Lessons Learned from Implementing a Language-Agnostic Dependency Graph Parser

Francesco Refolli[1], Darius Sas[2] and Francesca Arcelli Fontana[1]
[1]*Università degli Studi di Milano-Bicocca, Italy*
[2]*TXT Arcan, Italy*

Keywords: Source Code, Software Analysis, Dependency Graphs.

Abstract: In software engineering, automated tools are essential for detecting policy violations within code. These tools typically analyze the relationships and dependencies between components in large codebases, which may be written in various programming languages. Most available tools, whether free or proprietary, rely on third-party software to perform statistical analyses. This approach often requires a separate tool for each programming language, which can lead to high maintenance efforts, and even relying on a standardized technology such as Language Servers has several drawbacks. This paper investigates the feasibility of removing language-specific dependencies in the construction of dependency graphs by using two libraries: Tree Sitter and Stack Graph. After analyzing the capabilities of these technologies, their application in this context is demonstrated, and the effectiveness and accuracy of the proposed solution are evaluated.

1 INTRODUCTION

Software development is a long, complex process that is often prone to errors and failures. There are countless cases in which software becomes so complex to develop and maintain that it is abandoned in favor of a new implementation. Different anomalies at the design and implementation levels can be taken into account to tackle technical debt (Avgeriou et al., 2016), such as code (Fowler and Beck, 2002) and architectural smells (Azadi et al., 2019), and different metrics and technical debt indexes can be computed to evaluate software quality (Avgeriou et al., 2021). There are tools able to detect both *code* and *architectural smells*, indicators of the presence of anomalies in code, architecture, and design through the analysis of the dependencies of functions, packages, classes, and other types of code components, but these tools rely on language-specific software to obtain such data. The main aim of this paper is to investigate whether it is possible to use two emerging technologies, Tree Sitter (Latif et al., 2023) (a parser generator tool) and Stack Graph (Creager and van Antwerpen, 2023) (a graph-based representation for reference resolution), to build a framework able to analyze source code written in different arbitrary programming languages while keeping unchanged and simple the core of an analysis tool and without adding more dependencies to its platform. With this aim a prototype of such framework written in Rust has been developed and compared with an existing tool known in the field, called Arcan. Arcan was developed for architectural smell detection and technical debt computation through source code static analysis (Sas et al., 2019). We decided to consider this tool as a reference for the new implementation since Arcan represents a significant example of *architectural analysis* tool, which can build and serialize the dependency graph from the source code. The tool identifies the architectural smells by exploiting the dependency graph it builds as well as various metrics and the Technical Debt computation of a project (Sas and Avgeriou, 2023), (Roveda et al., 2018). In this paper, we will generate dependency graphs from code and compare the produced artifact with the graph that can be exported from Arcan analysis. Figure 1 shows the dependency graph model used by Arcan. Each component is divided into three node types: unit, function and container, which correspond intuitively to classes/interfaces, functions and packages/namespaces. It was designed to represent the relationships of inheritance, inclusion, implementation, and more generic *dependency*. The dependency edge represents all possible types of dependencies related to the usage of types, access to fields, and other rela-

Refolli, F., Sas, D. and Fontana, F. A.
Lessons Learned from Implementing a Language-Agnostic Dependency Graph Parser.
DOI: 10.5220/0013277600003928
In *Proceedings of the 20th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2025)*, pages 484-491
ISBN: 978-989-758-742-9; ISSN: 2184-4895

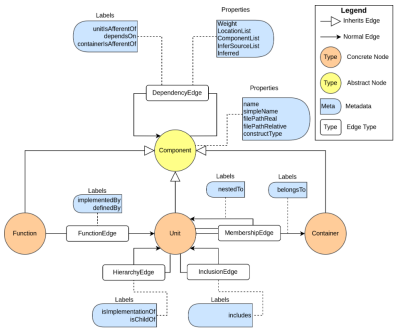

Figure 1: Dependency Graph Model built by Arcan.

tionships[1].

This study represents a first step towards the ability to perform analysis on heterogeneous code bases. This would lead to more abstract, general, and complex representations and algorithms, spanning from software quality analysis to memory safety checks or some advanced type checking of projects involving multiple languages.

2 RELATED WORK

In the software engineering literature, many studies and tools make use of dependency graphs for their analysis, but most of them either assume to start with a complete graph or use some *language frontends* to analyze the code passed as input, which is the mode of operation of Arcan, while few of them attempt to create a generic framework capable of extracting metrics and features from the source code. A technique used frequently is the translation of the input source code into an intermediate representation: from by-hand transformations into a domain-specific language (Marin and Rivero, 2018), to automated abstract syntax tree translations into a language-independent abstract syntax tree (Weiss and Banse, 2022), into metamodels like Famix (Ducasse et al., 2011) or into intermediate markup formats (Collard et al., 2013). Other lower-level approaches use language frontends to translate source code written in different languages into a common intermediate representation language like GIMPLE IR or LLVM IR (Grech et al., 2015), but those are often applied in compiler stages or similar code level tasks, therefore those IRs loose details of high-level code organization. The novel approach presented in this paper aims to create an efficient intermediate representation of the code starting from the concrete syntax tree without translating it into a language-independent syntax tree.

3 THE ARCAN SOLUTION

Arcan analyzes a software project by parsing its source code to build a dependency graph relying on several different third-party software (TPS): for each supported programming language, it needs a middleware that can parse the source code in advance to extract symbols and their references. The developers of Arcan and similar tools aspire to maintain and expand the support to several programming languages, which means integrating a multitude of TPS, namely, *at least* one for each new programming language they want to support. This architecture comes with several downsides. A suitable TPS for a specific programming language may not exist or if it does it may be deprecated, with little support by the community, it may lack some essential features or it may require an excessive amount of work to be integrated within Arcan. Additionally, some of these TPS may not satisfy the minimum requirements concerning *performance*. Indeed, for some programming languages (e.g. Java) there exist frameworks (e.g. Spoon, JDT) that provide efficient static analysis of Java source code. However, this may not be true for other languages (e.g. C#), where an appropriate library is completely missing and we are forced to rely on a language server [2] instead, which can be heavyweight and lacks an efficient method to get the outgoing dependencies of components, as the protocol is restricted to incoming dependencies only.

The current solution of having multiple TPS to process each programming language hinders the horizontal scalability of tools like Arcan, and therefore we are seeking a solution that is scalable, stable, and fast.

The Requirements for a New Solution. One of our main concerns is *maintainability*. A new method has to use the smallest possible number of third-party libraries, which should operate as part of the global strategy, therefore not limited to some languages or features, and should be the basis of an abstraction layer that allows us to have a unified model for all source languages. This is important whenever a tool such as Arcan has to change due to mutated needs in terms of how the graph has to be built, thus limiting the number of submodules that the developers have

[1]The reference enumeration of dependencies considered by Arcan can be found at http://doi.wiley.com/10.1002/spe.2421

[2]The Language Server protocol is used between a tool (the client) and a language smartness provider (the server) to integrate features like auto complete, go to definition, find all references and alike into the tool

to correct. It is also worthy of notice that sometimes third-party software introduces breaking changes, the more libraries you use, the more time will be spent on correction of dependencies. And even when relying on a small set of libraries, it is important that the addition of support for one programming language does not imply any task harder than understanding how a language is structured in terms of semantics (core components definable, file inclusion method, etc) and syntax (how components are encapsulated in the code). It is preferable if the integration of a language is made easier by isomorphism with other languages' structures. A more practical but key aspect is certainly performance: the new approach should be faster and lighter than the current implementation. If a tool needs to analyze a large codebase with millions of lines of code or multiple versions of a project to study its evolution over time, it is vital that source code analysis does not become a bottleneck.

4 A LANGUAGE INDEPENDENT APPROACH

4.1 Core Technologies

After a careful analysis of the state of the art in the field of Lexical Analysis (DeRemer, 1976), two promising Rust libraries have emerged and seem suitable for our research purpose: Tree Sitter e Stack Graph.

Tree Sitter (Latif et al., 2023) is a library for generating parsers that yield Concrete Syntax Trees, which contain the expansion of the language grammar used while parsing. **Stack Graph** (Creager and van Antwerpen, 2023) is a new data structure presented as an evolution of the Scope Graph (Zwaan and van Antwerpen, 2023) which allows to resolve references in the code and is promoted as suitable for vertical scaling since it builds large graphs incrementally, file by file. In order to automate the construction of Stack Graphs starting from the source code, the developers of Stack Graph have also implemented *Tree Sitter Stack Graph*, which requires for each language the fabrication of a *"grammar"* containing pairs of declarations that associate a piece of Concrete Syntax Tree (expressed as S-Expressions queries) to a block of procedural code that executes the actual instructions that build the graph.

4.2 A General Model

Since the Concrete Syntax Tree reflects the nature of a language's syntax, we cannot define a single set of rules for all languages. Therefore we developed a general model to simplify the creation of Tree Sitter Stack Graph grammars and to better unify algorithms and data structures that will be used to build the dependency graph.

4.2.1 Nodes

Generally, a Tree Sitter query language introduces atomic nodes for identifiers (ex: in Java *(identifier)* and *(type_identifier)*), while the nodes that concatenate multiple identifiers, such as *qualified name* and type declarations, recursively contain themselves or the atomic nodes. Since reference resolution in Stack Graph is done by maintaining a stack of symbols (*Symbol Stack*) initially empty and populated by encountering *push* or *pop* nodes, which instruct the solver to push or pop a symbol on the stack, we assigned such roles to Stack Graph nodes representing chains of identifiers. A *refkind* or *defkind* marker is also added to the ending of the identifier chain to identify references and definitions along with their reference/construct type.

By composing chains of *push* and *pop* nodes, we can resolve qualified names: the concatenation of *push* nodes X and Y with a direct edge will find a *pop* node for symbol Y and then a pop node for symbol X. More complex searches are implementable, such as type resolution: with an appropriate chain, we can jump from the reference *point.x* to the variable declaration of *point*, to the structure declaration of *Vec3D* and then to its field x. This technique is often called *ScopesAsTypes* (van Antwerpen et al., 2018), but the general idea of node chaining will go under the name of *resolution bridge*.

4.2.2 Optimization of the Graphs

When dealing with import statements we want to reduce the amount of useless jumps within a search. For example, instead of providing a direct edge to a bridge that would push "utils" and "org" to the stack, we can add a preventive couple of pop-push nodes to filter in only those paths which have on the stack the symbol "Tool". This technique could be applied to all the identifiers, but it is limited to the first node of a chain for simplicity and to reduce the size of the graph, which, as we will see, is already of considerable size.

Furthermore, in order to reduce the number of paths created during the exploration, we need to reduce the number of cycles in the graph, although the graph cannot be completely acyclic since it must be possible during the search to return to the root node of the graph to allow it to descend in the next subtree

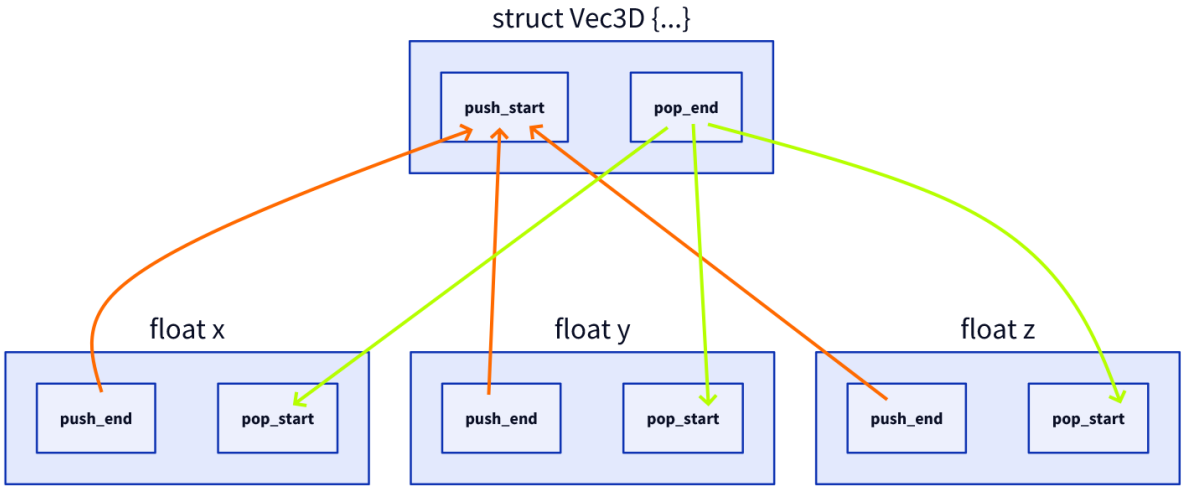

Figure 2: Example of Unit Composition.

to explore and find next definitions of the *resolution bridge*.

4.2.3 Composition

We further organized Stack Graph nodes in *units*, which are the building blocks of the graph and are defined with five fields. Each unit has a local root node called *scope*, which connects it in the direction of the global root. The *push_start/end* nodes define the upward path used to resolve references (ex: class's package), while the *pop_start/end* nodes represent the path for defining the unit (ex: class' body). These fields can either contain concrete nodes or references (in variables) to nodes defined elsewhere (in order to combine and concatenate these units). An example of composition of units is shown in Figure 2 where a unit for *Vec3D* defines a scope for its fields.

An implementation of *Tree Sitter Stack Graph* grammar for Java is given in the replication package of our prototype implementation [3].

4.3 Building the Dependency Graph

Before the dependency graph can be built, all references inside the Stack Graph should be resolved. Using Stack Graph built-in resolution algorithm, we can process every *refkind* marked node and if a match is found, this pair is saved inside the *reference map* [4]. Unfortunately, some references may not have been resolved because it is too complex (require too many jumps inside the Stack Graph) or the symbol could not be found in the code base (e.g. third-party libraries or standard library).

Then our algorithm explores such a graph like a tree from the global root, avoiding cyclic paths by storing a map of already visited nodes. During this step the *definition map* [5] is built associating every *defkind* marked node in the Stack Graph with its component node in the Dependency Graph. This cache also accommodates the semantics of some languages, like *C/C++*, which allow a component to be defined or declared multiple times. A pointer to the parent dependency graph node is kept (initially null) in order to introduce the *definedBy* dependency. An extra check is done on the *kind* of the two definitions involved in a *definedBy* dependency edge: for consistency, if their kind is equal (e.g. struct *Display* defined inside struct *Computer*) another edge is used instead ("Display" *nestedTo* "Computer"). For every encountered *refkind* marked node, a dependency edge should be created if a pair $\langle referenceNode, resolvedNode \rangle$ exists in the *reference map*. Then a dependency edge is added between the *parent* node of the reference and the pointed component that we can find via the *definition map*. Markers *defkind* and *refkind* also guide us when defining edge labels or component types.

We implemented serialization for GraphML (Brandes et al., 2013) to compare the dependency graphs produced by Arcan and our prototype. In Figure 3 we show an example of a dependency graph produced with the execution of the prototype on a trivial piece of code.

[3] https://doi.org/10.5281/zenodo.13982944

[4] $reference\ map : StackGraph.Node \rightarrow StackGraph.Node$

[5] $definition\ map : StackGraph.Node \rightarrow DependencyGraph.Node$

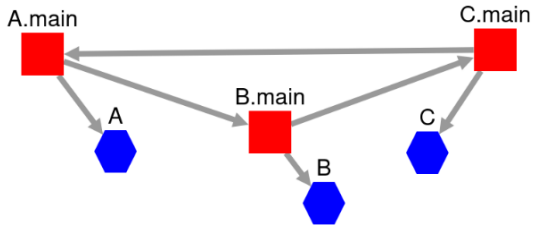

Figure 3: Example of Dependency Graph.

5 BENCHMARK TESTS

A dual testing strategy has been applied: some unit tests for dependencies and edge cases were created to ensure that the language features have been implemented correctly and a functional benchmark has been used in order to understand which dependencies (of which complexity) can or cannot be resolved with Arcan and our prototype.

5.1 A Custom Testing Framework

During the development of the grammar for Java and the algorithm for building the dependency graph, we needed a way to test single features against source code to ensure they were correctly recognized and collected as well. Following this idea, a lightweight testing framework has been embedded inside the command line interface of the prototype. Given a list of source files and a target language, two lists of tuples are specified as expected results of test cases (referred to as *policies*): nodes and edges that must be present in the dependency graph built upon these source files. Nodes are always enumerated by their fully qualified name, in both those lists. As edges and nodes in the dependency graph have attached a *kind* attribute which identifies the type of construct or relationship they represent, we also test that value to be correct.

In Table 1 we outline the content of all tests implemented with a flag indicating if the *policy* is satisfied. Some tests verify if a feature has been implemented correctly, while others if there are conflicts between detection features.

5.2 Comparing Arcan and the Prototype

Arcan uses the Pruijt et al's (Pruijt et al., 2017) benchmark to evaluate its precision, which consists of a wide code base that implements various kinds of dependencies, conveniently listed in a table. In order to check those dependencies against our prototype we decided to use the previously introduced embedded framework and configure a test according to this table, adapting entries to the way it recognizes dependencies. A key difference is that while Arcan attaches dependencies only to top-level classes (i.e., "components"), the prototype associates them with actual parent components. This includes all types of subordinate objects, such as subclasses and methods.

Table 1: Results of unit and integration tests for Java language.

Test Name	Dependencies	Is It Recognized?
implementation_bridge	calls	✓
class_methods_with_parameters	definedBy	✓
class_implementation	isImplementationOf	✓
class_with_attributes	definedBy	✓
class_methods_with_attributes	definedBy	✓
class_method_call	calls	✓
type_inference	calls, accessField	✓
enums	definedBy, accessField	✓
class_inheritance	isChildOf, nestedTo	✓
object_creation	usesType	✓
class_type_usage_nested_packages	definedBy, usesType	✓
nested_classes	nestedTo	✓
casts_type	castsType	✓
class_type_usage	definedBy, usesType	✓
throws_type	throwsType	✓
imports	definedBy, includes, usesType	✓
interface_inheritance	isChildOf	✓
type_inference_with_packages	calls, accessField	✓
extension_bridge	calls	✓
class_constructors	definedBy	✓
class_field_access	accessField	✓
class_packages	definedBy, includes	✓
annotation	usesType	✓
array_creation	usesType	✓
class_inheritance_with_packages	isChildOf	✓
class_methods	definedBy	✓

Both our prototype and Arcan were run against the benchmark source code, and the identified dependencies are recorded in Table 2. In the table, a mark indicates whether each dependency was correctly resolved. The prototype correctly detects only 22 of the 32 total dependencies present in the benchmark, compared to the 24 detected by Arcan. However, it should be noted that with the prototype, the dependencies do not belong to a generic "dependsOn" dependency (which happens with Arcan), each label is descriptive of the dependency that exists between the two components.

Finally we decided to launch Arcan and the prototype in a free analysis against various real world projects, which are enumerated with version and size (computed with *cloc* [6] in Table 3. We measured both the execution time and the similarity (using Jaccard's index) of the generated graphs, which were carefully adapted into a common format due to the previously mentioned differences. Specifically, dependencies attached to child components in the prototype's graph were simplified to match the top-level component dependencies in Arcan's graph.

As we can see in Table 4, contrary to our expectations the prototype is much slower than Arcan, and

[6] **cloc** https://github.com/AlDanial/cloc

Table 2: Benchmark results of the prototype and Arcan.

Source	Target	Dependency	Prototype	Arcan
Access				
AccessClassVariable	CheckInDAO	accessField	✓	✓
AccessClassVariableConstant	UserDAO	accessField	✓	✓
AccessClassVariableInterface	ISierraDAO	accessField	✓	✓
AccessEnumeration	TipDAO	accessField	✓	✓
AccessInstanceVariableRead	ProfileDAO	accessField	✗	✓
AccessInstanceVariableWrite	ProfileDAO	accessField	✗	✓
AccessInstanceVariableConstant	UserDAO	accessField	✗	✓
AccessInstanceVariableSuperClass	CallInstanceSuperClassDAO	accessField	✗	✗
AccessInstanceVariableSuperSuperClass	CallInstanceSuperClassDAO	accessField	✗	✗
AccessObjectReferenceAsParameter	Base	accessField	✓	✗
AccessObjectReferenceWithinIfStatement	Base	accessField	✓	✗
Annotations				
AnnotationDependency	SettingsAnnotation	usesType	✓	✓
Call				
CallClassMethod	BadgesDAO	calls	✓	✓
CallConstructor	AccountDAO	usesType	✓	✓
CallInstance	ProfileDAO	calls	✗	✗
CallInstanceInnerClass	CallInstanceOuterClassDAO	calls	✗	✗
CallInstanceInterface	CallInstanceInterfaceDAO	calls	✗	✗
CallInstanceSuperClass	CallInstanceSuperClassDAO	calls	✗	✓
CallInstanceSuperSuperClass	CallInstanceSuperClassDAO	calls	✗	✓
Declaration				
DeclarationExceptionThrows	StaticsException	throwsType	✓	✓
DeclarationParameter	ProfileDAO	usesType	✓	✓
DeclarationReturnType	VenueDAO	usesType	✓	✓
DeclarationTypeCast	ProfileDAO	castsType	✓	✓
DeclarationTypeCastOfArgument	ProfileDAO	castsType	✓	✓
DeclarationVariableInstance	ProfileDAO	usesType	✓	✓
DeclarationVariableLocal	ProfileDAO	usesType	✓	✓
DeclarationVariableLocal_Initialized	ProfileDAO	usesType	✓	✓
DeclarationVariableStatic	ProfileDAO	usesType	✓	✓
Import				
domain.direct.violating	AccountDAO	includes	✓	✗
Inheritance				
InheritanceExtends	HistoryDAO	isChildOf	✓	✓
InheritanceExtendsAbstractClass	FriendsDAO	isChildOf	✓	✓
InheritanceImplementsInterface	IMapDAO	isImplementationOf	✓	✓

Table 3: Summary of tested codebases.

Project	Language	Version	Size (in LOC)
JUnit4	Java	4	30K
JUnit5	Java	5	100K
ANTLR	Java	4	180K
Fastjson	Java	1	50K

Table 4: Execution time (in seconds) of both the prototype and Arcan.

Project	Tool	Min	Max	Mean execution time
JUnit4	prototype	65,82	69,07	65,88
JUnit4	Arcan	13,850	17,079	14,611
JUnit5	prototype	132,87	134,75	134,44
JUnit5	Arcan	42,542	47,613	44,186
ANTLR	prototype	171,65	175,49	172,64
ANTLR	Arcan	19,222	20,140	19,691
Fastjson	prototype	N/A	N/A	N/A
Fastjson	Arcan	66,932	71,094	69,071

in some cases, it does not terminate in an acceptable time. We also probed the similarity of the dependency graphs being built by computing the Jaccard Similarity Index [7] for the projects where the analysis with our prototype terminated. Results can be seen in Table 5. Not surprisingly the graphs produced are very divergent in both nodes and edges that have been added.

[7]https://www.nature.com/articles/234034a0

Table 5: Graphs analysis with Jaccard index.

Project	Jaccard(nodes)	Jaccard(edges)	Jaccard(graph)
junit4	37,26%	23,19%	25,42%
junit5	47,41%	15,13%	19,73%
antlr4	82,18%	44,25%	48,43%

An explanation emerges recalling what has been said before: our prototype attaches dependency edges to the actual components which are involved, while Arcan attaches almost all of them to the class in which they are contained.

6 DISCUSSION AND LESSONS LEARNED

Combining *Tree Sitter* and *Stack Graph* allows to support multiple programming languages while keeping the core elaboration algorithms unchanged. However, some critical issues have emerged making this solution still difficult to implement, but different lessons and hints emerged from this work.

The implemented dependency extraction tool achieves decent precision, comparable to Arcan, but faces noteworthy scalability challenges. First, the Stack Graph generated from the codebase is excessively large due to the integration of numerous scopes

at various levels. This layered composition is necessary for accurate reference identification, accommodating the flexible naming conventions typical of programming languages. Additionally, while Tree Sitter supports incremental updates to the Concrete Syntax Tree by processing only the changes between source code versions, Stack Graph lacks this capability, requiring a complete re-analysis of the source code with every new change. Furthermore, the prototype currently does not distinguish between references to standard or third-party libraries and components defined within the codebase, leading to inefficient graph exploration and increased memory consumption. Stack Graph's Domain Specific Language is well-designed and provides developers with all the tools they may need to process the code. However, a fundamental shortcoming is the lack of a function in the domain-specific language that allows to establish whether a node has an attribute or the value of a certain attribute makes it impossible to avoid conflicts between multiple types of references. The grammar can be designed to minimize these issues as much as possible, but depending on the language being analyzed, some conflicts may be inevitable, and certain features might need to be partially implemented as a workaround.

A further issue regards the impossibility of detecting some dependencies. Apart from external library references, there are also references (e.g. involving several jumps between the subgraphs representing different source files) in which simply the reference resolution algorithm is not powerful enough to solve them. Exploring a redundant graph is expensive and it is not even possible to increase the tolerance on cyclic paths since it would make the resolution failure of some third-party references more expensive. This last possibility has been explored but has not led to concrete results, given that most of the time the program runs out of RAM available to its process even with small codebases (such as the benchmark's one).

All these issues significantly impact memory usage and the speed of reference resolution, making this implementation suitable only for small to medium-sized projects.

7 THREATS TO VALIDITY

According to the internal threats to validity, the presented approach is based on a rather new technology that was applied only to GitHub code navigation feature. Since it is a novel method we had to create from scratch a strategy in order to work with it, keeping in mind the limitations of the domain specific language and of the stack graph resolution algorithm. The scheme of implementation for Tree Sitter Stack Graph could be non-optimal: different and more efficient schemes for implementing such artifacts could be found in the future. Still, the use of the Stack Graph library was optimized as much as possible through advanced constructs and parameter customization, with valuable assistance from one of its maintainers. Since the beginning of this study, the Stack Graph repository has been updated and partially rewritten to improve its API and name resolution algorithm.

According to the external threats to validity, although the implementation example for Java covers most of the constructs commonly used in programming languages, some language-specific features can require more complex solutions. In order to further prove the generalizability of this model, a draft implementation has been produced for other languages [8] with different code inclusion mechanisms (C/C++) or module naming rules (Python). In general, implementing a Tree Sitter Stack Graph grammar for a programming language has the same complexity as learning the structure of its concrete syntax tree and its semantics, which is surely easier than implementing from zero a source code analyzer (if a TPS library doesn't exist) since it would require at least same amount of effort and knowledge. Although certain languages may present challenges, it is likely that all constructs can be represented within a Tree Sitter Stack Graph grammar: for example, adding a component "File" is useful for having C-like code inclusion.

8 CONCLUSIONS

In this work, we have described the prototype that we implemented by exploiting two emerging technologies, Tree Sitter and Stack Graph. Results show that the current implementation performs decently for small to medium-sized projects, but further development is necessary to enhance scalability and improve accuracy. As outlined in Section 6, the main limits are represented by the reference resolution algorithm implemented by Stack Graph and the complexity of producing artifacts (*Tree Sitter Stack Graph* grammars) to build automatically a Stack Graph of the source code. However, the Stack Graph visit to build the dependency graph is linear-time and relatively fast, so it may be possible to use this Stack Graph to create an efficient intermediate representation. Since a reference can be seen as a chain of identifiers (a given string "a.b.c" can be tokenized with its delimiter as

[8] Grammars produced for this study can be found inside the directory "assets/tsg" of the replication package

[”a”, ”b”, ”c”]), the references can be resolved by exploring the component graph, treated as a prefix tree. The resolution process could start from the locality node of the reference (for example the function in which it is contained) and walk up the prefix tree until the current prefix identifier of the reference (the first identifier of the chain) is found as a child node of the current scope node. Then, a downward search can be carried out until all the identifiers of the reference are exhausted (each time a prefix identifier matches, it is removed from the array).

REFERENCES

Avgeriou, P., Kruchten, P., Ozkaya, I., and Seaman, C. B. (2016). Managing technical debt in software engineering (dagstuhl seminar 16162). *Dagstuhl Reports*, 6(4):110–138.

Avgeriou, P., Taibi, D., Ampatzoglou, A., Arcelli Fontana, F., Besker, T., Chatzigeorgiou, A., Lenarduzzi, V., Martini, A., Moschou, A., Pigazzini, I., Saarimaki, N., Sas, D. D., De Toledo, S. S., and Tsintzira, A. A. (2021). An Overview and Comparison of Technical Debt Measurement Tools. *IEEE Software*, 38(3).

Azadi, U., Fontana, F. A., and Taibi, D. (2019). Architectural smells detected by tools: a catalogue proposal. In *Proc, of the II Int. Conf, on Technical Debt, TechDebt@ICSE 2019, Montreal, Canada, 2019*, pages 88–97. IEEE / ACM.

Brandes, U., Eiglsperger, M., Lerner, J., and Pich, C. (2013). Graph markup language (graphml). In Tamassia, R., editor, *Handbook on Graph Drawing and Visualization*, pages 517–541. Chapman and Hall/CRC.

Collard, M. L., Decker, M. J., and Maletic, J. I. (2013). srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International conference on software maintenance*, pages 516–519. IEEE.

Creager, D. A. and van Antwerpen, H. (2023). Stack graphs: Name resolution at scale. In Lämmel, R., Mosses, P. D., and Steimann, F., editors, *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands*, OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

DeRemer, F. (1976). Lexical analysis. In Bauer, F. L. and Eickel, J., editors, *Compiler Construction, An Advanced Course, 2nd ed*, volume 21 of *Lecture Notes in Computer Science*, pages 109–120. Springer.

Ducasse, S., Anquetil, N., Bhatti, M. U., Cavalcante Hora, A., Laval, J., and Girba, T. (2011). MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Research report.

Fowler, M. and Beck, K. (2002). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Grech, N., Georgiou, K., Pallister, J., Kerrison, S., Morse, J., and Eder, K. (2015). Static analysis of energy consumption for llvm ir programs. SCOPES ’15, page 12–21, NY, USA. ACM.

Latif, A., Azam, F., Anwar, M. W., and Zafar, A. (2023). Comparison of leading language parsers – antlr, javacc, sablecc, tree-sitter, yacc, bison. In *2023 13th International Conference on Software Technology and Engineering (ICSTE)*, pages 7–13.

Marin, V. J. and Rivero, C. R. (2018). Towards a framework for generating program dependence graphs from source code. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*, SWAN 2018, page 30–36, New York, NY, USA. Association for Computing Machinery.

Pruijt, L., Köppe, C., van der Werf, J. M. E. M., and Brinkkemper, S. (2017). The accuracy of dependency analysis in static architecture compliance checking. *Softw. Pract. Exp.*, 47(2):273–309.

Roveda, R., Fontana, F. A., Pigazzini, I., and Zanoni, M. (2018). Towards an architectural debt index. In *Proceedings - 44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018*, pages 408–416. IEEE.

Sas, D. and Avgeriou, P. (2023). An architectural technical debt index based on machine learning and architectural smells. *IEEE Transactions on Software Engineering*, pages 1–27.

Sas, D., Avgeriou, P., and Fontana, F. A. (2019). Investigating instability architectural smells evolution: An exploratory case study. In *2019 IEEE Int.Conf. on Software Maintenance and Evolution, ICSME, OH, USA, 2019*, pages 557–567. IEEE.

van Antwerpen, H., Bach Poulsen, C., Rouvoet, A., and Visser, E. (2018). Scopes as types. *Proc. ACM Program. Lang.*, 2(OOPSLA).

Weiss, K. and Banse, C. (2022). A language-independent analysis platform for source code. *CoRR*.

Zwaan, A. and van Antwerpen, H. (2023). Scope graphs: The story so far. In *Eelco Visser Commemorative Symposium, 2023, Delft, The Netherlands*, volume 109 of *OASIcs*. Schloss Dagstuhl.