Master Thesis Nr. enter number

Soot-Based Configuration Generator for Analysis Writers

am: 24.05.2021

FRAUNHOFER INSTITUTE FOR
MECHATRONIC SYSTEMS DESIGN
Zukunftsmeile 1
D-33102 Paderborn

Contents

1 Introduction

Static analysis is a software engineering technique that is used to examine the application code without executing it. It helps in performing code analysis, compiler optimizations, code inspection, identification of bugs and security vulnerabilities, etc. Multiple open-source frameworks like Wala [17], Soot [14] are available to perform static analysis for Java programs.

Soot is an open-source static analysis and code optimization framework for Java and Android applications. It provides the users with various options and configurations to perform static analysis. Various analyses like call graph construction [1], points-to analysis [15], intraprocedural analysis [12], interprocedural analysis [12], live variable analysis [18], taint analysis [2], demand-driven pointer analysis [6], arithmetic expression analysis, etc can be performed using the Soot framework. Soot can take input from Java, Andriod, Jimple [14], Jasmin [10] representation and can produce output in the form of Jimple, Java bytecode, Android bytecode, or Jasmin representation. It can transform bytecode (.class) files into Jimple code [14] and also Jimple code back to Java bytecode. Soot framework can be used as a command-line tool or as a library by adding the relevant dependencies. In this thesis, I focus only on using Soot as a library.

Soot execution consists of different phases as represented in Figure 1-1. Each of these phases may include sub-phases. Each of these phases in Soot is implemented with the packs. Packs are the collection of transformers that further implement the sub-phases of phases in Soot [13]. Transformers contain instructions to convert the Java bytecode to Jimple code in a specified manner. In Figure 1-1, T1, Tn are used to represent transformers. Phase 1, Phase n represent different phases of Soot execution. Figure 1-1 depicts that the Java bytecode undergoes various phases in the process of conversion to Jimple code. For example, Soot has a phase called *JB* which is used to build the Jimple bodies [13]. Jimple bodies provide Jimple representation of an input bytecode method. *Empty Switch Eliminator* (JB.ese) is one of the transformers in the *JB* phase which is used to eliminate the empty switch statements present in the input Java bytecode method during its conversion to Jimple code. Some of the transformers in Soot are enabled by default to perform optimizations in the Jimple code. Soot provides users with multiple options to manipulate these sub-phases or build their customized transformers to configure Soot as

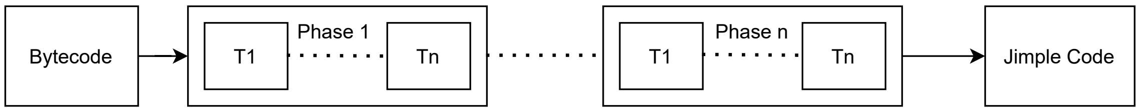


Figure 1-1: Soot Bytecode Transformation to Jimple

per their requirements. These options are called phase options.

There are five kinds of phase options [13]:

- Boolean options: These options take *true* or *false* values. For example, an option called *enable* takes these boolean values to activate or deactivate a particular feature in Soot.
- Integer options: These options take integer values. For example, an option called *jdkver* takes an integer value and sets the JDK version for the analysis.
- Multivalued options: These options take a value from a set of allowed values specific to that option. For example, an option called *propagator* takes a value from its specified set {iter, worklist, cycle, merge, alias, none}. Its default value is worklist.
- Floating-point options: These options take floating-point values. For example, *java-version* takes a floating-point value to force the Java version of bytecode generated by Soot.
- String options: These options take arbitrary string values as input. For example, an option called *cp* takes a string argument and sets it as the classpath for Soot (refer to section 2 for Soot classpath explanation).

With help of these options, the users can modify the default behavior of Soot and perform analysis as per their requirements.

Soot framework is in place for years and still, many novice users face issues while configuring Soot due to a lack of documentation and prerequisite knowledge specific to Soot. As mentioned above Soot provides its users with multiple options [13] for configuration. Users are mostly aware of certain basic options and most of them lack knowledge about the multiple options that could enhance the performance of Soot. Working on certain options is also not clear to the users due to a lack of usage and documentation. Issues faced by the users are discussed in detail in the next section.

The main aim of this thesis is to provide support for the novice users of Soot in terms of its configuration and also encourage the users to utilize Soot's multiple options to perform their analysis.

2 Problem Description

Soot framework was introduced in the late ’90s. After its introduction, over the years it has undergone many changes. As the field of static analysis gained importance [21] and got enhanced with new techniques, Soot was also updated with new features that support these new techniques. For example, the Geometric Encoding technique [19] for points-to analysis was introduced in Soot after its publication in 2011. Soot provides various options and configurations to perform the points-to analysis using the Geometric encoding technique after its introduction. Hence there is substantial addition of new features to the Soot framework over these years. With the addition of new features, the list of options provided by Soot has also increased.

Two main issues that the users face while working with Soot are:

- Lack of knowledge regarding the options that are available in Soot, and usage of multiple Soot options.
- Configuration of Soot like setting up of Soot classpath.

These issues arise mainly due to a lack of documentation, and prerequisite knowledge related to Soot.

Currently, the Soot framework provides its users with 22 phases and each of these phases contains multiple options to configure the Soot framework according to their application needs. Most of the users are not aware of many features that exist in the Soot framework and they are not able to gain complete benefit from these features. For example, when the users are constructing the call graph they can opt to include/exclude internal library function calls by using an option -no-bodies-for-excluded. In this scenario, Soot only considers application calls for the construction of call graph and exclude internal calls between library functions. In case if the resulting call graph is huge and causes out-of-memory exceptions then the users can exclude internal calls of library functions by using this option. If the users are not aware of this option, then they face issues related to out-of-memory exceptions if they are trying to construct call graphs for larger applications. With a large set of Soot options, it is not feasible for the users to be aware of all the options present in Soot by going through the available documentation.

Whenever a user decides to work with the Soot framework, they must have to attain knowledge about the Soot classpath [13], application classes [13], library classes [13], phantom classes [13], etc. There is also documentation available for the users to get started with Soot. It helps the users to gain some basic knowledge regarding application classes, library classes, phantom classes and also guides them on how they have to set the Soot classpath, etc. Even with this documentation, the users face issues in understanding the concepts of the above-mentioned Soot classpath and various classes.

Users mostly face challenges in setting Soot configurations and get stuck with errors related to setting up of the Soot classpath. There is also a GitHub repository for Soot [14] where the users can raise the issues that they are facing and get help from the maintainers of the Soot framework. As an open-source project, Soot is maintained on a voluntary base. Therefore, it lacks maintenance and more adequate support for its users.

To better understand the issues that most users have, we have analyzed 625 issues from Soot's repository separately to identify the issues that were related to the configuration of Soot and usage of Soot's options. We wanted to identify the issues related to the configuration of Soot, usage of Soot's options, documentation-related support, etc. To analyze the issues, we went through details of the issues by considering the conversations between the Soot users and Soot maintainers. Based on their conversations we identified the issues that were related to the configuration of Soot like the setting of the Soot classpath, Usage of the Soot options like retaining original variable names, etc. We had an inter-rater agreement between two people in 80% of the analyzed issues for considering or not considering the issues to be related to the configuration of Soot or usage of its options. Further, we discussed together, our 20% disagreements and identified almost 32% of the analyzed 625 issues were related to either configuration of Soot or usage of its options. Most of the issues we identified were related to the Soot classpath setting, usage of the Soot options to alter call graph or extract the original variable names, etc.

In this thesis, I address the above-mentioned problems by designing a method that the users can refer to and explore the multiple options that are available in Soot. Users can consider only those options that are related to their applications such as call graph construction, dead code analysis, etc. This reduces the burden on the users while dealing with multiple options of Soot. To provide support, I will also develop a tool that helps the user to configure Soot automatically. This will help the users to overcome the issues related to the setup of the Soot classpath. The tool will generate a stub project based on users' Soot configuration setup. Next section provides detailed explanation regarding the proposed solution to address the above-mentioned issues.

3 Contribution

The primary goal of this thesis is to help novice users of Soot with the configuration and usage of its options. To address the issues mentioned in Section 2, I will propose a systematic approach for the users to specify the configurations of Soot. This approach also includes a tool that will help the users with the configuration of Soot. To provide support to the users with the various configurations of Soot, I will develop a feature model that represents the various features of Soot that can help the users to configure Soot for their application depending on their needs. With this feature diagram, the users can identify which features of Soot they have to choose for their application depending on the kind of analysis and optimization they are looking for. This thesis also includes the development of a code generator tool that automates the process of Soot configuration setup and generates a Maven project [5] with stub code.

3.1 Feature Diagram

Feature diagrams are kind of tree-like structures that are used to represent various features/functionalities of the software product lines [11]. They are used to specify, usage of which features are mandatory or optional. They also provide information about alternate features that can be used. For presentation, the feature model is being split into multiple sub-feature models. Below mentioned figures Figure 3.1.1, Figure 3.1.2, and Figure 3.1.3 represent the feature models.

Figure 3.1.1, presents the feature model with the main Soot options. Node Soot represents a concrete type and consists of five children. The Black dot above the child nodes

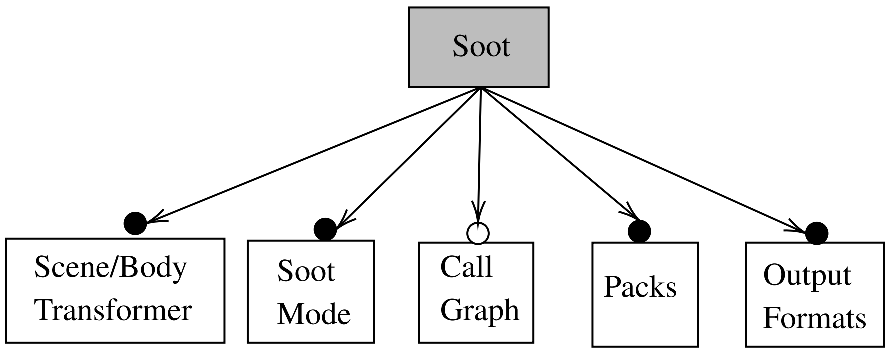


Figure 3.1.1: Feature Model Soot Main Options

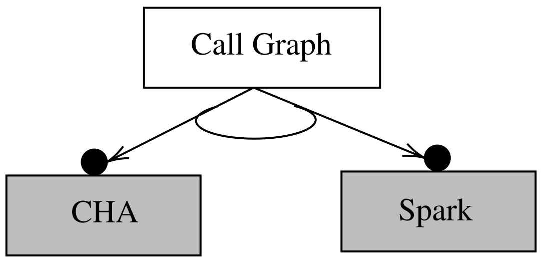


Figure 3.1.2: Feature Model Call Graph Options

represents they are mandatory and the hallow white dot represents optional. Each of the child nodes is an abstract type.

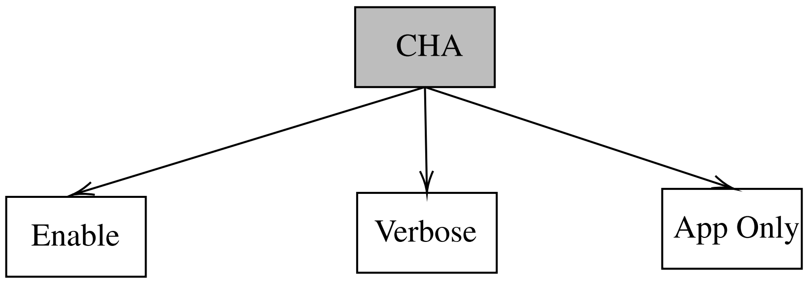


Figure 3.1.3: Feature Model CHA Options

Figure 3.1.2 represents the feature model for call graph options. The Call graph is an abstract node. It has two alternative and mandatory children Class Hierarchy Analysis (CHA) [4], SPARK [9]. Likewise again CHA and Spark will have their sub-feature models representing the various options available in Soot for them and so on.

Users can refer to these feature models to choose the options they need for their analysis. Suppose if a user wants to construct a call graph, he can focus on the feature model that refers to the call graphs and choose the variant of call graph they want to use in their analysis such as CHA, SPARK, etc. Each variant of these call graphs has multiple options available to further configure the call graph algorithms. Figure 3.1.3 represents the feature model of CHA options. Users can retrieve the statistics of the call graph by setting an option called verbose. They can choose to include only the application calls in the call graph using App only option. By using these options of CHA users can construct the call graph as per their requirements. These three feature models are just an example and the rest of Soot options will also be represented as feature models in a similar manner.

3.2 Soot Code Generator

Soot Code Generator tool will be based on the designed feature model. Users will be provided with a user form consisting of a wizard where they can select to enable or disable the particular feature of Soot. This user form will be in sync with the feature model that is designed and will have these below mentioned options to specify :

- Soot classpath.
- Application classpath/Library classes path.
- Phantom class option.
- Call graph options.
- Transformers options including custom transformers.
- Packs options.
- Solver options (Optional).
- Alias options (Optional).

Once the users select the options required for their configuration needs and specify a path to their .class files that are needed to be analyzed by Soot, a tool will be developed to set up the Soot configurations with these features and generate a Maven project containing stub code in the form of a zip file. Users can unzip this zipped file and import it as a Maven project in the IDE of their choice.

Figure 3.2.1 refers to a business process model that represents the actions of users throughout the process of stub project generated using the Soot Code Generator tool. Suppose a user wants to construct a call graph and analyze and identify the methods that are not reachable then the users can refer to the feature model and choose the options referring to the call graph as per their needs and generate a stub project. Once they have generated the Stub project and imported it as a Maven project then they can just add the compiled files to the path specified by them in the configuration wizard of the tool and generate a call graph and analyze them.

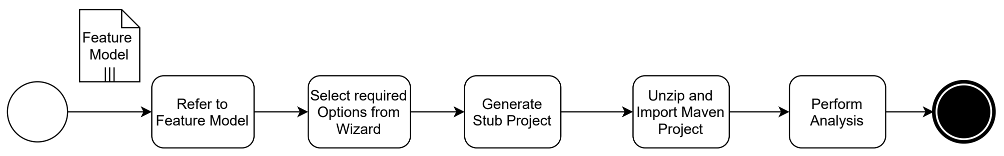


Figure 3.2.1: User Business Process Model

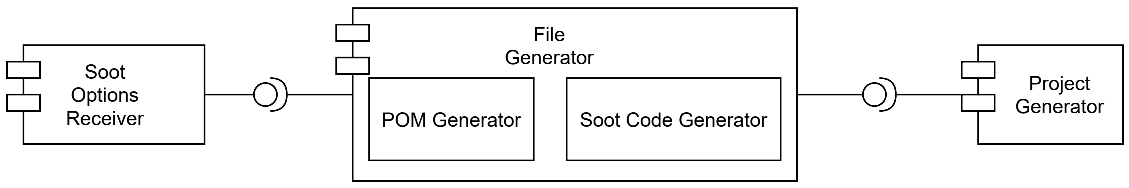


Figure 3.2.2: Soot Code Generator Component Diagram

Figure 3.2.2, provides the component diagram of the Soot code generator tool. It consists of three main components:

- Soot Options Receiver: This component contains the instructions that are necessary to receive the options from the user.
- Files Generator: This component consists of two sub-components POM Generator and Soot Code Generator. POM generator will contain the instructions to generate the pom.xml file with the required dependencies and properties. Soot Code Generator will consist of the instructions that are required to generate the necessary .java files required to set up Soot based on the options provided by the users from the wizard.
- Project Generator: This component will contain the instructions required to generate a Maven project containing stub code as a zip file. Once the zip file is generated, the users can unzip the file and import them as a Maven project.

Figure 3.2.3 provides the business process model of the entire design and development of the feature model and the Soot Code Generator tool. Initially, the feature model will be designed and developed. The Feature model will be similar to the ones that are presented above in the figures Figure 3.1.1, Figure 3.1.2 and, Figure 3.1.3. Based on the feature model, the wizard of the Soot Code Generator tool will be designed. As mentioned above, this wizard will be designed to capture the Soot classpath, Paths to different kinds of classes utilized in Soot, different packs, etc. Templates will be created to capture Soot configuration options selected by the users from the wizard. These templates will have placeholders and based on the options selected, placeholders will be updated in these templates. After modifying the templates, a Maven project containing stub code will be generated as a zip file. Users can import this to the IDE of their choice and continue with their analysis.

With time constraints initial focus will be on options that mostly pertain to optimization like options included in the JOP packs of Soot [13] and call graph construction and generate Maven projects. In case if time permits, the feature model will be enhanced with

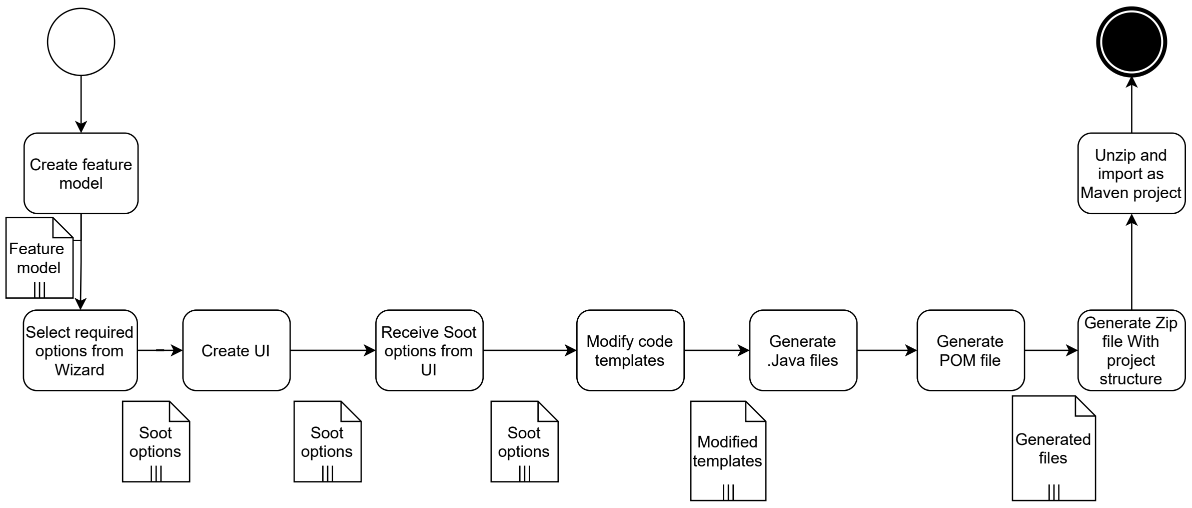


Figure 3.2.3: Design and Development Business process Model

options for solvers (Heros, Boomerang), and the tool will be enhanced to generate Gradle projects [8] as well.

The tool will be implemented as a REST API. For tool implementation, Spring Boot [16], React [7] and Xtend [20] languages will be used. Xtend is used to perform the code generation. Xtend is mainly used to generate Java source code. All the existing libraries of Java language can also be used with Xtend language. Spring Boot provides functionalities for the creation of REST API. React is a library of JavaScript that provides functionalities for creating user interfaces. React will be used to design the user wizard of the tool.

4 Evaluation

The evaluation of the designed feature model and Soot Code Generator tool will be performed by answering the following two research questions.

1. **RQ1 Does the features and options provided in the feature model, completely cover the entire set of options required to perform the specific analysis?**
   It must be evaluated that options provided for the features are opted and provide the necessary functions of Soot to perform the required tasks like dead code analysis, call graph construction, taint analysis, etc. For this purpose, interviews with the Soot experts (5 members) will be conducted and feedback will be collected from them on the designed model. The Completeness and correctness of the designed model for analyses will be verified. Based on their feedback, if there is a need to modify or enhance the feature model, it will be modified.

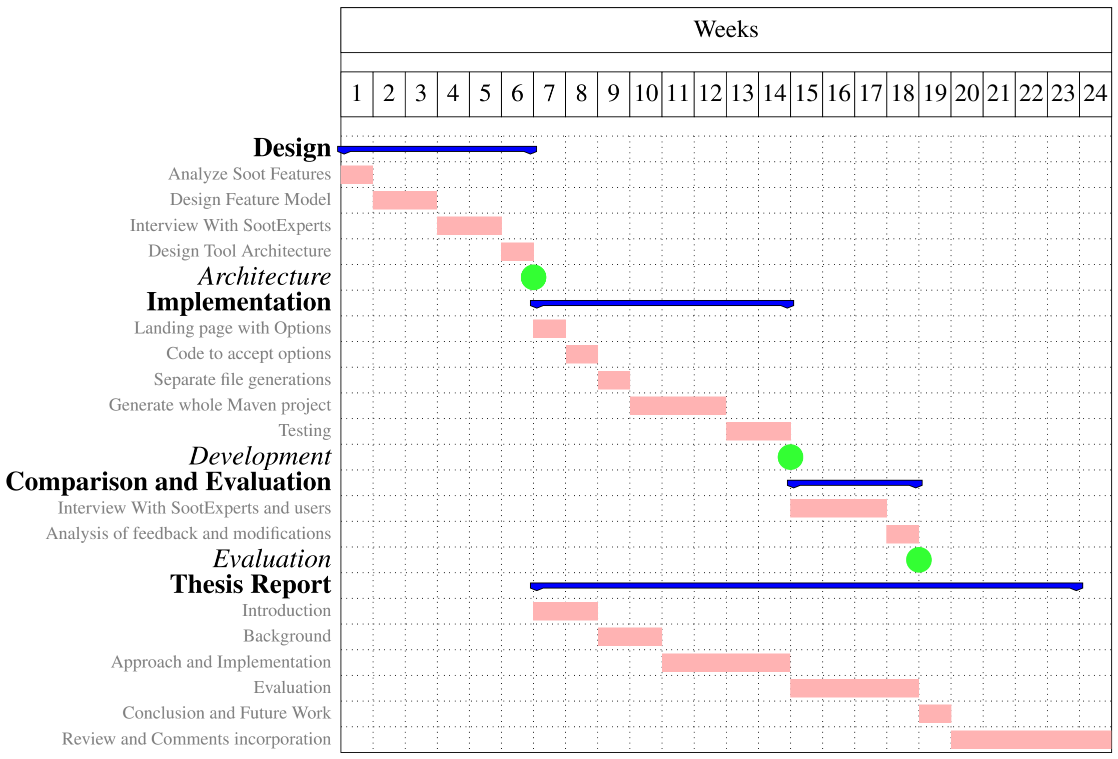
2. **RQ2 How useful is the Soot Code Generator?**
   This will be performed using the within-subjects [3] technique. The within-subject technique is used in scenarios where the same user is asked to test multiple user interfaces, multiple tools, etc. Here, I will set up interviews with the users of Soot (5 members). Users will be asked to set up the Soot configuration using the Soot Code Generator tool by specifying the Soot classpath, application classes path, library classes path, and other related options. Users will also be asked to manually perform Soot set up without using the tool. For these interviews with the Soot users, I will choose a simple call graph construction analysis. Users will be asked to perform the call graph construction by using the tool as well as without the tool. Their feedback regarding their experience with the tool will be noted. The amount of time taken to configure Soot with the help of the tool and without it will be noted and compared. The number of lines of code generated from the tool in comparison with the manual setup will also be compared. Based on the users' feedback modifications could be performed if needed.

5 Preliminary Structuring

The preliminary structure of the thesis is described as follows:

1. **Introduction** Includes motivation on the usage of Soot for performing various techniques of static analysis.
2. **Problem Description** Includes an overview of the problems with the Soot configuration and usage of the Soot options.
3. **Contribution** Describes the feature model and the Soot Code Generator tool designed to automatically perform the Soot configuration and generate a stub project.
4. **Implementation** Includes technical details about the Soot Code Generator tool implementation and guidance on how to use it.
5. **Evaluation** Includes details about the various interviews conducted for the evaluation of the Soot Code Generator tool and the designed feature model and details about the efficiency of the Soot Code Generator tool.
6. **Conclusion and Future Work** Summarizes the contributions made as part of the thesis and points out the areas for future development.

6 Time Plan



The time plan depicts four phases as mentioned below:

- Design: In this phase, various features of Soot will be analyzed and a feature model representing the various features of Soot will be designed. Once the feature model is in place, interviews with the Soot experts will be conducted and the feedback will be collected. This phase also includes the design of the Soot Code Generator tool architecture. By the end of this phase Architecture of the tool, milestone will be completed.
- Implementation: This phase involves the implementation of the Soot Code Generator tool. It includes the development of the user wizard, modification of the code templates to capture the user configurations, and generation of the stub project. By the end of this phase Development of the tool, milestone will be completed.
- Comparison and Evaluation: This phase includes evaluation of the Soot Code Generator tool. Interviews with the Soot users will be conducted and the feedback will

be collected. By the end of this phase Evaluation of the tool, milestone will be completed.

- Thesis Report: This phase includes drafting various sections of the final thesis report.

The proposed schedule is only the estimation and can be overlapped in time. Buffer time is included in the time allocated to write the thesis report which can be used in accommodating any unexpected events or difficulties faced.

7 Bibliography

[1] K. Ali and O. Lhoták. Application-only call graph construction. In *European Conference on Object-Oriented Programming*, pages 688–712. Springer, 2012.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[3] G. Charness, U. Gneezy, and M. A. Kuhn. Experimental methods: Between-subject and within-subject design. *Journal of Economic Behavior & Organization*, 81(1):1–8, 2012.

[4] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101. Springer, 1995.

[5] T. A. S. Foundation. Maven. `https://maven.apache.org/`, 2021. Online; accessed May 2021.

[6] N. Heintze and O. Tardieu. Demand-driven pointer analysis. *ACM SIGPLAN Notices*, 36(5):24–34, 2001.

[7] F. Inc. React framework. `https://reactjs.org/`, 2021. Online; accessed May 2021.

[8] G. Inc. Gradle. `https://gradle.org/`, 2021. Online; accessed May 2021.

[9] O. Lhoták. Spark: A flexible points-to analysis framework for java. 2002.

[10] J. Meyer and T. Downing. *Java virtual machine*. O'Reilly & Associates, Inc., 1997.

[11] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 139–148. IEEE, 2006.

[12] M. Sharir, A. Pnueli, et al. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences ..., 1978.

[13] Soot. Soot commandline options. `https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/options/soot_options.htm`, 2021. Online; accessed May 2021.