
  return flag;
}
```

Fig. 10. Second method detected as clone

TABLE III
CLONES FOUND IN THE CHESS PROJECT

GAST Java	GAST C#
Initialize	ValidateLinesStraightHorizontalD
Initialize	ValidateLinesStraightVertical
ValidateDiagonalRightDown	ValidateDiagonalLeftDown
ValidateDiagonalRightDown	ValidateLinesStraightHorizontalD
ValidateDiagonalRightDown	ValidateLinesStraightVertical
ValidateDiagonalLeftDown	ValidateLinesStraightHorizontalD
ValidateDiagonalLeftDown	ValidateLinesStraightVertical
ValidateLinesStraightHorizontalD	ValidateLinesStraightVertical
ValidateL	ValidateR
ValidateMove	SetPiece

C. GAST applications

The conversion of source code from a specific programming language to a universal language enables the evaluation of software quality and maintainability. In our study, we conducted an analysis of the JDT project to generate code analysis tests for key applications. The resulting JDT GAST was stored in a Neo4j database, a graph-oriented database that facilitates the visualization of classes, methods, and relationships, and enables the calculation of various metrics.

To determine the indirect connections between nodes, we devised metrics that capture the methods of classes and establish associations between them [23]. Figure 11 illustrates two distinct classes, JavaMethodFiltersTable and TypeFilterAdapter, each with methods that call other methods, allowing us to establish two types of associations. The first type, CALLS, signifies that one method in the code invokes another method. The second type corresponds to OWNS_METHODS, indicating that the method is a member of the class.

Moreover, employing a universal language analysis enables the collection of quantitative and analytical software measurements. Examples include metrics such as lines of code (LOC), methods called (CALLS), methods being called by other methods (CALLED BY), and cyclomatic complexity (CYC). The work by Navas-Su et al. [23] provides detailed explanations of how these metric values were computed. Table IV presents the results of each metric for the two JDT approaches.

The primary objective of gathering these metrics is to enhance software maintainability and facilitate informed decision-making to ensure positive effects of code modifications. Cyclomatic complexity (CYC), a particularly relevant metric, provides insights into the extent of a method's utilization within a software project. This understanding allows for an assessment of the potential impact of modifying, eliminating, or creating a new method that depends on others (or vice versa), and can assist in identifying ineffective approaches.

Furthermore, the application of GAST in academic settings offers additional benefits, particularly in supporting programming instructors and guiding students. GAST automates the examination of various aspects of computational thinking, including flow control, data representation, problem decomposition, and the identification of common programming errors. As a result, it can effectively contribute to student assessments. By leveraging GAST, instructors can assess and evaluate students' levels of computational proficiency, providing valuable assistance in teaching activities. This automated evaluation process enhances efficiency and ensures consistent, objective assessment criteria, benefiting both students and educators.

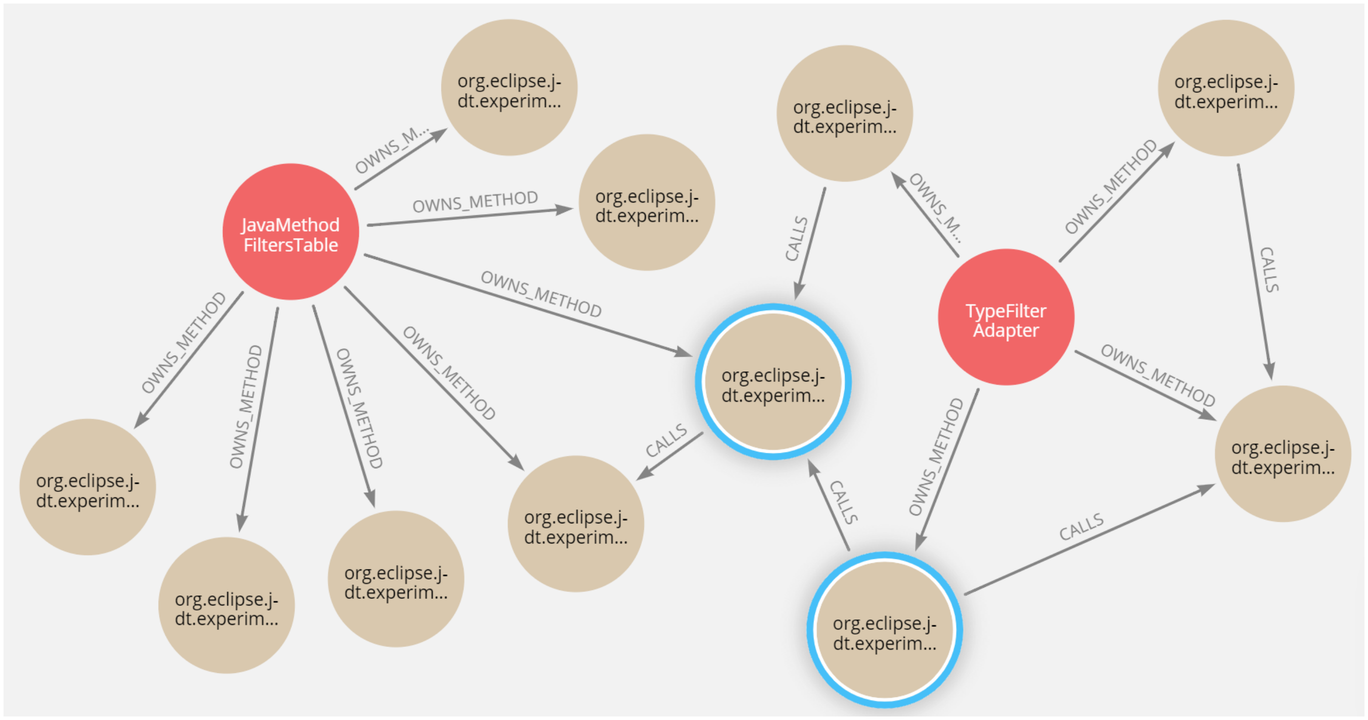


Fig. 11. Two specific classes of the JDT project JavaMethodFiltersTable and TypeFilterAdapter represented by the red circles, with their respective method membership relationships and calls to other methods.

V. Conclusions

The evaluation of software quality heavily relies on source code analysis, which traditionally requires the development of language-specific metrics to accommodate the unique syntax of each programming language. In this paper, we propose an alternative approach by introducing a universal structure based on the MOF 2.0 specification.

This universal structure serves as a representation for multiple abstract syntax trees from various programming languages. Its design emphasizes extensibility, allowing for the inclusion of new languages and enabling comprehensive support for quality control and software maintenance tasks.

The adoption of a universal representation for multiple programming languages contributes to standardizing the field of software analysis. By creating metrics once for all languages, this approach offers the advantage of reusability as new languages are integrated into the universal structure, enhancing flexibility in analysis.

The Generic Abstract Syntax Tree (GAST) plays a crucial role in establishing equivalence among elements from diverse programming languages. This allows for consistent analysis techniques to be applied across languages belonging to different paradigms. By utilizing a single structure with adaptable metric definitions, it becomes possible to compare the behavior of components in RPG, Java, C#, and other languages. Even though these components may differ, it can be demonstrated that they are substantially equivalent, and metrics can be derived accordingly for software developed in various languages.

However, the method's limitation lies in the conversion process from language-specific Abstract Syntax Trees (ASTs) to the GAST, as it relies solely on the structure of the AST. Each language requires a unique mapping procedure, specifying equivalence criteria for every syntactic element. To ensure accurate transformation of all syntactic elements into the GAST the other modules that use the GAST do not need to verify the completeness of the syntactic elements due because they are checked during the process of constructing the transformer.

TABLE IV
Metrics for the doubleClicked and doButtonPressed methods of the JDT project

Metric name	doubleClicked	doButtonPressed
calledBy	0	2
calls	2	1
cyc	2	6
fcyc	2	6
fhal	1180.96	965.09
floc	22	19
fnom	4	2
hal	33	846.49
loc	2	18
nom	1	1
rcyc	2	7
rhal	33	1001.35
rloc	2	21
rnom	1	3

The results obtained showed that the validation of the mapping is a time-consuming task. However, it represents an advantage because the other modules that use the GAST do not need to verify the completeness of the syntactic elements due to the checking performed when building the transformer.

The outcomes of our study clearly demonstrate the GAST's efficacy in facilitating program analysis, cross-language code comparisons, and educational assessments. This research paves the way for the development of advanced source code analyzers, software metric collection and calculation tools, and training resources that seamlessly operate across multiple programming languages.

VI. Future Work

The GAST serves as a foundational framework for various ongoing initiatives within our research team. To thoroughly validate and understand its potential limitations and challenges, we are actively converting source code from different languages to the GAST and conducting comprehensive studies.

Exploring the reverse process of generating source code from the general abstract syntax tree and producing code in specialized languages is a topic of future investigation. Currently, the GAST structure is successfully used to generate code in Java, C#, and Python. This opens up possibilities for the development of many-to-many language translators, enabling code generation in multiple target languages from a single GAST representation.

To expedite the syntactic verification process for the two trees, we propose the incorporation of parallelism into the analysis workflow. By leveraging the computational power of modern systems, parallel processing can be employed to simultaneously analyze multiple files. This approach efficiently distributes the workload across threads or processors, resulting in faster analysis and significantly reduced turnaround time. Harnessing the potential of parallelism enhances the speed and efficiency of analysis, ultimately improving the overall effectiveness of the verification task.

The GAST project also finds utility in the analysis of malware code. Decompiling binary files and converting them to GAST representation allows for the extraction of associated assembly code. By employing search algorithms, it becomes possible to identify coding patterns associated with malicious code, enabling early detection of malware.

Furthermore, the GAST serves as the foundation for a novel initiative in clone detection across programs written in different languages or different versions of the same language. Our experiments involve improved meta-data and semi-structural code-to-code comparisons, utilizing deep learning techniques for resemblance analysis of digital images derived from GAST representations, and structural GAST-based similarity analysis

References

[1] S. Nanz and C. A. Furia, “A Comparative Study of Programming Languages in Rosetta Code,” p. 778–788, 2015, https://doi.org/10.1109/ICSE.2015.90.

[2] Z. Mushtaq, G. Rasool, and B. Shehzad, “Multilingual Source Code Analysis: A Aystematic Literature Review,” *IEEE Access*, vol. 5, pp. 11 307–11 336, 2017, https://doi.org/10.1109/ACCESS.2017.2710421.

[3] M. D’Ambros, H. Gall, M. Lanza, and M. Pinzger, “Analysing software repositories to understand software evolution,” pp. 37–67, 2008, ISBN: 978-3-540-76440-3.

[4] Sonar, “Sonarqube,” *Electronic*, Jun 2023. [Online]. Available: https://www.sonarsource.com

[5] O. Nierstrasz, S. Ducasse, and T. Gîrba, “The Story of Moose: an Agile Reengineering Environment,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 1–10, 2005, https://doi.org/10.1145/1095430.1081707.

[6] M. Papadakis, M. Delamaro, and Y. Le Traon, “Mitigating the Effects of Equivalent Mutants with Mutant Classification Strategies,” *Science of Computer Programming*, vol. 95, pp. 298–319, 2014, https://doi.org/10.1016/j.scico.2014.05.012.

[7] F. A. Bastidas and M. Pérez, “A Systematic Review on Transpiler Usage for Transaction-Oriented Applications,” in *2018 IEEE Third Ecuador Technical Chapters Meeting (ETCM)*. IEEE, 2018, pp. 1–6, https://doi.org/10.1109/ETCM.2018.8580312.

[8] D. L. Whitfield and M. L. Soffa, “An Approach for Exploring Code Improving Transformations,” *ACM Transactions on Programmming Languages Systes*, vol. 19, no. 6, p. 1053–1084, nov 1997, https://doi.org/10.1145/267959.267960.

[9] K. An, N. Meng, and E. Tilevich, “Automatic Inference of Java-to-Swift Translation Rules for Porting Mobile Applications,” in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, 2018, pp. 180–190, https://doi.org/10.1145/3197231.3197240.

[10] T. Dirgahayu, S. N. Huda, Z. Zukhri, and C. I. Ratnasari, “Automatic Translation from Pseudocode to Source Code: A Conceptual-Metamodel Approach,” in *2017 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom)*. IEEE, 2017, pp. 122–128, https://doi.org/10.1109/CYBERNETICSCOM.2017.8311696.

[11] N. Shetty, N. Saldanha, and M. Thippeswamy, “CRUST: AC/C++ to Rust Transpiler Using a “Nano-parser Methodology” to Avoid C/C++ Safety Issues in Legacy Code,” pp. 241–250, 2019, ISBN:978-981-16-1344-9.

[12] M. Drissi, O. Watkins, A. Khant, V. Ojha, P. Sandoval, R. Segev, E. Weiner, and R. Keller, “Program Language Translation Using a Grammar-Driven Tree-to-Tree Model,” *arXiv preprint arXiv:1807.01784*, 2018, https://doi.org/10.48550/arXiv.1807.01784.

[13] “Tree-to-Tree Neural Networks for Program Translation, author=Chen, Xinyun and Liu, Chang and Song, Dawn, journal=arXiv preprint arXiv:1802.03691, year=2018, note=”https://doi.org/10.48550/arxiv.1802.03691”.”

[14] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, “Unsupervised Translation of Programming Languages,” *arXiv preprint arXiv:2006.03511*, 2020, https://doi.org/10.48550/arXiv.2006.03511.

[15] P. Yin and G. Neubig, “A Syntactic Neural Model for General-Purpose Code Generation,” *arXiv preprint arXiv:1704.01696*, 2017, https://doi.org/10.48550/arXiv.1704.01696.

[16] M. Rabinovich, M. Stern, and D. Klein, “Abstract Syntax Networks for Code Generation and Semantic Parsing,” *arXiv preprint arXiv:1704.07535*, 2017, https://doi.org/10.48550/arXiv.1704.07535.

[17] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, “Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation,” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 574–584, https://doi.org/10.1109/ASE.2015.36.

[18] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, “Migrating Code with Statistical Machine Translation,” in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 544–547, https://doi.org/10.1145/2591062.2591072.

[19] OMG. (2016) About the Meta Object Facility Specification, Version 2.5.1. [Online]. Available: https://www.omg.org/spec/MOF

[20] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013, ISBN=978-1-934356-99-9.

[21] Mapstruct. (2018, jul) Mapstruct Java Bean. [Online]. Available: https://maven.apache.org/

[22] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, “Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271, https://doi.org/10.1109/SANER48275.2020.9054857.

[23] J. Navas-Sú and A. González-Torres, “A Method to Extract Indirect Coupling and Measure its Complexity,” in *2018 International Conference on Information Systems and Computer Science (INCISCOS)*. IEEE, 2018, pp. 186–192, https://doi.org/10.1109/INCISCOS.2018.00034.