NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

A Language Server for Soufflé Datalog

Ioannis N. Daridis

Supervisors: **Yannis Smaragdakis,** Professor NKUA
Sifis Lagouvardos, PhD Student

ATHENS

OCTOBER 2022

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

A Language Server for Soufflé Datalog

Ιωάννης Ν. Δαρίδης

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Σήφης Λαγουβάρδος, Διδακτορικός Φοιτητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2022

BSc THESIS

A Language Server for Soufflé Datalog

Ioannis N. Daridis

S.N.: 1115201700028

SUPERVISORS: **Yannis Smaragdakis,** Professor NKUA
Sifis Lagouvardos, PhD Student

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

A Language Server for Soufflé Datalog

Ιωάννης Ν. Δαρίδης
Α.Μ.: 1115201700028

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Σήφης Λαγουβάρδος, Διδακτορικός Φοιτητής

ABSTRACT

Software development is increasingly supported by programming language tools and IDEs, in order to meet the increase in software complexity and computing power. Programmers nowadays also expect a level of automation like automatic syntax error detection or a level of auto complete when editing their source-code. In this thesis we explore how one can use the Language Server Protocol and other technologies to add tool support to a novel programming language like Soufflé Datalog, a logic programming language used for static program analysis. We developed a Language Server for Soufflé, which can be used inside a plugin for a variety of source-code editors or IDEs, compatible with LSP, to add smart functionality and support for Soufflé.

SUBJECT AREA: Programming Language Tools

KEYWORDS: Soufflé, Datalog, LSP, tooling, IDE

ΠΕΡΙΛΗΨΗ

Τα τελευταία χρόνια η ανάπτυξη λογισμικού υποβοηθείται σε μεγάλο βαθμό από εργαλεία και βιβλιοθήκες που προσφέρονται ως μέρος των γλωσσών προγραμματισμού, καθώς και από Ολοκληρωμένα Περιβάλλοντα Ανάπτυξης. Επίσης, πλέον οι προγραμματιστές αναμένουν ένα επίπεδο αυτοματοποίησης την ώρα της συγγραφής κώδικα, όπως αυτόματη συμπλήρωση κώδικα ή επισήμανση συντακτικών λαθών. Σε αυτή την εργασία εξερευνούμε το πως μπορεί να χρησιμοποιηθεί το πρωτόκολλο LSP και άλλες τεχνολογίες για την προσθήκη λειτουργιών υποστήριξης, όπως αυτές που αναφέρθηκαν, σε μια νεα γλώσσα, τη Soufflé Datalog. Η Soufflé είναι μια γλώσσα λογικού προγραμματισμού, που χρησιμοποιείται για στατική ανάλυση προγραμμάτων. Αναπτύξαμε έναν Language Server, ο οποίος μπορεί να ενσωματωθεί σε πρόσθετο οποιουδήποτε συγγραφέα πηγαίου κώδικα συμβατό με το πρωτόκολλο LSP, για να προστεθούν "έξυπνες" λειτουργίες και υποστήριξη για τη γλώσσα Soufflé Datalog.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Εργαλεία γλωσσών προγραμματισμού

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Soufflé, Datalog, LSP, tooling, IDE

Αφιερωμένο στους γονείς μου

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Software development is becoming increasingly complex, as our computers keep getting faster and modern languages continue to evolve to meet growing needs. As this trend continues, we see the need for better tools supporting programming languages, tools such as linters, static analyzers, debuggers, and so on. Programmers are now accustomed to having these tools available through an Integrated Development Environment (IDE) or a source-code editor. We also expect some level of automation, such as auto complete and type hints, to be at our disposal. This fact is true for all the major industrial languages, but often the smaller, more academically focused languages are left behind in the support from major IDE or source-code editor providers. With the introduction of the Language Server Protocol this can be changed.

The Language Server Protocol was designed by Microsoft, to enable the decoupling of the language tools and the source-code editors. This allows the language vendors to write a single target for their tools, rather than repeating the same job over and over, also enabling a greater degree of separation of concerns. It is now an open-source project and standard and it is used by major languages, for instance Java and Typescript, and editors, such as VS Code and Eclipse IDE.

We will present the development of a language support tool, a Language Server, for the Soufflé Datalog language [25], a logic programming query language used primarily for static program analysis. Notable Soufflé applications include the DOOP framework [8], the Gigahorse toolchain [14] and Securify2 [2], to name a few. We explore how we can add some of the familiar functionality one might expect from a modern source-code editor to a novel language, with the help of the LSP, and the challenges and design choices this entails. During the implementation we see a great overlap with technologies used in a language compiler front end, namely: grammar construction, parsing, semantic analysis and symbol table construction.

Below we introduce the contents of this thesis:

- In chapter 2 we introduce the Datalog and Soufflé languages, along with the LSP and how it operates, followed by ANTLR, which was used for the parsing.
- In chapter 3 we present a bottom up view of how the Soufflé Language Server is implemented, starting from the source-code parsing and ending with the user facing features.
- In chapter 4 we show how we tested and evaluated our tool and the problems and limitations of the current implementation.
- In chapter 5 summarize the conclusions we made.

2. BACKGROUND

2.1 Datalog

Datalog [9] is a declarative logic programming language designed for deductive database querying. It is a syntactical subset of Prolog, but the semantics of the two languages are quite different. The main strength of Datalog is the bottom-up evaluation of the proof trees, until a fixpoint is reached, when there is a finite set of facts, and thus the program always terminates. This principle is also referred to as forward chaining, and is in contrast to the backward chaining, used in languages like Prolog.

A Datalog program consists of a finite set of Horn clauses [17], called facts and rules. The general shape of a rule clause is: $L :- L_1, ..., L_n$, where each L_i is a literal of the form $p_i(t0, ..., t_k)$. Fact clauses, are represented like the rules but with an empty body. For a Datalog program to be guaranteed to produce a finite set of facts, and thus to always terminate, a number of safety conditions must hold, namely:

- Each fact must be *ground*, meaning it does not contain any variables
- Each variable which occurs in the head of a rule must also occur in the body of the same rule.
- Impose stratified negation in rules

Initially designed for relational database querying, Datalog introduces two sets of clauses: the Extensional Database (EDB), the set of ground facts stored in a relational database and the Intenional Database (IDB), the Datalog program itself. Lately though, Datalog has seen a revival outside of the traditional database querying, in static program analysis, through the use of different frameworks based on vaious implementations of Datalog.

2.1.1 Soufflé

Datalog, like many logic programming languages, does not have a formal implementation. `Soufflé` [25], rather than just a Datalog implementation, is a programming language inspired by Datalog with some extensions of its own [18]. Focused more on static program analysis, Soufflé overcomes some limitations, by permitting programmers the use of functors (intrinsic, user-defined, records/constructors, etc.) and the use of non finite domains.

As stated in their website[1]: *Soufflé is short for Systematic, Ontological, Undiscovered Fact Finding Logic Engine. The EDB represents the uncooked Soufflé and the IDB causes the Soufflé to rise, i.e., monotonically increasing knowledge. When it stops rising and a fixed-point is reached, the result is a puffed-up ready-to-eat Soufflé.*

Soufflé is designed to fix some of the challenges of logic programming by translating the programs to efficient C++ code, leveraging modern computer hardware, including multi-core computers and parallelization. It achieves that by using advanced compiler techniques namely Futamura Projections [12], staged-compilation and partial evaluation.

[1] `https://souffle-lang.github.io/docs.html`

In addition to its default optimizations [5, 27], Soufflé also allows programmers to optimize their programs in various ways and at different levels. At the program level, Sideways Information Passing Strategy [7] can be selected. At the relation level users can enable the magic-set transformation [6] or select the data structures used for each relation [19][20][22]. Finally, at the rule level the programmer can alter the evaluation order of recursive rules via query plans.

2.2 Language Server Protocol (LSP)

Modern software development is becoming increasingly polyglot, but the tools supporting the languages, such as source-code editors, linters etc, remain limited. It is economically infeasible to build the toolchains to support each language, from the ground up. The `Language Server Protocol`[2] [11] was designed to solve this exact problem. It allows the decoupling of the source-code editor and the language processing, by delegating the language smarts to a Language Server, which then communicates with the editor with Inter-Process Communication. This way every editor compatible with the LSP can provide feautures akin to go-to-definition or autocomplete, for a greater variety of languages.

The Language Server Protocol uses JSON-RPC for the communication between the Language Server and the source-code editors implementing the protocol, called Language Clients. This design enables a language agnostic type of communication, between the server and the client, where they communicate in terms of document URIs and generic data types. In the code listing 1 we can see how the actual messages are structured and in figure 2.1 we can see an overview of the communication sequence.

2.2.1 LSP4J library

The Language Server Protocol team does not provide an implementation, rather just the specification for the protocol. The `LSP4J` library[3] is a Java implementation of the LSP, developed by the Eclipse Foundation, for use in the Eclipse IDE and other editors. It is a wrapper for LSP JSON-RPC calls, but can also be used as a generic JSON-RPC implementation for Java. In the code listing 2 we can see an example of the library usage.

Because the Java programming language is so prevalent the LSP4J library integrates well with many other tools. The LSP4J library was used for the implementation of many tools, namely: the Language Server for the Ballerina programming language [16], the MagpieBridge tool for static analysis [21] and the ExtendJ compiler [26]

[2] `https://microsoft.github.io/language-server-protocol/`
[3] `https://github.com/eclipse/lsp4j`

```
{
    "jsonrpc": "2.0",
    "id" : 1,
    "method": "textDocument/definition",
    "params": {
        "textDocument": {
            "uri": "file:////VSCode/Playgrounds/cpp/use.cpp"
        },
        "position": {
            "line": 3,
            "character": 12
        }
    }
}

{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "uri": "file:///VSCode/Playgrounds/cpp/provide.cpp",
        "range": {
            "start": {
                "line": 0,
                "character": 4
            },
            "end": {
                "line": 0,
                "character": 11
            }
        }
    }
}
```

Listing 1: Example of a LSP JSON-RPC request/response

```
public class SouffleTextDocumentService implements TextDocumentService {

    public SouffleTextDocumentService(SouffleLanguageServer) {...}

    @Override
    public void didOpen(DidOpenTextDocumentParams) {...}

    @Override
    public void didChange(DidChangeTextDocumentParams) {...}

    @Override
    public void didClose(DidCloseTextDocumentParams) {...}

    @Override
    public void didSave(DidSaveTextDocumentParams) {...}

    @Override
    public CompletableFuture<Either<List<? extends Location>, List<? extends LocationLink>>>
    ↪  definition(DefinitionParams) {...}

    @Override
    public CompletableFuture<Hover> hover(HoverParams) {...}

    @Override
    public CompletableFuture<Either<List<CompletionItem>, CompletionList>> completion(CompletionParams) {...}

    @Override
    public CompletableFuture<WorkspaceEdit> rename(RenameParams) {...}

    @Override
    public CompletableFuture<SignatureHelp> signatureHelp(SignatureHelpParams) {...}
}
```

Listing 2: Example of the LSP4J library usage

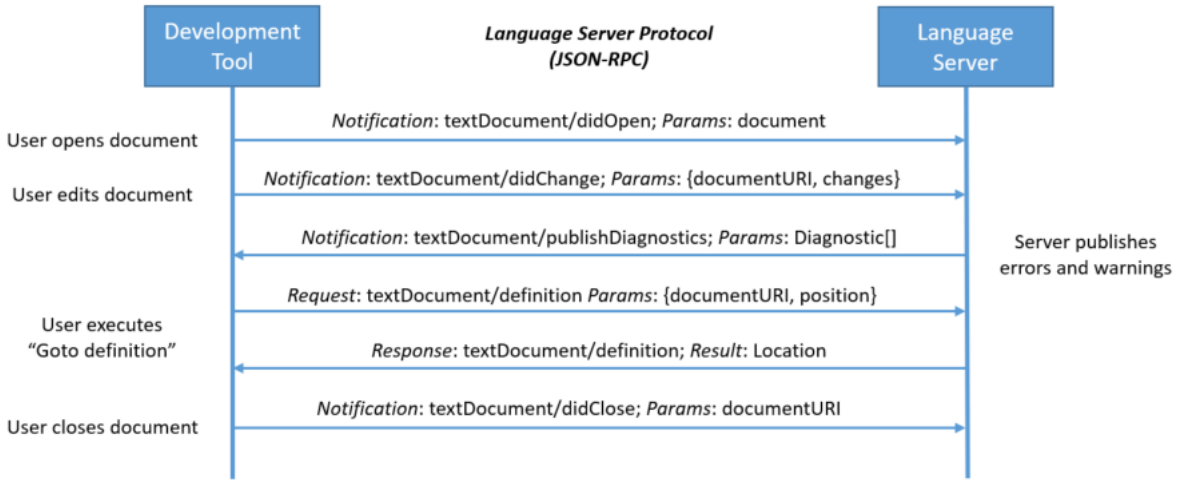


Figure 2.1: The Language Server Sequence (as taken from their website)

2.3 ANTLR

Parsing of a computer program refers to the process of breaking it down to its component pieces and checking if they conform with the rules of the language's grammar. A grammar is a set of rules describing a language's syntactic structure. A parser is a program that, given a language grammar, checks if the input conforms to the grammar and, if it does, it produces a parse tree, a data structure for further analysis of the input. The parser can be written by hand, for simpler language grammars, but the complexity rises quickly, so nowadays we have tools for automatically producing parsers, called parser generators.

`ANTLR`[4] (ANother Tool for Language Recognition) [23] is a parser generator, developed by Terence Parr, built to solve many of the problems programmers were facing with other parser generator tools. It uses a top down LL(*) parsing technology, making it very fast. The new v4 iteration of ANTLR introduced the the Adaptive LL(*) technology [24], making it possible to express a greater variety of grammars, without worrying about left recursion. ANTLR provides a scanner generator, as well, and it integrates the regular expressions into the grammar file, making it a more complete and easy experience to generate a parser for any language you want. It also produces parse tree walkers and visitors, and exposes APIs for parse tree manipulation for all the languages it supports. This encourages the decoupling of the grammar and the underlying implementation, making the grammar reusable and extensible.

[4] `https://www.antlr.org/`

3. THE LANGUAGE SERVER

In this chapter we present how the Soufflé Language Server is implemented and the features it provides. As discussed on section 2.2 the architecture consists of a Language Client and a Server. For the purposes of this thesis we chose Visual Studio Code as the client, and the server is part of a broader plugin for VS Code we implemented, so the majority of the assumptions we made or the examples we'll give are based on that fact. The server is implemented in the Java programming language using the LSP4J library version 0.16, which targets version 3.17 of the LSP and it is packaged in the form of a jar file inside the plugin.

This is a brief introduction of the Language Server: When the language client recognizes the Soufflé Datalog file type (.dl) it starts the server and makes a connection to it, via the standard I/O system. When the connection is established, the `initialize` command is sent and after the server sets up all its internal state and announces its capabilities to the client, it finishes initialization and proceeds to read, parse and process all the Soufflé files on the current project directory. As soon as the processing is complete the server is ready to be used in ”interactive mode” and handle any further requests it receives from the client. When a user saves a file, the parsing and processing of the file is being performed again and the internal state of the server is refreshed to mirror the current state.

3.1 Parsing the input

One of the main aspects of our tool, in order to provide the smart features, is the ability to correctly read and parse the source-code being edited. For that we need to access the source-code and pass it on for parsing and further analysis, which will then extract the information we need. The LSP and lsp4j library provide the document URI so that we can feed it to the parser, which then reads the document text and proceeds with the parsing stage, syntax error checking and information retrieval.

3.1.1 Soufflé

The current implementation of the Soufflé language uses a grammar[1] written for the GNU Bison parser generator [10]. For the purposes of our tool we needed to port the grammar to ANTLR. The process was quite straightforward, because even though the original $LALR$ grammar contains left recursion, the ANTLR4 parser generator, as mentioned (2.3), is well equipped to handle it. All we had to do was copy the regular expressions and grammar rules and change the syntax so it conforms with ANTLR grammar syntax. The resulting grammar[2], after some modifications for the preprocessor handling, combines the regular expressions for scanning and the grammar for parsing into one file.

The Soufflé language uses C style comments, both one line and multiline, so we incorporated them into the grammar. Instead of ignoring them, we used ANTLR's lexical channel feature to save the comments for later use, as documentation.

[1] `https://github.com/souffle-lang/souffle/blob/master/src/parser/parser.yy`

[2] `https://github.com/jdaridis/souffle-lsp-plugin/blob/master/grammar/Souffle.g4`

3.1.2 C Preprocessor

Soufflé makes use of the C preprocessor in order to include files or to define macros for reuse later in the source-code. We noticed that this feature is widely used in the industry and in popular open source libraries, so our tool need to be robust enough to handle it.

The parsing proved to be a bit of a challenge, because the syntax differs from that of Soufflé. In addition to that, the heavy use of macros made it difficult to differentiate between macro and legitimate Soufflé code, without running the preprocessor ourselves, thus adding time and complexity. The solution we came up with is to introduce another parsing stage and the corresponding grammar[3], before the main stage, in order to parse the input correctly.

The parser reads the whole input and ignores most of the text until it reaches a preprocessor directive (#include, #define), then it breaks it down to its components. When the whole input is parsed, the resulting syntax tree is traversed by a `Visitor` object and when a macro definition is reached, we save the definition in a dictionary (`HashSet`) for later use. The dictionary is then passed to the next, and main stage of parsing, and is being used to find which identifier is a macro use. We achieve that by inserting code into the scanner so that when it encounters a macro use it produces a different lexical token and then we change the grammar to include the new special preprocessor token.

3.2 Extracting the information

The LSP is simply the mechanism that enables the communication between the code editors and the language tools. It is the tool's responsibility to provide the smart features (for instance go to definition or auto complete) on top of the language. To accomplish that, we need to perform semantic analysis and extract all the useful information, and for that, we need efficient data structures, to hold and query all the information and provide the intelligence to our tool.

The core elements of a Soufflé program are *relations* and logic statements on these relations. Each relation, has a set of *typed attributes*. A *fact* is a logic statement which unconditionally holds true, and a *rule* is a conditional logic statement, that has one (or more) relation(s) as a head predicate and one or more literals as a body. The rule literals can be either relations or attribute constraints. Soufflé has, also, introduced the concept of *components*, to make logic programs modular and reusable. Therefore, if we want to reference some element that's inside a component we need its fully qualified name.

Once the parsing is complete, we are left with the parse tree for each file. The ANTLR runtime provides us with an implementation of the Visitor pattern [13] and an interface to extend it and traverse the parse tree. Since Soufflé is a statically typed language and the types and relations are known at compile time, it is wise to collect information about them ahead of time, by performing two passes of the parse tree traversal.

Firstly, for all the files parsed, the `SouffleDeclarationVisitor` visits all the Relation, Type and Component declaration nodes and saves the information about the declaration in a special data structure. If there are comments above the declaration, we access them through ANTLR's hidden lexical channel, then process and save them with the declaration.

[3] `https://github.com/jdaridis/souffle-lsp-plugin/blob/master/grammar/Preprocessor.g4`

Then we proceed with the next stage, where the `SouffleUsesVisitor` visits all the other Soufflé grammar rules. When it visits a Soufflé rule node, it queries the server to find the definitions of the involved relations, containing useful information, such as the declaration site, and stores them alongside the rule. A similar process is performed for the facts. As mentioned, with the introduction of components sometimes we need a fully qualified name, in order to correctly reference a relation or rule, so we store the component name alongside the relation name, and when the `SouffleUsesVisitor` visits a fully qualified name node, it has all the information it needs to perform a correct query for the definition.

3.2.1 Infrastructure

In this subsection, we present how it all works under the hood. First of all, we needed a data structure to efficiently support the random navigation throughout the document, a user might perform. The solution we came up with, is a balanced binary search tree, using the range of each grammatical scope as a key and the caret position as a query key. The `TreeMap`[4] class in Java has the properties we want, because it is implemented using a Red-Black tree [15]. We also wanted to keep some information about the context and lexical scope of each grammatical structure, so we implemented the following hierarchy, shown in figure 3.1, loosely mirroring the structure of a Soufflé program. With the modified data structure we achieve a time complexity of $O(m \log n)$, where m is the scope depth and n is the number of nodes in each scope level. The query stops when it reaches a full match either at a leaf or an internal node.

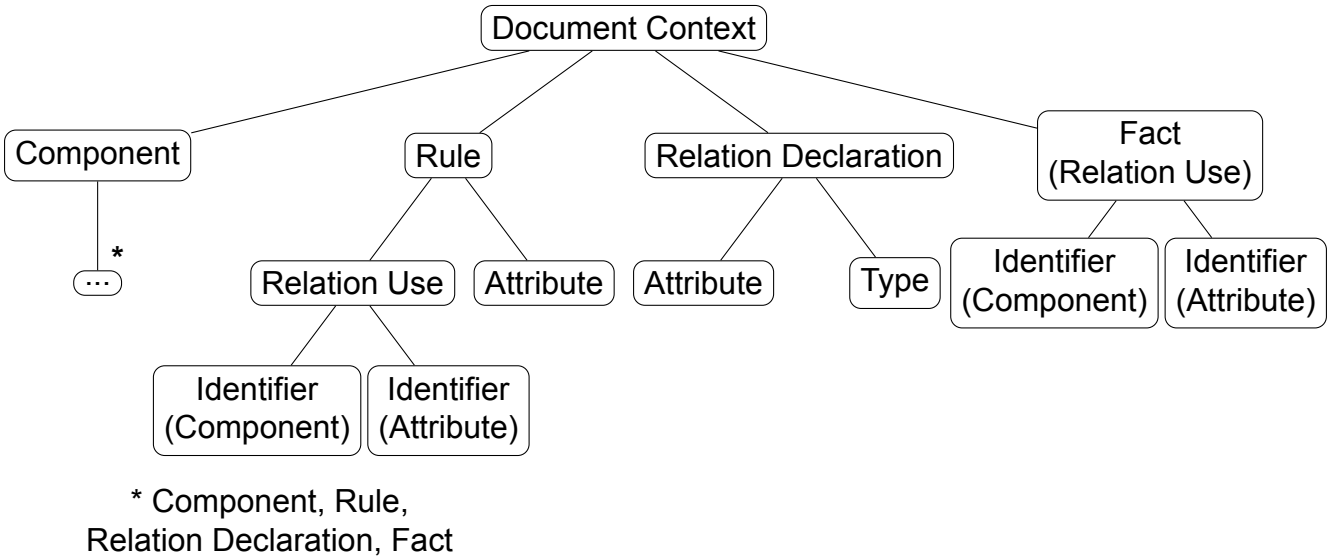


Figure 3.1: Soufflé Context Hierarchy

As mentioned, in each level we also keep information about the lexical scope. The scopes are unique per level, but there can be some overlap at times. Also, we note that each Soufflé Component carries its scope with it. We make use of two different data structures depending on the situation. If we want to perform a query using a specific name of a symbol we use a `HashMap` and retrieve a list of all the symbols with the given name. Alternatively, if we want a spacial query, for example involving the cursor position, we use a `TreeMap` for a more efficient search.

[4] `https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html`

The figure 3.2 shows the complete Soufflé modeling we came up with, including all the information about each Soufflé component we found useful. The basic block is the SouffleSymbol class, so we analyze it further. It contains the basic identifying information of the symbol in question, such as its name and range (line & column number) and the type of symbol for easy differentiation between components. Each symbol has associated with it a declaration and the URI of the document it is located in. Optionally, it also has a documentation (taken from comments) and a component, used for fully qualified names. We then extend the base class `SouffleSymbol` and add all the extra information we might need about each symbol.

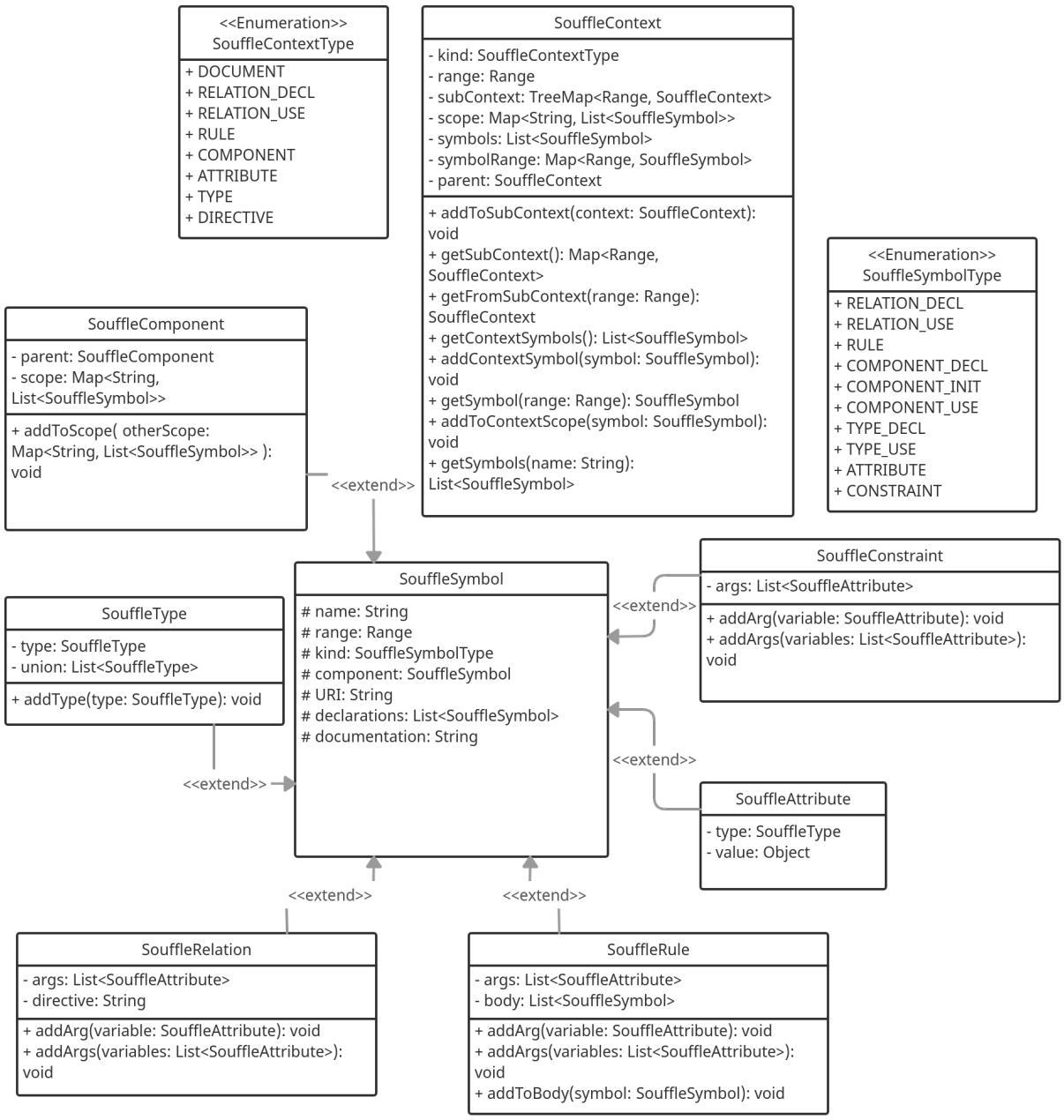


Figure 3.2: Soufflé model UML Class Diagram

3.3 Features

3.3.1 Syntax error reporting

After the parsing of a file, if a syntax error occurs, we notify the user about it by providing them with the line, column and an error message, as shown in figure 3.3. In order to produce more helpful error messages, we had to change ANTLR's error reporting mechanism and provide our own messages. Then we send an error message to the client, which presents the user with the error.

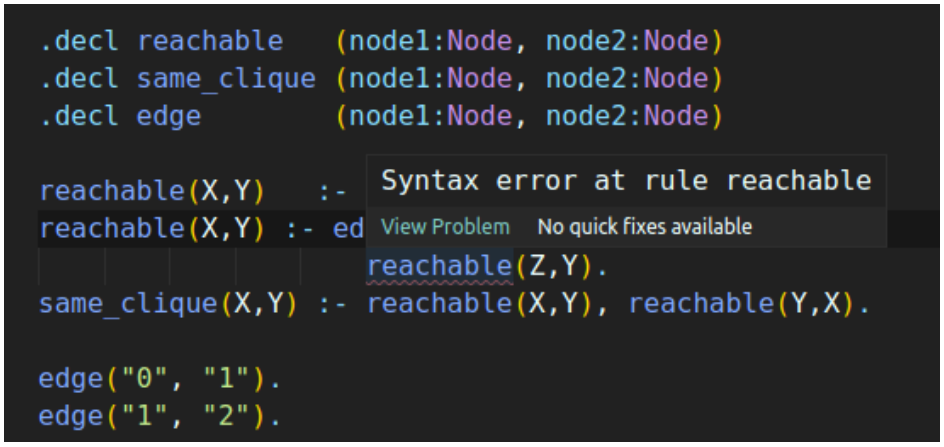


Figure 3.3: Syntax error message

3.3.2 Hover

When a user hovers the mouse over a specific object, we query the server by traversing the Soufflé context hierarchy object, using the cursor position as a key, and provide them with relevant information about that object. Specifically, when they hover over a relation or a component, we show the full signature, alongside with any documentation we might have collected about it. Alternatively, when they hover over a variable, we show the variable name and its type.

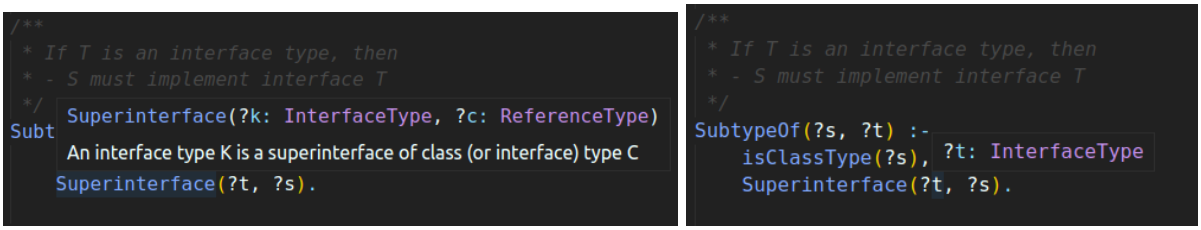


Figure 3.4: Hover popup examples

3.3.3 Go to Definition

In Soufflé each relation needs to be declared ahead of time, so when a user asks for the definition of a relation, we have it available in the Soufflé object and can provide it. We point them to the declaration site, alongside with the file it is located in. The same process is being performed for the types and components. If, in a given project, there are multiple declarations we present the user with a list to choose from.

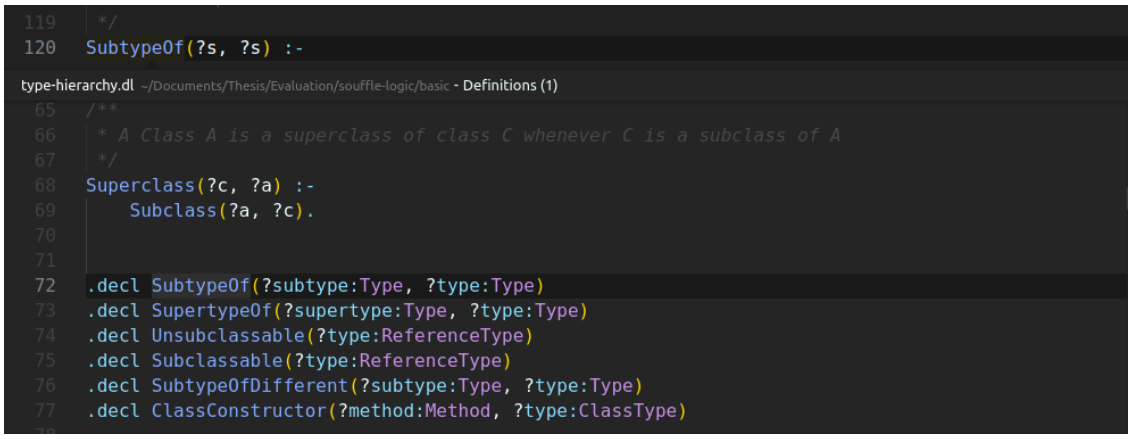


Figure 3.5: Go to definition example

3.3.4 Go to Type Definition

We follow the same process, as with the plain definitions, but because Soufflé is statically typed, we found it useful to add this shortcut for when the user wants to navigate directly to a variable's type definition.

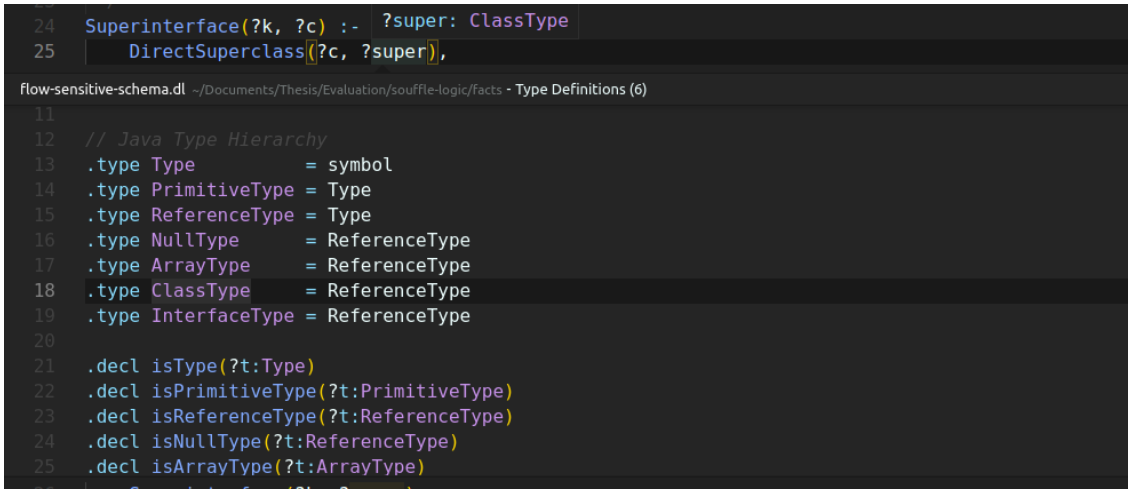


Figure 3.6: Go to type definition example

3.3.5 Auto complete

When a user starts typing, we provide a list of auto complete suggestions, contextually relevant to what the user is typing. This list contains relations, types or Soufflé directives, which we have acquired by querying the Soufflé context objects of all the the files in the project. It is then filtered according to the context the user is typing in or the trigger word. For example, a dot will trigger directive auto complete and a colon a type auto complete.

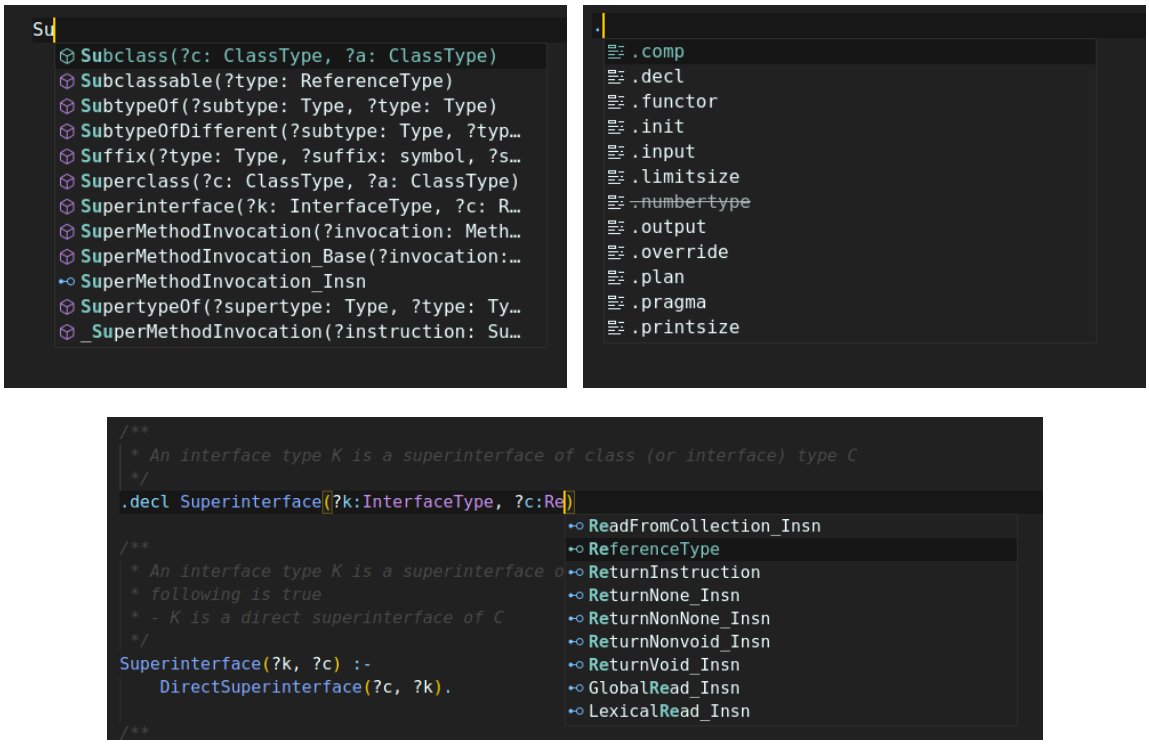


Figure 3.7: Auto Complete examples

3.3.6 Signature Help

When the user is in the process of writing a relation, we show a popup with the relation signature alongside with its attributes and their type, with the current attribute being highlighted. Because the file parsing is being performed after the user has saved the file, we couldn't perform the query for the relation while the user is typing it, so we had to find a different way. The solution we came up with is to use the ANTLR's parsing error reporting, on the live, and thus not yet grammatically correct text, to obtain the last token the user has written and use it as a query key in the Soufflé context object.

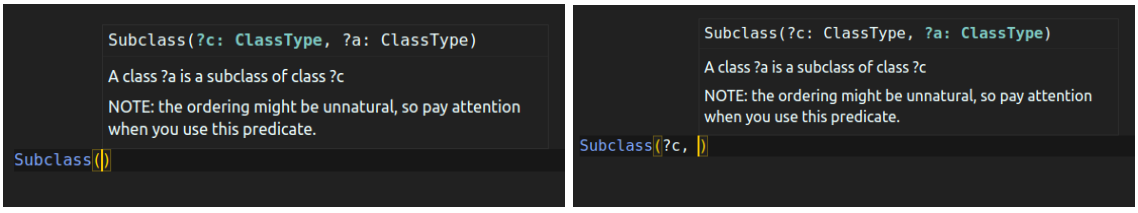
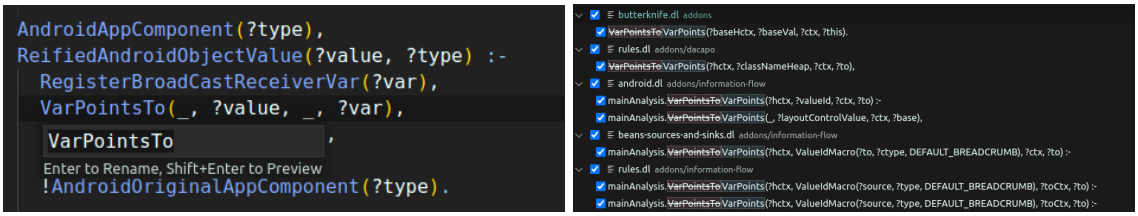


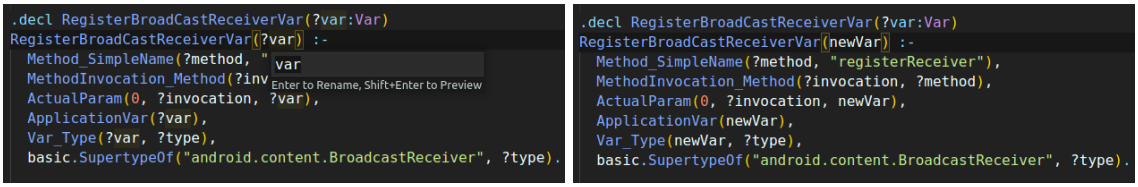
Figure 3.8: Signature Help example

3.3.7 Rename

We provide the user with the ability to rename any symbol they want, either at a global or at a local level. The Soufflé context object, proves to be instrumental in the implementation of this feature, because we can determine the context, and thus the scope, of the symbol in question and perform the rename on all the relevant symbols.



(a) Global rename



(b) Local rename

Figure 3.9: Rename examples

3.3.8 Find all references

Perform a full search of the project for all appearances of a certain symbol, by querying all the project's Soufflé context objects. We then provide the user with a list of all the appearances and locations within the file.

Figure 3.10: Find references example

3.3.9 Find all rules for relation

If a user wants to find all the rules a certain relation is the head predicate in, we perform a query of all the project's Soufflé context objects, for rule symbols only, and return to the user a list of all the rules, alongside with the files and locations within the files.

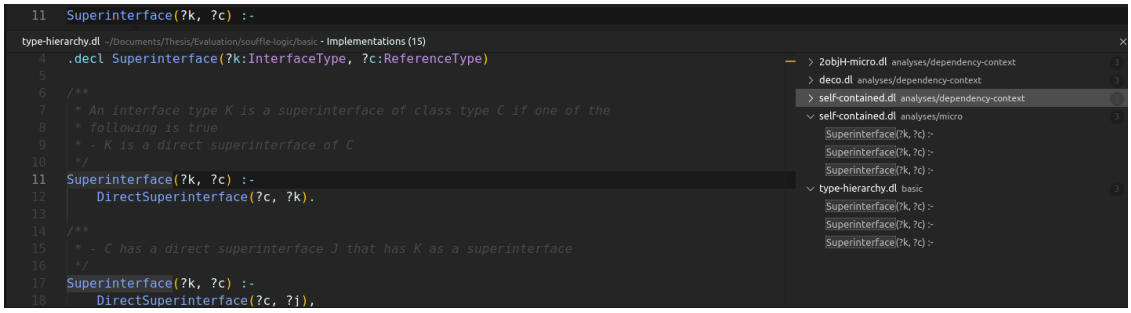


Figure 3.11: Find all rules example

3.3.10 Document symbols

We provide the client with a list of all the symbols contained in a specific file alongside with their type, for example a relation or a rule. We also provide a way to connect the symbols with each other. For example, for each relation we provide all the rules, this relation is the head predicate, or for a component we provide the scope of it. In the case of Visual Studio Code, as shown in figure 3.12, it uses these information to construct a file outline and help the user navigate the file.

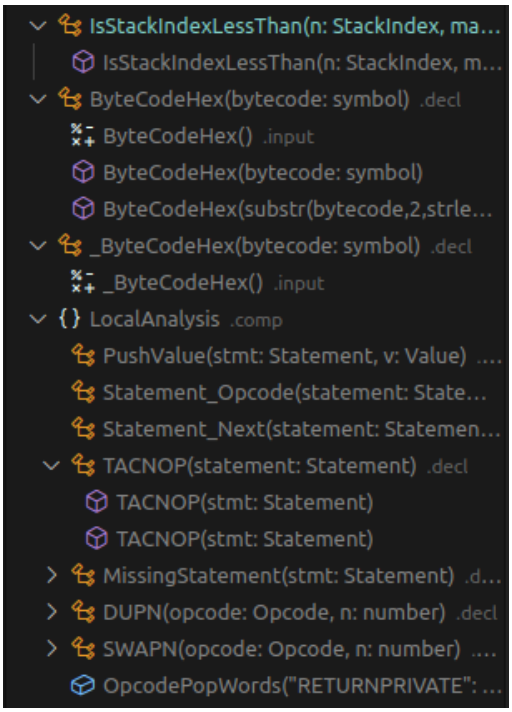


Figure 3.12: Document outline example

4. EVALUATION

In this chapter, we present how we test and evaluate our tool. We test the server in two code editors compatible with the LSP, namely Visual Studio Code and Sublime Text. The main development was carried out in the VS Code editor, so this is our reference platform. The testing was performed on a modern mid-tier laptop with an Intel i7-1065G7 processor with $4\ cores$ and $8\ threads$ and a frequency of $3.9Ghz$ running Ubuntu 20.04 LTS. The system is also equipped with $16GB$ of RAM.

For the evaluation we used the libraries found on the Souffle site [1], alongside with some of our own examples. For each library, we keep track of how many Soufflé Datalog lines it contains. The metrics we chose were *the startup time, memory footprint* and finally the *file coverage*. A more extensive breakdown is given below:

Time We calculate the server's startup time, from the moment the server is started and the `initialize` command is sent, until the `initialized` command. In that time, the server parses all the files and builds the context objects for later use. We figured it is the best time window, since the bulk of the work is being performed in that time. This time is also auto reported by the language client. To reduce the system noise, we took 5 measurements and calculated their geometric mean.

Memory Footprint We note how much RAM, in mebibytes (MiB), the server uses when it is fully loaded and operational. This should give an idea of the total memory footprint of the server, since most of the structures are in memory from the start.

File Coverage In any given project, the core objective is to be able to edit and work with as much of the project's files as possible. For this reason, we count how many of the Soufflé Datalog files, located in the project, our tool can correctly parse and process, and then offer a project coverage percentage.

In the table 4.1 we can see all the data we gathered for each library and in the figure 4.1 we can see the scaling of the server's startup time and RAM usage in conjunction with the Soufflé Datalog lines of each library.

Table 4.1: Soufflé Language Server evaluation

Library	File Coverage	Time (ms)	Memory Footprint
Soufflé Benchmark (65.759)[a]	47 / 47 (100%)	$4.382,6$	754.9 MiB
Doop (44.793)	229 / 229 (100%)	$4.077,9$	450.2 MiB
Gigahorse (13.595)	46 / 46 (100%)	$2.324,7$	307.0 MiB
Securify2 (3.365)	65 / 66 (98.48%)	$1.603,8$	170.0 MiB
Ddisasm (15.437)	60 / 60 (100%)	$1.750,9$	272.0 MiB
Vandal (636)	5 / 5 (100%)	$963,6$	78.0 MiB
cclyzer++ (15.670)	127 / 127 (100%)	$2.221,6$	228.0 MiB
Dynamic-datalog (2.307)	3 / 3 (100%)	$775,2$	80.9 MiB
Ghc-grin (717)	8 / 22 (36.36%)	800.2	107.0 MiB
Insieme compiler[b] (1.505)	10 / 31 (32.26%)	$1.060.0$	101.0 MiB

[a] Number of Soufflé Datalog lines in library, at the time of testing

[b] Uses custom preprocessing script and keywords outside of grammar

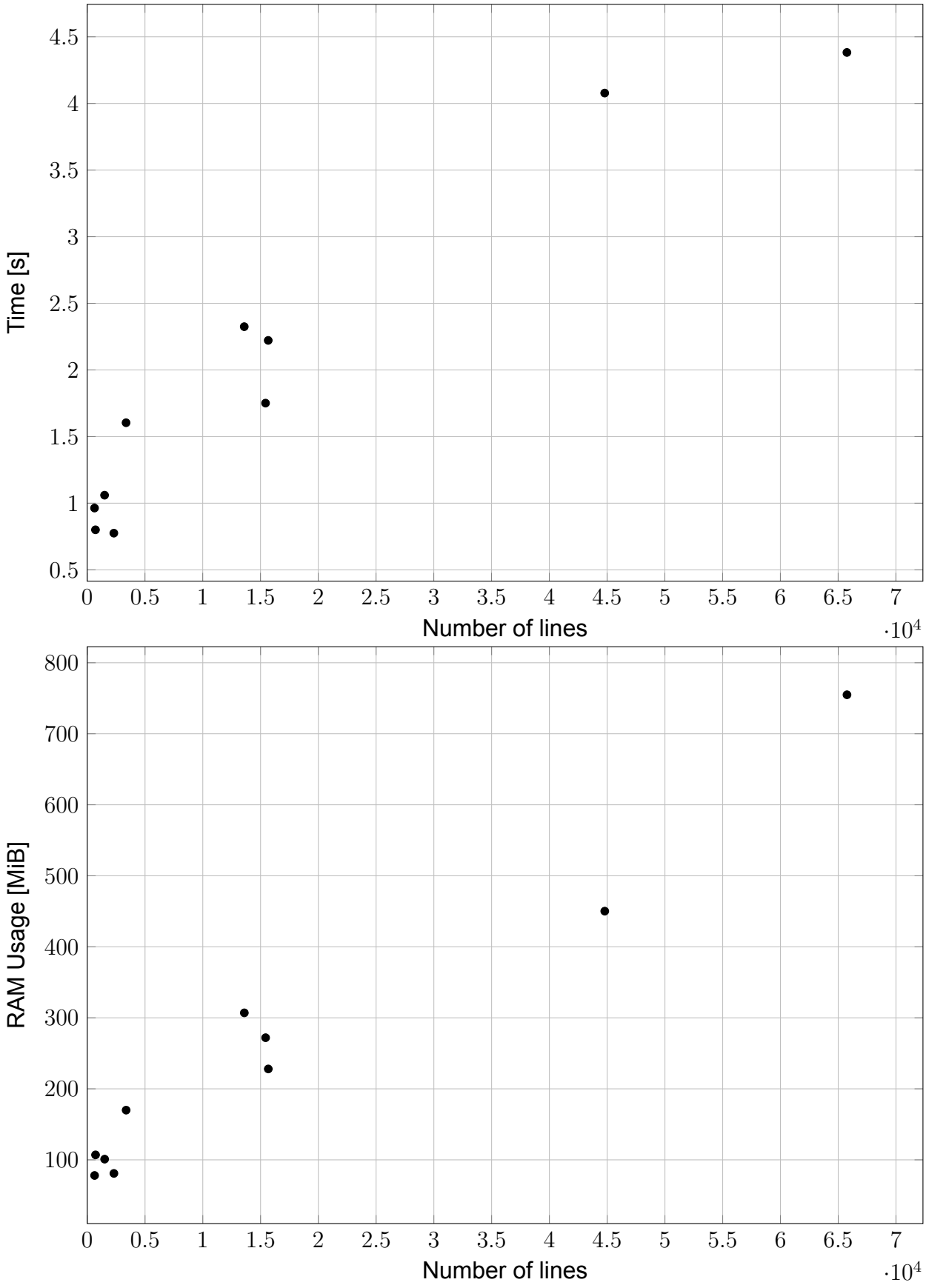


Figure 4.1: Scatter plots of the total project lines vs the time and RAM usage

4.1 Libraries used

In this section we provide a quick overview of the main libraries used in our evaluation and testing. We note that we will not present all the libraries used for the evaluation, but only the ones we used extensively during the development of the server.

4.1.1 Doop framework

Doop [8] is a points-to analysis framework for Java programs. It contains a variety of pointer analysis tools. Doop was originally written for the LogiQL datalog dialect and implementation and was later ported to Soufflé [4].

4.1.2 Gigahorse

Gigahorse [14] is a decompiler toolchain, for the Ethereum Virtual Machine (EVM) bytecode. It converts EVM bytecode to a higher level 3-address code representation, enabling different analyses on smart contracts both new and already deployed in the blockchain, without needing access the original source-code.

4.1.3 Securify2

Securify version 2.0 [2] is security scanner for smart contracts written in the Solidity programming language and run on the Ethereum blockchain. The security analyses are written in Soufflé Datalog.

4.1.4 Soufflé Benchmark

The team behind the development of Soufflé team has built a library [3], meant to be used as a benchmark tool for the performance of a local installation of Soufflé. It is a large compilation of Soufflé Datalog files, used for stress testing and benchmarking.

4.2 Problems and limitations

During the testing we identified some weaknesses in our tool. Notably, in certain libraries the server failed to correctly parse and process all the files present in the project, thus lowering the project coverage. After investigation, we found that the reason for that was the extensive use of the C preprocessor. The libraries in question are using a mixture of conditional compilation guards (#ifdef) and nested macro definitions, making it difficult for our preprocessor parser to correctly process them. In particular, the use of conditional compilation guards, where the complete program text was not available ahead of time, was often a point of failure, due to the nature of the parser.

A proposed solution to the above problems might be to add an additional stage, executing the C preprocessor itself and then keeping track of the changes in the source-code. That solution would require the user to have a C compiler installed on their machine, but also would add complexity and extra startup time to the server.

5. CONCLUSIONS

In this thesis we explored how support for programming languages in source-code editors can be extended, even for novel languages. We used the Language Server Protocol for the implementation of a support tool for Soufflé Datalog. We found that the extensive use of the C preprocessor in popular Soufflé programs, presented a problem for us, because we couldn't correctly parse the whole text in all cases. For the implementation we used most of a compiler front end stack, from grammar construction and parsing to the construction and use of an efficient symbol table.

ABBREVIATIONS - ACRONYMS

LSP	Language Server Protocol
VS Code	Visual Studio Code
IDE	Integrated Development Environment

BIBLIOGRAPHY

[1] Applications | Soufflé; A Datalog Synthesis Tool for Static Analysis — souffle-lang.github.io. https://souffle-lang.github.io/applications. [Accessed 04-Oct-2022].

[2] GitHub - eth-sri/securify2: Securify v2.0 — github.com. https://github.com/eth-sri/securify2. [Accessed 04-Oct-2022].

[3] GitHub - souffle-lang/benchmarks: Datalog benchmark suite — github.com. https://github.com/souffle-lang/benchmarks/. [Accessed 04-Oct-2022].

[4] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. Porting doop to soufflé: A tale of inter-engine portability for datalog-based analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, page 25–30, New York, NY, USA, 2017. Association for Computing Machinery.

[5] Samuel Arch, Xiaowen Hu, David Zhao, Pavle Subotić, and Bernhard Scholz. Building a join optimizer for soufflé. In *Logic-Based Program Synthesis and Transformation: 32nd International Symposium, LOPSTR 2022, Tbilisi, Georgia, September 21–23, 2022, Proceedings*, page 83–102, Berlin, Heidelberg, 2022. Springer-Verlag.

[6] I. Balbin, G.S. Port, K. Ramamohanarao, and K. Meenakshi. Efficient bottom-up computation of queries on stratified databases. *The Journal of Logic Programming*, 11(3):295–344, 1991.

[7] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '87, page 269–284, New York, NY, USA, 1987. Association for Computing Machinery.

[8] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.*, 44(10):243–262, oct 2009.

[9] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[10] Robert Paul Corbett. *Static Semantics and Compiler Error Recovery*. PhD thesis, EECS Department, University of California, Berkeley, Jun 1985.

[11] Microsoft Corporation. Language server protocol. Technical Report 3.17.

[12] Yoshihiko Futamura. Partial evaluation of computation process, revisited. *Higher Order Symbol. Comput.*, 12(4):377–380, dec 1999.

[13] Erich Gamma, editor. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, Reading, Mass, 1995. pp 331.

[14] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: Thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186, 2019.

[15] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21, 1978.

[16] Nadeeshaan Gunasinghe and Nipuna Marcus. *Implementing a Language Server*, pages 23–33. Apress, Berkeley, CA, 2022.

[17] Alfred Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16(1):14–21, 1951.

[18] Xiaowen Hu, Joshua Karp, David Zhao, Abdul Zreika, Xi Wu, and Bernhard Scholz. The choice construct in the soufflé language. In Hakjoo Oh, editor, *Programming Languages and Systems*, pages 163–181, Cham, 2021. Springer International Publishing.

[19] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. Brie: A specialized trie for concurrent datalog. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'19, page 31–40, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. A specialized b-tree for concurrent datalog evaluation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 327–339, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Linghui Luo, Julian Dolby, and Eric Bodden. MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper). In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:25, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[22] Patrick Nappa, David Zhao, Pavle Subotić, and Bernhard Scholz. Fast parallel equivalence relations in a datalog compiler. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 82–96, 2019.

[23] Terence Parr. *The definitive ANTLR 4 reference*. The pragmatic programmers. The Pragmatic Bookshelf, Dallas, Texas, 2012. OCLC: ocn802295434.

[24] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll(*) parsing: The power of dynamic analysis. *SIGPLAN Not.*, 49(10):579–598, oct 2014.

[25] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 196–206, New York, NY, USA, 2016. Association for Computing Machinery.

[26] Fredrik Siemund and Daniel Tovesson. Language server protocol for extendj. 2018.

[27] Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. Automatic index selection for large-scale datalog computation. *Proc. VLDB Endow.*, 12(2):141–153, oct 2018.