A Framework For Efficient Code Fixes Using Reference Attributed Grammars

Charlie Mrad
D16, Lund University, Sweden
ch3045mr-s@student.lu.se

Abstract

Code fixes tools are essential for any developer and are often a feature of the more popular IDEs available today. They provide an efficient way for a programmer to learn potentially better ways to write their code and fix the potentially erroneous code automatically. The problem today comes from a lack of efficiency in how quickly the code fixes can be found and generated and how language analysing extensions for IDEs are being developed. This paper proposes a framework that can efficiently find potential errors in Java code using IntraJ and provide quick-fix options. To avoid spending a significant amount of time implementing and maintaining LSP support for the framework Magpie Bridge is used. This combination provides a solid foundation to quickly provide an expandable array of code fixes. The initial evaluation seems to outperform Soot on a similar test case. There is still plenty of room for improvement on the proposed framework, but it appears to hold up reasonably well as a starting point.

1 Introduction

Static source code analysis provide helpful assistance in the software development process, allowing the programmer to detect critical problems, e.g., security flaws, run time errors, unintended behaviour. Recent studies [2, 3] have shown that the analysis needs to be responsive, evident in its feedback and results should be easy to access, i.e., collected in one central place, such as the integrated development environments (IDE). In this paper, the possibility of bringing the IntraJ [9] framework for static analysis into IDEs is explored to improve the user experience when developing in Java.

1.1 Introduction to IntraJ

IntraJ is a control-flow analysis tool for the Java language that uses the language-independent framework IntraCFG [9] to construct the control-flow graphs (CFGs). IntraJ superimposes the CFGs on top of the Abstract Syntax Tree (AST) nodes of interest and is not tied to the underlying nested structure of the AST. Overall, IntraJ provides an efficient and precise way of constructing CFGs for Java programs and can be used to create Java source code analyses. IntraJ is built as an extension of the ExtendJ [4] Java compiler. ExtendJ is written with JastAdd [6] and uses the Reference Attributed Grammar formalism, enabling on-demand evaluation and declarative specification. This allows ExtendJ to be easily extended with new modules such as IntraJ being a prime example. Similar to ExtendJ, IntraJis written using RAGs, meaning that the analyses performed by IntraJ are evaluated on-demand. Therefore, if an analysis is left unused, it does not have to be calculated and will not slow down the overall computation.

The static analyses IntraJ provides can be a valuable tool for helping programmers during the development process, and while IntraJ can provide useful information about analysis results it is all done through a command line interface. This is not ideal as the developer has to manually run commands as well as read through and match analysis results with the corresponding source code it refers too. That is where the Language Server Protocol (LSP) can come into play in order to facilitate the creation of extensions for IDEs and allow IntraJ to display its results in the IDE text editor itself. This is accomplished by deferring the IDE features related to language support to external language-specific servers which communicate with the IDE according to LSP specifications [1].

1.2 Introduction to Magpie Bridge

While LSP does provide a manageable way to integrate language support into multiple IDEs without requiring IDE specific code, it does not eliminate the IDE integration work entirely. Many IDE extensions still suffer from a significant portion of the code base being dedicated to LSP communications which becomes especially evident for smaller projects which can sometimes only have as little as $\frac{1}{3}$ of the code dedicated to actual language analysis as illustrated in Table 1. In order to attempt addressing this issue an LSP abstracting framework called Magpie Bridge was developed that allows users to simply set up their code to implement the Magpie Bridge interface and initialize an LSP server which in turn handles all the LSP communication for the extension [8].

1.3 Contributions

This paper ultimately provides an implementation of an extendable source code analysis plugin for Java using IntraJ to attain Java-specific control-flow information and Magpie Bridge to simplify the use of LSP and reduce the amount of code required significantly. The plugin is intended to

Course paper, EDAN70, Lund University, Sweden
January 12, 2022.

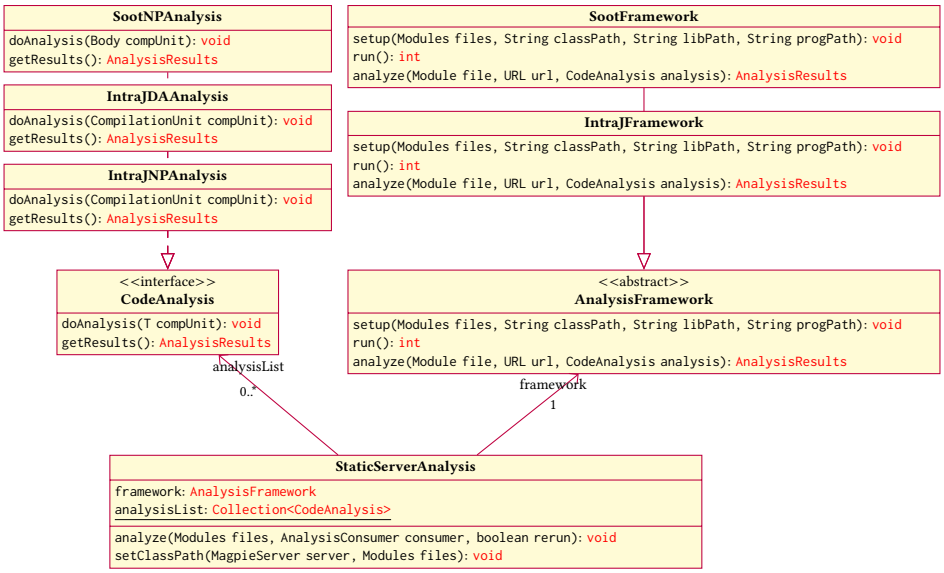

Figure 1. Design structure of the source code

Tool	Analysis	Plugin	Overhead (%)
SPLift	1317	3317	71.6
CogniCrypt	11753	18766	61.5
PMD	117551	33435	22.1

Table 1. Brief table derived from Linghui Luo et al. paper on Magpie Bridge [8]. Shows the worst examples of LSP code overhead. The Analysis and Plugin are expressed in LoC.

accommodate future work to extend the analysis capabilities by adding analysis types, JastAdd files for IntraJ and other analysis frameworks. The resulting extension brings the IntraJ control flow analysis framework into a more user-friendly and accessible format that can be employed whenever an IDE has LSP support compatible with Magpie Bridge, without requiring the work overhead that comes with implementing LSP support for the extension itself. However, for this paper, only Visual Studio Code is tested and evaluated.

Also demonstrated is the increased efficiency gained from using Magpie Bridge by counting lines of code and comparing to the aforementioned values in Table 1, which implement LSP support of their own. Additionally, the performance difference between IntraJ and Soot is showcased, Soot being another analysis framework that does not use reference attributed grammars [7]. This is done by measuring the time for the completion of an analysis from when it starts executing in the extension until it is finished and all the results are handed over to Magpie Bridge. These measurements are done on the ANTLR 2.7.2 project, specifically the *ANTLRParser.java* file. This is done to illustrate both how the extension performs on a sizable test case, and that the extension works on source code not written with the plugin in mind. Summarizing, the contributions of this paper are the following:

- A tool for finding precise analysis results efficiently using IntraJ and providing options for quick code fixes.
- The implementation of an extendable framework for creating static analysis extensions for multiple IDEs through Magpie Bridge.
- Evaluation of the execution time performance of IntraJ compared to Soot, a framework that does not use reference attributed grammars to perform its analyses.
- Demonstration of some of the utility of Magpie Bridge and how it affects LSP code overhead.

The rest of the paper is structured as follows. The first Section describes the implementation itself, how it works and the code structure. Section 3 discusses the evaluation process with the results of the performance evaluation and a look at the amount of code overhead generated for LSP communication. Finally, related works are discussed in Section 4.

2 The extension

The extension implemented is primarily targeted at performing static analyses on Java code. It is possible to extend it to support any language if supported by both the analysis framework being used and Magpie Bridge but currently this has not been done. The extension framework implemented in this paper is designed to be fairly flexible. It consists of mainly two parts, one core extension framework component and one extendable custom analysis component. Figure 1 gives a brief overview of the general structure. The way it

works is by defining an analysis framework and a set of code analyses to go along with it. The code analyses and analysis framework are represented by the `CodeAnalysis` interface and `AnalysisFramework` abstract class. It is then the job of a Magpie Bridge server analysis to define the interaction with Magpie Bridge and this is represented by `ServerAnalysis` which can be seen in Figure 1 as well. The method for extending the analyses is determined by if the analysis framework needed is already supported or not. If it is supported then all that is needed is to write another class that implements the `CodeAnalysis` interface and add an instance of that class to the static `analysisList` in `ServerAnalysis`. However, if the framework is not supported the a class that extends `AnalysisFramework` has to be written and as of the time of writing, some minor adjustments are still needed in `StaticServerAnalysis` to use the new framework. After that the analysis itself can be added just as if the framework was supported initially.
If IntraJ or ExtendJ is used as AnalysisFramework it is possible to write additional JastAdd files for the extension that are copied over and included in the respective framework before building. This is how the string equality analysis is implemented for IntraJ; a JastAdd file[1] was written that adds an additional warning message for equality checks and a new type of warning message was added that provides more complete source code positional information. This was then leveraged to create a string equality analysis that finds these warnings and displays them.

2.1 Performing an analysis step-by-step

Performing an analysis and displaying the results in the IDE starts with Magpie Bridge and ends with Magpie Bridge. The first step is to activate one of the predefined analysis triggers. This tells Magpie Bridge to fetch information about which files to analyze and sends them through to the ServerAnalysis. The ServerAnalysis the uses the supplied AnalysisFramework and all active analyses (defined as CodeAnalysis types) to perform the complete analysis. This procedure follows three steps as follows.

- First the AnalysisFramework is set up with any initialization that is required. For IntraJ this means calculating all the arguments for compilation and createing a new instance of IntraJ.
- Second the AnalysisFramework is run, performing any calculations it needs to do in order to run the analysis. IntraJ needs to be run with the arguments from the following step to build up the graph representation of the code required for analyses.
- Lastly the ServerAnalysis iterates all enabled analyses and runs the AnalysisFramework dependent analysis with the currently active framework. For IntraJ all that is needed is to find the relevant compilation units and invoke the relevant analysis method which may return warnings that are then converted to analysis results.

3 Evaluation

Evaluation of the extension has been performed on the metrics of source lines of code compared to other analysis extensions and analysis speed for IntraJ and Soot [7] using a null-pointer analysis.

3.1 Source lines of code

The number of lines of code excluding comments and blank lines are what will be used as the source lines of code (SLOC) metric. The implementation as of this writing contains a total of 1183 SLOC out of which 187 lines are purely used for evaluation, bringing the total SLOC used for analysis down to 996 lines. Further out of that number 75 lines come from a slightly modified file (MySourceCodeReader.java) from the Magpie Bridge code base that serves as a temporary fix for an issue until Magpie Bridge is updated. This brings the total down further to 921 SLOC. This includes code for two IntraJ analyses, one ExtendJ analysis and one Soot analysis. IntraJ provides a null-pointer analysis and a dead assignment analysis, ExtendJ provides a String equality analysis[2], and Soot also provides a null-pointer analysis but based on the Soot framework instead for the purposes of comparing to IntraJ.
Additionally there is no code specifically meant to handle LSP communication since all of it is handled by Magpie Bridge. However, the code used to communicate with Magpie Bridge still had to be written which took up 375 SLOC. To compare with the values from Table 1 that is an overhead of 40.1%, in other words 40.1% of the the code written for this extension is used for LSP communication, indirectly through Magpie Bridge. See Table 2 for the updated version that compares this extension with the rest of the entries from before. These results puts this extension below both SPlift and CogniCrypt in terms of overhead. PMD is the only one performing better in that regard still. Note however that the code base for PMD is very big and from looking at the original table [8] there looks to be a pattern where smaller projects suffer from worse overhead.

3.2 Analysis execution time

The analysis execution time is measured by running repeated analyses on the same source files a multitude of times. After each run the time taken is recorded and another run initiated if needed. A sample size of 10000 measurements per evaluation stage was used. Important to note is that the measurements are taken 200 iterations into each run in order to avoid measuring the start-up times and instead measure

[1] JastAdd files need to be in ./src/jastadd for the pre-build copy step to work.

[2] Simply checks if .equals(..) or '==' is used for string operands in Java

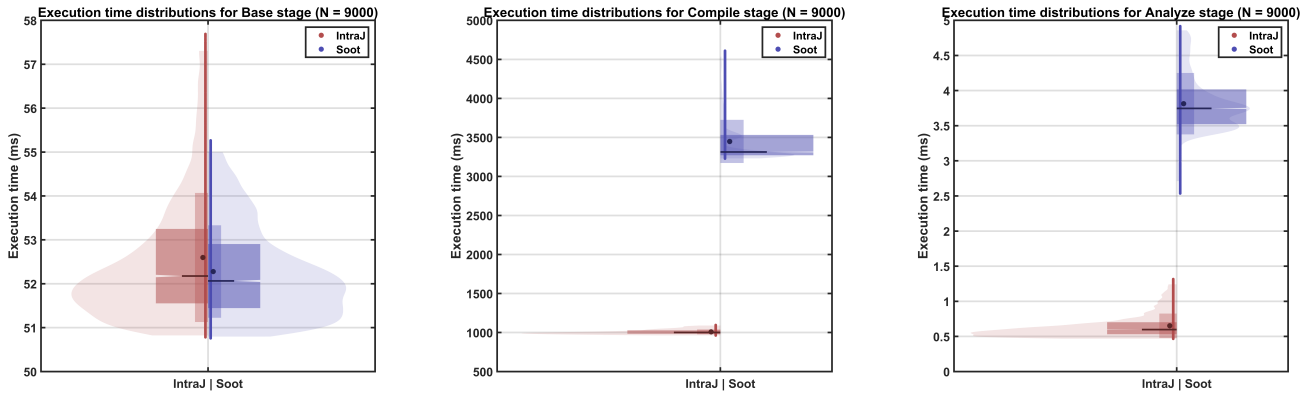

Figure 2. Execution time results for each stage of the evaluation. From left to right you have the base, compile and analyze stage. **N** in the titles stands for the number of measurements being plotted after outliers that are more than 3 standard deviations away from the average have been eliminated. For each figure the shorter black line represents the median, the black star represents the average, the thin colored box shows one standard deviation from the average in either direction, the thick colored box with the notches represents the first and third quartiles whilst the notches represent the 95% confidence interval around the median.

Tool	Analysis	Plugin	Overhead (%)
`SPLift`	1˙317	3˙317	71.6
`CogniCrypt`	11˙753	18˙766	61.5
`Our extension`	546	375	40.1
`PMD`	117˙551	33˙435	22.1

Table 2. Updated version of Table 1, now including the extension implemented in this paper.

steady-state times, which should be more representative of actual usage of the extension over a period of time. The only source file used to run the evaluation is *ANTLRParser.java* from ANTLR 2.7.2[3]. The analysis is specifically performed on Visual Studio Code, as this is the only IDE this extension has been tested on. Soot [7] was chosen to compare against IntraJ due to Soot being a well known and established analysis framework and to try and illustrate some of the performance differences between using a reference attributed grammar (IntraJ) and not doing so. The computer used to run the evaluation is a Lenovo Thinkpad T460s and it has the following hardware specifications.

- CPU: Intel Core i5-6300U, 2.4 GHz
- OS: Windows 10 Pro 64-bit
- RAM: 16 GB DDR4
- Graphics: Intel HD Graphics 520
- Storage: 128 GB SSD, SAMSUNG MZNTY128HDHP-000L1

The evaluation happens in three stages, dubbed Base, Compile and Analyze. The Base stage runs no framework functionality at all and is the same across both IntraJ and Soot. What is measured in Base consists of all background code that supports the analysis, most notably the calculation of the analyzed projects source paths, library paths and class path. This calculation will take a long time first time it is run due to Magpie Bridge initiating a project service. Therefore the project service initiation is done separately before evaluation begins in the interest of keeping the measurements fair for both frameworks. The Compile stage runs everything from the Base stage and also runs all the setup code required for the framework to run analyses. For IntraJ this means parsing the relevant code into an abstract syntax tree (AST) and inserting the control-flow attributes where they are needed. In the case of Soot a conversion from the internal representations (IR) used by Wala to the Jimple IR used by Soot is required. The time measured for the Compile stage is only the framework setup code, specifically `setup(...)` and `run()` methods from the `AnalysisFramework` class in Figure 1 The Analyze stage runs everything from the Compile stage and additionally runs a null-pointer exception analysis. For IntraJ the built-in null-pointer analysis is used while for Soot a custom made forward analysis is used. This custom made analysis is based on a sample implementation made by Navid Salehnamadi[4] and may not be optimal, however given how simple the analysis is it is hard to imagine how to improve on execution time without changing the underlying Soot code or something similar. The time measured for the Analyze stage is only the time it takes from starting the

[3]https://www.antlr.org/

[4]https://github.com/noidsirius/SootTutorial

analysis thread which runs the analysis until that thread has finished.
The results of the comparison for each stage outlined above is visualized in Figure 2. Seemingly IntraJ perfoms much faster with the Analyze stage median execution time being around 0.5 ms whilst Soot measures in at around 4 ms. However, the biggest performance load for Soot seems to come from the Compile stage and after further investigation and profiling the code using JProfiler 12.0.4 it turns out that the majority of the Compile stage is spent converting the Wala IR into Jimple. Further it turns out that the bad performance is mostly due to a large number of instantiations of temporary objects done in the Wala source code. In other words, this is not intrinsically due to how Soot operates but due to the fact that Magpie Bridge requires a conversion to use Soot analyses.

The Base evaluation stage displayed to the left in Figure 2 is largely the same for both IntraJ and Soot as expected since this stage is independent of any analyses and analysis frameworks. Both frameworks the Base stage results have a median of about 50 ms which means that the supporting extension code outside of the framework and analysis specific code takes around that long to execute.

4 Related work

Previous attempts at bringing the various errors and warnings from ExtendJ into the IDE using LSP have been made in the past as part of similar projects to the one presented in this paper [5, 10]. These older attempts managed to implement LSP support to display diagnostic messages over the code in-editor and showcased how their extensions then could display ExtendJ errors and warnings in a handful of IDEs. The extension implemented for this paper performs a similar function in that it displays some warnings, however for the purposes of this paper no ExtendJ native warnings or errors are considered[5]. The biggest difference between the work done in those previous attempts and this one is the use of Magpie Bridge to abstract the LSP support away from the extension itself. This has provided many benefits such as out-of-the-box ready LSP features that can be used without needing to know the technical details of LSP and simplified extension building for multiple IDEs that can be provided by Magpie Bridge and does not require the developer to figure out how to build for a particular IDE. For example, the Visual Studio Code extension building script used to build the extension was provided by Magpie Bridge, and the quick fix IDE prompts were part of the Magpie Bridge analysis result handling code. Another addition that sets this work apart from the previous is that the results displayed can be powered not only by ExtendJ but also by IntraJ, Soot and other analysis frameworks or compilers.

[5]The string equality check is not being considered native as it stems from an extension made to ExtendJ

5 Conclusion

From the evaluation results we can draw two main conclusions. First, the SLOC count results seems to indicate that the workload for implementing LSP support is indeed smaller when using Magpie Bridge. This is not strong evidence for this feature of Magpie Bridge as not enough cases are considered and this project is not of commercial scale by any means. But with more research and application of LSP abstracting frameworks such as Magpie Bridge the picture might become clearer.
Secondly, the execution time of IntraJ does seem to be smaller than that of Soot. Focusing only on the Analyze stage results, they seem to paint a promising picture of IntraJ efficiency in performing static intraprocedural analysis. It is however important to bear in mind that these results are derived from evaluating the performance on a single project (ANTLR 2.7.2), on a single file within said project (*ANTLRParser.java*), using only a single analysis (Null-pointer analysis). There is no guarantee that IntraJ will remain faster when more files, projects and analyses are considered. Another important distinction between IntraJ and Soot is that unlike IntraJ, Soot allows for more than just intraprocedural analysis. In addition to intraprocedural analysis Soot provides functionality to do interprocedural analysis as well as transforming the byte code for optimization purposes. Yet another consideration to be made is that the speed difference observed between Soot and IntraJ is definitely attributable to RAGs on-demand evaluation feature. More controlled tests would be needed to determine this.
Finally as an additional note it should be mentioned that the Compile stage execution times could potentially be improved for IntraJ. Right now the Compile stage code runs IntraJ on the full project, classpath, source files, and libarary files included. For the purposes of intraprocedural analysis technically only the file to be analyzed needs to be examined, however this would most likely produce compilation errors wiht IntraJ. If those compilation errors are ignored and the AST is built with all CFG attributes present, then IntraJ should still be able to run the analyses required. Doing this could improve the Compile stage times for IntraJ considerably and more importantly it would decouple the Compile stage code execution time from the project size, instead only depending on the individual file size to be analyzed. This could be part of some future work done on this extension. Other work that can still be done would include extending the analysis suite to support more types of analyses, extending the number of supported analysis frameworks, including a `AnalysisFramework` reference in `CodeAnalysis` so that the `StaticServerAnalysis` class does not have to know about the `AnalysisFramwork` and can thus perform analyses regardless of which framework they belong to since all framework related information is embedded in the analysis class. Also for the evaluation, there is a lot of work to be

done still; evaluating on other projects and files and doing more controlled tests that can better determine where performance differences arise would be the two most prominent places to start.

Acknowledgments

I would like to thank my supervisor Idriss Riouak for his patient guidance and help with troubleshooting.

References

[1] Hendrik Bünder. 2019. Decoupling Language and Editor-The Impact of the Language Server Protocol on Textual Domain-Specific Languages.. In *MODELSWARD*. 129–140.

[2] Lisa Nguyen Quang Do. 2019. *User-centered tool design for data-flow analysis.* Ph.D. Dissertation. University of Paderborn, Germany.

[3] Lisa Nguyen Quang Do, James Wright, and Karim Ali. 2020. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering* (2020).

[4] Torbjörn Ekman and Görel Hedin. 2007. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*. 1–18.

[5] Joakim Ericson. 2018. Language Server Protocol for ExtendJ. (2018).

[6] Görel Hedin and Eva Magnusson. 2003. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming* 47, 1 (2003), 37–58.

[7] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. Soot-a Java bytecode optimization framework. In *Cetus Users and Compiler Infrastructure Workshop*. 1–11.

[8] Linghui Luo, Julian Dolby, and Eric Bodden. 2019. MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Alastair F. Donaldson (Ed.), Vol. 134. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 21:1–21:25. https://doi.org/10.4230/LIPIcs.ECOOP.2019.21

[9] Idriss Riouak, Christoph Reichenbach, Görel Hedin, and Niklas Fors. 2021. A Precise Framework for Source-Level Control-Flow Analysis. In *21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM*.

[10] Fredrik Siemund and Daniel Tovesson. 2018. Language Server Protocol for ExtendJ. (2018).