Identifying Dead Code in Java Programs with Call Graphs: a Comparative Study of Semantic and Syntactic Approaches

1rd Arianna Bianchi
s222890
s222890@dtu.dk

2st Kajsa Aviaja Pedersen
s220422
s220422@student.dtu.dk

3nd Kári Sveinsson
s212434
s212434@dtu.dk

4rd Mariana Santos
s233360
s233360@dtu.dk

***Abstract*—In this paper, we conduct a comparative analysis of two static approaches aimed at identifying dead code in Java programs. Our investigation involves a syntactic analysis tool utilising parse trees, and an abstract interpreter that interprets JVM bytecode. Both these tools strive to find an over-approximation of all callable functions in a program. These are then compared to each other to see which is performing best and to a call graph created by running an interpreter on a program with no parameters, which serves as the ground truth. We see that both ways produce an over-approximation on the subset of Java we handle, where the syntactic approach is faster, and the semantic approach is closer to the ground truth. Despite it being slower than the syntactic analysis, the semantic analysis should be preferred due to the more precise result.**

***Index Terms*—Call Graph, Interprocedural Control Flow, Syntactic and Semantic Program Analysis, Static Program Analysis, Dead Code**

I. Introduction

Call graphs are widely used tools to help with the analysis and understanding of computer programs. They are valuable not only to aid programmers with understanding the flow of software code, but also highlight possible areas for improvement within the code, and can be used for compilers for optimisation [1]. In projects with a large codebase, it becomes increasingly hard to keep track of which code blocks are actually being executed or not. The presence of pieces of code that will never be run poses two main problems:

- It makes the program code more difficult to understand since the developer is most likely expecting that every piece of code is used at one point or another. This becomes a problem when cleaning up old code. [2]
- Can often be an indicator of logic errors, and is also considered a “code smell". [3]

These problems are more common than one might think at first, and there are multiple examples where a large percentage of code was unused in production systems. [2] [3] [4]

To create call graphs, you usually do an interprocedural control flow analysis. An interprocedural analysis looks at the dynamic relationships between procedures within a program and can be used to obtain a call graph – a directed graph that represents which methods are called by which other methods. Each edge represents a function p, and an edge (p_i, p_j) exists if p_i can invoke p_j. [5]

While the terms “unreachable code" and “dead code" can be considered to be code that can never be executed – because there’s no possible execution path to reach them –, there’s another different meaning to “dead code". It can refer to code blocks which would get executed but do not produce any changes to the program. Most of this code will be removed by the compiler. [3] In this project, we will not consider the alternative meaning of dead code.

In this project, we will look at the interprocedural flow and call graphs in Java programs. In particular, we want to answer the following question:

Considering accuracy, speed and complexity: is semantic analysis better than syntactic analysis for creating call graphs in Java programs to identify dead code?

The syntactic analysis will be performed on the Java code, using the parser generator tool *Tree-sitter* [6]. We expect that this simplistic approach will provide unsatisfactory results, returning a graph with too many false positives. However, this analysis should be faster, since it is reading a text file and analysing an Abstract Syntax Tree (AST).

In the semantic analysis, we will analyse Java bytecode, converted to JSON objects using the tool *jvm2json* [7]. We will use sign abstraction for integers and references, and we also abstract over Boolean values to support if/else statements and loops. We expect this solution to provide a graph that is more precise, with fewer false positives. However, using some kind of widening, or other kinds of abstraction, might yield even better results. It’s important to note that the bytecode obtained from compiling the Java code can be subject to some optimisation. However, we don’t expect this to have an influence on our work or results, and will therefore use the compiler without opting out of optimization.

In this paper, we make the following contributions:

- There are many different program analysis techniques that could apply to this problem. In Section II we will explore the two methods subject to comparison. Other techniques, such as dynamic analysis could be used to produce function call graphs. We will analyse alternative solutions in Section V.
- The semantic analysis produces a significantly more robust graph, that is better aligned with reality. Our Python

implementation and analysis of results are described in Section III and IV.

II. Preliminaries

The syntactic and semantic analysis are two different types of static analysis. This means that they are performed without having to actually execute the code. These two techniques differ in that the syntactic analysis only focuses on the structure of the source code – the syntax and that the semantic analysis considers the meaning of the statements in the code – how the program behaves.

This section aims to introduce the theory behind these techniques, and to justify their effectiveness.

A. Syntactic Analysis

Programs are normally written in a formal language, also known as source code. Syntactic analysis is a type of static analysis and is a type of analysis performed upon these text files, by analysing the code using a specific formal grammar. Syntactic analysis is an important step in the compilation of any program, and is part of the parsing phase. But how can a sequence of characters be converted into an executable program?

The source code is first converted into a sequence of tokens. Lexical analysis – also referred to as lexing, scanning, or tokenization – is the phase responsible for converting a regular text file into a stream of **tokens**. Tokens are meaningful strings, which are categorized based on regular expressions [8]. For instance, tokens will be identified as being keywords, identifiers or symbols, among others. Unnecessary information will be discarded at this point, such as white-space characters.

For example, the expression `a = 2 + 3` could be converted into the following tokens:

```
<'a', Name> <'=', Assign>
<'2', Constant> <'+', BinOp> <'3', Constant>
```

The following phase is the syntax analysis, also called parsing, and it deals with the hierarchical structure of the tokens [9], determining the syntax of the source program [8]. This involves verifying that the sequence of tokens is valid and well-formed, according to the grammar rules in place. This step normally results in a tree-like structure, called a **Concrete Syntax Tree (CST)**. A simplified version is more often used, and is called an **Abstract Syntax Tree (AST)**. The key distinction between the two lies in the omission of certain information in the abstract version, focusing only on what is essential for a syntax tree. For instance, in the ASTs, the parenthesis around mathematical expressions may be omitted because they can be inferred from the tree structure [9]. An example AST for the aforementioned expression is depicted on Figure 1.

The next important phase in the compilation process is semantic analysis, which we will get back to in Subsection II-B. Instead, we will now focus on two different syntactic analysis techniques: (1) using regular expressions, and (2) analysing ASTs.

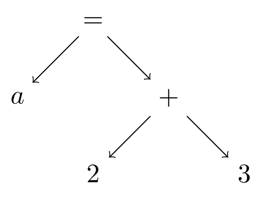

Figure 1. Example of an AST for the expression `a = 2 + 3`

To understand the characteristics of these analysis techniques, one must be aware of the different types of formal grammar that exist.

Grammars are classified into the Chomsky Hierarchy based on their power and the expressiveness of the languages they generate. The levels are described in the following points, where a is a terminal symbol, A, B are non-terminal symbols and α, β, γ are string consisting of terminal and non-terminal symbols.

- **Type-3 Grammars – Regular Languages:** Regular languages can be obtained by regular expressions. The production rules are $A \to a$ and $A \to aB$. These languages are quite fast to process since they can be interpreted by a finite automaton – also known as a state machine –, and as such, they are useful for pattern matching and text processing.
- **Type-2 Grammars – Context-free Languages:** Context-free languages can be recognised by a pushdown automaton (PDA). The production rule is $A \to \alpha$. Most programming languages are belong to this category.
- **Type-1 Grammars – Context-sensitive Languages:** These grammars generate languages that can be recognised by a linear bounded non-deterministic Turing machine. The production rule is defined as $\alpha A \beta \to \alpha\gamma\beta$.
- **Type-0 Grammars – Unrestrained Languages:** The languages generated by Type-0 grammars are also known as recursively enumerable, or Turing-recognisable languages. These include all formal grammar, and generate the set of languages that a Turing machine can recognise. The production rule is $\gamma \to \alpha$, where γ is non-empty.

As stated, most programming languages are considered to be context-free. Formally, context-free languages (CFL) are languages that can be generated by a context-free grammar (CFG). A CFG is a tuple:

$$G = (V, \Sigma, R, S)$$

Where V is a finite set of non-terminal variables, Σ is a finite set of terminals, R a set of production rules of the form $A \to \alpha$ with $A \in V$ and $\alpha \in (V \cup \Sigma)^*$, and $S \in V$ is the start variable.

Therefore, a language L is context-free if there exists a CFG G where $L = G(L)$:

$$L(G) = \{v \mid S \Rightarrow_G^* v, v \in \Sigma^*\}$$

Here $S \Rightarrow_G^* v$ denotes that v can be produced by applying 0 or more productions from the CFG G [10].

In our syntactic analysis, we look at a pre-generated AST. This way it is fairly straightforward to search through the various elements and quickly find the information that we need. Even though regular expressions can be used as a tool to perform syntactic analysis, using this approach wouldn't be sufficient for our use case, since they can correctly only parse regular languages.

We assume that the pre-generated AST's are a true representation of the Java source code, and so we just have to make sure that we do find all nodes that represents a function call, so that we get an over-approximation of what functions are actually called. By only looking at the AST, we do not care about how the program is run, but merely look at what the source code contains. This ensures that we can get an over-approximation of the program, but it will most likely not be very precise.

In our implementation, we parse the AST in a dept-first fashion. We will delve into implementation details in Section III-B.

B. Semantic Analysis and Abstract Interpretation

While syntactic analyses focus on the program structure, semantic analyses look at the meaning, or behaviour of the program. There are multiple ways to do this kind of analysis but for the purpose of this paper, we will only discuss **abstract interpretation**.

When reasoning about the semantics of a program, we strive to know every state of the program. This, however, is not feasible since we cannot run programs with every possible input. Instead, we can try and abstract over these values to find a good approximation of what the actual states would look like. One example of this is an abstraction over integers, where we only look at the sign of integers instead of the actual number. We'll expand on this later.

To understand what lattices are, we first introduce **partially ordered sets**. A partially ordered set $(S, \sqsubseteq)$ consists of a non-empty set S and a binary relation $\sqsubseteq$ that $\forall x, y, z \in S$ satisfies the 3 properties:

1) Reflexivity: $x \sqsubseteq x$
2) Anti-symmetry: $x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$
3) Transitivity: $x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$

We define $S = \{+, -, 0, \bot, \top\}$ as a partially ordered set, where $\sqsubseteq$ denotes the ordering. Then $\bot \sqsubseteq 0$ says that 0 is a safe approximation of $\bot$.

We extend this definition and say that for $X \in S$ and $y, z \in S$ then

- $X \sqsubseteq y$ denotes that y is an upper bound for X if $\forall x \in X : x \sqsubseteq y$
- $z \sqsubseteq X$ denotes that z is lower bound for X if $\forall x \in X : y \sqsubseteq x$.

We can also define a least upper bound $\bigsqcup X$ as:

$$X \sqsubseteq \bigsqcup X \wedge \forall y \in S : X \sqsubseteq y \Rightarrow \bigsqcup X \sqsubseteq y$$

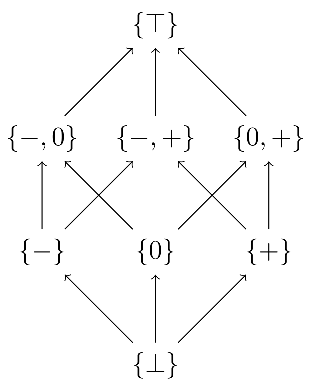

Figure 2. Sign-Abstraction Power Lattice

and a greatest lower bound $\bigsqcap X$ as:

$$\bigsqcap X \sqsubseteq X \wedge \forall y \in S : y \sqsubseteq X \Rightarrow y \sqsubseteq \bigsqcap X$$

We can now introduce **lattices**. A lattice is a partially ordered set $(S, \sqsubseteq)$ where there for all pairs $\forall x, y \in S$ exists:

- a least upper bound: $\bigsqcup\{x, y\} = x \sqcup y$ (join)
- a greatest lower bound: $\bigsqcap\{x, y\} = x \sqcap y$ (meet)

When doing abstract interpretation it is also important to talk about **complete lattices**. A complete lattice is also a partially ordered set $(S, \sqsubseteq)$ where there for all subsets $\forall X \subseteq S$, exist a greatest lower bound and a least upper bound.

For every set S, we can create a complete lattice $(P(S), \sqsubseteq)$ where $P(S)$ is the powerset of S. For the set $S' = \{+, -, 0\}$, we can create the complete powerset lattice $P(S')$ which is shown in Figure 5.

When using lattices, we want to make sure that our abstraction is as precise as possible. To ensure this we use **fixed points**. A fixed point is a value that does not change under any given transformation. That is, for a function $f_L \rightarrow L$, we say that x is a fixed point if $f(x) = x$.

A program can have multiple fixed points. However, to obtain the most precise solution, we are only interested in the least fixed point $f^n(\bot) \sqsubseteq lfp(f)$.

We know from Kleen [5], that, given a complete lattice L with finite height, every monotone function on L has a unique least fixed point:

$$lfp(f) = \bigsqcup_{i \geq 0} f^i(\bot)$$

where a function $f : L \rightarrow L$ over lattice L is monotone when:

$$\forall x, y \subseteq L : x \sqsubseteq y \rightarrow f(x) \sqsubseteq f(y)$$

Because of this we now know that for every function f over a complete lattice L with finite height, there will always exist a *a most precise solution*.

However, if we have a lattice with infinite height, there does not necessarily exist a least fixed point. So, when using abstractions with lattices with infinite height it is necessary to do some sort of widening or narrowing. This is, however, not the case for our analysis, so we will not touch further upon this.

When using abstractions, we want to make sure they are an over-approximation. For this, we can use **Galois connections**. A Galois connection is a pair of monotone functions, one abstraction function $\alpha : L_1 \to L_2$, and one concretisation function $\gamma : L_2 \to L_1$, over the complete lattices L_1 and L_2, that satisfies the properties:

1) $\forall x \in L_1 : x \sqsubseteq \gamma(\alpha(x))$
2) $\forall y \in L_2 : \alpha(\gamma(y)) \sqsubseteq y$

Property (1) states that the abstraction is always safe, but can lose precision. Property (2) states that it should give the most precise abstraction, and it is usually the identity function.

1) *Soundness in abstract Interpretation*

When using an abstract interpreter, we want to make sure that it produces an overestimation, i.e. the analysis is sound. Formally speaking, we want to ensure that, for all concrete states $\mathcal{C}$, and all abstract states $\mathcal{A}$, then the following is true:

$$\alpha(\mathcal{C}) \sqsubseteq \mathcal{A}$$

We do this by showing that all abstract functions $\mathcal{A} \to \mathcal{A}$ and concrete functions $\mathcal{C} \to \mathcal{C}$ are monotone. Then, if the function $af : \mathcal{A} \to \mathcal{A}$ is a sound abstraction of $cf : \mathcal{C} \to \mathcal{C}$, that is, if:

$$cf \circ \gamma \sqsubseteq \gamma \circ af$$

then we know that [5]:

$$lfp(cf) \sqsubseteq \gamma(lfp(af))$$

and by that, we can show soundness.

2) *Our Abstractions*

In our abstract interpreter, we only abstract over references, Booleans and integers.

The abstraction lattice over references is rather simple and is shown in Figure 3.

$$\begin{tikzcd}
\{\top\} \\
\{ref\} \arrow[u] \\
\{\bot\} \arrow[u]
\end{tikzcd}$$

Figure 3. Reference-Abstraction Lattice

In our interpreter, all references correspond to a natural number, so the abstraction function α_R maps from $\mathbb{N}$ to our reference lattice.

$$\alpha_R(r) = \begin{cases} \bot, & \text{if r} = \emptyset \\ ref, & \text{if r } \sqsubseteq \{1,2,3...\} \cup \texttt{None} \\ \top, & \text{otherwise} \end{cases}$$

$$\gamma_R(r) = \begin{cases} \emptyset, & \text{if } r = \bot \\ \{1,2,3...\} \cup \texttt{None}, & \text{if } r = ref \\ \text{some reference}, & \text{otherwise} \end{cases}$$

The lattice used for the Boolean abstractions is also quite simple and is seen in Figure 4. It should be noted that in our abstract interpreter, we do not use Boolean variables, but the abstraction is used for conditions in if/else statements and loops.

$$\begin{tikzcd}
& \{\top\} & \\
\{T\} \arrow[ur] & & \{F\} \arrow[ul] \\
& \{\bot\} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 4. Boolean-Abstraction Lattice

The abstraction and concretization function for Boolean values very similar to the is a lot like the one for references, but instead of mapping from $\mathcal{Z}$ we map from Boolean values.

$$\alpha_{Bools}(b) = \begin{cases} \bot, & \text{if } i = \emptyset \\ F, & \text{if b only contains false values} \\ T, & \text{if b only contains true values} \\ \top, & \text{otherwise} \end{cases}$$

And the corresponding concretization function:

$$\gamma_{Bools}(b) = \begin{cases} \emptyset, & \text{if} s = \bot \\ \{F\}, & \text{if } s = F \\ \{T\}, & \text{if } s = T \\ \{T,F\}, & \text{otherwise} \end{cases}$$

We abstract over integers using sign abstraction. The lattice we use is seen in Figure 5.

$$\begin{tikzcd}
& \{\top\} & \\
\{-\} \arrow[ur] & \{0\} \arrow[u] & \{+\} \arrow[ul] \\
& \{\bot\} \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

Figure 5. Sign-Abstraction Lattice

We use the abstraction function α_{Int} that maps from the lattice over Integers to the sign-lattice:

$$\alpha_{Int}(i) = \begin{cases} \bot, & \text{if } i = \emptyset \\ -, & \text{if } i \sqsubseteq \{-1,-2,-3,...\} \\ 0, & \text{if } i = 0 \\ +, & \text{if } i \sqsubseteq \{1,2,3...\} \\ \top, & \text{otherwise} \end{cases}$$

And the corresponding concretization function:

$$\gamma_{Int}(s) = \begin{cases} \emptyset, & \text{if} s = \bot \\ \{-1, -2, -3...\}, & \text{if } s = - \\ 0, & \text{if } s = 0 \\ \{1, 2, 3...\}, & \text{if } s = + \\ \mathbb{Z}, & \text{otherwise} \end{cases}$$

We see that all the pairs of abstraction and concretization functions form Galois connections.

We will now argue that our analysis will produce an overestimate, i.e., is sound, by using the proposed abstractions.

We know that all lattices we use for our abstraction $\mathcal{A}$ are complete and of finite height, and as we just mentioned, the abstraction and concretization functions form Galois connections.

Therefore, to ensure that our analysis is sound, we need to make sure that all functions, both abstract and the corresponding concrete function, are monotone, and that all abstract functions are sound abstractions of their concrete counterparts.

Then, following Section II-B1, we can prove that the analysis is sound.

Since our abstract interpreter interprets JVM bytecode, we only have to make sure that all operations we use are a sound abstraction of the concrete bytecode operations.

To do this, we follow the examples from Table I for the relevant cases, but we will not display every case here. Since we only support a limited set of operations, it is fairly manageable to make sure every operation is a sound abstraction.

$+$	$\top$	$+$	0	$-$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$
$+$	$\top$	$+$	$+$	$\top$	$\bot$
0	$\top$	$+$	0	$-$	$\bot$
$-$	$\top$	$\top$	$-$	$-$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

$\times$	$\top$	$+$	0	$-$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$
$+$	$\top$	$+$	0	$-$	$\bot$
0	$\top$	0	0	0	$\bot$
$-$	$\top$	$-$	0	$+$	$\bot$
$\bot$	$\bot$	$\bot$	0	$\bot$	$\bot$

$=$	$\top$	$+$	0	$-$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$
$+$	$\top$	$\top$	F	F	$\bot$
0	$\top$	F	T	F	$\bot$
$-$	$\top$	F	F	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

$\neq$	$\top$	$+$	0	$-$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$
$+$	$\top$	$\top$	T	T	$\bot$
0	$\top$	T	F	T	$\bot$
$-$	$\top$	T	T	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

$<$	$\top$	$+$	0	$-$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$
$+$	$\top$	$\top$	F	F	$\bot$
0	$\top$	T	$\top$	F	$\bot$
$-$	$\top$	T	T	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

$\leq$	$\top$	$+$	0	$-$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$
$+$	$\top$	$\top$	F	F	$\bot$
0	$\top$	T	T	F	$\bot$
$-$	$\top$	T	T	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Table I
CALCULATIONS FOR $S = \{\top, \bot, +, -, 0\}$

III. Implementation and Setup

We have implemented three parts, all of them using the Python programming language:

1) Interpreter
2) Syntactic analysis
3) Semantic analysis (abstract interpreter)

The source code can be found in this GitHub repository. The project code is all within the `project/` directory.

A. The Interpreter

The interpreter only works on a subset of Java, it only supports operands of type `int`/`integer` and `ref`. It runs on JVM bytecode in a JSON format and consists of two main functions which are called recursively. The first one takes specifications for a function as input, finds the function, and extracts an array of bytecode, which is then interpreted by the other function. This is done by going through the array of bytecode recursively and acting on each element, by matching on the operand, and increasing the index for each call, unless an operand specifies an index. When encountering an invoke operation, the first function is called with the specified function.

The Interpreter works using an operand stack, a call stack and a heap. The operand stack is the local stack that is used for push, pop and local calculations. The call stack is the local variable array, constructed as a Python dictionary, since we want to access it by index, and contains the parameters that are passed to a function. The heap is the long-term storage and contains all objects. It is also constructed as a Python dictionary where the pointers are keys.

B. Syntactic analysis

The syntactic analysis takes as a starting point an Abstract Syntax Tree produced by the tool Tree-sitter [6].

We are then traversing this tree structure in a depth-first fashion, and saving relevant information into corresponding data structures. Performing this on all Java source code files of the project, we obtain a collection of important information about classes, methods, variables, and invocations present in the source code. After having this data collected, an algorithm will match the invocations to the methods declared on the actual classes, ensuring that they exist.

To display the actual edges, the user has to select a specific function. The program will get information on this function, and all its subsequent invocations.

C. Abstract interpreter - Semantic analysis

The abstract interpreter is built the same way as the interpreter, but we abstract over elements. Since we only support integers and references, we only need to abstract over these and Boolean values since we want to support if/else statements and loops. For this, we use the abstraction functions defined in Section II-B2. In if/else statements where the boolean abstraction is $\top$, i.e. it can either be true or false, we execute both branches of the conditional statements. In while loops we do something similar, when the condition is potentially true and false we execute the loop once with the most precise abstraction possible.

IV. Evaluation and Results

To answer our research question, we must assess the precision, speed and complexity of the two analyses under study. In the following sections, we will delve into the details of each of these different evaluation categories.

A. Accuracy

To evaluate the accuracy of our static solutions, we compared the results towards those produced by the semantic analysis – the interpreter. We assume the results from the interpreter to be our ground truth – the absolute correct values.

Each edge in our results can be considered to be one of the following types:

- **True positive** – the graph edge is present both on the true graph and on the tool's graph.
- **False positive** – when the tool identifies an edge that is not present on the true graph.
- **False negative** – when the true graph has an edge that the tool didn't identify.

We can also enumerate a fourth type of outcome: **true negatives**. This would correspond to the edges that weren't identified both by the true graph and by the tool. For our problem, it doesn't make sense to use it, since we only identify edges, and we don't keep track of how many *non-edges* we have.

To evaluate the validity of our results, we decided on computing the **precision** and **recall** metrics for our tools. We also combined these two metrics in what is known as the **f-score**.

- **Precision**
  Precision is a measure of how exact – or close – the obtained results are to the correct values. It can be computed by Equation 1.

$$precision = \frac{true\ positives}{true\ positives + false\ positives} \tag{1}$$

- **Recall**
  Recall can be considered as a rate of the true positive values. It measures how well the edges are identified, and is calculated by Equation 2.

$$recall = \frac{true\ positives}{true\ positives + false\ negatives} \tag{2}$$

- **F-score**
  Neither precision nor recall are useful on their own to assess the accuracy of results. The F-score is the harmonic mean of precision and recall and is commonly used to assess the accuracy of the results. It can be computed by Equation 3.

$$Fscore = \frac{precision * recall}{precision + recall} \tag{3}$$

Applying these equations to our results for our example project, we obtain the results in Table II.

	Syntactic Analysis	**Semantic Analysis**
Precision	0.4231	0.6957
Recall	0.5789	1.0000
F-score	0.4889	0.8205

Table II
Precision, recall, and F-score from syntactic and semantic analysis.

From Table II, we observe that the precision varies significantly between the syntactic and semantic analyses, with values of 0.423 and 0.6957, showing that the semantic analysis is 64% more precise than the syntactic one. This is due to the fact that the AST is generated from the source code, and it represents what the source code contains. It doesn't look at how the program is run, and it creates an over-approximation of the program. Hence the precision is affected negatively.

Another observation regarding the measurements pertains to recall. The recall from the syntactic analysis is not as high as the recall from the semantic analysis, i.e., 0.5789 and 1.000, respectively, which varies by approximately 72%. Recall is a measure to identify the true positive rate, so this indicates that the semantic analysis is not a true over-approximation. This, however, is because the implementation of the syntactic analysis is missing some edge cases that our semantic analysis picks up. This is most likely because the constructor of the `java.lang.Object` class invocation in the bytecode generated from *Jvm2Json* goes unnoticed by the syntactic analysis, as it does not operate on byte-code, but directly on Java source code.

Finally, the harmonic mean of precision and recall, called F-score, is 0.4889 and 0.8205 for syntactic and semantic analysis respectively. It symmetrically represents both precision and recall in one metric, and the closer this measurement is to the value 1, the more balanced are precision and recall. The F-score for the semantic analysis is higher than the one for the syntactic analysis by approximately 67%, representing a better performance for the semantic analysis.

B. Speed

To measure the performance, or speed, of the tools, we used the `timeit` Python tool [11]. The two tools were one right after the other, on the same computer. Even though testing performance in only one computer isn't very reliable, we tried to keep testing conditions as stable as possible.

The tool allowed us to measure the time it took to run each tool 50 times, providing us with this value. We ran this procedure 100 times, obtaining a list with 100 values. By analysing the data we obtained the average values and standard deviation shown in Table III, in milliseconds. The time corresponds to the average running time of one execution of the tool.

The values show that the semantic analysis takes about 27% longer to conclude its analysis. Considering that the program under analysis doesn't have a size comparable to a real-world application, it's difficult to use this value to conclude about

Syntactic Analysis	Semantic Analysis
$6.897 \pm 0.369ms$	$8.778 \pm 0.405ms$

Table III
MEANS AND STANDARD DEVIATION OF SYNTACTIC AND SEMANTIC ANALYSIS.

the actual speed difference between the tools. However, because the semantic analysis involves more complex logic and structures, we estimate that it will have a worse performance in almost all cases.

C. Complexity

When looking at complexity we look both at the tools used for the analysis and the complexity and scalability of the theory behind each type of analysis.

We could also look at the size and line count. This, however, is not representative of the complexity, since the coding style used for the 2 tools is different and would therefore not give any insight into how it would differentiate them.

The semantic analysis works on a parse tree generated by the Tree-itter, utilizing Java source code as input. The abstract interpreter relies on *Jvm2Json*, a tool that transforms Java bytecode into JSON. So for the abstract interpreter, we both need a Java compiler to compile the Java code to JVM bytecode and the tool *Jvm2Json* for the conversion. Hence, the abstract interpreter can be seen as more complex due to its involvement in additional steps compared to the semantic analysis.

If we look at the theory behind the tools, then the static analysis mainly relies on parsing trees, whereas the abstract interpreter relies on lattice theory and on what promises you can make when using those. However, it is not feasible to compare those methods and say which one is the most complex. We can instead argue that the syntactic analysis works on most of the Java language since this is supported by Tree-sitter and that it is able to parse most parse trees constructed by this. Whereas the abstract interpreter only works on a small subset of Java, it is not very easy to extend since we would both have to define lattices primitive type in Java and implement the bytecode operators that are not yet implemented in the interpreter. This then leaves the syntactic analysis as the least complex tool which is preferable for development and maintenance.

V. Discussion and Alternatives

The syntactic analysis is simpler, faster, and far more scalable than the semantic analysis, but it does not yield as precise results as the semantic analysis nor does it provide such good guarantees.

However, the abstract interpreter, used for the semantic analysis can produce as imprecise results as the syntactic analysis, depending on the lattice used. For example, the lattice we use for abstracting objects (see Figure 3) is very simple, and if every abstraction was used with a lattice that simple, the abstract interpreter would not yield better results than the syntactic analysis, and in that case, the syntactic analysis would be a better choice, due to it being faster and more scalable. However, when using an abstract interpreter for analysis, it could initially be a good idea to make every abstraction as simple as the one seen in Figure 3, and then later only extend the lattices to get a more precise result. This way it is possible to fine-tune the abstraction in such a way that it fits the objective of the analysis. So even though the abstract interpreter isn't as easy to implement for a whole programming language as the syntactic analysis we used, there is a much higher possibility of fine-tuning it to get as close to the truth as possible.

As mentioned before, the abstract interpreter we have used for the semantic analysis only works on a small subset of Java. We only support objects and integers and only 19 of the 100+ bytecode operations in JVM. But even so, the semantic analysis should still be able to give a far more precise picture on the whole Java language than the syntactic analysis.

Another approach could have been Concolic execution which is a type of dynamic analysis, that tries to execute every possible branch of the program. It is a mixture of concrete and symbolic execution and the idea is to systematically explore every path in the program, using symbolic values for concrete executions of the program. A SMT solveris usually used to generate new inputs to ensure that all paths get executed and explored. This way every executable path in a program is executed, and therefore it gives a very precise estimate of what can actually happen in a program. Concolic execution is very usefull and can potentially be very efficient for finding bug, but it is still limited since it takes up a lot of computing power to explore the large number of paths one can execute in a program [12]

Using Concolic execution we could have gotten a very precise estimate of what functions are called. However, concolic executions, being a dynamic analysis, will produce an underestimate of what functions are actually called. Since our goal is to produce a call graph to find unreachable code this would not be suitable, since it could potentially create a call graph with fewer functions than what is actually callable, and this would defy the purpose of the analysis as we want to know what functions are most definitely not called.

So, in order to improve our implementation, it would be most obvious to first extend the number of types and operations our abstract interpreter supports, that way we could get a better picture of how much the 2 different approaches differ.

VI. Conclusion

In this paper, we studied a syntactic and a semantic approach for finding dead code in Java programs. We see that our syntactic tool that analyses parse trees is a lot faster than the semantic approach but it does not yield a result that is as precise as the semantic analysis. The result from the abstract interpreter is a lot more precise, and despite the slower runtime, it is preferable to the syntactic approach due to it being significantly closer to the actual truth.

References

[1] G.C. Murphy, D. Notkin, and E.S.-C. Lan. An empirical study of static call graph extractors. In *Proceedings of IEEE 18th International Conference on Software Engineering*, pages 90–99, 1996.

[2] William Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. Antipatterns: Refactoring software, architectures, and projects in crisis. 1998.

[3] Simone Romano, Christopher Vendome, Giuseppe Scanniello, and Denys Poshyvanyk. A multi-study investigation into dead code. *IEEE Transactions on Software Engineering*, 46(1):71–99, 2020.

[4] Sebastian Eder, Maximilian Junker, Elmar Jürgens, Benedikt Hauptmann, Rudolf Vaas, and Karl-Heinz Prommer. How much does unused code matter for maintenance? In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1102–1111, 2012.

[5] Anders Møller and Michael I. Schwartzbach. Static program analysis. Department of Computer Science, Aarhus University, https://cs.au.dk/~amoeller/spa/spa.pdf, October 2018.

[6] Tree-sitter github page. https://tree-sitter.github.io/tree-sitter/. Accessed on 21 October 2023.

[7] Christian Gram Kalhauge. Jvm2json project. https://gitlab.gbar.dtu.dk/chrg/jvm2json. Accessed on 21 October 2023.

[8] A. Jayanthila Devi P. S. Aithal Vaikunta Pai T. A systematic literature review of lexical analyzer implementation techniques in compiler design., 2020.

[9] Tao Guo Jianxin Wang Ding Mal Baojiang Cui, Jiansong Li. he expressive power of the statically typed concrete syntax trees, 2021.

[10] John C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill Higher Education, 2nd edition, 1997.

[11] Timeit module python. https://docs.python.org/3/library/timeit.html. Accessed on 21 October 2023.

[12] Bing Chen, Qingkai Zeng, and Weiguang Wang. Crashmaker: An improved binary concolic testing tool for vulnerability detection. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, page 1257–1263, New York, NY, USA, 2014. Association for Computing Machinery.